Composable Dynamic Voltage and Frequency Scaling and Power Management for Dataflow Applications

Kees Goossens[1], Dongrui She[1], Aleksandar Milutinovic[2], Anca Molnos[3]
[1]Eindhoven University of Technology [2]University of Twente [3]Delft University of Technology
{k.g.w.goossens, d.she}@tue.nl, a.milutinovic@utwente.nl, anca.molnos@tudelft.nl

***Abstract—Composability* means that the behaviour of an application, including its timing, is not affected by the absence or presence of other applications. It is required to be able to design, test, and verify applications independently. In this paper we define composable dynamic voltage and frequency scaling (DVFS) hardware, and composable power management. We ensure that the functional and temporal behaviours of an application are not affected by other applications, even when they are power managed.**

For dataflow applications with worst-case execution times per task, our power management is also *predictable*, i.e. guarantees end-to-end real-time requirements, even when the application is mapped on multiple processors that are power managed independently. Our method can be used with various DVFS architectures, such as on-chip and off-chip VF regulators.

Our FPGA implementation models a system with multiple tiles, each containing a processor with local memory running a real-time operating system (RTOS) and power management. Tiles are interconnected by a network on chip, and communicate using shared memories. Experiments indicate energy savings of 68% w.r.t. no power management, and 40% w.r.t. power gating only. We also demonstrate composability and predictability on the platform in the presence of power management.

I. Introduction

Low energy consumption is important for systems on chip (SOC). *Dynamic voltage and frequency scaling* (DVFS) is often used to trade a linear processor slowdown for a potentially quadratic decrease in energy consumption. This trade-off has been exhaustively addressed for single processors, and for multi-processor SOCs [1], [2]. We propose an architecture that uses existing DVFS hardware to build composable and predictable SOCs running multiple applications.

To deal with the complexity of system design and verification, the concept of *composability* has been advocated [3]–[6] and practised [7]. Behaviours of different applications are independent, to be able to develop, test, verify, and execute applications in isolation before they are integrated to a system. A composable SOC then ensures that applications already integrated do not affect the newly added application, and vice versa.

Embedded systems contain a mix of best-effort applications, and those that have (hard or soft) real-time requirements, such as radio pipes, and video and audio decoding. *Predictability*, or timeliness, is required to guarantee that each application meets its deadlines, while composability ensures that multiple (real-time) applications are independent. Predictability is not easy to ensure when applications use multiple shared resources (processors, interconnect, memories).

Predictability and composability in the SOC domain have been demonstrated but without [6] real-time operating system (RTOS) or power management. The composable RTOS used here was introduced in [8], and a predictable power-managed RTOS in [9]. Here all elements are combined, for predictable and composable low-power power management, i.e. the temporal (and possibly functional) behaviour of an application is not affected by the power management of other applications.

A. Contributions

In this paper we focus on streaming applications that can be expressed in variable-rate dataflow, possibly with cycles and data dependencies. They may be mapped on multiple resources. Task code and data fit in the processor scratch pad, and tasks use explicit communication between tasks using shared (remote) memories [6]. We assume a fixed mapping of tasks of different applications to multiple processors. Processors run the same RTOS, but schedule and manage power independently. We focus on dynamic power, and extend prior work on composability by introducing composable DVFS (hardware) and dynamic power management (software). Although not addressed here, static power can be taken into account by modifying the next-frequency function (Section V-D). For dataflow applications for which a worst-case execution time (WCET) for each task is given, our approach is also predictable, i.e. guarantees end-to-end throughput and latency even in the presence of power management.

We first define a hardware architecture per tile (processor, local bus and memories) to enable composable and predictable DVFS. We assume the fast DVFS hardware of [10]. The essential idea is to *schedule* DVFS *operations at precise points in the future*. We remove the variation in RTOS behaviour and VF transitions by scheduling subsequent operations at their worst-case completion times. As a result, RTOS time slices are independent in terms of duration of VF transitions, enabling composability: an application's number and timing of processor cycles is independent of other applications. Moreover, at any frequency, the number of cycles allocated to an application time slice can be budgeted (minimum and maximum bound on the number of cycles), which is required for predictability. In an earlier paper [8] a basic tile version without DVFS was introduced.

The DVFS hardware is the first step to ensure that each time slice contains a budgeted number of processor cycles (instructions). However, to guarantee an absolute finishing time, we must also bound the time that each instruction takes. This is often not the case for processors, because a load instruction, e.g. to a shared off-chip DRAM, can take thousands of cycles to complete. (In theory even forever, if the slave does not respond.) Because most embedded processors do not support interruptable load instructions (hardware-multi-threaded processors being the exception), we equip the processor with direct memory access (DMA) controllers that effectively implement *interruptable load instructions*. The disadvantage is that the application must be aware where its data are mapped, either in local memories (when it can use normal loads), or remotely (when it must use a DMA to access it). Fortunately, many embedded-system applications use local scratch pads and explicit synchronisation (via local or remote memories) between tasks. DMA access is then hidden in the communication library without changes to the application, as we show later.

The software that uses the DVFS hardware consists of several parts. First, drivers to program the VF operating points, and interrupt service routines. Second, each processor independently runs the same RTOS, which uses *two-level scheduling: composable between applications, and (optionally) predictable within applications*. The former uses time-division multiplexing (TDM). The latter schedules tasks within each application with an application-specified scheduler (currently TDM, CCSP [11], or round robin). Finally, explicit inter-task (and inter-processor) communication with the C-HEAP protocol [12] allows the RTOS *to monitor the progress and slack of tasks*. For real-time applications with given worst-case execution times of tasks, the RTOS *computes minimal operating points such that deadlines are always met*, using the application's slack.

The remainder of this paper first discusses related work in Section II and background in Section III. It then follows the structure of the above contributions in Sections IV-V. Section VI contains experimental results, and Section VII concludes.

II. Related Work

Related work falls in two categories: predictable dynamic power management, and composable and predictable SOC architectures. We know of no prior work on composable power management, which is the contribution of this paper. Since we change operating point at every RTOS time slice, we require efficient per-tile VF scaling hardware that allows fast and fine-grained VF switching, such as [10], [13].

[1], [2] give good overviews of low-power techniques. [14] surveys methods that suit real-time multiprocessors. Here we only consider those allowing inter-task dependencies, as we also do. [15]–[20] consider applications described as directed acyclic task graphs (DAG), [9], [21] tackle the more general case of cyclic dataflow task graphs, and [22] models application using network calculus, expressing dependencies implicitly, via arrival curves. [15] combines DVFS and adaptive body biasing in heterogeneous platforms; [16] iteratively distributes slack over the DAG's nodes. [18] selects the optimal number of (symmetric) processors and their VF points, to minimize system power under throughput constraints. [20] proposes a global scheme that requires inter-processor synchronization when slack is reclaimed. Here, the different RTOS instances do not communicate. [17] extends the DAG targeted work considering conditional task graphs. [9] considers streaming applications modeled as cyclic dataflow graphs. [9]'s RTOS is closest to our approach, but is not composable, nor implemented on FPGA. [21] uses a similar dataflow model and streaming applications as in [9], exploring the design space for memory size and processor voltage frequency levels, while meeting a throughput constraint, but without processor sharing. [22] proposes a DVFS strategy in two phases: off-line worst-case bounds that are conservatively improved at run time.

Composability for SOC design has been gaining ground [3]–[7]. It has been demonstrated for different components of a SOC: networks on chip (NOC) [23], [24], memory controllers [25], and processor real-time operating system (RTOS) [8], [9], [26]. [6] combines a number of these components into a composable SOC. However, none of these address (composable) power management.

III. Background

We focus on dynamic *power*, given by $P_{dyn} = \alpha C V^2 f = \alpha C V^2 w/t$, where α is the switching activity, C is the switched capacitance, and V and $f \in [f_{min}, f_{max}]$ define the VF operating point. Alternatively, work w is the number of cycles executed in time t. The *energy* spent is then $E_{dyn} = P_{dyn} t = \alpha C V^2 w$. To minimise energy, the voltage must be scaled to the lowest value supporting the frequency required to meet a deadline. For simplicity, we define the operating point by f alone, and we assume a linear relationship between V and the maximum f it supports.

Our platform (see Fig. 1) is composable: each resource (computation, communication, storage) can be partitioned using time-division multiplexing (TDM) in smaller "virtual resources" that are allocated to multiple applications, such that the functional and temporal behaviours of the virtual resources are independent and do not interfere. Each task receives a budget (TDM slots) in a given period (TDM wheel). TDM wheels of different resources may not be aligned, because they may run at different variable frequencies. Tasks are scheduled only based on their allocated budget, and not based on any application or task deadline. We use a composable and predictable NOC [23]. The processors with local memories use an extension of the RTOS of [8], [9], [26], which is composable and predictable (Section V).

The application programming model comprises tasks with a variable-rate dataflow semantics, communicating using finite-capacity FIFO channels. Tasks (actors) fire when all their input data and space for their all outputs are present, and then

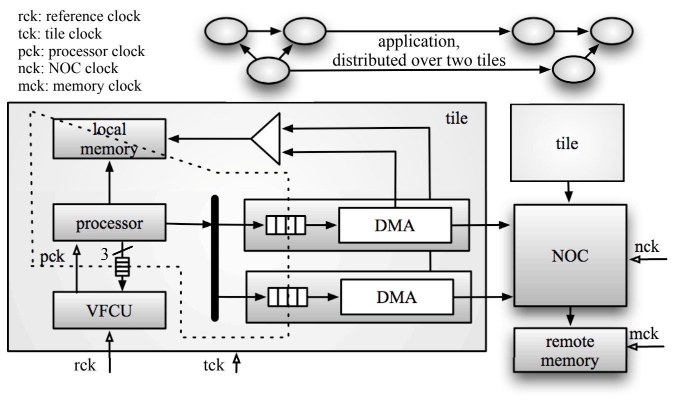

Fig. 1. Our composable and predictable platform.

execute without stalling on other resources (but they may be preempted). The code of the task, and its data reside in the local memories (scratch pads).

We use the dataflow paradigm to model the applications, the platform, and the binding of applications to the platform [23]. For multi-rate cyclostatic dataflow applications with a WCET for each actor, it is then possible to compute their end-to-end throughput and latency, given their binding to the platform [27].

An application contains a number of communicating tasks. Some tasks have a *worst-case work* (*wcw*), which is defined as the worst-case number of clock cycles that the task takes. *wcw* takes WCET seconds at f_{max}. Note that because tasks run from local memory after their input data and output space is available, their work does not depend on the frequency. However, the actual work of a task is often less than *wcw*; the difference is spare capacity, or *slack*. Slack also arises from early data arrival, pessimistic modelling, or when the set of running applications does not use the full capacity of the platform. Slack can be expressed in work (cycles) or, at a given operating point, in time (remaining time until the budget depletes).

Slack is used to reduce the frequency and voltage to reduce the power and energy. Our innovation lies in the fact that we do this composably (eliminating interference between applications) and predictably (respecting application deadlines). It is important to note that *slack can only be distributed within applications*. Otherwise, an application would receive more capacity (cycles, bandwidth, memory) than when other applications would be present. Although this may seem positive, it changes its temporal behaviour, and verification then becomes dependent on other applications, which is not composable.

IV. Composable DVFS Hardware

The clock and power domains of the system are shown in Fig. 1 by the dashed lines, running through several blocks. The local memory is a dual-port memory, but the tile architecture can be changed to use single-port memories, if required. The VF control unit (VFCU) and direct memory access (DMA) interfaces use bisynchronous FIFOs to decouple the clock and power domains. In the experimental FPGA set-up, the MicroBlaze-VFCU communication uses the fast simplex link FIFOs (FSL), and the local memory and DMAs use Xilinx asynchronous memories. We now describe the VFCU and DMA controllers.

A. Voltage/Frequency Control Unit (VFCU)

The VFCU provides the clock to a processor tile. It has three functions, independently programmable by the processor (Fig. 2). 1) Change the operating point to f_{new}, starting *at time t in the future*. 2) Disable the clock to the tile from now until a time t in the future. 3) Generate an interrupt to the processor at time t in the future. Time is expressed in *local wall time*, i.e. a time counter that always runs at maximum frequency. Wall times of different tiles may be different (in terms of frequency, phase, drift), because it is difficult to provide an accurate high-speed time reference to tiles that are distributed over a chip. Each tile locally power manages tasks such that they receive their local budgets, and do not miss local deadlines. Monotonicity of our dataflow methodology [27] then guarantees that no end-to-end deadlines are violated. Our design is independent of the actual VF hardware. However, we require fast per-tile VF switching, and for ASIC implementation would use [10]. In our FPGA prototyping platform we implemented the VFCU as a frequency-divider with clock gating with the timers, as described above.

The essence of being able to delay the actions (change, disable, interrupt) until a point in the future, is that it allows us to remove variation in timing of what happened before, by programming them to occur at the worst-case completion time of preceding actions.

For example, as Fig. 2 shows with the intervals $[T1, T2]$ and $[T3, T4]$, depending on how the clock is generated, transitions from one operating point to another may take different or varying durations. For example, a high to low voltage transition is faster than vice versa, or depending on the clock distribution network, externally supplied clocks may come from different PLLs with different characteristics. Because the clock may not be stable for some time (the grey areas in the figure), the disable function removes the clock from the processor until the clock is guaranteed to be stable. This can also emulate a halt instruction, if absent from the processor.

Section V defines the infrastructure we use to avoid interference between applications when changing frequency (composability), and to budget a minimum number of cycles per time slice (predictability). But first we limit the time each instruction takes.

B. Interrupts and Direct Memory Access (DMA)

We use interrupts to ensure that tasks cannot exclusively claim the processor. To bound the interrupt rate, required to guarantee a minimum processing capacity, only the RTOS uses interrupts. The RTOS can accurately schedule an interrupt to initiate a task switch by programming the VFCU. However, on most processors individual instructions are not interruptable. For example, in our set-up, a load instruction to a local

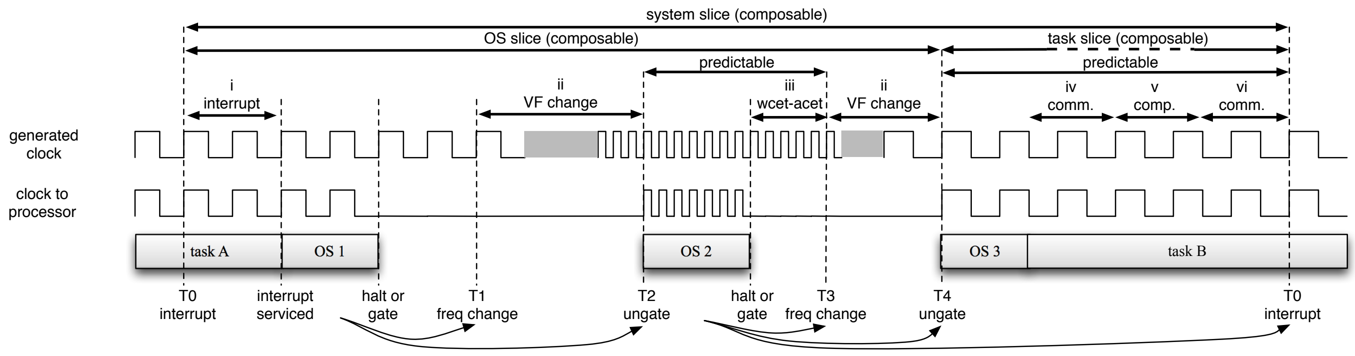

Fig. 2. OS and task time slices. (Not to scale: the OS time slice is less than 8% of the task time slice.)

memory over the bus can take up to five cycles. But a remote load can take an arbitrary number of cycles, depending on the speed of the bus, NOC, and slave. For a shared remote DRAM the time during which no task switch can take place because a processor cannot be interrupted, can be thousands of cycles.

We address the relatively small, and bounded, delay on the interrupt service due to completion of instructions such as multiplication and local loads, by assuming the worst-case completion, and eliminating variation using the VFCU as described above. But this technique cannot be used for the long completion time of remote load instructions. Instead we convert remote load instructions in *local load instructions*, by using a direct memory access (DMA) controller on the local bus. For a remote load (or store) instruction, the processor programs the DMA with source and destination addresses, burst size, and starts it. It then repeatedly polls the DMA until the data has been transferred from remote memory to local memory (or vice versa). Even though the DMA may be blocked on a remote read, the processor can be interrupted after each polling local read, which only take a few cycles to complete on the (uncontended) local bus.

A task must now distinguish local and remote data, using DMA for the latter. However, this is a natural fit with our dataflow application model, where input data must be present in the local memory when a task (actor) fires. As detailed below, the communication library ensures that data from tasks on remote tiles is copied by the DMA to the local memory before a task is started by the RTOS to operate on them. Each FIFO channel has a DMA, and copying of data takes place in parallel with other DMAs as well as computation. Copying has a known worst-case completion time due to performance guarantees in the NOC and shared remote memories [23]. The DMAs consist of a simple finite state machines and clock-domain crossing, and are small.

V. Composable Power Management

In this section we describe how time slices are made composable, and then how applications and tasks are scheduled.

A. Composable Time Slices

We define a system time slice to be an RTOS slice, followed by a task slice, as illustrated in Fig. 2. Composable power management relies on system time slices that have a constant duration, to decouple successive tasks (possibly belonging to different applications). Predictable power management and scheduling depends on guaranteeing a minimum number of clock cycles to a task in its time slice.

Composability is not trivial. First, changing the operating point can take a variable (even application-dependent) number of cycles. Second, the RTOS takes a variable number of cycles to execute in its RTOS time slice. Finally, the interrupt to end a time slice is served after a variable number of cycles. Using a DMA for remote load instructions, this takes at most five cycles in our MicroBlaze-based FPGA prototype (allowing the multiplication but not the division instruction). Using Fig. 2 we explain how we create composable time slices. Interrupts are used to switch from one task to the next, with RTOS running in between to decide which task should run next and at what frequency, based on budgets and slack.

1) At $T0$ the VFCU sends an interrupt to task A running at frequency f_A. It may take up to five cycles for it to be serviced and to start the RTOS, as shown by (i) in the figure.

2) The RTOS programs the VFCU to change to f_{max} at $T1$, and to re-enable the clock at $T2$. It then halts itself (or tells the VFCU to gate the clock). This removes the variation in interrupt service time (i) and VF change (ii). At $T1$ the VFCU starts changing the frequency, which may take a variable time (ii), and may include a period where the clock is unstable (the grey area in the figure).

3) At $T2$, when the clock is stable at f_{max}, the processor's clock is re-enabled. The RTOS saves task A's context, and schedules a new task B to run at frequency f_B (discussed below). It programs the VFCU to change to frequency f_B at time $T3$, to re-enable the clock at $T4$, and to generate an interrupt at $T0$. It then halts itself. $T4$ removes variation in both RTOS execution time (iii) and frequency changing (ii). At $T3$ the frequency changes, and is stable by $T4$, when the RTOS restarts.

4) At this point the RTOS restores task B. For each FIFO in the current firing rule of the task, the task programs a dedicated DMA to copy input data for the task from remote to local memory. It polls the DMA(s) until they have completed (iv). The task computes the output data given the input data, both

in local memory (v). Finally it copies the output data of each FIFO from local memory to remote memory, using the DMA(s) (vi).

The Ti are chosen such that all activities scheduled before them are finished in the worst case, such as interrupt service latency (i in the figure), crossing clock domains and frequency changes (ii), and RTOS scheduling (iii).

B. Composable and Predictable Scheduling

Given composable time slices, the RTOS decides which task is executed in each time slice, and at what operating point.

First of all, inter-application and intra-application scheduling have different requirements. The former must be composable, and the latter (optionally) predictable. This is implemented with a two-level scheduler that first uses time-division multiplexing (TDM) between applications, and then an application-specified scheduler (currently predictable TDM or best-effort round robin) that picks a task from the application. The pseudo code illustrates the scheduling (please ignore the IF statement for now).

```
s := -1;
forever do
  s := (s+1) mod NRSLOTS;
  app := tdm_table[s];   // TDM between apps
  task := task_scheduler[app]();
  if task = idle then
    // task cannot fire or has finished
    // pick another eligible task in this app
    task := task_slack_scheduler[app]();
  fi
  restore_context(task); // iv
  jump to task;          // v
  while true do ; od;    // wait for interrupt
od
```

At this point, by fixing the frequency to f_{max}, our RTOS is composable and predictable, but not yet power optimised. We can apply prior work [6], [9], [27] to compute the system's performance given scheduling budgets for the applications and their tasks (length of TDM wheel, and which slots per task). The remaining challenge is to extend this to variable operating points. The first step is to detect and compute the slack available to tasks (V-C), followed by computation of operating points (V-D).

C. Slack Scheduling

The scheduling algorithm shown in the previous section first determines the application to be run, and then calls the application's task scheduler. If no application or task is scheduled, because the TDM slot was not assigned, the task is waiting for input data or output space, or the task finished before using its entire budget, then the time slice is slack. It can be used in various ways. 1) It can be left unused, and the processor can be clock-gated. 2) It can be given to another task in the same application to increase throughput or reduce latency. 3) Or it can be given to another task in the same application, and reduce the task frequency to save power. Each application has a task slack scheduler that decides what to do.

Slack is handed out one slot at a time, because it is detected when the task scheduler finds no eligible task to run. Although this is suboptimal in terms of power, this slack accounting is easy, and would be complex otherwise [9].

D. Scheduling with DVFS and Slack

Our scheduling approach must know the worst-case work of each task. Since the RTOS starts and stops tasks, it also knows the actual work spent by the task on its last firing (invocation). As a result, we can observe slack generated by a task, and pass it to itself in the next iteration, or to other tasks. Slack that is received by a task is used to lower its frequency.

Each task is characterised by its worst-case work *wcw* measured in cycles, and its budget B that is the time (in seconds) allocated to it during each period (corresponding to the number of TDM slots in each TDM wheel revolution). We assume that we can scale the frequency from 0 to f_{max} in N steps. (We deal with f_{min} later.) T_{slice} is the duration of the task time slice (in seconds). $C_{switch} \ll T_{slice} \cdot f_{max}$ is the constant number of cycles (about 150) of the RTOS task-switching functionality in that time (OS3 in Fig. 2).

When a task starts it is allocated B seconds (computed below) to finish its worst-case work (*wcw* in cycles), corresponding to a number of time slices. We keep track of its remaining work $w[i]$ (starting at *wcw*) and remaining allocated budget $b[i]$ (starting at B). A task runs either in an allocated slice that is part of its budget (and its budget $b[i]$ will reduce), or in a slack slice that was not used by another task (and $b[i]$ will not reduce). Either way, we compute the task frequency in the current slice $f[i]$: essentially, the remaining work divided by the remaining budget. Without slack, a task would run at f_{max}, as its remaining work and budget reduce in tandem. But slack slices reduce work for free (without a budget reduction), and allow a lower frequency.

We formalise this as follows. For each task, the remaining work $w[i]$ (in cycles) at the start of slice i, is defined by $w[0] = wcw$, and

$$w[i+1] = \begin{cases} 0 & \text{task finished} \\ w[i] & \text{task does not use slice } i \\ w[i] - f[i] \cdot T_{slice} & \text{task uses slice } i \end{cases} \tag{1}$$

The budget left at the start of slice i is $b[0] = B$, and

$$b[i+1] = \begin{cases} b[i] - T_{slice} & \text{task uses allocated slice } i \\ b[i] & \text{otherwise (slack slice, or not scheduled)} \end{cases} \tag{2}$$

$$f[i] = \begin{cases} max(f_{min}, \left\lceil \frac{w[i]}{(b[i]+s[i]) \cdot f_{max}} \cdot N \right\rceil \cdot f_{max}/N) & \text{task uses slice } i \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $s[i] = T_{slice}$ for a slack slice, and 0 otherwise. Essentially, it computes the ratio of work left ($w[i]$ in cycles) over the maximum possible work that can be done in the remaining time budget plus slack ($(b[i] + s[i]) \cdot f_{max}$). This is then rounded and normalised to the N operating points ($1 \cdot f_{max}/N$ to $N \cdot f_{max}/N$), with a minimum f_{min}.

This is illustrated in Fig. 3 for a time wheel of 12 slices, and a task with 5 allocated slices (A) and 3 slack slices (S). It shows the remaining work, budget, and the frequency

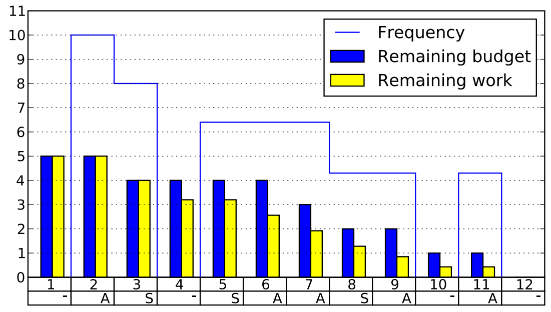

Fig. 3. Frequency, remaining work and budget (at start of slice).

when DVFS is used according to the above algorithm (with continuous operating points). If no DVFS is used, the remaining work would be identical to the remaining budget, and the frequency would be constant $f_{max} = 10$.

The initial budget for a task (rounded up to a slice duration) is $B = \lceil wcw'/(f_{max} \cdot T_{slice}) \rceil \cdot T_{slice}$. Lowering the frequency means using more slices, and hence incurring the C_{switch} (fixed number of cycles) overhead more often. wcw' adds this extra work to *wcw*. In our case, $C_{switch} \approx 150$ cycles, or about 0.3% of $T_{slice} \cdot f_{max}$ (1 ms at 50 MHz). Hence wcw' adds less than 2.5% to *wcw*. In any case, the difference is reclaimed as slack, if it is not used.

Essential for predictability, tasks always finish within their allocated budget, even when scaling. This holds because, after starting, a task operates entirely on the processor and the local tile, and does not depend on other resources (NOC or remote memories). As a result, the WCET of the task scales with frequency. Techniques like workload decomposition [28] are therefore not required. Note that the communication (copying of input data and output) is decoupled from the computation, since it is performed by the DMAs. These run at a fixed frequency (see Fig. 1), and finish in a bounded time due to performance guarantees in the NOC and shared remote memories [23].

E. Applications

As the RTOS copies input and output data to and from local memory, a task is essentially a function call operating on (a pointer) to input data and output space. The user writes the tasks as functions. The `set_firing_rule` function allows the task to change the firing rule, for the description of variable-rate dataflow applications.

VI. Experimental Results

We implemented the tile, comprising MicroBlaze processor, DMAs, and local busses in RTL VHDL. The VFCU is implemented by subsampling the fixed reference clock. We connected two tiles to an Æthereal NOC [29], together with a shared SRAM. The tiles run the RTOS, while a third MicroBlaze connected directly to the NOC boots the NOC and tiles. It acts as a monitor, by forwarding information sent from the tiles via MicroBlaze FSL links to a host PC, using a serial port. The entire system was prototyped on FPGA.

The RTOS uses TDM between applications, and TDM (with and without slack management) or round robin (always with slack management) within each application. f_{max} is 50 MHz, and f_{min} is 6.25 MHz, scalable in 8 steps. The system time slice is 1.08 ms, of which 7% is always used by the RTOS. Three slack policies are implemented: *None*, where slack is not used, *Self*, which passes slack only to the next invocation of the same task, and *Next*, which passes slack to the first eligible task in the same application (the ordering is based on the task control block list of the RTOS).

We implemented a JPEG decoder with 3 tasks on two processors, to demonstrate the variable-rate dataflow capability. We also created a parametrised synthetic task that can be configured to exhibit different test behaviours (actual work-load distributions, variable numbers of input and output tokens, etc.), and instantiated multiple times to create different applications. Tasks on different tiles communicate using the composably-shared remote memory, which is accessible via the NOC.

We performed three kinds of experiments, demonstrating predictability, composability, each without and with slack management and power saving. They are discussed in turn.

A. Predictability

We use a single synthetic application, consisting of a chain of four tasks, the first two mapped on a different processor from the last two. Each task has a random actual work between 0 and its *wcw* equal to either 16 or 32 (in units of 1/8$^{\text{ths}}$ of a slot, corresponding to the 8 operating points) and a corresponding budget of 2 or 4 slots out of a total of 12 slots. Fig. 4 shows the number of finished invocations of task 2 on processor 1 (the others are similar). With the Self and Next policies, without DVFS (Self/Next in the figure) the throughput is higher and the frequency the same. With DVFS (Self/Next+DVFS) the throughput is marginally higher but the frequency is lower. Next performs better than Self because it has more potential recipients for generated slack.

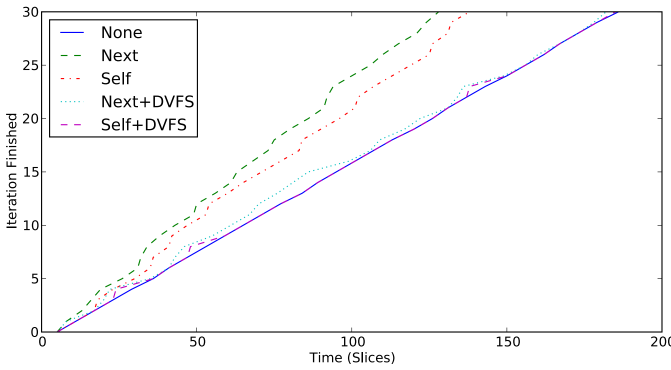

Fig. 4. Number of finished task invocations, without and with DVFS.

An alternative view on this is the total slack in the system, using Next, as shown in Fig. 5. Without frequency scaling slack accumulates, but with frequency scaling it remains low as it is used to run slower. Thus, assuming that the application

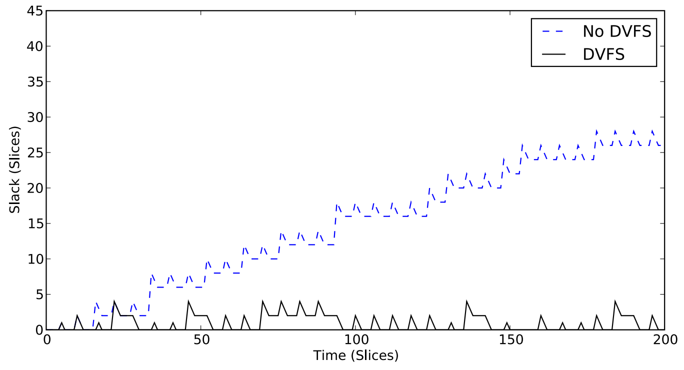

Fig. 5. Total slack, using Next without and with DVFS.

was configured with correct budgets and met its deadlines without DVFS, the Self and Next policies with DVFS also meet them, since at any point in time their number of finished iterations is always higher than for the None policy.

B. Composability

We map the first task of the JPEG decoder on the first processor, and the remainder on the other processor. Both sub-applications use round-robin scheduling, but no frequency scaling because JPEG is a variable-rate application with no (realistic) worst-case execution time. To demonstrate composability, a three-task pipeline application is also mapped on each processor. The pipeline applications are executed (1) with round-robin scheduling, and with TDM (2) with and (3) without slack management. Fig. 6 shows the number of finished task invocations of JPEG and the pipeline application on processor 1, for the three scheduling policies. Like in Fig. 4, the throughput of the pipeline varies with the slack policy (lower three lines). However, the JPEG measurements are identical, giving an indication of composability. Moreover, the JPEG schedules and finishing times per task iteration (not shown) are also identical, giving more proof of composability. Note that the *entire CompSOC platform is composable*: processor tiles, NOC, and shared memories.

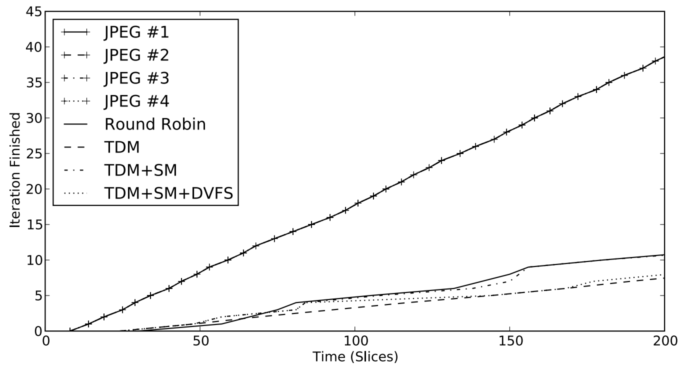

Fig. 6. Number of finished task invocations.

C. Energy Savings

We use the synthetic application shown in Fig. 1 with a random work load per task. The RTOS always uses its worst-case budget (7% of the system time slice). The energy cost of VF switching is ignored. We perform four experiments: (1) both tiles use TDM with no slack management; (2) additionally, idle slices are clock gated; (3) additionally both tiles use slack management, and one uses frequency scaling; (4) both tiles use frequency scaling. Table I shows the normalised energy of the synthetic application. It reduces by 42%, compared to only clock-gating idle slices, and 68% compared to no power management at all. The processor utilisation rises, when slack is used to run slower. Other experiments give similar results.

TABLE I
ENERGY CONSUMPTION FOR RANDOM WORKLOAD

Tile 0 Tile 1	Fixed Fixed	Gate Gate	DVFS SM	DVFS DVFS
Normalised Energy	1.95	1.00	0.74	0.51
Including RTOS	1.82	1.00	0.78	0.58
Slices Total	1808	1808	1806	1817
Slices Used	1233	1233	1528	1757
Average Processor Utilisation	68%	68%	85%	97%

Note that the finishing time (Slices Total in the table) is the same in both cases without DVFS, since the schedules are identical, only the power of idle slots changes. When applying DVFS, however, the task schedules on each processor change, which means that the data that is communicated between tiles is generated at different times. It is therefore transported in different TDM slots of the NOC, and stored in the shared memory in different TDM slots of the memory controller. This leads to a different (possibly earlier) finishing time *for the synthetic application only*; other applications are not affected. Moreover, note that the later finishing time of 1817 slices still meets the application deadline; the executions with 1806 and 1808 slices finished early (with unused slack).

VII. CONCLUSIONS

We introduced power management of embedded applications programmed using the variable-rate dataflow paradigm. Multiple independent applications, each of which may be real-time or not, are present in the system at the same time. They are mapped on a platform consisting of multiple tiles, a network on a chip, and distributed shared memories. The challenge addressed in this paper is composable and predictable power management, i.e. the functional and temporal behaviour of a power-managed application is not affected by the presence or absence of other (power managed) applications. Real-time applications are power managed such that no deadlines are missed.

We achieve composable and predictable power management through two techniques. First, at the hardware level, the voltage-frequency control unit (VFCU) enables the *programming of* DVFS *operations at precise points in the future* (relative to the tile's wall time). In particular, the initiation of a VF transition, the gating and ungating of the processor clock, and generation of interrupts to the processor can be scheduled. The operating system (RTOS) uses the VFCU to remove the variation in the interrupt service time, the VF transitions, and the RTOS behaviour, essentially by programming the DVFS operations at the respective worst-case completion times.

Second, direct memory access (DMA) units are used to ensure that processor instructions finish in a (small) bounded time. In particular, *load instructions to remote memories are replaced by* DMA *transfers*, otherwise they can take an arbitrary time to complete. During this time many processor cannot serve an interrupt. This could violate composability because an application waiting for a response of a remote memory can block another application from starting on the processor.

Using the VFCU and DMAs, the RTOS implements a *two-level arbitration scheme: composable time-division multiplexing (*TDM*) between applications, and round robin or predictable* TDM *between tasks of an application*.

The proposed platform has been prototyped on a FPGA. Taking into account that the RTOS uses 7% of each system time slice, for a JPEG application the energy reduces by 42%, compared to only clock-gating idle slices, and 68% compared to no power management at all. The JPEG application's behaviour was shown to be independent of other (power-managed) applications (composable).

REFERENCES

[1] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *Proceedings of the IEEE Transactions on VLSI*, 2000.

[2] V. VENKATACHALAM and M. FRANZ, "Power Reduction Techniques For Microprocessor Systems," *ACM Computing Surveys*, vol. 37, no. 3, pp. 195–237, 2005.

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

[4] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty, "Model-integrated development of embedded software," *Proc. of the IEEE*, vol. 91, no. 1, pp. 145–164, Jan. 2003.

[5] G. Gössler and J. Sifakis, "Composition for component-based modeling," *Lecture Notes in Computer Science (LNCS)*, vol. 2852/2003, pp. 443–466, 2004.

[6] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Transactions on Design Automation of Electronic Systems*, vol. 14, no. 1, pp. 1–24, 2009.

[7] *ARINC Specification 653*, Avionics Application Software Standard Interface, January 1997.

[8] A. Molnos, A. Milutinovic, D. She, and K. Goossens, "Composable processor virtualization for embedded systems," in *Proc. Workshop on Computer Architecture and Operating System Co-Design (CAOS)*, ser. Lecture Notes in Computer Science (LNCS). Springer, Jan. 2010.

[9] A. Molnos and K. Goossens, "Conservative dynamic energy management for real-time dataflow applications mapped on multiple processors," in *Proc. Euromicro Symposium on Digital System Design (DSD)*, Aug. 2009.

[10] M. Meijer, J. Pineda de Gyvez, and R. Otten, "On-chip digital power supply control for system-on-chip applications," *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pp. 311–314, 2005.

[11] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in *Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. Washington, DC, USA: IEEE Computer Society, Aug. 2008, pp. 3–14.

[12] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset 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," *ACM Transactions on Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, 2002.

[13] W. Kim, M. Gupta, G. Wei, and D. Brooks, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *IEEE 14th International Symposium on High Performance Computer Architecture, 2008. HPCA 2008*, 2008, pp. 123–134.

[14] O. S. Unsal and I. Koren, "System-level power-aware design techniques in real-time systems," in *Proceedings of the IEEE*, 2003, pp. 1055–1069.

[15] L. Yan, J. Luo, and N. Jha, "Joint dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 7, pp. 1030–1041, 2005.

[16] A. Manzak and C. Chakrabarti, "A low power scheduling scheme with resources operating at multiplevoltages," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, no. 1, pp. 6–14, 2002.

[17] D. Shin and J. Kim, "Power-aware scheduling of conditional task graphs in real-time multiprocessor systems," in *Proceedings of the 2003 international symposium on Low power electronics and design*. ACM New York, NY, USA, 2003, pp. 408–413.

[18] M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini, "Application-specific power-aware workload allocation for voltage scalable MPSoC platforms," in *2005 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005. Proceedings*, 2005, pp. 87–93.

[19] R. Mishra, N. Rastogi, D. Zhu, D. Mosse, and R. Melhem, "Energy aware scheduling for distributed real-time systems," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 2003, p. 9.

[20] C. Shen, K. Ramamritham, and J. Stankovic, "Resource reclaiming in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 4, pp. 382–397, 1993.

[21] J. Zhu, I. Sander, and A. Jantsch, "Energy efficient streaming applications with guaranteed throughput on MPSoCs," in *Proc. ACM international conference on Embedded software (EMSOFT)*, 2008, pp. 119–128.

[22] A. Maxiaguine, S. Chakraborty, and L. Thiele, "DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs," in *International Conference on Hardware Software Codesign: Proceedings of the 3 rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis; 19-21 Sept. 2005. 2005*. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA,, 2005.

[23] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET Computers & Digital Techniques*, 2009.

[24] C. Paukovits and H. Kopetz, "Concepts of switching in the time-triggered network-on-chip," in *Proc. Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008, pp. 120–129.

[25] B. Akesson, A. Hansson, and K. Goossens, "Composable resource sharing based on latency-rate servers," in *Proc. Euromicro Symposium on Digital System Design (DSD)*, Aug. 2009.

[26] 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, Jul. 2008.

[27] O. M. Moreira and M. J. G. Bekooij, "Self-timed scheduling analysis for real-time applications," *EURASIP Journal on Advances in Signal Processing*, 2007, article ID 83710.

[28] K. Choi, R. Soma, and M. Pedram, "Dynamic voltage and frequency scaling based on workload decomposition," in *Proc. Int'l Symposium on Low Power Electronics and Design (ISLPED)*, Newport Beach, California, USA, 2004, pp. 174–179.

[29] K. Goossens and A. Hansson, "The Aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *Proc. Design Automation Conference (DAC)*, Jun. 2010.