Conservative Dynamic Energy Management for Real-Time Dataflow Applications Mapped on Multiple Processors

Anca Molnos[1]
NXP Semiconductors, The Netherlands
Email: a.m.molnos@tudelft.nl

Kees Goossens
Computer Engineering Department
Delft University of Technology, The Netherlands
NXP Semiconductors, The Netherlands
Email: kees.goossens@nxp.com

***Abstract*—Voltage-frequency scaling (VFS) trades a linear processor slowdown for a potentially quadratic reduction in energy consumption. Complex dependencies may exist between different tasks of an application. The impact of VFS on the end-to-end application performance is difficult to predict, especially when these tasks are mapped on multiple processors that are scaled independently. This is a problem for real-time (RT) applications that require guaranteed end-to-end performance.**

In this paper we first classify the slack existing in RT applications consisting of multiple dependent tasks mapped on multiple processors independently using VFS, resulting in static, work, and share slack. Then we concentrate on work and share slack as they can only be detected at run time, thus their conservative use is challenging. We propose SlackOS, a dynamic, dependency-aware, task scheduling that conservatively scales the voltage and frequency of each processor, to respect RT deadlines. When applied to a H.264 application, our method delivers 22% to 33% energy reduction, compared to dynamic RT scheduling that is not energy aware.

***Keywords*-Real-time; DVFS; Multi-processor; Dataflow;**

I. Introduction

Minimizing energy consumption is important for SOCs in general, and especially for embedded SOCs that operate on battery power. In such energy-aware SOCs *voltage-frequency scaling* (VFS) 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 executing independent applications [1], [2].

However, the applications executed on such platform have inherent *data dependencies* among their constituent tasks. Embedded SOCs often implement *real-time* (RT) applications, e.g. (portable) multi-media or automotive applications. It is crucial to guarantee application performance, such as throughput and latency, to avoid severe quality degradations, or deadline misses. Thus, beyond per-task guarantees, *end-to-end* application guarantees are required. To offer such guarantees, worst-case assumptions have to be made. Nevertheless, during execution, the system might behave "better" than in the worst case, (e.g. earlier data arrival, shorter execution times). Thus the system might be over-provisioned, so at run-time it has *slack*. Conservatively saving energy using this slack is complex because the *data dependencies* among application tasks, which may be mapped on *multiple processors*. Existing approaches for energy minimization in such RT applications, using the slack at run-time, are restricted to static tasks order [3], [4], or require the processors to synchronize at each VFS action [5].

[1]Currently with the Computer Engineering Department, Delft University of Technology, The Netherlands.

Dataflow analysis tackles the problem of mapping of RT applications on multiprocessors [6], [7]. At design-time, among other system parameters, the dataflow method calculates the budget allocated to each task in a given time period on a processor, such that, regardless of the scheduling order of tasks, no application deadline is missed. To produce a datum, an application has to perform some *work*. When dimensioning these budgets, the worst-case data arrival and work are considered. At run-time, as long as each task gets its budget, the end-to-end application performance is guaranteed. Thus, an Operating System (OS) needs only to enforce these budgets on each processor, regardless of data availability, task progress, or execution time.

The problem that we solve in this paper is to find the VF operating points for each task such that the energy is minimized, while still guaranteeing the end-to-end performance. First we identify the types of slack potentially existing in such applications: static, work and share slack. Work and share slack cannot be determined and conservatively used at design time. Hence, we propose SlackOS, a dynamic, run-time RT task scheduling OS as described above, extended with energy management functionality. SlackOS detects the slack after it is "produced", and it allocates it to a task that is ready for execution (which may result in task reordering). The processor operating point is scaled down such that at worst case the scheduled work does not take more than its original budget plus the slack. This, plus the fact that the design-time analysis is valid regardless of the task order, guarantees that, if the original application meets its deadlines, the VF-scaled one also will. SlackOS implementation requires the following extensions to the original OS: (1) each task signals its progress (to detect early termination), and (2) tasks can be checked for input data and output space availability (to determine task readiness). Experimentally we observed that SlackOS achieves a 22%-33% energy reduction in a H.264 application running on two processors, compared with non energy-aware scheduling. This reduction represents 60% and 94% of the maximum achievable energy save, obtained when the processors constantly run at the minimum frequency. However, deadlines are missed, in this case, whereas our energy-aware scheduler is guaranteed to never miss a deadline. We compare SlackOS only with a non energy-aware scheduler because, to our knowledge, no method exist to save energy conservatively, for real-time data-dependent applications, with end-to-end deadlines and run-time task reordering.

In the remainder of this paper, we introduce the dataflow model of applications and architectures, and the energy consumption model (Section II). We identify three slack types: static, work and share slack in Section III. SlackOS is presented in Section IV, experimental results in Section V, related work in Section VI, and we conclude in Section VII.

II. Background and models

In this section we model applications and their mapping on processors as dataflow graphs. The worst-case *execution time* of a task is modeled by its worst-case *work* combined with the *operating point* (rate of work) of the processor it is bound to. The worst-case *response time* models that fact that multiple tasks share a resource [6]. Then we present the *energy* dissipation model for our SOC, as a function of the processor's operating points.

A. Dataflow application model

A static dataflow graph is a directed graph $G = (V, E)$ consisting of a set nodes $v \in V$ known as *actors*, and a set of edges $d \in E, d = (v_x, v_y)$ representing dependencies among actors. The actors synchronize by communicating tokens, in a FIFO order, following the direction of the edges. For edge (v_x, v_y), v_x is a predecessor of v_y, and v_y a successor of v_x. Each dataflow actor *fires* for an infinite number of activations, based on its firing rule. A firing rule specifies, for one activation, for each incoming and outgoing edge, the number of input tokens required and the number of tokens produced, respectively. The static dataflow model naturally describes a streaming application: a task is an actor, and a task iteration is an actor firing. FIFO communication between two tasks is represented as a pair of opposing edges; one modeling the communicated data, and the other modeling the available inter-task buffer space.

B. Dataflow architecture model

We assume a general architecture template as described in [8], where tasks share resources (processors, memories, interconnect) using starvation-free arbiters. We denote the set of tasks by T and the set of resources by R. In this paper we focus on the processor resources, leaving the use of memory and interconnect resources for future work. Each task t_i is statically bound to one processor p_k (tasks do not migrate), by the function *bind* : $T \rightarrow R$. Multiple tasks may have dependencies as described above, may be bound to a single resource, but resources are independent. The application, its resource usage, and arbitration policies are modeled in dataflow. As a result, after the application is bound to a platform instance, we can analyze RT properties, such as end-to-end application throughput, arbiter settings, and buffer sizes [6], [7]. We illustrate this dataflow model in Figure 1 that shows an application with 4 tasks (circles) and three or four processors (squares), bound in two ways: (A) each task has its own processor, and (B) two tasks share the same processor. A task's self-edge expresses that tasks are not re-entrant, i.e. cannot start a new iteration before finishing the current one. For clarity, in this example the inter-task communication buffers are considered infinite, i.e. task dependencies miss the opposing edge.

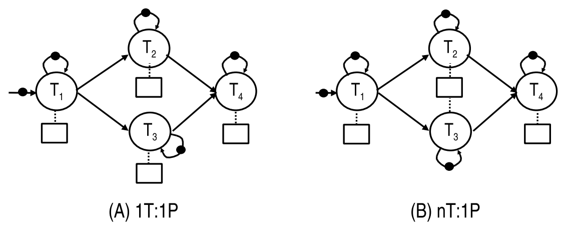


Figure 1. Task to processor binding examples.

Each t_i requires from its processor p_k, in each iteration j, an amount of work $w_{i,j}$, measured in cycles. The worst-case work of t_i is $wcw_i = \max_{\forall j}(w_{i,j})$. Traditionally, each p_k is operating at the maximum frequency f_k^{max} for which the processor's hardware is synthesized. Thus the *execution time* $et_{i,j}$ of t_i's iteration j depends on the requested work and the processor's operating point, $et_{i,j} = w_{i,j}/f_k^{max}$. Worst case $wcet_i = wcw_i/f_k^{max}$.

A task t_i is *ready* to execute when its input data and output space are available, i.e. in the model the actor has enough input tokens to fire. t_i is ready for its j^{th} iteration at time $r_{i,j}$ and it completes it at $c_{i,j}$. Typically, multiple tasks share a processor, therefore each one can use only a fraction of the processor's total capacity. Due to serialization, a task cannot always execute immediately when it is ready, as another task may be using the processor. Resource sharing is modeled in the dataflow analysis by using instead of $et_{i,j}$, the *response time* $rt_{i,j}$, defined as the difference between the ready and the completion time of the task: $rt_{i,j} = c_{i,j} - r_{i,j}$. The worst-case response time $wcrt_i$ is a function of $wcet_i$, of the scheduler type and settings, and of the worst-case data arrival. Note that with no processor sharing (1T:1P binding), the worst-case response time of a task equals its execution time ($wcrt_i = wcet_i$), as a task can start immediately when it is ready, whereas otherwise $wcrt_i \geq wcet_i$.

In our platform, we use preemptive time-division multiplexing (TDM) scheduling, with or without static ordering [7], but any other starvation-free arbitration could be used. Figure 2 shows an example of a TDM schedule, with a repetition period P_k, from which t_i has a slice S_i representing a fraction of the available processor capacity. As visible in Figure 2 the worst data arrival moment is when the slice of t_i just finished and the best case data arrival moment is when the data is available from the very beginning of t_i's slice. Thus, in the case data arrives at the worst case one slice is wasted in the sense that t_i cannot be executed. However, for the real-time analysis the worst-case data arrival has to be taken into account when deriving the guaranteed throughput that can be delivered by the system, thus one extra slice is budgeted. In this case, the worst-case response time is [6]:

$$wcrt_i^{TDM} = (P_k - S_i)\left\lceil \frac{wcet_i}{S_i} \right\rceil + wcet_i. \tag{1}$$

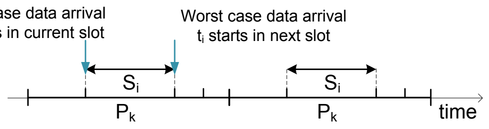


Figure 2. TDM worst/best case data arrival.

General arbitration can invalidate RT guarantees [9], but the dataflow execution model is *monotonic*, i.e. starting early only reduces the worst-case completion time of successors As a result, the minimum end-to-end throughput can only increase due to slack.

To analytically determine the minimum end-to-end throughput that an application mapped on an architecture can deliver, we use the set of all cycles $\mathcal{C}$ in the dataflow graph. In conventional dataflow analysis, the cycle mean of a cycle C_l is defined as $cm(C_l) = \frac{1}{\Delta_l} \sum_{t_i \in C_l} wcrt_i$, where Δ_l is the sum of initial tokens on a cycle [10]. The RT application's throughput $\mathcal{T}$ that can be guaranteed is:

$$\mathcal{T} \leq \frac{1}{\max\limits_{C_l \in \mathcal{C}}(cm(C_l))}. \tag{2}$$

Aiming to reduce energy, we extend prior dataflow model work by considering that each processor p_k has multiple independent VF operating points. Each iteration j of task t_i bound to p_k can have its own operating point, denoted by its frequency $f_{k,i,j}$. Voltage is derived from frequency, as described in Subsection II-C. Note that a task's requested work $w_{i,j}$ is independent of the operating point of the processor, but the execution time $et_{i,j}$ depends on the requested amount of work $w_{i,j}$ and on p_k's operating point: $et_{i,j}(f_{k,i,j}) = \frac{w_{i,j}}{f_{k,i,j}}$. In the rest of this paper all the times ($et_{i,j}, rt_{i,j}$, etc.) correspond to the maximum frequency f_k^{max}, unless explicitly specified otherwise.

C. Energy dissipation model

The *dynamic power* depends on the switching activity α, the capacitance of the physical processor's circuit C, the supply voltage V, and the clock frequency f: $P_{sw} = \alpha C V^2 f$. The *energy* is $E_{sw} = P_{sw} t$, where t is the length of time during which energy is dissipated. Since $t = w/f$, i.e. executing the required work w at operating point f, an alternative formulation is: $E_{sw} = \alpha C V^2 w$.

We assume a linear dependency between the voltage and the frequency, as experimentally determined in [11], with maximum frequency f^{max} and voltage V^{max} of the circuit. We also assume a minimum voltage V^{min} (and a corresponding f^{min}) to the voltage scaling, below which the energy is frequency independent, thus VFS does not save any energy. Hence a processor's frequency ranges from f^{min} to f^{max}. The energy is $E_{sw} = \beta f^2 w$, (where β is an adequacy constant) and VF scaling trades a linear processor slowdown for a quadratic energy reduction.

In this paper we assume that the VF scaling is implemented as proposed in [12], thus its time penalty is $\tilde{2}$00ns (for voltage swing from 1.2V down to 0.6V for an chip in 0.13um CMOS), i.e. negligible at the time granularity of tasks. Moreover, the VFS energy overhead is assumed to be zero (as the VF is scaled at most once per task iteration, as we present in Section IV, and the energy overhead of the scaling is typically significantly smaller than the energy spend in an iteration). Therefore, if a multiprocessor executes N tasks each of which process M_i iterations at different frequencies $f_{i,j,k}$, the total dissipated energy is:

$$E_{tot} = \sum_{i=1}^{N} \sum_{j=1}^{M_i} \beta f^2_{bind(t_i),i,j} w_{i,j} \tag{3}$$

In the following section we give a taxonomy for the *slack*, the base for our energy management method.

III. Slack and its use

In this section we introduce *slack*, which intuitively corresponds to unused capacity of the processors. Then we formulate the energy minimization problem than we want to solve in this paper, discuss possible strategies to reduce energy and their applicability in the RT domain. Finally, we categorize slack and comment on its effects.

In a platform as the one introduced above, slack exists because of variations: (1) in the *application* behavior, (2) in the *environment* where the application executes, and (3) between the *model* (application, architecture, binding) and the real life. The application related slack occurs because: (a) not all cycles in the dataflow graph are equally long and (b) tasks require less work than the worst-case for which the system was provisioned. The execution environment related slack is present when data arrives earlier than assumed when computing $wcrt_i$'s; The modeling slack occurs when (a) the computation of $wcrt_i$ and $wcet_i$ is not tight, and (b) resources are over-allocated due to discretization, i.e. rounding up to fixed-size time slices.

In this paper, we tackle the following *energy optimization problem*: find the operating points $f_{k,i,j} \leq f_k^{max}$ for each task firing such that the energy E_{tot} is minimized, while still guaranteeing the throughput $\mathcal{T}$ for which the application is designed (assuming f_k^{max}).

When a task completes s time units earlier, the time available for the next ready task t_i on the same processor is $wcet_i + s$. Hence the operating point can be conservatively scaled down to: $f_{k,i,j}(s) = \frac{wcw_i}{wcet_i + s}$, and the throughput of the application still guaranteed. Considering the relation among the $wcet_i$ and f_k^{max} in Subsection II-B:

$$f_{k,i,j}(s) = f_k^{max}\left(1 - \frac{s \cdot f_k^{max}}{wcw_i + s \cdot f_k^{max}}\right). \qquad (4)$$

The optimization problem above can be solved by employing an Energy Manager (EM). An EM's policy can be categorized using several criteria:

1. Employment phase. The policy can be applied prior to execution (design time), or during execution (run time).

2. Safety. A policy is *conservative* if it guarantees deadlines are not missed, and *non-conservative* otherwise. Often, non-conservative energy management predict and use future slack, maybe missing deadlines.

3. Slack certitude. In a system, the current *proven* slack is the one that it has been generated in the past and it and not lost (details in Section III-D). Proven slack can be used conservatively, but for unproven (speculative) slack this only holds if its lower limit can be computed upfront (e.g. at design time). The difference between proven and unproven slack refers only to the point in time when the slack is considered (namely after or before it was generated).

4. Scope. A *local* policy uses only information about tasks on the local processor, avoiding exchange and synchronization of information between processors.

We solve the formulated optimization problem by employing a conservative run-time energy manager, that requires only information local to each processor. The main justification is that distributed scheduling is more complex, especially because schedulers on different processors are not synchronized, and inter-processor state synchronization is relatively slow. Moreover, to preserve scalability, we would not like to add extra inter-processor dependencies and synchronization points. Below, we split this slack in two main categories the static and dynamic slack; the latter is split in work and share slack.

A. Static slack

Static slack is present even when all tasks require their worst case work, because some cycles means $cm(C_l)$ may be smaller than the inverse of the guaranteed throughput. Thus the difference between $1/\mathcal{T}$ and $cm(C_l)$ represents the static slack of each cycle C_l. The $f_{k,i,j}$ can be conservatively determined at design or run-time, i.e. by "stretching" all cycles via VFS, until they equal $1/\mathcal{T}$. This requires knowledge not local to a task (which task belongs to which cycles) that however is available at design-time, at no extra energy cost for an Energy Manager (EM) regardless if it uses the slack at design- or run-time.

Figure 3(A) presents the static slack for the 1T:1P binding in Figure 1(A). No processor is not, so $wcet_i = wcrt_i$. Each task represents a cycle and the longest one in this example is given by t_1. For simplicity we consider that the desired throughput equals $1/wcet_1$. In this case tasks t_2, t_3 and t_4 have static slack, as $wcet_{2,3,4} < wcet_1$ in each iteration.

B. Work slack

In practice not all task iterations require the worst case amount of work, as the work $w_{i,j}$ is data dependent. The *work slack* of a task in an iteration is given by the difference between $wcet_i$ and $et_{i,j}$. We consider the slack due to $wcet_i$ inaccurate computation to be included here. In the literature work slack is typically called dynamic slack, but we use the term work slack to distinguish it from the share slack introduced below, that also has a dynamic nature.

Figure 3(B) presents the work slack in addition to the static slack in Figure 3(A). The throughput is still given by $wcet_1$, and each task has a slack equal to the difference between its execution time in each iteration and $wcet_1$.

The work slack can be conservatively utilized at run-time only if it proven (i.e. after it is produced). A run-time speculative slack estimation is subject to miss-prediction, thus may lead to a deadline miss. Hence, past slack can be used when setting the operating points of future iteration(s) of the slack recipient(s). To determine the work slack, an EM has to know $wcet_i$ (available at design time) and $et_{i,j}(f_{k,i,j})$. Determining $et_{i,j}(f_{k,i,j})$ at run-time requires tasks to signal to the EM their *computation progress* (i.e. start and completion time). To exploit the slack, another task should be started earlier. To be sure that is possible, the task's readiness (i.e. the corresponding actor's firing rules) should be checked. On our platform, this information is locally available to each processor.

C. Share slack

As already mentioned, the binding of multiple tasks on a single processor is in practice more common than a 1 to 1 mapping, and the response time accounts for the serialization and arbitration effects. In the TDM case, due to discrete scheduler quanta or to slices allocation, a task might not start immediately when its data is available. Thus processing capacity is budgeted to account for the worst case of these effects. However, in practice data might arrive earlier than expected, hence the system has *share slack*.

Figure 3(C) illustrates the effects of serialization on the slack in Figure 3(B). The difference among the two figures is that in (C) t_2 and t_3 shared the same processor. Furthermore, an example of share slack is presented for a TDM arbitration scheme, in Figure 3(D). There t_2 cannot start immediately after its input data is available, as the slice belongs to t_3.

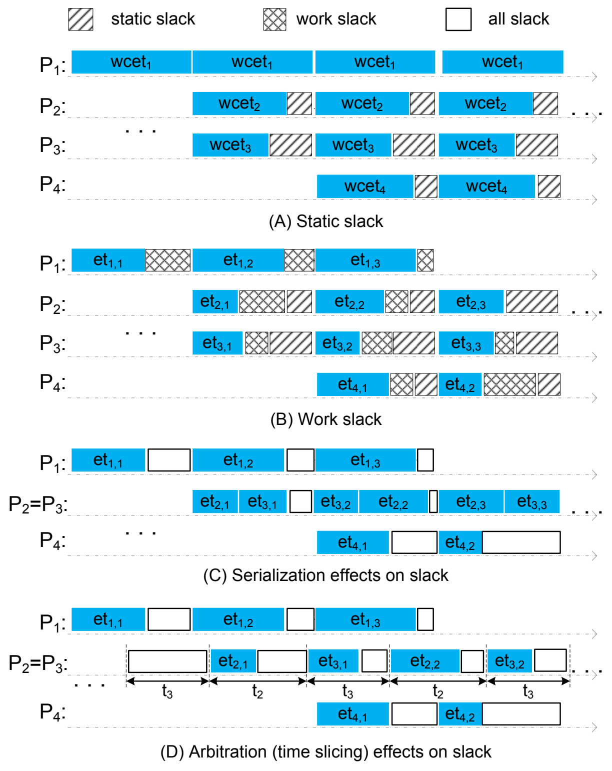


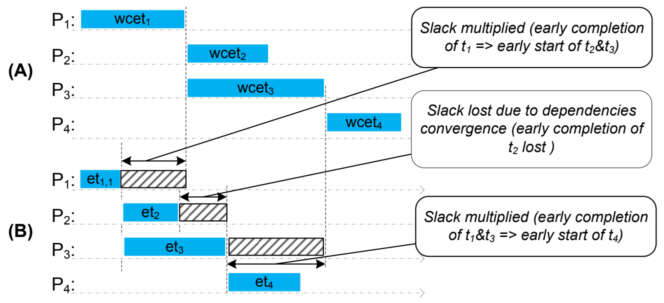
Figure 3. Static, work and share slack.

Similar to the work slack, the share slack nature is dynamic. However, its amount is known at design time (being the budget needed to cover for the worst-case data arrival). Nevertheless its occurrence in time is dynamic, as the actual data availability is dictated at run-time. Thus, conservative exploitation of share slack requires information about the task's readiness (locally available, as already mentioned) and it is possible if the slack is proven, same as for work slack.

D. Slack propagation

Regardless of its source, an effect of slack is early task completion, hence possibly early ready time of its successors, some perhaps executing on other processors. Hence, these successors may start early, hence end earlier than the worst case, causing slack to occur. Therefore slack produced on one processor, can "multiply" to other processors. However, an early task completion does not always result in an early start of its successors. (i.e. not all slack propagates). When a task has more than one predecessor, its earliest starting time is the maximum among the finishing times of its predecessors, hence the slack of some of its predecessors is lost when multiple data dependencies "converge" in a task. Oppositely, a divergence of dependencies, i.e. multiple outgoing edges from t_i, causes the slack of t_i to multiply, in the sense that all its predecessor are potentially enabled earlier. Slack multiplication actually means that the work slack of a task causes share slack to another task on a different processor. In this manner the slack can be "accumulated" in the system during several consecutive iterations.

These effects are exemplified in Figure 4 for an iteration of the task graph in Figure 1(A). Two cases are presented: the case (A) in which tasks execute in their worst case time, and the case (B) in which tasks take less than the worst case time, thus have slack. We can observe that the early completion of t_1 (bound to P_1) may cause t_2 (bound to P_2) and t_3 (bound to P_3) to start earlier, thus the slack multiplies (propagates from P_1 to P_2 and P_3). Moreover, we can see that the early completion of t_2 does not result in an early start of its successor, t_4 (a convergence of dependencies). In this manner the slack of t_2 can be lost (note however that if more task would have been bound to P_2, one of them could have used this slack). However, in case (B) t_4 starts earlier than in (A) because of t_1 and t_3 slack that propagates to P_4.



(A)
P1: wcet1
P2: wcet2
P3: wcet3
P4: wcet4
Slack multiplied (early completion of t1 => early start of t2&t3)
Slack lost due to dependencies convergence (early completion of t2 lost)
(B)
P1: et1,1
P2: et2
P3: et3
P4: et4
Slack multiplied (early completion of t1&t3 => early start of t4)

Figure 4. Slack multiplication&loss from a processor to another, along data dependencies

Above we considered the ideal case when the inter-task buffers have an infinite size. However, in practice that is not the case, and the size of the buffers limit the amount of slack that can be accumulated over consecutive iterations. We illustrate this effect in Figure 5 using a simple example of a producer (t_1) and a consumer (t_2), each bound to a processor, and communicating via a FIFO. We compare two cases: the one in which the FIFO can store only 1 token and the one in which its size is 2 tokens. We present t_1's accumulated slack in several consecutive iterations. This slack is defined as the time between the last possible completion of an iteration (without missing a deadline), and the actual iteration completion. In this example we denote with $slack^m_{i,j}$ the slack of t_i accumulated till iteration j, when the FIFO has the size m. For simplicity, in this example we assume that the deadlines of the application are at the end of each t_2 iteration and that only t_1 has slack. t_1 can restart its next iteration as long as it has space in the FIFO. Thus, when the FIFO size is 2, the number of t_1 iterations that can be executed until filling the FIFO is larger than in case the FIFO size is 1. Regardless its size, after the FIFO is full the slack does not accumulate anymore, (for t_1, e.g. $slack^{(1)}_{1,2} = slack^{(1)}_{1,3} = slack^{(1)}_{1,4} = \ldots$ and $slack^{(2)}_{1,4} = slack^{(2)}_{1,5} = \ldots$). We can still use the slack generated by t_1 after FIFO filling, by allocating it to a different task that is ready for execution. For clarity, in this example we considered only the static slack, but the same considerations hold true also for the dynamic slack. In conclusion, the larger the FIFO the more accumulated slack (for example $slack^{(1)}_{1,3} < slack^{(2)}_{1,3}$, as visible in Figure 5). This is an interesting observation for applications for which an increase in quality of service (QoS) is desired (instead of VFS), but that extra QoS requires a certain amount of slack. For such applications the buffer sizes should be large enough to allow sufficient slack accumulation.

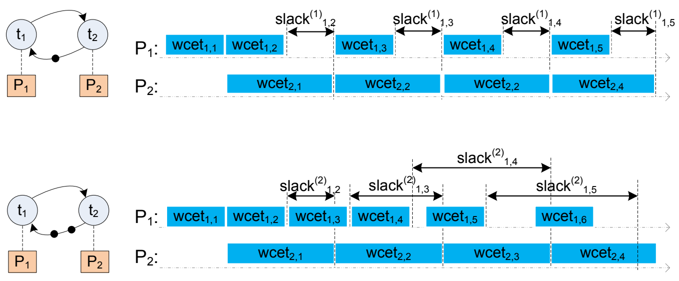


Figure 5. Slack accumulation for different FIFO sizes

IV. SlackOS - The Energy Aware Task Scheduler

In this section, we describe SlackOS, an operating system extension, to solve energy minimization problem. The focus of this paper is to exploit the dynamic slack, thus static slack is not discussed further. However, this is not a limitation, because the static slack can also be used at run-time, as mentioned in Subsection III-A.

As a starting point we have an existing task scheduler running on each core of a multiprocessor. This scheduler comes in two flavors: (1) pure Time Division Multiplexing (TDM) and (2) TDM mixed with static task ordering (TDM+SO). In TDM+SO a group of tasks are executing, sequentially, always in the same order, in a single time slice, the advantage being that it may decrease the worst-case response times (hence increase the achievable guaranteed throughput). For TDM+SO we regard such group of tasks as a single task. For both these flavors we assume that the schedule repetition period P_k and each task slice S_i are computed at design-time [6], [7]. The instantaneous frequency of a processor p_k is shortly denoted with f_k.

In both scheduling schemes, due to time slicing, the slack is not contiguous, i.e. it is also sliced. When a task t_i finishes earlier, its unused budged is distributed among one or more of the t_i slices, thus a t_i produced slack is:

$$s_{i,j} = cs_{i,j} + nS_i \cdot S_i \tag{5}$$

where $cs_{i,j}$ is the slack present in the current slice, and nS_i is the number of slices left for t_i to finish the current iteration. An example of this slack is visible in Figure 6, where we assume that in the worst case a t_i iteration execution time equals the size of the slice S_i. In this example t_i has both its data available earlier, and its iteration shorter than in the worst case, thus its slack is $s_{i,j} = cs_{i,j} + S_i$.

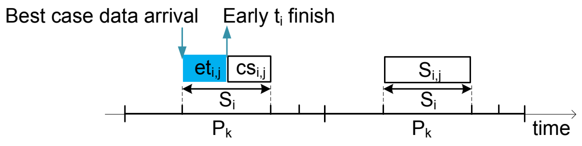


Figure 6. Example slack t_i, $s_{i,j} = cs_{i,j} + S_i$.

Subsequently, the energy management policy has to decide along the following dimensions for which several choices are possible:

1. VFS decision points: at an iteration end, at the beginning of a slice and/or at a point inside of an iteration.

2. Slack recipients number: at a decision point, the entire proven slack can be given to one or more recipient tasks. The advantage of having more slack recipients is that it offers a finer granularity management which may lead to higher energy savings. However, intuitively, giving slack to more than one recipient requires a larger administration overhead.

3. Identity of slack recipient(s) (slack balancing): at a given point more tasks might be available to use the slack. A simple order in which to pick the available recipient would be the scheduling order. However, it might not always enable slack multiplication, as introduced in Section III-D.

4. VFS granularity: VFS can be performed once, at the beginning of a task iteration, or several times, also in the middle of the iteration. If VFS is performed only once per task iteration some slack occurring later in the system might be left unused. However, scaling several times per iteration assumes either support for measuring the inside iteration computation progress or that the iteration execution time varies linearly with its required work over every iteration part, and not only over the entire iteration.

5. Idle slice management: if at the beginning of a slice no task is ready, SlackOS can decide to idle the entire slice or check again for tasks' readiness with a given periodicity. If the slices are large, it is likely that periodic checking for slack recipients decreases the energy consumption, however one has to keep in mind that each such check costs energy.

In this approach, SlackOS makes VFS decisions both at the end of an iteration and slice, gives the slack to a single recipient, searching for ready tasks in scheduling order, and scales only once, at iteration start. We leave the investigation of the other options and the slack balancing as subject for future research.

The SlackOS functionality requires extending the TDM as follows:

1. tasks' states; Task states are required to determine whether a task is executing or not, if it can start an iteration, etc. Hence a task can be: (a) *RUNNING* while executing an iteration on a processor, (b) *PREEMPTED*, when it is already started an iteration, has not finished it yet, but was preempted from the processor, and (c) *BLOCKED* when it has no input/output to progress.

2. ready tasks check; SlackOS can conservatively perform VFS only if it knows that the slack recipient can immediately start. Hence SlackOS administrates a table accounting for each task's input/output data/space, thus it can check the task's readiness. Being "ready" does not need to be an explicit task state, as immediately after checking the data/space availability the task goes either into the *RUNNING* or *BLOCKED* state.

3. computation progress; SlackOS has to be aware of each iteration end, thus the tasks or an underlying library have to signal the iteration progress (i.e. its end) to SlackOS.

The proposed energy manager changes the processor's frequency using Equation 4, as presented in Algorithm 1. We do not explicitly distinguish among the work and share slack, as it makes no difference for VFS. The slack slices nS_i accounting required to calculate the slack is done at t_i's iteration end (increase nS_i) and at t_i's slice end (decrease nS_i), to calculate the slack according to Equation 5. When an iteration ends, if the slack cs_i is larger than *MINSLACK* (the slack threshold), $cs_{i,j} > MINSLACK$, there is a potential for energy saving in that slice, by starting a task earlier and scaling down VF. SlackOS first checks if the next iteration $j+1$ of the same task can be started (continuing the same task is cheaper than switching tasks, due to the task switching overhead). If that is the case, t_i starts its next iteration, and f_k is set considering the entire t_i's slack $s_{i,j}$. If not, the SlackOS searches for a ready task, t_x, in TDM schedule order. If one is found, the f_k is changed considering only $cs_{i,j}$. This choice can cause slack waste, and it is subject for future work. However, passing slack that occurs in future periods complicates the scheduler, i.e. to ensure no deadline is missed, it has to check if the future slack does not occur after the latest moment when the slack recipient should complete its iteration. If no ready task is available to use the slack, the remainder of the slice is left idle. *MINSLACK* is a constant that reflects the break even point among the potential savings and the overhead of calculating and switching the operating point.

At the beginning of t_i's slice, if t_i is the *PREEMPTED* state, it simply resumes execution. Otherwise, if t_i is the *BLOCKED* state, SlackOS checks if it cannot start, i.e. checks for data. If so, t_i is started, and if moreover, t_i is having slices left from its previous iteration ($nS_i \neq 0$), the frequency is scaled accordingly. When t_i is not ready, and another task t_x can start earlier, t_x is scaled using t_i's current slice and possibly its own left slices ($S_i + nS_x \cdot S_x$). Otherwise the slice is left idle. Note that we do not aim to reduce the energy dissipated while the processors are running idle; this can be done by simply turning them off when idle.

Algorithm 1: SlackOS energy manager.

```
// At end iteration $j$, task $t_i$;
$nS_i++$;
if ($t_i$ is ready) then
    $f_k = f_{k,i,j+1}(cs_{i,j} + nS_i \cdot S_i)$;
    $t_i \rightarrow RUNNING$;
else
    if (found $t_x$ ready) then
        $f_k = f_{k,x,h}(cs_{i,j} + nS_x \cdot S_x))$;
        $t_x \rightarrow RUNNING$;
    else idle slice;

//At end slice, iteration $j$, task $t_i$;
$nS_i--$;
if ($t_i$ is RUNNING) then $t_i \rightarrow PREEMPTED$;
else $t_i \rightarrow BLOCKED$;

//At start slice of iteration $j$, task $t_i$;
if ($t_i$ is PREEMPTED) then
    $t_i \rightarrow RUNNING$;
else
    if ($t_i$ is ready) then
        $f_k = f_{k,i,j}(nS_i \cdot S_i)$;
        $t_i \rightarrow RUNNING$;
    else
        if (found $t_x$ ready) then
            $f_k = f_{k,x,h}(S_i + nS_x \cdot S_x)$;
            $t_x \rightarrow RUNNING$;
        else idle slice;
```

We would like to stress the fact that SlackOS is not limited to platforms modeled as dataflow. From the application and platform model SlackOS has to know only the budget and the worst case execution time for each iteration of each task. Hence the energy-managed platform misses no deadline if the initial design meets them, and the system is monotonic (giving more budget to a task does not delay its completion). Summarizing, the novelty of SlackOS resides in the fact that it conservatively scales VF and allows reordering of tasks at run-time, in shared multi-processors executing RT tasks with complex dependencies, requiring only local information.

V. Experimental results

In this section, we first briefly introduce the experimental platform and then we compare the energy consumption of an existing real-time multiprocessor as it is, with the same one but this time managed by SlackOS.

The simulation platform follows the tiled template of the predictable multiprocessor in [8], and has 2 ARM processors connected by a NoC [14], and sharing an external DDR memory, as visible in Figure 7. Each ARM has a local memory to store local data as well as FIFOs data and administration structures. The data communication outside a tile is controlled by a Communication Assist, CA (with a functionality similar with Direct Memory Access (DMA)). For instance when a task writes data into a FIFO, it actually writes it in the local tile memory, and the CA forwards the data to the local memory of the FIFO consumer. This ensures that the computation and communication can be performed in parallel, and the processor performance is not heavily penalized by long latency accesses to remote memories.

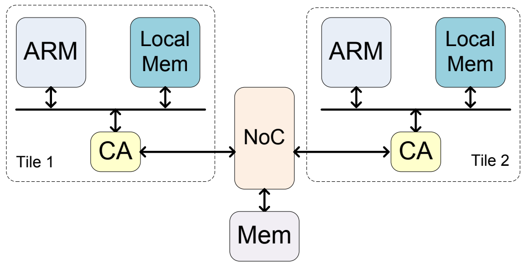


Figure 7. Multi-processor experimental platform

On the platform briefly presented above we execute an H.264 decoder application with 10 tasks [13], first scheduled using TDM, then using TDM+SO. The H.264 is presented in Figure 8 (for readability, we omit the self-edges, and each arrow represents a FIFO, corresponding to a pair of data-flow edges). The *entropy decoder*, the *loop filters* and the *frame buffer* tasks are mapped on the first ARM processor and the *transform decoders* are mapped on the second one. As the H.264 is a large application, and the freely available ARM instruction-set simulator (i.e. SWARM [15]) is a slow one, in order to be able to exercise input data streams with a large number of frames (72 in this paper) we utilize a fast simulation method. This method requires an initial step of gathering the execution times of each task iteration $et_{i,j}$ running at f_k^{max}. The $wcet_i$ is calculated for each task as the maximum $et_{i,j}$. This step is a slow one, but it is performed only once. Then, whenever we experiment with an energy saving policy we accurately simulate only the inter-task FIFO communication, and the SlackOS code. To speed-up the simulation we only account for the tasks execution time (the equivalent of a SystemC "wait()"). In case VFS is apply during an task iteration, the time that iteration takes is scaled accordingly, $et_{i,j}(f_{k,i,j}) = \frac{et_{i,j} f_{k,i,j}}{f_k^{max}}$ (as it can be inferred from Section II).

In both scheduler cases, TDM and TDM+SO, the parameters (size of the time slices, TDM repetition period, tasks order) are determine at design-time using dataflow methods as mentioned in Section II, and the scheduler performs task switching at fixed points in time. We augment this scheduler with SlackOS features, as described in the previous section.

We consider that the maximum frequency of the ARM cores is 200MHz and the frequency below which the voltage does not scale anymore is 91MHz [11]. We carried out experiments for the following three cases: (WCET) each task requesting its worst-case work wcw_i, and no VFS, (ACET) each task requesting its actual work $w_{i,j}$, and no VFS, and (SlackOS) each task requesting its actual work and the energy managed as in the section above. The energy is

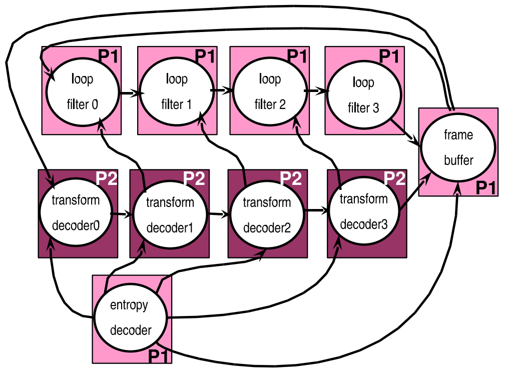


Figure 8. H.264 experimental application

presented cumulatively, per frame, normalized to the highest energy consumption experimented (i.e. the ACET energy consumption of the 72nd frame). We compare the energy consumed when using SlackOS only with the one of an energy unaware multiprocessor. The reason is that, to our knowledge, no method exist to save energy conservatively, for real-time data-dependent applications, with end-to-end deadlines and task reordering.

In the following we present: (1) the energy savings obtained when applying our strategy, and (2) the average processors utilization (the higher the processor utilization, the smaller the slack left unused). The results include the time and energy spent in the execution of the SlackOS's code, and do not include the energy spent when ARMs are idle (if the idle energy would have been included the differences between ACET and SlackOS would be even larger as the energy-managed platform is idle for less time). We investigate the processors utilization over a large range of minimum processor frequencies $f^{min} \in \{200, 125, 91, 50, 33, 25\}$MHz. Note that under 91MHz theoretically the energy can not be reduced anymore, however here we want to measure SlackOS potential to use the slack.

For an H.264 decoder scheduled with TDM, the SlackOS reduces the energy consumption with 22% when compared with the energy spent in ACET (Figure 9). The ideal energy saving achieved when the processors run constantly at f^{min} is 37%, when compared to ACET. Note however that when the processors run continuously at f^{min}, deadlines can be missed. SlackOS saves 60% from the maximum possible energy reduction, while guaranteeing zero deadline misses. As visible in Figure 10, the processor utilization numbers for the SlackOS saturate after an $f^{min} = 91$MHz, and do not reach the values of WCET utilization. This suggests that, in the case of TDM scheduled H.264, SlackOS does not fully utilize the slack. The underlying reasons are: (1) SlackOS exploits only the intra-slice slack cs_i when passing it among tasks, and (2) when no task is ready to use the slack the entire rest of the slice is spent in idle. (3) the frequency is set once per iteration, thus the slack cannot be passed to a *PREEMPTED* task. Moreover, we observed that for the H.264 case most of the times the slack could be passed only inter-task, as the next iteration of the current task was typically not available.

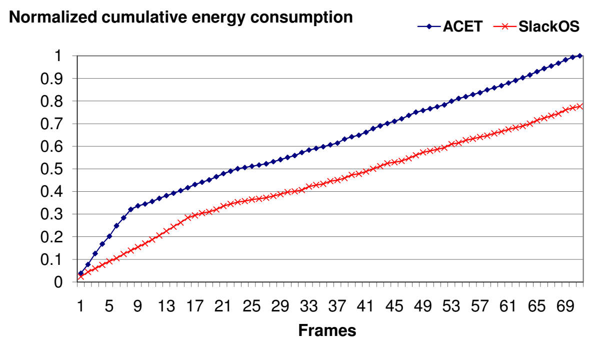


Figure 9. H.264 normalized energy, cumulative, per frame (TDM)

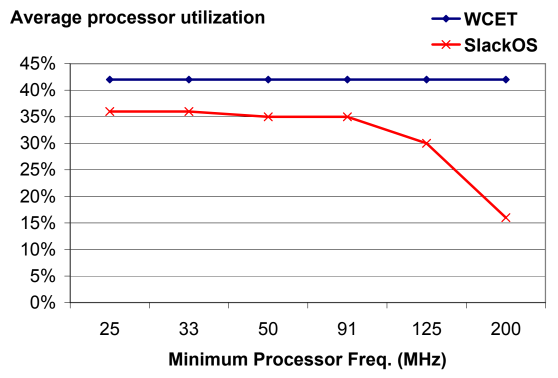


Figure 10. Avg. procs. utilization (TDM)

In the case of TDM+SO, the energy saving when the ARMs run constantly at f_k^{min} is 35%, when comparing with the energy spent in ACET. Here our policy achieves very close to ideal savings while keeping the throughput guaranteed (the energy dissipated using SlackOS is 33% from ACET's energy, as visible in Figure 11). The processor utilization, as observed in Figure 12, goes very soon close to the WCET case. This suggest that in this case SlackOS uses the entire slack. The difference with the pure TDM based policy is that in the TDM+SO case a ready slack recipient was always found, thus the slack was not lost.

VI. Related work

A good overview of low power techniques is presented in [2]. Most of the existing techniques for energy reduction in multi-processors differ from ours because tackle use static slack, at design-time. One of the first methods to handle dependent tasks [16], performs simultaneous task assignment and scheduling. The authors of [17] introduce power-aware static slack distribution and priority adjustment. In [18] they propose an optimal solution for combined supply voltage

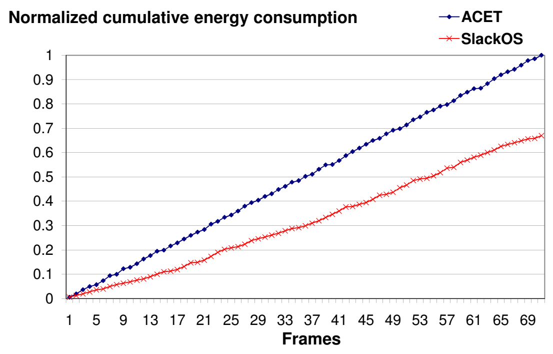


Figure 11. H.264 normalized energy, cumulative, per frame (TDM+SO)

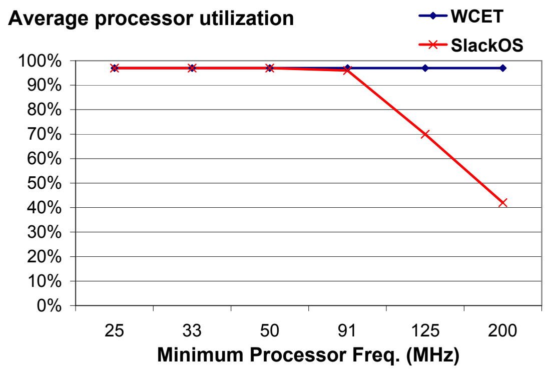


Figure 12. Avg. procs. utilization (TDM+SO)

and body bias selection for multiprocessor systems executing applications with time constraints. The work in [19] tackles acyclic task graphs. In [20] the switching and leakage power is minimized via VFS and adaptive body biasing, under time constraints, by iteratively scaling the task that are likely to save the most power. Manzak et al. [21] propose a distribution of slack among the nodes in the data-flow graph. The distribution procedure approximates the minimum energy relation iteratively derived using the Lagrange multiplier method. In [22] a scheme for proportional static slack distribution taking parallelism into account is presented. The method in [23] calculates the task ordering and voltage assignment for a platform with one scalable processor and multiple I/O devices. As these methods bring interesting energy improvements, however they do not utilize the dynamic slack conservatively.

Moreover, two design and run time methods for dealing with static slack exist. The method in [24] calculates off-line the Pareto operating points corresponding to task running on different resources. An energy-performance optimal point is chosen at runtime, depending on the tasks and resources in the system. Shin et al. [25] present a DVFS method for conditional task graphs. At design time an initial task ordering and stretching is determined, and further refined at run-time depending on the actual data arrival.

Furthermore, several approaches using work slack are proposed and deliver promising energy reductions. Ruggiero et al. [26] select the optimal number of (symmetric) processors and their VF points, to minimize system power under throughput constraints. Simunic et al. [27] propose a closed control-loop energy regulation that assumes exponential arrival times and offers globally optimal control. In [28] the power utilization of application's consisting of task graphs running on systems with rechargeable batteries is maximized. However all these methods are non-conservative.

The approaches closest to our work are [1], [3], [4], [5]. Maxiaguine et al.[1] propose a DVFS strategy in two phases: worst-case off-line bounding and improving these bounds at runtime, conservatively. The method is based on network calculus and can trade buffer sizes for energy savings, but it does not support resource sharing. The methods in [3] and [4] deal with acyclic task graphs. In [3] a new VF point is determined based on the amount of work left for each task, involving complex book-keeping. In [4] static and work slack is conservatively exploited for statically scheduled task graphs, hence task reordering is not possible at run-time. In [5] a conservative slack reclamation that deals with dependent tasks is proposed. However this method requires extra synchronization among the processors at each reclamation action. Moreover, preemption and task reordering are not supported, limiting the slack usage possibilities.

VII. Conclusions

In this paper, we investigated conservative energy management for real-time (RT) applications with multiple dependent tasks, mapped on multiple independent processors. Using dataflow models we classified different slack types, which can be used to reduce dynamic energy: static, work and share slack. For work and share slack we proposed SlackOS, a run-time, dependency-aware, task scheduler that conservatively scales down the processors' voltage and frequency, to reduce energy while meeting the application throughput constraints. Its implementation extended on an existing MPSoC platform with dependencies and energy-awareness features, and uses only information local to the processor to compute new processor operating points (enabling SlackOS's scalability with the number of processors).

When applied to an H.264 decoder SlackOS saves 22% and 33% energy in comparing to a non-energy-aware execution, for a task scheduler based on TDM and a TDM and static order, respectively. At a cost of missing deadlines, an execution with the processors constantly running at the minimum frequency would only achieve 37% energy savings, when compared to a non-energy-aware execution. Thus SlackOS delivers 60% and 94% energy reduction, while guaranteing zero deadline miss.

References

[1] A. Maxiaguine *et al.*, "DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs," in *CODES ISSS*, 2005.

[2] V. Venkatachalam *et al.*, "Power reduction techniques for microprocessor systems," *ACM Computing Surveys*, 2005.

[3] R. Mishra *et al.*, "Energy aware scheduling for distributed real-time systems," in *IPDPS*, 2003.

[4] D. Roychowdhury *et al.*, "A voltage scheduling heuristic for real-time task graphs," in *DSN*, 2003.

[5] C. Shen *et al.*, "Resource reclaiming in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, 1993.

[6] M. Bekooij *et al.*, "Dataflow analysis for real-time embedded multiprocessor system design," in *Dynamic and Robust Streaming in and between Connected Consumer Electronic Devices*, 2005.

[7] O. Moreira *et al.*, "Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor," in *EMSOFT*, 2007.

[8] M. Bekooij *et al.*, "Predictable embedded multiprocessor system design," in *SCOPES*, 2004.

[9] J. Reineke *et al.*, "A Definition and Classification of Timing Anomalies," in *WCET*, 2006.

[10] R. Reiter, "Scheduling parallel computations," *Journal of the ACM*, 1968.

[11] A. Burchard *et al.*, "Complex SoC power characterization," in *NXP-R-TN 2007/00005*, 2007.

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

[13] P. de With *et al.*, "On the design of multimedia software and future system architectures," in *Embedded Processors for Multimedia and Communications*, 2004.

[14] K. Goossens *et al.*, "The thereal network on chip: Concepts, architectures, and implementations", *IEEE Design and Test of Computers*, 2005.

[15] M. Dales. SWARM - Software ARM. [Online]. Available: *http://www.cl.cam.ac.uk/ mwd24/phd/swarm.html*

[16] F. Gruian, "System-level design methods for low-energy architectures containing variable voltage processors," in *Workshop on Power-Aware Computer Systems*, 2001.

[17] B. Gorji-Ara *et al.*, "Fast and efficient voltage scheduling by evolutionary slack distribution," in *ASP-DAC*, 2004.

[18] A. Andrei *et al.*, "Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems," in *DATE*, 2004.

[19] H. Kimura *et al.*, "Emprical study on reducing energy of parallel programs using slack reclamation by DVFS in a power-scalable high performance cluster," *IEEE International Conference on Cluster Computing*, 2006.

[20] L. Yan *et al.*, "Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems," in *ICCAD*, 2003.

[21] A. Manzak *et al.*, "A low power scheduling scheme with resources operating at multiple voltages," *IEEE Transactions on VLSI Systems*, 2002.

[22] S. Shaoxiong *et al.*, "Power minimization techniques on distributed real-time systems by global and local slack management," in *ASP-DAC*, 2005.

[23] P. Rong *et al.*, "Power-aware scheduling and dynamic voltage setting for tasks running on a hard real-time system," in *ASP-DAC*, 2006.

[24] P. Yang *et al.*, "Energy-aware runtime scheduling for embedded-multiprocessor socs," *IEEE Design and Test of Computers*, 2001.

[25] D. Shin *et al.*, "Power-aware scheduling of conditional task graphs in real-time multiprocessor systems," in *ISLPED*, 2003.

[26] M. Ruggiero *et al.*, "Application-specific power-aware workload allocation for voltage scalable MPSoC platforms," in *ICCD*, 2005.

[27] T. Simunic *et al.*, "Managing power consumption in networks on chips," *IEEE Trans. VLSI Syst.*, 2004.

[28] J. Suh *et al.*, "Dynamic power management of multiprocessor systems," in *IPDPS*, 2002.