Methods for Power Optimization in Distributed Embedded Systems with Real-Time Requirements *

Razvan Racu, Arne Hamann, Rolf Ernst
IDA†
Technical University of Braunschweig
D-38106 Braunschweig, Germany
{racu,hamann,ernst}@ida.ing.tu-bs.de

Bren Mochocki, Xiaobo Sharon Hu
Department of CSE‡
University of Notre Dame
Notre Dame, IN 46556, USA
{bmochock,shu}@cse.nd.edu

ABSTRACT

Dynamic voltage scaling and sleep state control have been shown to be extremely effective in reducing energy consumption in CMOS circuits. Though plenty of research papers have studied the application of these techniques in real-time embedded system design through intelligent task and/or voltage scheduling, most of these results are limited to relatively simple real-time application models. In this paper, a comprehensive real-time application model including periodic, sporadic and bursty tasks as well as distributed real-time constraints such as end-to-end delays is considered. Two methods are presented for reducing energy consumption while satisfying complex real-time constraints for this model. Experimental results show that the methods achieve significant energy savings without violating any deadlines.

Categories and Subject Descriptors

C.3 [**Special-Purpose and application-based systems**]: Real-time and embedded systems; C.4 [**Performance of systems**]: Modeling techniques; Performance attributes; Reliability, availability, and serviceability

General Terms

Design, Performance, Reliability, Verification

Keywords

real-time systems, power optimization, sensitivity analysis, Dynamic Voltage Scaling, evolutionary algorithms, timing analysis, SymTA/S

*This work is supported in part by the ARTIST2 Network of Excellence and by Deutsche Forschungsgemeinschaft under grant number ER168/18-4

†Institute of Computer and Communication Network Engineering

‡Embedded System Design Group

CASES'06, October 23–25, 2006, Seoul, Korea.

1. INTRODUCTION

Besides meeting timing constraints, energy consumption has become a major consideration in real-time embedded system (RTES) design as VLSI technology continues its remarkable advances. This is partially due to the proliferation of mobile systems with limited power resources, e.g. portable communication devices. Even in an energy-rich platform, power consumption has raised serious concerns [3] with respect to reliability and cost.

Since most RTESes only require the peak performance occasionally, an extremely effective approach to save power and energy is to dynamically adjust the operating clock frequency or operating modes, provided that the real-time performance can be guaranteed. Two circuit-level power-reduction techniques have been widely employed at the system level to achieve this: *Dynamic Voltage Scaling (DVS)*, which enables systems to operate under dynamically varied supply voltages and hence frequency; and clock/power gating, which dynamically cuts off clock or supply voltage to regulate the operation mode of a system. We refer to the latter as Low-Leakage Mode (LLM). To fully exploit the benefits of DVS and LLM in RTESes is a complex issue and has attracted the attention of researchers from both the real-time system community and the design automation community.

In this paper, we consider RTESes consisting of a set of applications mapped on a heterogeneous distributed system architecture. The applications may have distributed real-time constraints, like end-to-end deadlines. Different hardware components may employ individually controllable power management mechanisms. The objective is to minimize energy consumption of such systems while satisfying real-time constraints. Two distinguishing features of this work include comprehensive real-time application modeling and close interaction with timing analysis tools. We propose two approaches: one leverages the sensitivity analysis taking place in timing validation for energy reduction, while the other uses genetic algorithms to find Pareto-optimal system configurations satisfying both timing and power requirements.

The paper is organized as follows: in Section 2.1 we review the existing work on system level formal timing analysis. Section 2.2 shows the state-of-the-art in low power embedded system design. Section 3 introduces the notations and the model used for system specification. In Section 4 we present two methods that combine system timing analysis and power analysis to optimize the energy requirements in embedded systems with real-time and power constraints. The first approach uses the results of the sensitivity analysis of different system timing properties. The second approach is based on evolutionary search techniques. The two approaches are compared in Section 5 using a mobile gaming platform example.

2. ANALYSIS OF SYSTEMS WITH TIMING AND POWER CONSTRAINTS

2.1 Methods for formal timing analysis

System performance validation is key during the design of state-of-the-art real-time distributed systems. System simulation represents the analysis methodology that has been adopted for a very long time in practice. However, as a current trend, the single-processor architectures are replaced by large heterogeneous platforms based on multi-processor systems on chip (MpSoC). In spite of high applicability and diversity at the resource level, performance simulation faces complex adaptability issues when moving to the system level. One is due to weak corner-case coverage that drastically reduces the performance guarantees. The large runtime complexity and the impossibility to be applied at early design phases are other main draw-backs of system simulation.

As an alternative, formal performance analysis has got in the past years a lot of attention from the real-time community. The larger scalability of the formal techniques compared with the simulation approaches allows the system designer to quickly derive the system performance estimates. Event though it can cover all system corner-cases, formal analysis methods may suffer from accuracy issues, i.e. overestimated results. A lot of work has been already done or is currently in progress to capture the timing dependencies between the system components and to increase the result accuracy.

Since in this paper we use a formal timing analysis model for system performance verification, the rest of this section shortly reviews the main work done in this area.

The *holistic analysis approach* developed by Tindell [28] systematically extended the classical local analysis techniques, considering the scheduling influences along functional paths in the system. He proposed a performance verification model for distributed real-time systems with preemptive task sets communicating via message passing and shared data areas. Eles *et al.* [19] extended this approach for systems consisting of fixed-priority scheduled CPUs connected via a TDMA scheduled bus. Later on, Palencia *et al.* [18] extended the analysis for tasks with precedence relations and activation offsets.

Gresser [7] and Thiele [27] established a different view on scheduling analysis. The individual components or subsystems are seen as entities which interact, or communicate, via event streams. Mathematically speaking, the stream representations are used to capture the dependencies between the equations (or equations sets) that describe the individual components timing. The difference to the holistic approach (that also captures the timing using system-level equations) is that the compositional models are well-structured with respect to the architecture. This is considered a key benefit, since the structuring significantly helps designers to understand the complex dependencies in the system, and it enables a surprisingly simple solution. In the *compositional approach*, an output event stream of one component turns into an input event stream of a connected component. Schedulability analysis, then, can be seen as a flow-analysis problem for event streams that, in principle, can be solved iteratively using event stream propagation.

Richter *et al.* proposed a compositional analysis model based on event model interfaces [22]. The approach combines local scheduling techniques and event model propagation into a system-level analysis. The interfacing between the local components is realized using a set of comprehensible standard event models. Jersak [14] extended this approach for applications with complex task dependencies, including multiple activating inputs and cyclic task graphs. This method is used in SymTA/S analysis framework [11].

2.2 Methods for power optimization

Power consumption in a CMOS circuit includes two components: dynamic and static power. Dynamic power consumption consists of the switching power for charging and discharging the load capacitance, and the short-circuit power due to the non-zero rising and falling time of the input and output signals, while static power consumption is primarily due to leakage currents. The overall power consumption of a system can be expressed as

$$P_{total} = P_{dyn} + P_{leak} = \alpha C_L V^2 f + I_{sc} V + I_{leak} V, \quad (1)$$

where α is the switching activity, C_L is the load capacitance, V is the supply voltage, f is the system clock frequency, I_{sc} is the short-circuit current, and I_{leak} is the leakage current which consists of both the subthreshold leakage current and the reverse bias junction current in the CMOS circuit.

When the processor is in the active mode, lowering supply voltage (V in (1)) is one of the most effective ways to reduce dynamic power/energy consumption. Scaling supply voltage has a negative impact on the speed (i.e., frequency) of a circuit which is captured by the following expression:

$$f = \frac{(V - V_{th})^a}{k \cdot V} \quad (2)$$

where k is a device related parameter, V_{th} is the threshold voltage, and a is a technology dependent parameter ranging from 1.2 to 2. Therefore, scaling voltage is equivalent to executing a task at a lower frequency, i.e., prolong the task execution time. With voltage scaling, different tasks can be executed at different frequencies (i.e., task level) or different components can operate at different frequencies (i.e., resource level).

An effective technique to reduce the leakage power consumption is to switch a processor into the low-leakage mode when it is idle and switch it back to the normal execution mode when execution is required. Various circuit level techniques have been proposed related to LLM, e.g., power gating [20], input vector control [15], and body biasing [6]. It is important to note that mode changes could incur considerable energy and timing overhead. The work presented in this paper focuses on DVS techniques but the approach can be extended to LLM as well since LLM can be considered as a special case of DVS where speed of 0 is also a possible value.

System-level techniques play a major role for maximally harvesting the benefit provided by DVS. A number of researchers have studied simpler versions of the problem that we are interested in. Some do not consider real-time constraints (e.g., [12]) while others investigate only independent real-time tasks executed on multiple processors [5, 31]. Most research on systems with dependent tasks utilizes acyclic task graphs. For this task model, globally optimal solutions based on either nonlinear mathematical programming (for the continuous voltage case) or integer nonlinear mathematical programming (for the discrete voltage case) have been presented (e.g. [1, 30]). Though the solutions are quite comprehensive, the inherently high computational cost would deem such approaches not applicable to larger systems. A number of heuristic approaches have also been proposed to tackle similar problems. The work in [8, 29, 23, 31] solves the energy minimization problem for dependent tasks on multiple DVS processors. However, the acyclic task graph model can be rather inefficient if the periods of the tasks are not harmonic or when tasks are bursty. The task model under consideration of this paper is much more comprehensive and practical.

3. SYSTEM SPECIFICATION MODEL AND NOTATIONS

For the system specification, we use the SymTA/S model [11]. An application is modeled by a set of computation and communication tasks (application entities). The tasks are mapped to and executed on a set of processing and communication elements, representing the system architecture. Each task is characterized by its execution time interval, defined as the minimum and maximum time the task requires for a complete execution on the corresponding resource, assuming that no blocking occurs during execution. A task is activated due to an activating event. All activating events of a task are captured by an event stream. The behavior of the event streams is described using event models. The task may be periodically or sporadically activated, and additionally each of these models may be with jitter or with burst. The event models are characterized by a set of timing parameters like period, jitter or minimum inter-arrival distance. Any timing dependency between the activations of two tasks is represented by an event stream emerging from one task and entering the dependent task. The real-time behavior is described by a set of timing constraints, like task deadlines, end-to-end deadlines, and maximum jitter at system output.

When multiple tasks share the same resource, two or more tasks may request the resource at the same time. To arbitrate request conflicts, each resource is associated with a scheduler which selects a task for execution out of the set of active tasks according to some scheduling policy. The scheduling analysis calculates worst-case and best-case task response times, i.e. the maximum and minimum times between task activation and task completion taking into account the effects of scheduling. One may check the feasibility of the system by comparing the results of the scheduling analysis against the set of timing constraints. We say that a system is feasible if all constraints are fulfilled. Vice-versa, a system is infeasible if at least one constraint is violated. Note that the SymTA/S model is much more general than both periodic/aperiodic task models and task dependency graph models [11].

In terms of system architecture, we consider heterogeneous systems on a chip (SoC). Each processing element may have independently controllable power management mechanisms such as DVS. Voltage-island based design investigated by various researchers, for example [13, 16, 26], can readily implement the above architecture. With the continuous scaling of CMOS technology, this trend will gain even bigger momentum. We consider both the scenarios where power management decisions are made at the task level and at the resource level. At the task level, each task may be executed with its own speed/voltage, which is equivalent to DVS. At the resource level, each processing element may be assigned a unique speed/voltage and will not change during context switches, which is referred to as *multiple voltage* approach (MV). The voltage/frequency levels can be either continuous or discrete. The problem of interests is to minimize the energy consumption of a given system while satisfying real-time constraints.

4. POWER OPTIMIZATION UNDER TIMING CONSTRAINTS

In this section we present two methods for the power optimization of embedded distributed systems with real-time constraints. Section 4.1 presents a heuristic algorithm for power optimization based on the results obtained applying sensitivity analysis for the system parameters influencing the global energy demand of the system. Section 4.2 describes a stochastic power optimization approach based on evolutionary algorithms.

4.1 Sensitivity-based approach

As already mentioned in Section 2.2, the most effective way to reduce system power/energy is to switch the processor to a lower voltage level. From Equation 2 it can be observed that the processor clock frequency is proportional to the processor voltage level. However, a lower clock frequency will directly affect the timing characteristics of all components executed on that resource. By reducing the clock frequency of a processor by a factor sf leads to an increase of the execution times of the tasks mapped on that resource by a factor that can be conservatively approximated by $1/\mathsf{sf}$. Therefore, in systems with real-time requirements, modifications of the resource voltage levels must be performed such that no timing constraint is violated.

In this section we describe a heuristic algorithm for the optimization of system power/energy based on sensitivity analysis. Sensitivity refers to the robustness of the system timing properties with respect to changes, i.e., how far a particular parameter can be changed without affecting the feasibility of the system. The robustness of a parameter is characterized by the available slack obtained for that parameter, where the slack is computed using the sensitivity analysis framework presented in Section 4.1.1.

It is not difficult to see that the robustness of timing properties can be used to help decide how much a task execution can be slowed down before the system becomes infeasible. In the rest of this section, we describe our sensitivity analysis based power minimization algorithms applied at task level using DVS, or at resource level using the minimum supply voltage for each resource permitted by the set of constraints. Section 4.1.2 presents the overall approach of our optimization algorithms.

4.1.1 Sensitivity analysis framework

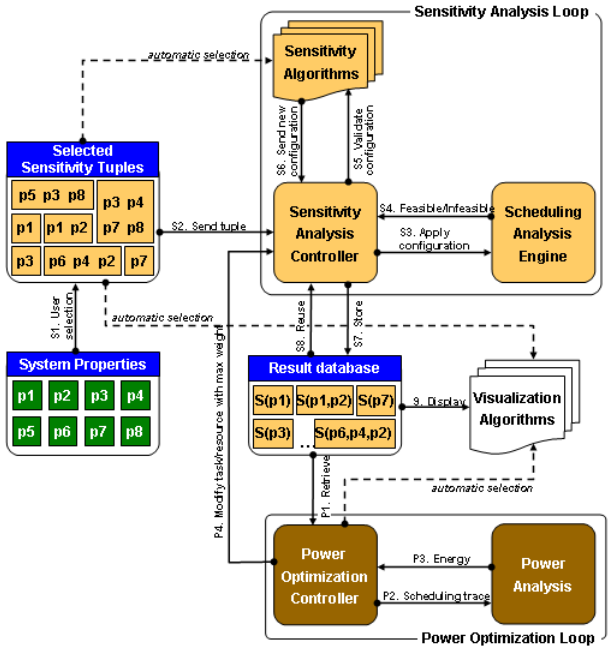


Figure 1: Sensitivity analysis framework

The diagram presented in Figure 1 shows the components of the sensitivity analysis framework and the corresponding analysis flow. The user selects (Step S1) from the pool of system properties, a set of *sensitivity tuples* on which the sensitivity analysis is performed. A sensitivity tuple, in this context, represents a data object contain-

ing a system property and a sensitivity objective defined for this property. The system properties used by the power optimization algorithm are the execution times of the tasks on which DVS can be applied, or the speed of the resources allowing MV. The sensitivity objective is the minimization of the scaling factors of the selected properties.

The sensitivity tuples are sent one by one to the sensitivity analysis controller (Step S2). The *sensitivity analysis controller* is responsible for directing the flow of information between modules. The *sensitivity analysis loop* contains the sensitivity analysis controller, the scheduling analysis engine and the sensitivity algorithms. At first iteration, the sensitivity analysis controller transmits (Step S3) the initial value of the selected system property to the scheduling analysis engine. It receives back (Step S4) the status of the analyzed system: *feasible* or *infeasible*. A detailed presentation of the compositional performance analysis implemented by the *scheduling analysis engine* is given in [11]. Based on status, the search space is defined and sent (Step S5) to the analysis algorithm. The algorithm selects a value in the search space and sends it back to the analysis controller (Step S6). The steps S3 to S6 are repeatedly carried out until the algorithm finds the desired value. Note that, the search space is defined in the first iteration by the sensitivity controller and adapted during each iteration by the analysis algorithm. When the analysis completes, the result is stored (Step S7) in the *result database*. Before first analysis iteration or sometimes during iterations, the sensitivity analysis controller might access (Step S8) the result database to reuse already existing information.

The scheduling analysis is performed using the SymTA/S analysis engine presented in [11]. The communication between the scheduling analysis engine and the sensitivity analysis controller is implemented using a client/server interface. The communication between the sensitivity analysis controller and the other modules is realized via function calls.

Once the selected properties are analyzed, the *power optimization controller* performs the weighting of the computed slack values. For each analyzed tuple (Step P1), a new system configuration is created, assigning the value determined using the above algorithm to the corresponding system property. This configuration is sent to the *power analysis* engine (Step P2). The power analysis computes, based on scheduling traces, the average power of the new configuration and sends it back to the power optimization controller. Based on the power value and the available feasibility slack, a weight is calculated for each system property (see Section 4.1.3 and 4.1.4). The property with the maximum weight is selected (Step P4) and the system configuration is modified accordingly. The entire algorithm is repeated until no slack left in the system.

The communication between the power optimization controller and the power analysis is realized using a client/server interface. The communication between the power optimization controller the other modules is realized by function calls.

The results are displayed (Step S9) by demand using visualization algorithms specific to each sensitivity tuple. The sensitivity, visualization and power optimization algorithms are automatically selected, depending on the user choices.

4.1.2 Heuristic power optimization

Algorithm 1 gives the overall approach to minimizing power/energy by a set of components that can be either tasks if task-level optimization is selected, or resources when the optimization is applied at resource level.

When applied at task level, the algorithm determines the minimum scaling factor of the execution time for each individual task. If applied at resource level, the algorithm computes the minimum scaling factor of the speed of each resource. The algorithms to determine the minimum scaling factors are presented in details in sections 4.1.3 and 4.1.4. After the scaling factors are computed, a weight is calculated for each component. The component weights are also presented in detail in the next subsections. The timing parameters of the component with the maximum weight are modified according to the minimum scaling factor computed for that component. In the next iterations, the new scaling factors of all components are recalculated and again modifications are applied for the components with the maximum weights. The iteration terminates when no more slack is available in the system, i.e., when all scaling factors have been minimized and all components have *zero* weights.

Algorithm 1 Minimize power on selected components

INPUT: the set of target components: $\mathcal{C}$;

OUTPUT: optimized power: P_{min};
the set of components with modified scaling factors: $\mathcal{S}$;

```
1: slackLeft = true;
2: while slackLeft do
3:    for all c_i ∈ C do
4:       compute minimum scaling factor of c_i: sf_opt^{c_i};
5:       compute weight of c_i: weight_{c_i};
6:    find c_w such that weight_{c_w} = max(weight_{c_i}), (∀)i such
      that weight_{c_i} ≠ 0;
7:    if c_w ≠ null then
8:       set sf_opt^{c_w};
9:       add (c_w, sf_opt^{c_w}) to S;
10:      slackLeft = true;
11:   else
12:      slackLeft = false;
13: compute power: P_min;
```

4.1.3 Task level

The minimum scaling factor of the execution time of a task (Algorithm 2) is determined using a binary search technique similar to those described in [21]. The binary search is performed on the interval bounded by the initial execution-time scaling factor, sf_{init} (usually 1), and the scaling factor that determines a maximum resource utilization equal to a value U_{max}. U_{max} is implicitly set to 100% to avoid resource overload, or it can be defined by system designer to specify an upper bound for the resource utilization.

The initial resource utilization, U_{CPU} can be determined using the following equation:

$$U_{CPU} = \frac{C_x}{\mathcal{P}_x} + \sum_{i=1, i\neq x}^{n} \frac{C_i}{\mathcal{P}_i}, \tag{3}$$

where C_i and $\mathcal{P}_i$ are the execution time and the activation period of task τ_i.

Scaling the resource frequency by factor sf during the execution of task τ_x leads to a new execution time of τ_x equal to $\frac{1}{\mathsf{sf}} \cdot C_x$. The maximum resource utilization U_{max} occurs when the execution time of τ_x equal to $C_{x,max}$:

$$U_{max} = \frac{C_{x,max}}{\mathcal{P}_x} + \sum_{i=1, i\neq x}^{n} \frac{C_i}{\mathcal{P}_i} \tag{4}$$

$C_{x,max}$ can be obtained by scaling the execution time of τ_x by a factor sf_{lowest}:

$$C_{x,max} = \frac{C_x}{\mathsf{sf}_{lowest}} \Rightarrow \mathsf{sf}_{lowest} = \frac{C_x}{C_{x,max}} \quad (5)$$

Replacing Equation 5 in Equation 4 we obtain:

$$U_{max} = \frac{C_x}{\mathcal{P}_x \cdot \mathsf{sf}_{lowest}} + \sum_{i=1, i\neq x}^{n} \frac{C_i}{\mathcal{P}_i} \quad (6)$$

If we denote by U_{τ_x} the load determined by task τ_x ($C_x/\mathcal{P}_x$) on CPU, from equations (3) and (6) it results that

$$\mathsf{sf}_{lowest} = \frac{U_{\tau_x}}{U_{max} - U_{CPU} + U_{\tau_x}} \quad (7)$$

Notice that the value returned by the binary search technique, sf_{opt}, is smaller than the initial scaling factor value, leading to a decrease in the CPU clock frequency during the execution of task τ_x, and therefore to a lower power dissipation.

If the initial system configuration is not feasible, the search interval is determined by the initial scaling factor sf_{init}, and the scaling factor for which U_{τ_x} is equal to a user-defined value U_{min}. U_{min} is defined by system designer and can be determined from the highest possible clock frequency available for that resource.

$$U_{min} = U_{\tau_x,min} = \frac{1}{\mathsf{sf}_{highest}} \cdot \frac{C_x}{\mathcal{P}_x} \Rightarrow \mathsf{sf}_{highest} = \frac{U_{\tau_x}}{U_{min}} \quad (8)$$

In this case, the binary search algorithm is looking for the minimum value of the scaling factor leading to a feasible system configuration. If such a value is found, then it is larger than sf_{init}. This actually leads to a higher CPU clock frequency during the execution of task τ_x. The obtained *feasible* configuration might have a higher power dissipation than the original *infeasible* configuration.

The system configurations corresponding to the values selected by the binary search algorithm are validated (analyze_system) by the *scheduling analysis engine* presented in Section 4.1.1.

The task weights in Algorithm 1 are calculated according to the following equation:

$$weight_{\tau_x} = (\mathsf{sf}_{init} - \mathsf{sf}_{opt}) \cdot U_{\tau_x} \cdot P_{CPU}, \quad (9)$$

where $(\mathsf{sf}_{init} - \mathsf{sf}_{opt})$ represents the available scaling factor slack, U_{τ_x} which is the load determined by task τ_x on CPU, and P_{CPU} is the power (energy per unit time) of CPU. The task weight linearly increases with the scaling factor slack. The larger the execution time scaling factor slack, the lower the supply voltage of the resource during the execution of the task. The weight linearly depends also on the task load, i.e. DVS is applied first on those tasks using the resource most of the time. Additionally, the tasks mapped on the resources with the highest power dissipation have priority for optimization. In case that the original system configuration is infeasible, the obtained scaling factor slacks will be negative, and the task with the maximum weight is the one whose computed scaling factor is the smallest. The algorithm determines the minimum additional power demand required by a feasible system configuration.

4.1.4 Resource level

Algorithm 3 determines the minimum scaling factor of a resource speed permitted by the set of constraints. Similar to task execution time scaling factor, a binary search technique is used. The search is again performed in two directions, depending on the initial system status: if the system is feasible, the search interval is determined by the initial speed factor sf_{init} (usually 1) and the value that leads to a maximum resource utilization equal to U_{max}. When the system is infeasible, the search interval is determined by the initial speed scaling factor sf_{init} and a scaling factor leading to a minimum resource utilization equal to U_{min}. U_{min} and U_{max} have been previously defined in Section 4.1.3.

Algorithm 2 Execution time minimum scaling factor

INPUT: initial scaling factor: $\mathsf{sf}_{init} = 1$;
initial resource load: U_{CPU};
task execution time: C_x;
task activation period: $\mathcal{P}_x$;
maximum load: U_{max};
minimum load: U_{min};
algorithm precision: ϵ;
OUTPUT: minimum scaling factor: sf_{opt};

```
1: U_τx = C_x/P_x;
2: sf_lowest = U_τx/(U_max − U_CPU + U_τx);
3: sf_highest = U_τx/U_min;
4: analyze_system;
5: if (system_feasible) then
6:    low = sf_init;
7:    high = sf_lowest;
8: else
9:    low = sf_highest;
10:   high = sf_init;
11: repeat
12:   middle = (high + low)/2;
13:   set sf = middle;
14:   analyze_system;
15:   if (system_feasible) then
16:      low = middle;
17:   else
18:      high = middle;
19: until (high − low) ≤ ε
20: set sf = sf_init;
21: sf_opt = low;
```

The initial resource utilization can be calculated using the following equation:

$$U_{CPU} = \sum_{i=1}^{n} \frac{C_i}{\mathcal{P}_i}, \quad (10)$$

When scaling the resource speed by factor sf, the execution times of all tasks mapped on that resource are scaled by factor $1/\mathsf{sf}$. If the system is feasible, the lower limit of the binary search interval is determined by the scaling factor value leading to U_{max}. This value can be determined using the following equation:

$$U_{max} = \frac{1}{\mathsf{sf}_{lowest}} \cdot \sum_{i=1}^{n} \frac{C_i}{\mathcal{P}_i} \Rightarrow \mathsf{sf}_{lowest} = \frac{U_{CPU}}{U_{max}} \quad (11)$$

Similarly, if the initial system configuration is infeasible, the upper bound of the search interval is determined by the scaling factor value leading to U_{min}:

$$U_{min} = \frac{1}{\mathsf{sf}_{highest}} \cdot \sum_{i=1}^{n} \frac{C_i}{\mathcal{P}_i} \Rightarrow \mathsf{sf}_{highest} = \frac{U_{CPU}}{U_{min}} \quad (12)$$

Similar to execution time scaling factor algorithm, the analyze_system procedure validates the system configurations corresponding to the values selected by the binary search algorithm.

The resource weights introduced in Algorithm 1 are calculated using the following equation:

$$weight_{CPU} = (\mathsf{sf}_{init} - \mathsf{sf}_{opt}) \cdot P_{CPU}, \quad (13)$$

Algorithm 3 Resource speed minimum scaling factor

INPUT: current speed factor: $\mathsf{sf}_{init} = 1$;
current resource load: U_{CPU};
maximum load: U_{max};
minimum load: U_{min};
algorithm precision: ϵ;
OUTPUT: minimum speed allowed: sf_{opt};

1: $\mathsf{sf}_{lowest} = U_{CPU}/U_{max}$;
2: $\mathsf{sf}_{highest} = U_{CPU}/U_{min}$;
3: analyze_system;
4: **if** $(system\_feasible)$ **then**
5: $low = \mathsf{sf}_{init}$;
6: $high = \mathsf{sf}_{lowest}$;
7: **else**
8: $low = \mathsf{sf}_{highest}$;
9: $high = \mathsf{sf}_{init}$;
10: **repeat**
11: $middle = (high + low)/2$;
12: set $\mathsf{sf} = middle$;
13: analyze_system;
14: **if** $(system\_feasible)$ **then**
15: $high = middle$;
16: **else**
17: $low = middle$;
18: **until** $(high - low) \leq \epsilon$
19: set $\mathsf{sf} = \mathsf{sf}_{init}$;
20: $\mathsf{sf}_{opt} = low$;

where $(\mathsf{sf}_{init} - \mathsf{sf}_{opt})$ is the scaling factor slack and P_{CPU} is the resource power dissipation. The resources with the maximum speed slack and the highest power dissipation are likely to be scaled down first. In case the initial system configuration is infeasible, the heuristic determines the minimum additional power demand required by a feasible system configuration.

4.2 Stochastic approach

In this section we present a stochastic approach for power optimization with DVS. It is based on a previously published design space exploration framework [10], which uses multi-dimensional evolutionary search techniques [4, 32].

We first give a short description of the exploration framework and how it is used to perform the power optimization (Section 4.2.1). We then give details on several important aspects: search space encoding (Section 4.2.2), the creation of an initial population used as starting point for the optimization (Section 4.2.3), and the variation operators used to guide the exploration (Sections 4.2.4 and 4.2.5).

4.2.1 Exploration framework

Figure 2 shows the compositional design space exploration framework [10] used in this work. It implements the standard exploration loop common to exploration frameworks based on evolutionary algorithms.

One unique characteristic of the utilized exploration framework is its compositional organization of the search space. Different parameters of a system, such as priority assignments, scaling factors, etc., are encoded as separate *chromosomes*. For optimization the user selects a subset of all system parameters. The chromosomes for these parameters form an *individual* and are included in the evolutionary optimization while all others are fixed and immutable.

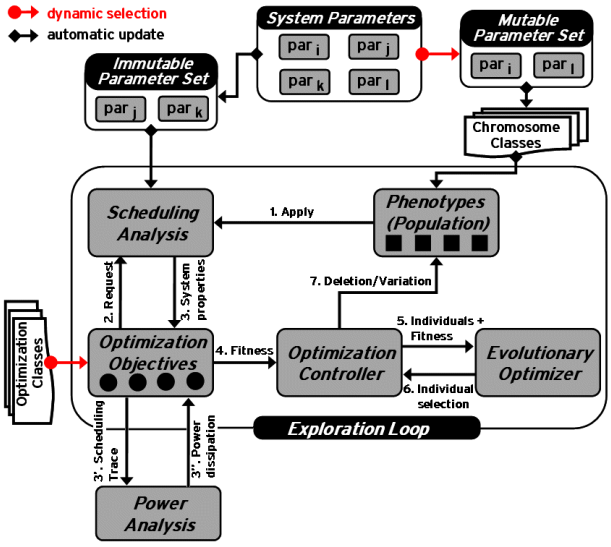


Figure 2: Exploration framework

Note that the variation operators of the evolutionary algorithm are applied chromosome-wise for individuals. More details on the exploration framework can be found in [10].

In this paper we explain in detail the DVS chromosome. Its algorithms (creation of the initial population, crossover, and mutation) are tailored for power minimization with DVS. Note that the DVS chromosome can be combined with previously developed chromosomes for priority optimization [10], TDMA time slot optimization [9], traffic shaping [10], etc.

During exploration the *scheduling analysis engine* analyzes the generated system configurations and determines properties subject to optimization (*optimization objectives*). In the case of power minimization addressed in this paper, scheduling traces are generated and sent to *power analysis*. Based on these traces power analysis calculates and returns the power dissipation.

Note that the utilized exploration framework can perform Pareto-optimization of multiple concurrent optimization objectives. Consequently, it is capable of determining trade-off curves (Pareto-fronts) between system power dissipation and other optimization objectives including timing, buffer sizes, etc.

4.2.2 Encoding

As previously explained the voltage levels assigned to tasks included into DVS are modeled by scaling factors for their core execution time intervals in this paper. Correspondingly, processor clock-rates are modeled by scaling factors. In both cases, a scaling factor of 1 corresponds to a nominal core execution time or clock-rate, respectively. In the following we refer to the set of tasks and processors included power optimization as *components*.

The scaling factors are directly encoded as vector containing one real number entry for each considered component. In the following we call such a vector of scaling factors an *individual*. Additionally, each individual is annotated with a boolean flag *working* indicating whether it represents a feasible system configuration or not.

The *working* flag is left unassigned after the creation of an individual in step 7 of the exploration loop. However, it is assigned once the individual is analyzed by the scheduling analysis engine, and thus before it can be used as parent individual for the generation of offsprings.

4.2.3 *Initial population*

Algorithm 4 describes the creation of the initial population. It creates α individuals, each with random initial scaling factors for the components included into power optimization. Thereby, the upper bound for the generated scaling factors is bound_{sf}.

The parameter k permits control over the creation of the initial population. Figure 3 visualizes the effect of k on the scaling factors generated during the creation of the initial population for $\text{bound}_{sf} = 1.5$.

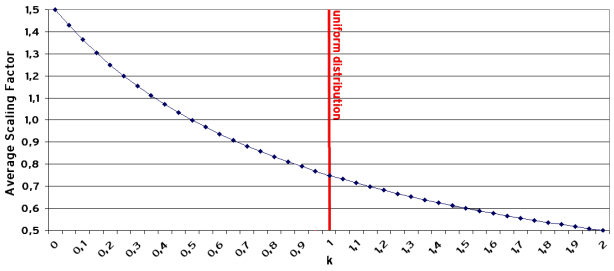


Figure 3: Average generated scaling factors for $\text{bound}_{sf} = 1.5$ as a function of k

We observe, that $k = 1$ leads to a uniform distribution, and that decreasing and increasing values for k increase or decrease the average value of the generated scaling factors, respectively.

A good choice for k can significantly increase the validity of the initial population, and thus decreases the time for the exploration to find good scaling factor assignments. For instance, if there is little slack in the system, low values for k enforce the generation of high scaling factors, and thus increase the probability that feasible system configurations are generated.

Algorithm 4 Initial Population

INPUT: Set of components included into DVS: $\mathcal{C} = \{c_1, \ldots, c_n\}$
Maximum bound for scaling factors bound_{sf}
Number $k > 0$
Initial population size α
OUTPUT: Initial population $\mathcal{P} = \{p_1, \ldots, p_\alpha\}$
1: **for** $(i = 1;\ i <= \alpha;\ i = i + 1)$ **do**
2: **for** $(j = 1;\ j <= n;\ j = j + 1)$ **do**
3: choose random $r \in]0, \mathsf{bound}_{sf}]$
4: $p_i.\mathsf{sf}_{c_j} = \frac{\mathsf{bound}_{sf}}{\mathsf{bound}_{sf}^{k}} \times r^k$

4.2.4 *Crossover operator*

The crossover operator described in algorithm 5 leads to the convergence of the obtained scaling factor assignments towards (locally) optimal solutions contained "between" individuals considered by the evolutionary algorithm. It takes as input two parent individuals A and B and generates one offspring C.

The offspring is created by randomly choosing scaling factors for the considered components in the intervals defined by the two parents (lines $9 - 11$). Thereby, k_{low} and k_{high} influence the generated scaling factors by shifting them towards the lower or the upper bound of these intervals, respectively. Note that choosing $k_{high} = k_{low} = 1$ leads to uniform distribution of the generated scaling factors.

The following rules are used to determine whether k_{low} or k_{high} is applied to the generated scaling factor for a specific component c_i (lines $5 - 8$):

Algorithm 5 Crossover Operator

INPUT: 2 parent configurations $A = \{\mathsf{sf}_{a_1}, \ldots, \mathsf{sf}_{a_n}\}$ and $B = \{\mathsf{sf}_{b_1}, \ldots, \mathsf{sf}_{b_n}\}$
$k_{high} : 0 < k_{high} \leq 1$
$k_{low} : 1 \leq k_{low}$
OUTPUT: offspring configuration $C = \{\mathsf{sf}_{c_1}, \ldots, \mathsf{sf}_{c_n}\}$
1: **for** $(i = 1;\ i <= n;\ i = i + 1)$ **do**
2: **if** $(\mathsf{sf}_{a_i} = \mathsf{sf}_{b_i})$ **then**
3: $\mathsf{sf}_{c_i} = \mathsf{sf}_{a_i}$
4: **else**
5: **if** $(A.\text{working} \wedge \mathsf{sf}_{a_i} < \mathsf{sf}_{b_i}) \vee (B.\text{working} \wedge \mathsf{sf}_{b_i} < \mathsf{sf}_{a_i})$ **then**
6: $k_{applied} = k_{low}$
7: **else**
8: $k_{applied} = k_{high}$
9: $\mathsf{difference} = \mid \mathsf{sf}_{a_i} - \mathsf{sf}_{b_i} \mid$
10: choose random numbers r and $s \in [0, \mathsf{difference}]$
11: $\mathsf{sf}_{c_i} = \min(\mathsf{sf}_{a_i}, \mathsf{sf}_{b_i}) + \frac{\mathsf{difference}}{\mathsf{difference}^{k_{applied}}} \times r^{k_{applied}}$

- k_{low}: if both parents correspond to feasible system configurations
- k_{high}: if both parents correspond to non-feasible system configurations
- k_{low}: if the parent corresponding to the feasible system configuration has a smaller scaling factor for c_i
- k_{high}: otherwise

4.2.5 *Mutation operator*

The crossover operator described in section 4.2.4 combines properties of two parent individuals to create new configuration alternatives. Of course, it is possible that the variety of the initial population is insufficient to find good solutions only by using the crossover operator. Additionally, the exploration may get stuck in a local optimum, without the possibility to reach globally better solutions.

Therefore, we introduce a mutation operator (algorithm 6), enabling the evolutionary algorithm to break out of local optima and to reach parts of the search space not yet considered. It takes as input one parent individuals A and generates one offsprings B.

The mutation algorithm increases and decreases scaling factors of randomly chosen components. Thereby, the overall variation percentage is bound by $\mathsf{budget}_{applied} \leq \mathsf{budget}_{max}$.

The variation of the scaling factors is guided to a certain extend (configurable through $\mathsf{prob}_{heuristic}$) by a heuristic strategy (lines $7 - 12$) decreasing or increasing the scaling factors of the considered components dependant on the parent configurations system status. For instance, if the parent configuration is feasible all components scaling factors are decreased trying to consume the remaining system slack for power optimization.

However, this heuristic strategy should be used in moderation, since it might be necessary to increase scaling factors of some components to allow further scaling factor decreases of different components in order to obtain optimal solutions. Therefore, $\mathsf{prob}_{heuristic}$ should not be chosen greater than 0.5.

Note that choosing $\mathsf{prob}_{heuristic} = 0$ leads (globally) to uniform distribution of $\mathsf{budget}_{applied}$ among the considered components.

Algorithm 6 Mutation Operator

INPUT: parent configurations $A = \{\mathsf{sf}_{a_1}, \ldots, \mathsf{sf}_{a_n}\}$
Maximum bound for scaling factors bound_{sf}
Integer budget_{max}
$\mathsf{prob}_{heuristic} : 0 \leq \mathsf{prob}_{heuristic} \leq 1$
OUTPUT: offspring configuration $B = \{\mathsf{sf}_{b_1}, \ldots, \mathsf{sf}_{b_n}\}$

```
choose random integer budget_applied ∈]0, budget_max]
while (budget_applied > 0) do
    choose random integer r ∈]0, min(budget_applied, 100)]
    budget_applied = budget_applied − r
    choose random integer i ∈ [1, n]
    choose random number h ∈ [0, 1[
    if (h < prob_heuristic) then
        if (A.working) then
            sf_bi = sf_ai × (1 − r/100)
        else
            sf_bi = min(sf_ai × (1 + r/100), bound_sf)
    else
        choose random boolean b
        if (b) then
            sf_bi = sf_ai × (1 − r/100)
        else
            sf_bi = min(sf_ai × (1 + r/100), bound_sf)
```

5. CASE STUDY

In this section an example system is presented and optimized using both the heuristic and stochastic techniques. The example system, illustrated in Figure 4, represents a cooperative mobile gaming platform with 2-way voice communication. Figure 4(a) shows the target architecture. There are essentially three processing elements, two ARM11 processor cores [2] operating at a peak frequency of 550 MHz and one GPU with two parallel pixel pipelines. Each processor has its own dedicated memory for independent processing. Data is shared via an on-chip bus, which is connected to off-chip components (such as the game pad and display) via a bus bridge. A separate bus connects the two ARMs to the wireless networking hardware. All three processing elements are assumed to be independently scalable, each with 32 voltage/frequency levels.

Figure 4(b) shows the example task set. The left portion of (b) shows the task dependencies via a set of directed acyclic graphs, while the right portion lists the name and [best, worst] case execution cycles of each sub-task. The double-ringed nodes are source/sink tasks, the single ringed nodes are execution tasks and the arrows represent communication tasks. Task one is representative of a racing 3D application from [17]. The task is activated 15 times a second by periodically receiving input from the user and position data from remote players. This information is used by the application to generate the new position of the local player each frame, as well as the raw triangles that make up the 3D scene. The local position is forwarded to the remote player, while the triangles are sent to the geometry stage for transformation and projection operations (i.e., matrix multiplication). From there, the transformed triangles are sent to the triangle setup stage where they are mapped to scan-lines (rows of pixels). The scan-lines are then sent to the render stage where the final color of each pixel is determined. Finally, the pixels are written to the frame buffer. Tasks 2 and 3 represent the encoding and decoding of VSELP voice traffic [24]. This allows the local and remote users to communicate during play.

Figure 4(c) shows how the tasks are allocated to the various processing and communication elements. The bandwidth requirements of each communication task are also displayed [25]. With this allocation at the maximum speed, the utilization of ARM1 is 68.79%, ARM2 is 49.73% and the GPU is 59.7%. The shared bus and communication bus are 20% and 2% utilized, respectively. From the utilization values, it is clear that some slack exists that can be converted into energy savings. However, from Figure 4(c) it is just as clear that the system is quite complex. The slack must be used without inducing significant delay or jitter, especially on the path from the user input to the frame buffer, as such a delay can significantly degrade the gaming experience. In the remainder of this section, the effectiveness of the sensitivity-based DVS algorithm on the example system will be compared to the stochastic approach.

To prevent excessive delay, an end-to-end deadline from τ_{11} to τ_{17} of *1 second* will be enforced.

Table 1 shows the results obtained using the task level heuristic algorithm described in Section 4.1.3. The initial average power of the system is $P_{init} = 3.51$ W. Table 1(a) shows the tasks on which DVS is applied and the corresponding clock frequency scaling factors. In Table 1(b) are presented the initial power values of each resource and the values obtained after optimization. The minimum system power is $P_{min} = 1.51$ W. The measured algorithm run-time is 91 seconds.

Table 2 presents the results determined by the resource level heuristic (Section 4.1.4). Table 2(a) shows the computed minimum scaling factors of the resource clock frequencies. The optimized resource power values are shown in Table 2(b). After optimization the average power in the system has been decreased from $P_{init} = 3.51$ W to $P_{min} = 1.65$ W. The algorithm run-time is 20 seconds.

Figures 5(a) and 5(b) compare the stochastic and the heuristic power optimization approaches with respect to quality of obtained results and runtime. Note that the power curve of the stochastic approach represents the best reached power dissipation after each generation. This curve was obtained by averaging the results of 100 exploration runs each considering 50 generations with 15 individuals.

The following parameters are defined for the algorithms used by the stochastic optimization:

- creation of the initial population: $k = 0.75$
- crossover operator: $k_{low} = 1.4$ and $k_{high} = 0.75$
- mutation operator: $\mathsf{budget}_{max} = 80$ and $\mathsf{prob}_{heuristic} = 0.5$

In the case of task level power optimization the stochastic approach finds in average a system configuration of comparable quality (1.51 W) to the configuration obtained by the heuristic approach after 10 generations. Afterwards, the stochastic approach finds better solutions in the average case. After 50 generation, for instance, the average power dissipation of the best found system configuration is equal to 1.43 W.

However, the better results of the stochastic approach are bought with higher runtime. With comparable runtime the stochastic approach reaches 1.54 W in the average case.

The situation for the resource level power optimization looks similar. After 4 generations the stochastic approach discovers in average a system configuration of comparable quality (1.65 W) to the configuration obtained by the heuristic approach. Afterwards, it finds better solution, reaching an average power dissipation of 1.47 W after 50 generations.

Again, in order to obtain the same quality of results as the heuristic approach, the runtime of the stochastic algorithm is higher. With comparable runtime the stochastic approach reaches 1.75 W in the average case.

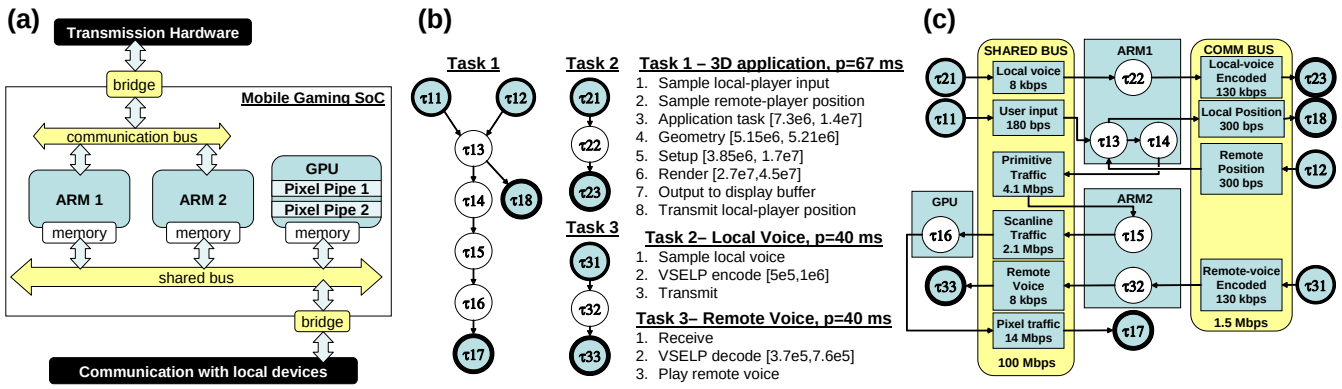


Figure 4: An example mobile gaming platform. (a) The system architecture. (b) The tasks with period p and core execution cycles [best case, worst case]. (c) The task-resource assignment with the associated communication traffic.

Table 1: Task level heuristic power optimization

(a)

Task	sf_{init}	sf_{opt}	*Mapping*
VSELP_ENCODE	1.0	0.48	ARM1
SETUP	1.0	0.75	ARM2
RENDER2	1.0	0.61	PIXEL_PIPE2
3D_APP	1.0	0.65	ARM1
VSELP_DECODE	1.0	0.1	ARM2
RENDER1	1.0	0.61	PIXEL_PIPE1

(b)

Resource	P_{init}	P_{min}
ARM1	0.65	0.43
ARM2	0.5	0.26
PIXEL_PIPE1	1.0	0.23
PIXEL_PIPE2	1.0	0.23
COMM_BUS	0.08	0.08
SHARED_BUS	0.28	0.28
Total power	3.51	1.51

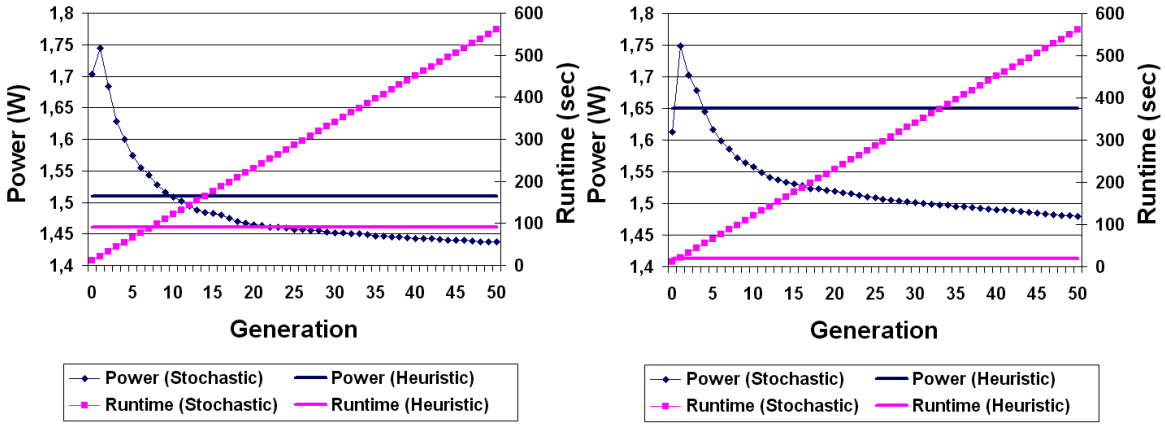
Table 2: Resource level heuristic power optimization

(a)

Resource	sf_{init}	sf_{opt}	*Mapping*
ARM2	1.0	0.64	[SETUP, VSELP_DECODE]
PIXEL_PIPE1	1.0	0.61	[RENDER1]
PIXEL_PIPE2	1.0	0.61	[RENDER2]

(b)

Resource	P_{init}	P_{min}
ARM1	0.65	0.65
ARM2	0.5	0.21
PIXEL_PIPE1	1.0	0.23
PIXEL_PIPE2	1.0	0.23
COMM_BUS	0.08	0.08
SHARED_BUS	0.28	0.25
Total power	3.51	1.65



(a) Task level power optimization (b) Resource level power optimization

Figure 5: Comparison of the heuristic and stochastic power optimization approaches

6. CONCLUSION

The previous approaches to system power optimization are limited by either simple task models and do not consider systems with timing constraints or are restricted to systems with local real-time requirements. In this paper we presented two methods for power optimization of complex embedded systems without violating any deadlines. As shown in the experimental section, the proposed methods can be easily applied to real-time systems with distributed functional and timing dependencies.

7. REFERENCES

[1] A. Andrei, M. Schmitz, P. Eles, Z. Peng, and B. M. Al-Hashimi. Simultaneous communication and processor voltage scaling for dynamic and leakage energy reduction in time- constrained systems. *ICCAD*, 2004.

[2] ARM11 MPCore. `http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html`.

[3] R. Bianchini and R. Rajamony. Power ans energy management for server systems. *IEEE Computers*, 37(11):68–75, November 2004.

[4] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA – a platform and programming language independent interface for search algorithms. `http://www.tik.ee.ethz.ch/pisa/`.

[5] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. *Int'l Conf. on Parallel Processing*, 2005.

[6] S. Duarte, Y. Tsai, N. Vijaykrishnan, and M. Irwin. Evaluating run-time techniques for leakage power reduction. *ASPDAC*, 2002.

[7] K. Gresser. An event model for deadline verification of hard real-time systems. In *Proceedings 5th Euromicro Workshop on Real-Time Systems*, pages 118–123, Oulu, Finland, 1993.

[8] F. Gruian and K. Kuchcinski. Lens: Task scheduling for low-energy systems using variable supply voltage processors. *Proc. Asia South Pacific Design Automation Conference*, pages 449–455, 2001.

[9] A. Hamann and R. Ernst. TDMA time slot and turn optimization with evolutionary search techniques. In *Proc. of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE)*, Munich, Germany, Mar. 2005.

[10] A. Hamann, M. Jersak, K. Richter, and R. Ernst. A framework for modular analysis and exploration of heterogeneous embedded systems. *Real-Time Systems Journal*, 33(1-3):101–137, July 2006.

[11] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, March 2005.

[12] J. Hu, Y. Shin, N. Dhanwada, and R. Marculescu. Architecting voltage islands in core-based system-on-a-chip designs. In *ISLPED*, 2004.

[13] W. Hwang. New trends in low power soc design technologies. *IEEE SoC Conf.*, 2003.

[14] M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, Technical University of Braunschweig, 2004.

[15] M. Johnson, D. Somasekhar, L. Chiou, and K. Roy. Leakage control with efficient use of transistor stacks in single threshold cmos. *IEEE Trans. on VLSI Systems*, 10, 2002.

[16] D. Lackey, P. Zuchowski, T. Bednar, D. Stout, S. Gould, and J. Cohn. Managing power and performance for system-on-chip designs using voltage islands. In *ICCAD*, 2002.

[17] B. Mochocki. Desktop2handheld: the porting of an opengl application to opengl/es. `http://www.nd.edu/~bmochock/race/ComputerGraphicsFinal_Mochocki.pdf`.

[18] J. C. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, 1998.

[19] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES)*, Estes Park, Colorado, USA, May 2002.

[20] M. Powell, S. Yang, B. Falsafi, K. Roy, and T. Vijaykumar. Gated-vdd: a circuit technique to reduce leakage in deep-submicron cache memories. In *ISLPED*, 2000.

[21] R. Racu, M. Jersak, and R. Ernst. Applying sensitivity analysis in real-time distributed systems. In *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, San Francisco, CA, USA, 2005.

[22] K. Richter. *Compositional Performance Analysis*. PhD thesis, Technical University of Braunschweig, 2004.

[23] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles. Energy-efficient mapping and scheduling for dvs enabled distributed embedded systems. *DATE*, pages 321–330, 2002.

[24] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2):20–30, March 2001.

[25] M. H. Sunwoo and S. Park. Real-time implementation of the vselp on a 16-bit dsp chip. *IEEE Transactions on Consumer Electronics*, 37(4):772–781, November 1991.

[26] E. Talpes and D. Marculescu. Toward a multiple clock/voltage island design style for power-aware processors. *IEEE Trans. on VLSI Systems*, 13, 2005.

[27] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, Geneva, Switzerland, 2000.

[28] K. Tindell and J. Clark. Holistic schedulability analysis for distributed real-time systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[29] L. Yan, J. Luo, and N. K. Jha. Combined dynamic voltage scaling and adaptive body biasing for heterogeneous distributed real-time embedded systems. *ICCAD*, pages 30–37, 2003.

[30] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. *Proceedings of the Design Automation Conference*, 2002.

[31] D. Zhou, N. AbouGhazaleh, D. Mosse, and R. Melhem. Power aware scheduling for and/or graphs in multi-processor real-time systems. *ICPP*, pages 593–601, 2002.

[32] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for multiobjective optimization. In *Proc. Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100, Barcelona, Spain, 2002.