Distributed Power Management of Real-time Applications on a GALS Multiprocessor SOC

Andrew Nelson and Kees Goossens
Eindhoven University of Technology
The Netherlands

ABSTRACT

It is generally desirable to reduce the power consumption of embedded systems. Dynamic Voltage and Frequency Scaling (DVFS) is a commonly applied technique to achieve power reduction at the cost of computational performance. Multiprocessor System on Chips (MPSoCs) can have multiple voltage and frequency domains, e.g. per-core. When DVFS is applied to real-time applications, the effects must be accounted for in the associated formal timing model.

In this work, we contribute our distributed multi-core run-time power-management technique for real-time dataflow applications that uses per-core lookup-tables to select low-power DVFS operating points that meet the application's timing requirement. We describe in detail how timing slack is observed locally at run-time on each core and is used to select a local DVFS operating point that meets the application's timing requirement. We further describe our static off-line formal analysis technique to generate these per-core lookup-tables that link timing slack to low-power DVFS operating points. We provide an experimental analysis of our proposed technique using an H.263 decoder application that is mapped onto an FPGA prototyped hardware platform.

Categories and Subject Descriptors

EDA3.1 [**System-level low-power design and thermal analysis, simulation and management**]

Keywords

Low-power design, Modelling and prediction, Multiprocessor systems, Real-time systems, Embedded Systems

1. INTRODUCTION

Power consumption of embedded systems is an ever growing design concern. It is of particular concern in portable devices, as there is a trade-off between the capacity (and hence physical size and weight) of the battery required, and the time between charges. Dynamic Voltage and Frequency Scaling (DVFS) is a common method used to reduce the power consumption of synchronous electronic systems. This method trades computational performance for a reduction in power consumption.

Real-time applications have timing requirements that must be met. Any performance reduction as a consequence of DVFS must not violate these requirements. Real-time applications are typically verified using formal modelling methods to derive timing guarantees. It is demonstrated in [23] how an application mapped onto multiple cores of a Globally Asynchronous Locally Synchronous (GALS) Multiprocessor System on Chip (MPSoC) is modelled as a Homogeneous Synchronous Dataflow Graph (HSDFG). By annotating the actors with Worst-Case Execution Times (WCETs), static worst-case bounds can be derived for the application's latency and throughput.

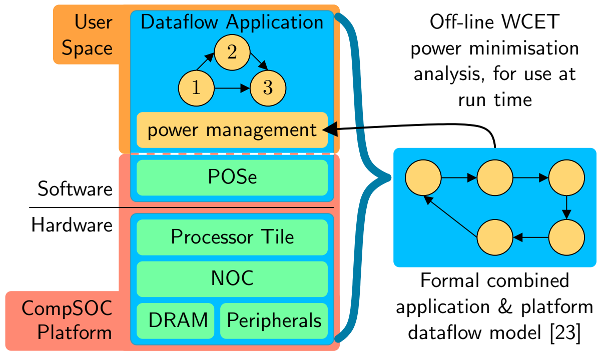

User Space
Dataflow Application
1
2
3
power management
Off-line WCET power minimisation analysis, for use at run time
Software
POSe
Hardware
Processor Tile
NOC
DRAM
Peripherals
CompSOC Platform
Formal combined application & platform dataflow model [23]

Figure 1: Application of our proposed power management technique to the CompSOC platform.

For Static-Periodic Schedules (SPSs), it was shown in [21] how each edge in an application HSDFG can be represented as a timing constraint that can be used as part of an off-line linear program to minimise the period of the SPS without violating causality. This is expanded upon in [22] to create an off-line convex optimisation enabling the power consumption of the application graph to be minimised while ensuring that the period of the SPS meets the application's throughput requirement. The described technique can only take advantage of static timing slack in the application's schedule, producing static Voltage and Frequency Scaling (VFS) levels for use at run-time.

In this work, we propose a distributed run-time power-management technique that uses locally (per-core) observed timing-slack as a criteria to select a low-power DVFS operating point, from a table that is derived in an off-line static analysis, that conservatively meets the application's timing requirement. Figure 1 illustrates how our proposed power management scheme is applied to the predictable multiprocessor CompSOC platform [10], by formally modelling the application and platform in a combined dataflow model for off-line analysis. The result of this formal analysis is a per-core DVFS look-up table that the local run-time application power manager uses to select a local DVFS level for the observed amount of local timing slack that is guaranteed not to violate the application's global timing requirement. To achieve this we contribute:

1. A method that uses observed local timing slack to infer a conservative amount of global slack that is used to select the DVFS operating point from an off-line derived table.
2. A technique to derive these DVFS look-up tables for use at run-time.
3. A case study analysis of our power management technique applied to an H.263 decoder application.

The rest of this paper is structured as follows. In the next section, we present the related work to our power management technique. We follow this by describing relevant background information in Section 3 before proceeding to describe the effects of DVFS on dataflow task scheduling in Section 4. We present our distributed run-time power-management technique in Section 5 that uses per-core DVFS tables that are derived in an off-line analysis of the application and platform. We describe our off-line analysis technique in detail in Section 6. In Section 7 we provide a case study analysis of our technique applied to an H.263 decoder application, before making concluding statements in Section 8.

2. RELATED WORK

Research on using DVFS as a processor power-management mechanism goes back at least two decades [5, 11]. In this section, we will attempt to relate our power-management techniques that we present in this paper to other relevant work or similar techniques that we are aware of. For a broader overview of power-management and DVFS techniques, we direct the reader to [1, 15].

Many DVFS techniques employ a notion of load when making DVFS decisions [4]. For whatever definition of load they each have, the power-management mechanism generally lowers the DVFS-level when processor load is low and increases it when load is high. Real-time applications need timing guarantees requiring formal mathematical analysis. Algorithms for static off-line and dynamic run-time energy optimal scheduling are proposed in [26] for independent real-time tasks with deadlines. The work in [17] extends the algorithms from [26] for tasks with arbitrary arrival times, deadlines and execution times. These techniques are limited to single processor systems, whereas the techniques in [2, 18, 19, 25] apply to multiprocessor systems.

The multiprocessor DVFS technique in [25] applies to tasks with precedence constraints that form a graph topology. An off-line power-management technique is proposed in [7, 8] to reduce energy consumption of periodically scheduled Kahn Process Network (KPN) applications on multiprocessor platforms. Off-line and run-time DVFS techniques are proposed in [6] for Finite State Machine Scenario Aware Dataflow (FSM-SADF) applications. The techniques described in [6] are not implemented for any platform. Details are not provided on how this would be achieved or on the impact of any implementation practicalities on their described technique. This work describes a run-time DVFS technique that uses and off-line analysis of an Homogeneous Synchronous Dataflow (HSDF) model of the combined application and platform, that is implemented on the CompSOC multiprocessor platform [10].

Distributed multiprocessor power-management techniques address the scalability problem of synchronising information

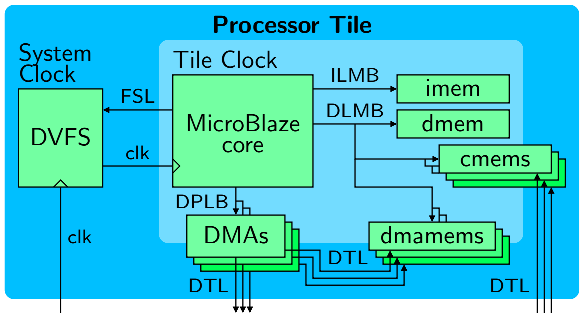


Figure 2: CompSOC processor tile.

with a centralised decision algorithm. Care must be taken when using local information to lower DVFS-levels as described in [16]. Local DVFS decisions affect global timing and therefore impact the ability of other cores to make local DVFS decisions, e.g. one core lowers its DVFS level causing other cores to have to increase their DVFS levels. Progress information is explicitly communicated for the technique in [16], whereas queue occupancy is used in [3] to implicitly communicate application progress. This is achieved in [3] for dataflow applications with a single identifiable producer and consumer task.

Buffer occupancy is used as a feedback in the closed DVFS control loop. It is not described in [3] how well their described technique works with small buffer capacities, i.e. of size one. Their technique works with "tuned" occupancy thresholds, but no description is given how these thresholds are derived. Our proposed DVFS technique is applicable to all dataflow applications that can be abstracted as an HSDFG. We use local application progress as an indicator of global performance, rather than buffer occupancy, enabling the use of small buffer capacities as may be required on resource constrained embedded systems. We describe our off-line analysis technique to derive the DVFS tables used by our power-management control loop in detail in Section 6.

A DVFS technique for single core mixed-criticality systems is proposed in [14]. The CompSOC multiprocessor platform supports virtual execution for mixed time-criticality systems [10]. The HSDF dataflow modelling technique [23] that produces the combined application and platform models used by our power-management technique also models the virtualisation. We leave an extension of the work in this paper to include power-management for multiprocessor mixed time-criticality applications, as future work.

3. BACKGROUND

We begin the explanation of our run-time power-management technique by describing the CompSOC hardware and software platform [10] to which it can be applied. A CompSOC platform consists of one or more processor tiles connected by the timing predictable Æthereal Network on Chip (NoC). Each tile exists in its own voltage and clock domain. The local processor is able to change its own voltage and frequency by programming the operating point on the local DVFS control module.

As illustrated in Figure 2, each processor tile consists of a MicroBlaze processor with local scratch pad memories, for instructions (imem) and data (dmem) that are accessible over single-cycle access I/DLMB buses. Direct Memory Access

(DMA) modules are used to enable off-tile communication by providing access to distributed shared memories over the NoC. Each DMA module has an associated scratch pad memory (dmamem) that is accessed by the local processor via the DLMB bus. The DMA modules are programmed over the single master DPLB bus, but have direct single-cycle access to their associated dmamem via a point-to-point link that uses the DTL protocol. The DMA modules read/write a specified amount of data from/to a specified location in its associated dmamem and writes/reads this data to/from a specified remote location accessible via the NoC. To enable data to be written directly to local tile memory, some local dual-port scratch pad memories (cmems) that are accessible from the local processor on the DLMB bus are also connected directly to the NoC using the DTL protocol.

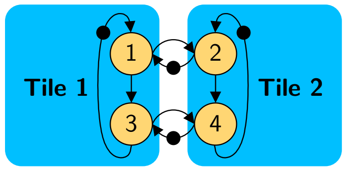
Real-time dataflow applications are executed using the Predictable Operating System (POSe) [23]. Both the computation and communication of the application executing on the CompSOC platform are predictable. The timing of the application and the platform hardware is modelled as a combined application and platform HSDFG [23]. It was shown in [22] how the power consumption of an HSDFG can be minimised for an SPS using a convex optimisation technique.

4. DATAFLOW TASK SCHEDULING

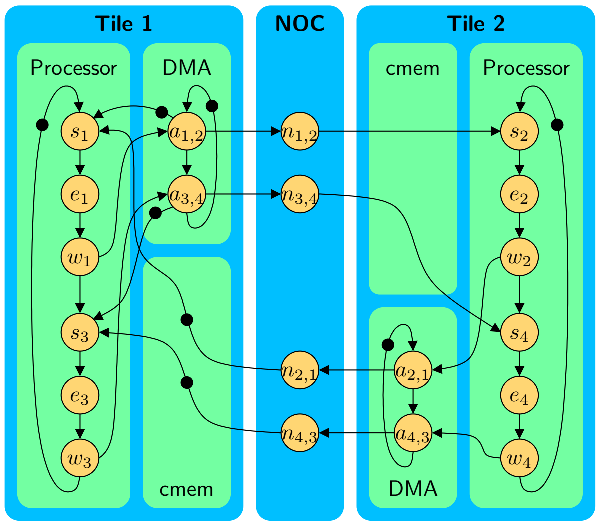
As a starting point, we assume that the dataflow application is mapped to the CompSOC platform, and that an HSDFG of the combined application and CompSOC platform has been constructed, as per [23]. A simplified HSDFG application and core mapping is presented in Figure 3a for illustration purposes. Nodes in the graph represent dataflow actors for modelling, and application tasks for execution. The edges represent dataflow channels and inter-task First In First Out (FIFO) communication. Figure 3b is the resultant combined application and platform HSDFG that is produced from the technique described in [23].

An HSDFG G is represented as a tuple (V, E, t, d). Vertices $v \in V$ in a dataflow graph are called actors and represent delay $t(v)$ due to computation, etc. Edges between vertices $(i, j) \in E$ are directed, with $i \in V$ being the producing actor and $j \in V$ the consuming actor, and represent inter-actor FIFO communication channels. Tokens represent atomic units of communication between actors. Initial token placement is represented by a black circle on the communication channel it occupies. The number of initial tokens on a channel is represented as $d(i, j) \in \mathbb{N}$.

The combined application and platform HSDFG from Figure 3b models task execution timing as actors e_v. The POSe Operating System (OS) executes dataflow applications following the dataflow Model of Execution (MOE). Tasks are executed following a per-core Static-Order Schedule (SOS). In keeping with dataflow actor firing rules, a task can only start execution whenever data and space is available on all of its incoming and outgoing FIFOs, respectively. The timing overhead associated with performing these checks for the application's tasks are represented by actors s_v. After the task has completed execution, the data that is produced is written to any remote locations using the DMA. The time taken to execute the DMA driver function to program the DMA is modelled using the w_v actor. The time taken by the DMA to write the data onto the NoC is modelled by actor $a_{i,j}$ where i and j are the producing and consuming task



(a) Mapped HSDF application with actor SOS per-tile.



(b) Combined application and platform HSDFG.

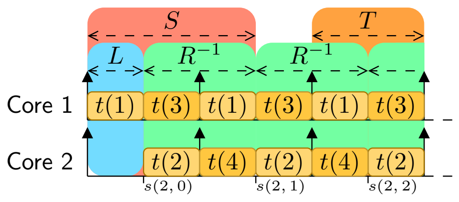
Figure 3: Application and platform HSDF.

ID, respectively. Actor $n_{i,j}$ represents the time taken for the last word of data written onto the NoC by the DMA to be written into the local cmem of the receiving task.

POSe executes applications following a Self-Timed Schedule (STS) while the application timing is conservatively modelled as an SPS. Modelling the application's timing as an SPS enables the power minimisation techniques that we describe in Section 8 to be applied. Application tasks are time triggered at set times when scheduled following an SPS. Executing the application following a data driven STS enables tasks to start earlier due to earlier finishing times of previous tasks. This enables applications to accumulate timing slack at the application level.

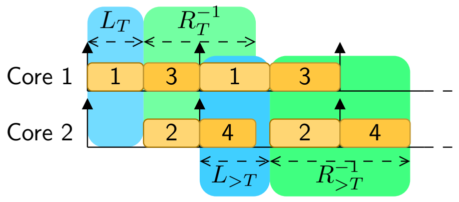
Figure 4a illustrates an example Worst-Case Self-Timed Schedule (WCSTS) for the application illustrated in Figure 3a. In this example, communication and other delays of the combined application and platform HSDFG are not illustrated for simplicity. Each actor v has its firing duration annotated with its WCET $t(v)$. We refer to the time taken to execute a single iteration of the application graph as the schedule length. The minimum rate at which the application completes graph iterations is determined by the duration T of the critical cycle or cycles of the application's WCSTS, which in this instance is the duration of the application's SOS on each core, i.e. $t(1)+t(3)$ and $t(2)+t(4)$. The application's critical cycle can be shorter than its schedule length. If this is the case, multiple application iterations execute concurrently to achieve the application's minimum execution rate. This can be seen in Figure 4a where the second iteration of actor 1 starts before the first iteration of actor 4 has completed.

Figure 4a also illustrates the schedule represented as an

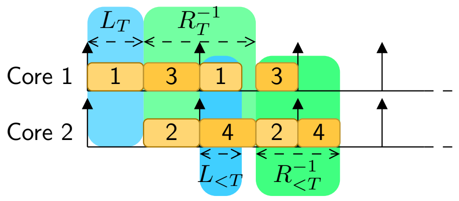


S
T
L
R^{-1}
R^{-1}
Core 1
Core 2
$t(1)$ $t(3)$ $t(1)$ $t(3)$ $t(1)$ $t(3)$
$t(2)$ $t(4)$ $t(2)$ $t(4)$ $t(2)$
$s(2,0)$
$s(2,1)$
$s(2,2)$

(a) WCSTS and SPS.



(b) SPS where the period of the second graph iteration $>T$ is longer than that of the first T.



(c) SPS where the period of the second graph iteration $<T$ is shorter than that of the first T.

Figure 4: Example schedules of the dataflow application in Figure 3a.

SPS. The start of each SPS iteration is indicated by upwards arrows. The SPS starts with a period of T and has a schedule length S greater than T. The first application graph iteration completes after the schedule length and with a rate of T^{-1} thereafter. Application graph iteration completions are therefore conservatively modelled using a latency-rate abstraction, i.e. after a latency of duration L the finishing times of the application can be conservatively modelled with a rate R. For an SPS schedule of an HSDF application, latency L and rate R components of the latency rate server abstraction are calculated from the schedule length S and schedule period T as follows:

$$R = T^{-1} \tag{1}$$
$$L = S - T \tag{2}$$

where $L \geq 0$ as $S \geq T$.

Timing slack is used to reduce the application's power consumption by lowering the DVFS level of the processors. The WCET of the tasks and hence the worst-case execution time of an iteration of the application graph increases, for decreasing DVFS levels. The off-line analysis technique described in Section 6, derives low-power frequencies for each core such that the application has a throughput of T^{-1}. Application-level timing slack is used to lower the application throughput at run-time, which causes the application schedule length to also increase. This can be seen in Figure 4b where the period of the second application graph iteration $> T$ is scaled using DVFS to be longer than the first T. Not only is the

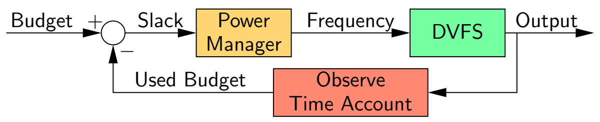


Figure 5: Per-core power-management control loop.

rate of execution of the graph $R_{>T}$ lower than R_T, but the latency $L_{>T}$ of the second iteration is greater than that of the first L_T. Care must therefore be taken when performing power-management to lower the frequency that the available slack is sufficient to bound the increase in application latency and the decrease in application execution rate.

Decreasing the application graph's period through DVFS decreases the latency $L_{<T}$ while increasing the application's rate of execution $R_{<T}$. This can be seen in Figure 4c where the period of the first application graph iteration is T and the second iteration is $<T$. In this instance the decrease in the latency $L_{>T}$ is bounded by the previous rate R_T. The second iteration of the graph immediately enters the rate component of its latency-rate server abstraction after the first iteration has completed. From Figures 4a & 4b, it can be seen that only the rate component of the latency-rate server abstraction needs to be taken into account when maintaining or decreasing the graph's period T, while Figure 4b shows that both components of the latency-rate server abstraction of the graph need to be taken into account when increasing the graph's period T.

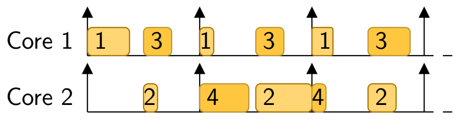
5. DISTRIBUTED POWER MANAGEMENT

Our distributed power management technique sets conservative DVFS levels on each core of the application without using explicit power management related communication between cores. Each core executes the power management code and performs local observations of the application's progress. The power manager on each core uses its independent observation to conservatively infer global application progress. Slack in the application's timing (i.e. how far it is ahead of schedule) is used to decrease the application's DVFS level for subsequent iterations of the graph. In this section, we describe our run-time power management technique in detail.

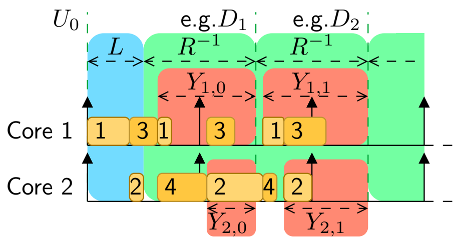
5.1 Per-core Power Management

Our power management technique applies to HSDF applications that are mapped onto one or more processing cores, e.g. the HSDF application that is mapped onto two cores that is illustrated in Figure 3a. The application's power management code is executed locally on each of the cores to which it is mapped. Our proposed power-management scheme is generalised as the control loop illustrated in Figure 5.

The application's real-time requirement is represented in POSe as a time budget on each core with the use of this budget recorded in a *time account*. The application is allocated a *budget* of time to perform a number of iterations of the application graph. The time taken to complete the number of iterations is the *used budget*. The difference between the budgeted amount of time and the actual time it took to complete the number of iterations is *timing slack*. For a real-time application that always meets its timing requirement, its timing slack can never be negative, i.e. it can not use

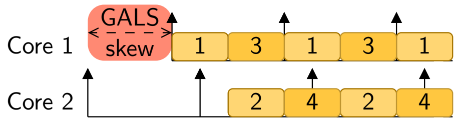


(a) SPS with same task start times as WCSTS but with varying task execution durations.

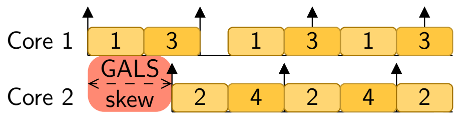


(b) STS with local slack observations $Y_{c,i}$.

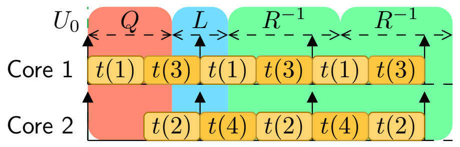
Figure 6: Example schedules using tasks with varying execution time.



(a) Core 1 timer running late.



(b) Core 2 timer running late.



(c) Bounds on application latency and rate for bounded GALS skew.

Figure 7: The effect of GALS timer skew on application timing.

more time than was budgeted. Positive slack means that the application is running ahead of schedule. The *power manager* on each core decides what *frequency level* the next iterations of the application graph will be executed at, given the amount of available timing slack.

Care must be taken that the frequency level that is chosen does not violate the application's timing requirement. This is especially difficult when making distributed power management decisions, as local application DVFS affects global application timing. Our proposed technique uses locally observed application progress and timing information to conservatively infer global progress and timing, i.e how many full application graph iterations have completed and how far ahead of schedule (global slack).

In our proposed technique, the power manager is a simple table lookup algorithm that selects the appropriate DVFS level (i.e. low power but also temporally conservative) for the available global timing slack. The derivation and use of these tables is explained in detail in Section 6.

5.2 Per-core Slack Observation

The application's progress is observed per-core, by comparing the actual time taken to perform N iterations of the application graph with the budgeted time to execute N iterations of the application graph. Using an H.263 video decoder as an example, if one video frame consists of 99 macro blocks ($x \times y$ size pixel blocks) and one graph iteration decodes a single macro block, then the application performance should be measured every $N = 99$ application graph iterations if it is to be measured once per decoded frame.

In this work, we consider the case where application tasks are mapped onto multiple cores, but each task is only mapped onto a single core. Tasks are scheduled on each core following a SOS to complete a single graph iteration. In this scenario, application progress that is measured locally is the time taken to complete N local SOS iterations. We consider an SOS complete if the first task of the next iteration of the SOS is able to execute. Tasks may have varying execution durations, and are modelled for timing analysis as following an SPS, as illustrated in Figure 6a. Tasks are actually executed following an STS, enabling variations in task execution duration to translate into application-level timing slack, as illustrated in Figure 6b.

Timing slack is measured locally by taking a timestamp U_0 at the start of the application's execution. Throughput constraints are translated into a set of absolute deadlines D_n with $n \in \mathbb{N}^+$ that are relative to U_0 at intervals equal to the budgeted b_N amount of time for N application graph iterations, i.e.

$$D_n = U_0 + n \times b_N \tag{3}$$

A timestamp U_n is taken after every N local SOS iterations, and compared with the deadline D_n to find the locally observed slack $Y_{c,n}$ as follows:

$$Y_{c,n} = D_n - U_n - Q \tag{4}$$

where $c \in \mathbb{N}$ is the core ID. Each core c can observe a different amount of local slack $Y_{c,n}$, with the global application-slack Z_n being the minimum of all locally observed slack, i.e. $Z_n = \min(Y_{c,n}), \forall c \in$ Core IDs. Temporally bounded GALS skew Q is explained in Section 5.3. Communicating the application's local progress information between the cores, and hence finding the exact global slack value, is not scalable. We propose conservatively deriving the global slack value on each core, i.e each core uses its own local slack value to derive a global slack value that cannot be greater than the actual global slack value. Each core may derive a different value for global slack, but as long as all of the derived values are not greater than the actual global slack value then all of the local DVFS values selected from the core's lookup table are guaranteed not to violate the application's timing requirements. This is explained in more detail in the following sections.

5.3 Slack Observation in GALS Systems

In GALS systems some clock skew can exist between clock domains, i.e. even if the clocks in different domains have the same frequency, they are not guaranteed to be synchronous.

This skew must be taken into account in the SPS timing analysis to ensure that the amount of available application-level slack is not over-estimated.

Figure 7 illustrates two GALS skew scenarios, for the WCSTS/SPS schedule from Figure 4a. Figure 7a illustrates the scenario that the timer used for reference on Core 1 is running behind the timer on Core 2, i.e. the timer on Core 1 reports an earlier time than the timer on Core 2 at the same instant. The synchronised time at which both cores start executing is therefore also skewed. In Figure 7a, this has the effect that the tasks on Core 1 are able to start executing later on Core 1 than on Core 2. The tasks on Core 2 are dependent on data from the tasks on Core 1, and therefore cannot start executing until tasks on Core 1 produce the required data. This has the effect that the tasks on Core 2 finish later than their worst case finishing time when using the timer on Core 2 as a reference.

Similarly, Figure 7b illustrates the scenario that the timer on Core 2 runs later than the timer on Core 1. The tasks on Core 1 initially have space available to produce data, as illustrated by the dataflow application's initial token placement in Figure 3a. After one iteration, the tasks on Core 1 must wait until task 2 completes so that task 1 can start executing its second iteration. The tasks on Core 1 finish later than their worst-case finishing time when using the timer on Core 1 as reference. The GALS skew must therefore be taken into account during timing analysis so that timing guarantees can be derived.

To be able to take the GALS skew into account the amount of skew must be temporally bounded. Algorithmic software [9] and hardware [20, 24] techniques have been proposed that achieve a temporal bound on timer skew. Figure 7c illustrates how the GALS skew bound Q is used in combination with the application's latency and rate bounds to conservatively bound the application's worst-case timing. The GALS skew bound Q is taken into account when measuring the local slack in Equation 4.

5.4 Conservative Global Slack Derivation

We use the properties of the application's dataflow formalism and its run-time scheduling to conservatively derive a global slack value on each core from locally observed timing slack. Application's execute at run-time following a static order STS that is conservatively modelled as an SPS. The derived global slack value is used to conservatively scale the application's SPS. The progress of the local application's execution is constrained by the availability of data/space on incoming/outgoing FIFO channels of actors that are mapped onto the local processor. The number of application graph iterations that the local application execution can be ahead of the application's execution on other processors is therefore dependent on the application's mapping, buffer capacity and initial buffer occupancy.

Tasks on each core follow a self-timed SOS that executes local tasks in order to complete a single iteration of the application graph. At the point when local application progress is measured, the local application SOS may have completed more iterations than the application's SOSs on other cores. To conservatively estimate global progress, we derive the worst-case time for the application to complete at least the same number of SOS iterations as the local core on all cores.

On each core, each SOS iteration cannot take longer than the period of the application's SPS when tasks are executed at the processor's lowest DVFS level $T_{\max}$. From the application's HSDFG, a local SOS on one core can be maximally ahead of a remote SOS on another core by the shortest path between an actor in the local SOS and an actor in the remote SOSs. In this case, a path between two actors is any set of directed edges in the HSDF that when traversed in the opposite direction starts at one actor and finishes at the other. The path length is calculated as the sum of initial tokens on all of the path's edges. A conservative assumption can be made about the number of iterations that any application SOS can be behind another SOS by taking the longest of all of the shortest paths between application SOSs.

For example, using the HSDFG example from Figure 3a, the shortest path B_1 from the actors on core 1 to the actors on core 2 is between actors 1 and 2, and actors 3 and 4 with a path length of one, i.e. $B_1 = 1$. The actors 1 and 3 can be maximally one iteration ahead of actors 2 and 4, respectively. For core 2, the shortest path B_2 is from actors 2 and 4 to actors 1 and 3, respectively. These paths have a length of $B_2 = 0$. This means that an iteration of actor 2 can only execute after the same iteration of actor 1. If the SOS on core 1 has completed N iterations, then actor 1 has completed N iterations and therefore actor 2 has completed at least $N - B_1$ iterations. This means that the SOS on core 2 has at least partially finished iteration $N - B_1$. We can therefore conservatively estimate that the SOS on core 2 has to complete $B_1 + 1$ iterations before it has completed N iterations. We therefore conservatively estimate the global slack Z_c for core c from the local slack Y_c as follows:

$$Z_c = Y_c - (B_c + 1) \cdot T_{\max} \tag{5}$$

This slack is used to relax the throughput requirement of the application to enable a DVFS level reduction.

5.5 Distributed Conservative DVFS

The locally derived conservative global slack value Z_c, where c is the local core ID, is used to derive the local processor's DVFS for the following N application graph iterations. For an application with an SPS period requirement of T_{req} the required period T_c for the following N iterations is calculated as follows:

$$T_c = T_{\text{req}} + \frac{Z_c}{N} \tag{6}$$

Using the off-line derived DVFS frequency and SPS period table, described in Section 6, our distributed power management technique selects the lowest DVFS level from the local table that is guaranteed to have an SPS period less than T_c. If the DVFS-level will be lower than the current level, then the increase in the application's latency must be taken into account, as described in Section 4. For DVFS-levels lower than the current level, the required period is calculated as follows:

$$T_c = T_{\text{req}} + \frac{Z_c - (L_{\max} - L_{\min})}{N} \tag{7}$$

where $L_{\max}$ is the maximum latency of the application graph that is derived when the application's tasks have worst-case timings and are executed with the minimum frequency level, and $L_{\min}$ is the application graph's minimum latency and is derived when the application's tasks have worst-case timings and are executed with the maximum frequency. $L_{\max} - L_{\min}$ is the maximum change in graph latency that can occur and therefore conservatively bounds all possible changes to graph latency. A greater value of global slack Z_c is therefore required to achieve a DVFS-level lower than the current DVFS level.

The selected DVFS level is maintained for N application graph iterations, until the local progress is re-evaluated and a new DVFS level is set.

6. OFF-LINE DATAFLOW ANALYSIS

In this work, we apply the static analysis technique described [22] to generate tables of DVFS operating points with the associated minimum amount of observed local timing slack (distributed observations that are local to each core the application runs on) required to conservatively operate at each point. Our run-time power-management technique observes the local timing slack then selects and sets the appropriate DVFS point from the table, as described in Section 5.

6.1 Minimising Power Consumption

It was shown in [22] how HSDFGs timing graphs of applications that are mapped onto a CompSOC platform instance, can be modelled as a convex optimisation to minimise application power consumption. Code 1 is an example of the HSDFG in Figure 3a represented as a Disciplined Convex Program (DCP) [13] for the CVX convex programming tool [12] for Matlab.

```
cvx_begin
  variable s(NUM_ACTORS) % Actor start times
  variable f(NUM_CORES)  % Core frequencies
  minimise(
    % Any power model that complies with DCP
    sum((3.353E-5*pow_pos(f,3)+2.065))
  )
  f >= MIN_FREQUENCY % Lower frequency bound
  f <= MAX_FREQUENCY % Upper frequency bound
  % per-edge constraints
  s(1) + T*d(3,1) >= s(3) + t(3)*inv_pos(f(1))
  s(2) + T*d(1,2) >= s(1) + t(1)*inv_pos(f(1))
  s(3) + T*d(1,3) >= s(1) + t(1)*inv_pos(f(1))
  s(4) + T*d(3,4) >= s(3) + t(3)*inv_pos(f(1))
  s(1) + T*d(2,1) >= s(2) + t(2)*inv_pos(f(2))
  s(2) + T*d(4,2) >= s(4) + t(4)*inv_pos(f(2))
  s(3) + T*d(4,3) >= s(4) + t(4)*inv_pos(f(2))
  s(4) + T*d(2,4) >= s(2) + t(2)*inv_pos(f(2))
cvx_end
```

Code 1: Formulation for the CVX Matlab tool.

DCPs consist of an objective function and constraints. Code 1 uses a power model as a minimising objective function. The particular power model used is not important for our technique beyond that it (or an approximation of it) must be able to be expressed following the rules of DCP. In this work, we use the following processor power model:

$$P(f) = 3.353 \times 10^{-5} f^3 + 2.065 \tag{8}$$

Where $P(f)$ in Equation 8 is the processor power consumption in nano-Joules and f is processor frequency in MHz (*We do not claim that this model is accurate for any particular processor*). As our technique is only concerned with the DVFS level of the processors, the minimising objective function in Code 1 minimises the sum of the power levels of all of the processors used.

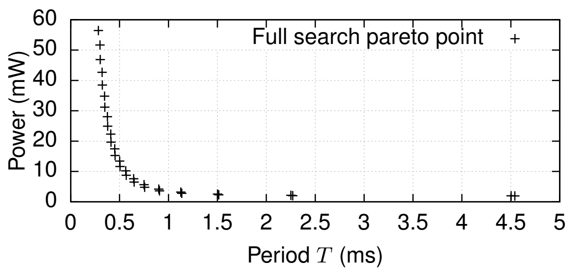


Figure 8: Pareto front from a full search using the model.

The objective function and constraints consist of variables and constants. The variables in Code 1 are, the start time `s(v)` of each actor `v` $\in \mathbb{N}$, and the frequency `f(c)` of each core `c` $\in \mathbb{N}$ (`MIN_FREQUENCY` and `MAX_FREQUENCY` are constant values representing the minimum and maximum available frequency levels in MHz). The constants in Code 1 are, the WCET (at `MAX_FREQUENCY`) `t(v)` of each actor, the number of initial tokens `d(i,j)` on the FIFO channel from actor `i` to actor `j`, and the period of the SPS `T` where `T` $\in \mathbb{R}$. The convex program solver scales the values of the variables to minimise the value of the objective function, i.e. Code 1 returns frequencies per-core `f(c)` that minimise the processor power consumption for a graph throughput of $\mathtt{T}^{-1}$.

6.2 Lookup Table Creation

We use the combined application and hardware platform DCP to generate the per-core table of DVFS operating points. For relatively small designs it is possible to do a full search of power consumption for all frequency combinations on all cores for small designs, and find the pareto optimal power versus frequency points from this set. Figure 8 presents such a pareto front for a full search of the example application that is illustrated in Figure 3. In this example, each processor has 16 available discrete DVFS levels. For a full search of the application mapped onto two cores, the period of 256 (16×16) DVFS combinations were derived and the pareto optimal in terms of power and period were selected to create Figure 8. However, we focus here on our sampling technique to derive the tables that is also applicable to larger designs that use more cores, where it may be undesirable or infeasible to perform a full search.

Using our sampling technique, we derive the tables by sampling the range of SPS period values T between the minimum period $T_{\min}$ and maximum period $T_{\max}$ of the worst-case graph. The DCP of the application is then solved for each value of T to find the per-core frequencies that produce the lowest power consumption while meeting the SPS period of T. Commonly, only a limited set of discrete DVFS operating points are available for scaling. To maintain temporal conservativeness in this case, the derived frequencies ($f \in \mathbb{R}$) are rounded up to the nearest available frequency levels and the period of the SPS derived for this new set of frequencies. This is achieved using a DCP similar to Code 1, where `T` is a variable, `f(c)` is a constant array of the rounded frequencies and the objective function is `minimise(T)`.

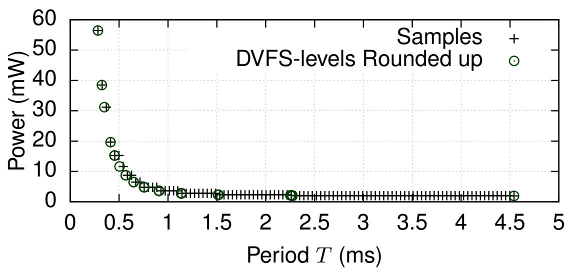
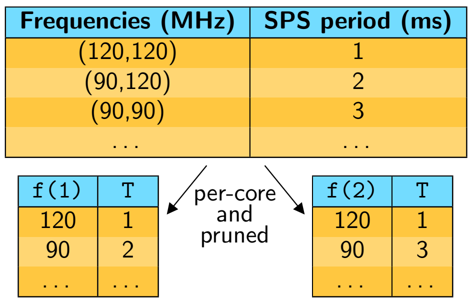


Figure 9: Pareto front from sampling the graph period range at regular intervals.



Frequencies (MHz)	SPS period (ms)
(120,120)	1
(90,120)	2
(90,90)	3
...	...

f(1)	T
120	1
90	2
...	...

f(2)	T
120	1
90	3
...	...

Figure 10: Example frequency table derivation.

This DCP is also used to find the SPS $T_{\min}$ and $T_{\max}$ by solving the DCP for `f(c)` values set to `MAX_FREQUENCY` and `MIN_FREQUENCY`, respectively.

Figure 9 presents the results of sampling the application's period range 100 times using the application's convex program to minimise power consumption. The samples can be seen to be on the same pareto front that was found using the full search and presented in Figure 8. The number of samples is configurable, with a higher number finding more suitable points once the DVFS-levels have been rounded up to available DVFS-levels.

The resultant rounded frequencies per-core with their associated SPS period are used to create per-core tables of frequency and associated SPS period, as illustrated in Figure 10. These tables are sorted according to the SPS period from lowest to highest. Multiple SPS periods can have the same associated frequency on a core. The tables are pruned to remove these duplicates, retaining only the lowest SPS period associated with the frequency level.

7. H.263 CASE STUDY

In this section, we proceed to demonstrate how our power management technique is applied to an H.263 decoder application. We demonstrate this using a four core Field Programmable Gate Array (FPGA) prototyped CompSOC platform instance. The H.263 decoder application, as illustrated in Figure 11, consists of five computational tasks (VLD, IQ, IDCT, MC+FR and UP) with data dependent task execution times. Table 1 presents the four task mappings that we used for our experimentation. These mappings were automatically generated to meet the H.263 decoders resource requirements, but are not intended to be power optimal or in anyway particularly suitable for a low-power design.

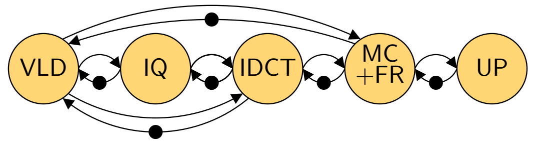


Figure 11: HSDFG of the H.263 decoder application.

No. Cores	B_c	VLD	IQ	IDCT	MC + FR	UP
1	-	1	1	1	1	1
2	0	1	1	1	2	1
3	2	1	1	3	2	3
4	4	1	1	3	2	4

Table 1: H.263 decoder task to core mappings.

The H.263 decoder requires 99 application graph iterations to complete a single frame of decoded video. Task execution times are data dependent and therefore the time taken to decode a frame depends on the type of frame and also the video being decoded. We use three videos (bus, tree and akiyo) to demonstrate our technique. These videos are encoded to have an I-frame every twelve frames and P-frames at other times. I-frames generally take longer to decode than P-frames. This can be seen in Figure 12 as a "dip" in frame rate every twelve frames, when the H.263 decoder is executed at the maximum DVFS level (MAX) and whenever static (VFS) is used. Static VFS is applied by setting the DVFS level once during initialisation to the level that meets the application's throughput requirement in the worst-case, as derived by our off-line analysis from Section 6 for a four frames per second requirement. As can be seen in Figure 12, the average frame rate for the VFS execution is close to eight frames per second. This is because the H.263 tasks are executing at less than the worst-case duration and the VFS technique is not able to use the dynamic timing slack to scale the voltage and frequency further.

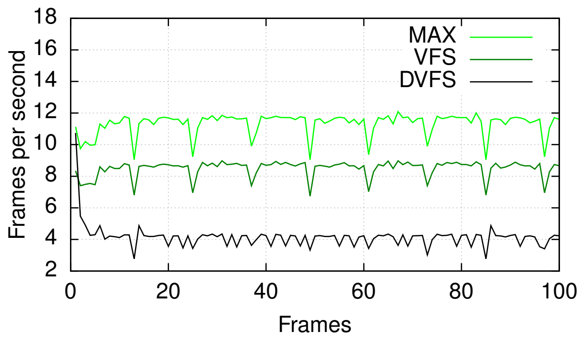


Figure 12: Decoding rates for the tree video (1 core).

Our distributed power management (DVFS) technique uses the dynamic timing slack to reduce the DVFS level further until the video is decoded at its required rate of four frames per second. Our power manager is invoked once per frame, i.e. every 99 application graph iterations. In Figure 13, the decoded rate occasionally dips below four frames per second, but the decoder does not miss any frame deadlines as our technique only decreases the DVFS level whenever the application progress is sufficiently ahead of the requirement to complete the decoding of the following frame even if it takes the worst case amount of time to decode, as is shown in Figure 13.

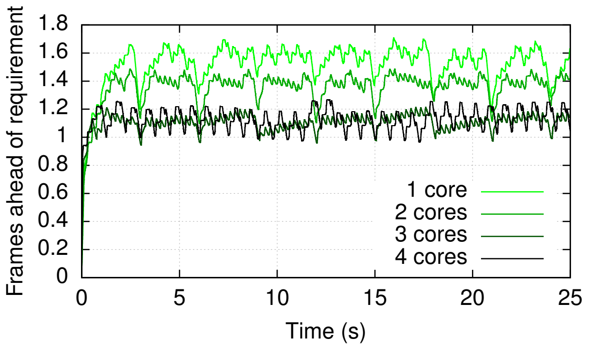


Figure 13: Buffered frames for the tree video.

The amount in advance of the requirement that our power management technique will produce output, and hence the amount of buffering that may be required, depends on the actual task execution time in comparison to the worst case off-line analysis. In Figure 13, it can be seen that the H.263 decoder ran up to approximately two frames in advance of the timing requirement. The amount of available memory for buffering can limit the amount the application can run ahead of schedule and hence the amount of application slack that is available for power management. The available memory for buffering is therefore a limitation of our power management technique with the effect being application and platform specific. In the case of the H.263 decoder, buffering two video frames is achievable in the Dynamic Random Access Memory (DRAM) of the CompSOC platform.

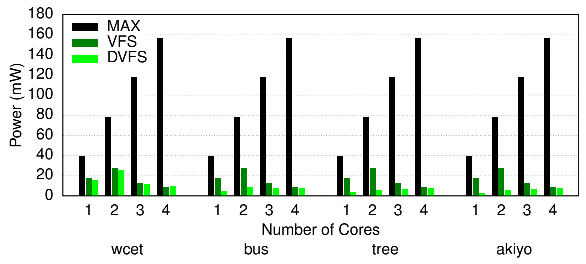


Figure 14: Reported H.263 decoder power consumption from the CompSOC FPGA prototype.

Figure 14 presents the power consumption and Figure 15 presents application throughput reported by the FPGA prototype of the CompSOC platform for all power management, application mapping and video input combinations. For this experimentation we add an additional video input called "wcet" that causes the H.263 decoder's tasks to always execute with their WCET. Figure 14 shows that both VFS and DVFS offer a significant power consumption improvement, over executing the application at the maximum DVFS-level, and that our DVFS technique offers an improvement over static VFS.

Figure 15 shows that this frame rate was met for all of the tested combinations. The static VFS is unable to use

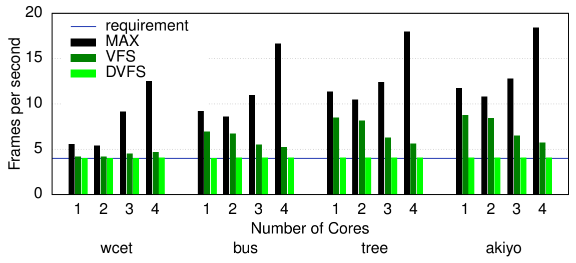


Figure 15: Reported H.263 decoder throughput from the CompSOC FPGA prototype.

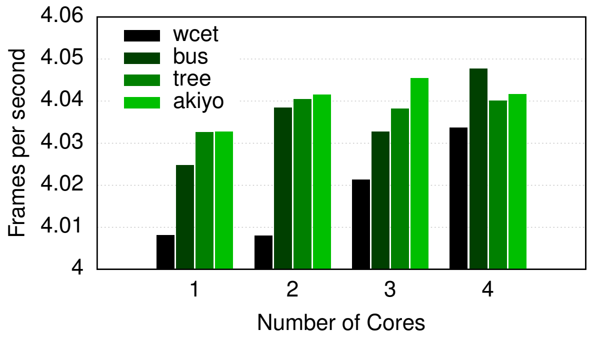


Figure 16: Frame rates achieved for the range of mappings and decoded videos using DVFS.

the dynamic timing slack and runs faster than required, whereas our DVFS technique uses almost all of the available application-level timing slack. This can be clearly seen in Figure 16 where our DVFS technique uses the application-level slack to within 1.25% above the application's timing requirement, over a 25 second time frame.

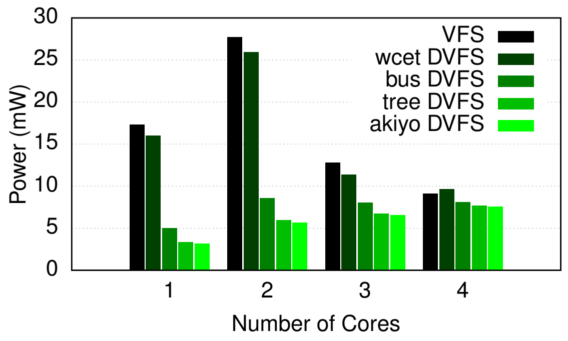


Figure 17: Power consumption due to our static and dynamic techniques.

Our DVFS power management technique is not guaranteed to outperform static VFS, as can be seen in Figure 17 for the four core mapping of the wcet video. This is because the wcet video does not have any variation in task execution times. The dynamic timing variations due to communication via the NoC does not produce enough application-level slack to offset the relatively small but additional computation required by our DVFS technique. Nevertheless, by using the available dynamic timing slack that the static VFS technique is unable to use, our DVFS technique achieves up to a 79.5% power reduction in comparison to using static VFS levels alone. This reduction can be seen in Figure 17 for the two core mapping of the akiyo video.

The more application-level timing slack an application has, the more effective our DVFS technique can be. This depends on many factors from hardware platform dimensioning, to dynamic task algorithm variations. Our power management technique uses low computational complexity table look-up, providing a low threshold of available application-level slack to produce power savings in comparison with static VFS alone. Nevertheless, we acknowledge that absolute numbers on power savings are for specific applications and platform configurations, or in other words, "your mileage may vary".

8. CONCLUSIONS

In this paper, we have shown how distributed power management of real-time applications can be achieved on a GALS MPSoC. We described our run-time power management technique that uses a static off-line worst-case analysis of a combined application and platform HSDFG to conservatively perform DVFS. Using an H.263 decoder application with various mappings on a CompSOC platform instance, we demonstrated our distributed run-time power management technique in comparison with what is achieved with using the static analysis alone to derive static VFS levels. Our distributed run-time power-management technique achieved up to a 79.5% decrease in power consumption, in comparison with using static VFS-levels alone.

Acknowledgements This work was partially funded by projects CATRENE CA505 BENEFIC, CA703 OpenES, CT217 RESIST; ARTEMIS 621429 EMC2 and 621353 DEWI.

9. REFERENCES

[1] S. Albers. Energy-efficient algorithms. *Communications of the ACM*, 53(5):86–96, 2010.

[2] S. Albers *et al.* Speed scaling on parallel processors. In *Proc. of the Nineteenth Annu. ACM Symp. on Parallel Algorithms and Architectures*, pages 289–298, 2007.

[3] A. Alimonda *et al.* A feedback-based approach to DVFS in data-flow applications. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(11):1691–1704, 2009.

[4] L. Benini *et al.* A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(3):299–316, 2000.

[5] T. D. Burd and R. W. Brodersen. Energy efficient CMOS microprocessor design. In *System Sciences, 1995. Proc. of the Twenty-Eighth Hawaii Int. Conf. on*, pages 288–297, 1995.

[6] M. Damavandpeyma *et al.* Throughput-constrained DVFS for scenario-aware dataflow graphs. In *Real-Time and Embedded Technology and Applicat. Symp. (RTAS), 2013 IEEE 19th*, pages 175–184, 2013.

[7] P. de Langen. *Energy Reduction Techniques for Caches and Multiprocessors.* PhD thesis, Delft University of Technology, oct 2009.

[8] P. de Langen and B. Juurlink. Leakage-aware multiprocessor scheduling. *Journal of Signal Processing Systems*, 57(1):73–88, 2009.

[9] D. Dolev *et al.* Hex: scaling honeycombs is easier than scaling clock trees. In *Proc. of the 25th ACM Symp. on Parallelism in algorithms and architectures*, pages 164–175, 2013.

[10] K. Goossens *et al.* Virtual execution platforms for mixed-time-criticality systems: The CompSOC architecture and design flow. *ACM SIGBED Review*, 10(3):23–34, 2013.

[11] K. Govil *et al.* Comparing algorithm for dynamic speed-setting of a low-power CPU. In *Proc. of the 1st Annu. Int. Conf. on Mobile computing and networking*, pages 13–25, 1995.

[12] M. Grant and S. Boyd. Cvx: Matlab software for dcp. `http://cvxr.com/cvx`, mar.

[13] M. Grant and S. Boyd. Graph implementations for nonsmooth convex programs. In *Recent Advances in Learning and Control*, pages 95–110. Springer.

[14] P. Huang *et al.* Energy efficient DVFS scheduling for mixed-criticality systems. In *Embedded Software (EMSOFT), 2014 Int. Conf. on*, pages 1–10, 2014.

[15] S. Irani and K. R. Pruhs. Algorithmic problems in power management. *ACM SIGACT News*, 36(2):63–76, 2005.

[16] P. Juang *et al.* Coordinated, distributed, formal energy management of chip multiprocessors. In *Low Power Electronics and Design, 2005. ISLPED'05. Proc. of the 2005 Int. Symp. on*, pages 127–130, 2005.

[17] P. Kumar and L. Thiele. p-yds algorithm: An optimal extension of YDS algorithm to minimize expected energy for real-time jobs. In *Embedded Software (EMSOFT), 2014 Int. Conf. on*, pages 1–10, 2014.

[18] T.-W. Lam *et al.* Energy efficient deadline scheduling in two processor systems. In *Algorithms and Computation*, pages 476–487. Springer, 2007.

[19] T.-W. Lam *et al.* Competitive non-migratory scheduling for flow time and energy. In *Proc. of the twentieth Annu. Symp. on Parallelism in algorithms and architectures*, pages 256–264, 2008.

[20] C. Lenzen *et al.* Clock synchronization with bounded global and local skew. In *Foundations of Comput. Science, 2008. FOCS'08. IEEE 49th Annu. IEEE Symp. on*, pages 509–518, 2008.

[21] O. Moreira *et al.* Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Trans. Comput.*, pages 188–201, 2010.

[22] A. Nelson *et al.* Power minimisation for real-time dataflow applications. In *DSD*, pages 117–124, 2011.

[23] A. Nelson *et al.* Dataflow formalisation of real-time streaming applications on a composable and predictable multi-processor SOC. *Journal of Systems Architecture*, 2015.

[24] M. Perner *et al.* Byzantine self-stabilizing clock distribution with HEX: Implementation, simulation, clock multiplication. In *DEPEND 2013, The Sixth Int. Conf. on Dependability*, pages 6–15, 2013.

[25] K. Pruhs *et al.* Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43(1):67–80, 2008.

[26] F. Yao *et al.* A scheduling model for reduced CPU energy. In *Foundations of Comput. Science*, pages 374–382, 1995.