Power-aware Scheduling for Embedded Systems under Min/Max Power and Timing Constraints *

Jinfeng Liu, Pai H. Chou, Nader Bagherzadeh
Dept. of Electrical & Computer Engineering
University of California, Irvine
Irvine, CA 92697-2625 USA
{jinfengl, chou, nader}@ece.uci.edu

Nazeeh Aranki, Nikzad "Benny" Toomarian
California Institute of Technology
Jet Propulsion Laboratory
4800 Oak Grove Dr., Pasadena, CA 91109 USA
{Nazeeh.I.Aranki, Nikzad.Toomarian}@jpl.nasa.gov

Abstract

Power-aware systems are those that must make the best use of available power. They subsume traditional low-power systems in that they must not only minimize power when the budget is low, but also deliver higher performance when required. Moreover, they must fully explore and integrate many novel power management techniques onto the whole system, not isolated components. Unfortunately, the power management techniques to date often cannot be incorporated into a unified framework. They are either over-specialized or fail to consider system-wide issues. This paper proposes a novel constraint-driven scheduling technique based on a graph-based model to integrate novel power management techniques and facilitate design-space exploration of power-aware embedded systems. Our application model captures min/max timing and min/max power constraints on computation and non-computation tasks through a new constraint classification and enables derivation of flexible system-level schedules. The scheduler computes a schedule that satisfies stringent min/max timing and max power constraints at all times. Furthermore, it also makes the best effort to satisfy min power constraint in an attempt to fully utilize free solar power or to control power jitter. Experimental results show that our automated technique yields designs that improve performance and reduce energy cost simultaneously compared to hand-crafted designs used in previous missions. This tool forms the basis of the IMPACCT system-level framework that will enable designers to aggressively explore many more power-performance trade-offs with confidence.

1 Introduction

Power management is becoming a central issue in embedded systems. It is particularly critical to systems that must carry their own energy source. As these systems are deployed in a diverse range of operating conditions, they must be able to adapt by shifting their power/performance curves.

This paper investigates key issues in power management for power-aware embedded systems. An example of such a system is the NASA Mars Pathfinder developed at JPL [1]. It draws power from a non-rechargeable battery pack and a solar panel to operate under extremely harsh weather and unsteady power conditions for hundreds of days. The power use is dominated by tasks in non-computational subsystems such as mechanical and thermal. These features pose several interesting problems that were not adequately addressed by previous work. First, such a system must be designed to be power-aware, rather than low-power. Second, it is critical that power management decisions be made at the system level, rather than only at the component level.

1.1 Power-aware vs. low-power

Today, many components and systems are designed to be *low-power*. However, we believe there is a critical difference with *power-aware* systems, which must make the best use of the available power by tracking the power availability from different sources and adapting to low-power or high-performance. They subsume low-power as a special case.

*This research was sponsored by DARPA PAC/C contract F33615-00-1-1719

In the Mars rover case, its designers constructed a low-power design. It incorporated some of the best low-power design techniques at all levels of abstraction. The rover has two power sources: a solar panel and a non-rechargeable battery. To strictly control power draw, its designers serialized all tasks, including driving, steering, obstacle detection, and heating motors. This low-power design allows the rover to operate for hundreds of days during daylight, and it sleeps at night. However, full serialization also means the rover moves as slowly as 10cm per minute, and it can only take a total of three high-resolution pictures and transmit them back to the base station per day.

A power-aware design can greatly improve the utility of the rover. We observe that the battery is non-rechargeable, and thus the free solar power would be wasted if not used while available. Since the solar power is not always available, the rover should be able to adjust itself to high-performance when the level of solar power is high, as well as adapt a low-power solution when free power is not available. In the existing design, the rover follows the same serial schedule regardless of the solar power level, and simply directs the excess energy to heating the wheels. An improved rover design should be able to outperform the existing low-power design (i.e., do more tasks in less time while saving battery energy) if it can take advantage of the free power with more parallelism in its schedule. We validate this prediction with experiments in the results section of this paper.

1.2 System-level power-aware design

We believe that power-aware designs must be done at the system-level, not just at the component level. Amdahl's law applies to power as well, not just performance. That is, the power saving of a given component must be scaled by its percentage contribution in an entire system. If a component draws only 2% of the power in a system, a 50% reduction in its power amounts to merely 1% saving to the system. Therefore, it is critical to identify where power is being consumed in the context of a system, not just the components in isolation.

In the case of the Mars rover, it turns out that some of the largest power consumers are not even in the digital computer, but rather they include the wheel motors, the steering motors, laser-guided obstacle detection, and the heaters. A successful power-aware design must consider these non-computation domains and coordinate their power usage as a whole system.

However, system-level power management poses more challenges to the designers, not only because of the increased complexity when considering many components as a whole system, but also due to the fact that many of the existing component-level power reduction technique yield counterintuitive results when applied to multiple resources and components. For example, reducing the voltage and frequency setting of a processor is a well-known technique to save CPU power. Unfortunately, it cannot be directly adapted to multiple resources, where tasks can be running in parallel. If one task is slowed down to save power, the time slots in its extra execution delay may overlap with some other tasks such that more power draw is resulted at the system level. Moreover, different components may need to coordinate with each other in their speed or modes. Therefore, a low-power decision in an isolated component may actually result in exceeding the power budget at the system level.

The designers in JPL were aware of this problem. Their solution was to serialize all tasks (not just computation, but also heating, turning, moving...) to avoid overlapping in power draw such that the existing low-power techniques can be applied to individual components while keeping the whole system low-power. This was the right decision in the past missions. However, hard-wiring the fully serial solution is over-constraining the design, because it does not exploit parallelism as a way to improve performance and power.

1.3 New application model and design tools

Power-aware systems pose new challenges for power management techniques in design space exploration for a variety of power/performance trade-offs. It is no longer sufficient to find just one good design point; instead, the same design must cover a range of behaviors depending on the operating conditions such as variable power and performance constraints. Further, a system-level power manager must integrate many novel power management

techniques across different domains (both computational and non-computational) into the same design framework, although these techniques are normally incompatible with each other.

To address these issues, our approach is to support power-aware design with a system-level design tool. One of the lessons learned from the Mars rover was that, without a tool, the designer had no choice but to embed many power-management decisions in the implementation. As a result, they were forced to design conservatively and could not consider more than one or two design alternatives. The purpose of our tool is to enable the exploration of many more points in the design space, so that additional knowledge about the mission can be incorporated to refine the design without requiring dramatic redesign.

In this paper, we propose a new system-level model that captures the essential power/performance features across both computational and non-computational domains in a concise, rigorous form. We present a new constraint classification to enable our constraint-driven power manager to meet both min/max timing and min/max power constraints on the entire system. The basis of our tool-based methodology is to capture different timing relationships and power characteristics in our new constraint model that effectively exposes the potential design alternatives to assist power management decisions.

The work presented in this paper represents one of the core tools in a larger design framework, called IMPACCT. The designers input a high-level behavioral specification of the design in terms of communicating processes and constraints. These processes have been assigned to run on specific execution resources. The scheduling tool in this paper constructs a constraint graph and performs power-aware scheduling. The output is then fed to another tool that performs optimizations and synthesis of power managers at the architectural level.

This paper is organized as follows. Section 2 reviews related work, and Section 3 describes the application example in more detail. We present the problem formulation in Section 4 and graph-based scheduling algorithms in Section 5. Then, we discuss experimental results in Section 6 followed by our concluding remarks and future work.

2 Related Work

Prior works have addressed minimization of power usage at the system level. Their common goal is to minimize power usage while maintaining a satisfactory level of performance or meeting real-time constraints. However, these low-power techniques often cannot be directly adapted in power-aware systems.

2.1 Subsystem shutdown

Shutting down idle subsystems such as network interfaces, hard disks, and displays can save a significant amount of power in a system. The shutdown decision can be based on fixed idle times of individual subsystems, although such approaches are less than satisfactory. Proposed improvements either attempt to make the timeout adaptive to the actual usage pattern, or use profiling to predict the proper time to shutdown and power up subsystems [45, 9, 46].

While it is important to manage the power of subsystems, unfortunately these techniques have several limitations. First, they do not handle timing or power as *constraints*. Instead, they treat time and power as costs or penalty to minimize. Second, their power-awareness is limited in the sense that they do not distinguish between free power (such as solar sources) vs. expensive power (non-rechargeable battery). These policies are oblivious to the current state of the energy source or their discharge characteristics, which are becoming an increasingly important consideration in power management.

2.2 Low-power scheduling by dynamic voltage scaling

Many real-time scheduling techniques have been proposed to date, but only recently have researchers started to address power issues with the objective of minimizing power usage. For example, rate-monotonic scheduling has been extended to scheduling variable-voltage processors. The idea is to save power by slowing down the processor just enough to meet the deadlines.

Dynamic voltage scaling (DVS) was first proposed by Weiser et al. in [48], and the aspect of energy minimization was analyzed by Yao et al. and the optimal off-line schedule is given [50]. Hong, Qu et al. presented a design

methodology to minimize energy in embedded systems based on variable-voltage processors [13, 13, 38]. Hong et al. also proposed an off-line scheduling heuristic for non-preemptive systems in [11, 10], and an on-line algorithm for mixed workload of both sporadic and periodic tasks in [12]. Their energy minimization techniques can guarantee deadlines of periodic tasks while making its best effort to meet the deadlines of sporadic tasks. Ishihara and Yasuura analyzed the optimal schedule for a processor that can operate in several discrete voltages in [14] and proposed an integer linear programming solution. Okuma, Ishihara and Yasuura proposed a DVS scheme [27, 28] that always guarantees deadlines for all tasks, although the energy may not be optimal. Shin et al. presented a run-time checking mechanism that can shut down the processor or adjust the processor speed [43] and an algorithm to minimize energy for periodic tasks [44]. Quan and Hu improved this technique by finding an optimal schedule for both periodic and sporadic tasks [41]. Luo and Jha incorporated the system cost into account and proposed a DVS scheme that also reduces cost [21]. Other techniques in recent literature can be seen in [44, 18, 17, 16, 15, 2, 3, 47, 42].

Such techniques have several limitations. First, they are CPU schedulers that minimize CPU power, rather than power managers that control subsystems and task executions. Second, the idea of slowing down the processor speed cannot be applied to managing the power of a system consisting of components in both computational and non-computational domains. The energy reduction is based on the fact that by decreasing the supply voltage, the execution delay grows linearly and the power consumption drops quadratically. This is generally not true for other execution resources except CMOS circuit. Although these schedulers meet deadlines, they do not consider max power as a constraint. When tasks can run in parallel, even if individually they are well within the max power, together as a system they may easily exceed it in low supply-power situations, causing the system to malfunction.

2.3 Scheduling for instruction-level energy minimization

Some recent studies use profiling to estimate the energy consumption of individual instructions. Such techniques are extended to instruction scheduling in an effort to minimize the energy consumption of program execution on a processor. Parikh et al. presented a scheduling scheme [30, 29] based on profiling information that captures not only energy consumption for individual instructions, but also the inter-instruction energy consumption between two consecutive instructions. Therefore, the scheduler can choose an optimal sequence to execute the instruction stream such that the energy can be minimized.

Although such technique to date can only handle dispatch sequence in instruction-level on one processor, it could be extended to handle processors with deeper pipelines and multiple execution units, although it will be difficult to model the energy cost by profiling in more sophisticated processors. Further, it can be extended to multiple execution resources if the cost model is accurate enough. Moreover, the current technique does not handle either power or timing constraints.

2.4 Power management in communication networks

A few research works have addressed the issues in energy reduction in communication networks with quality-of-service (QoS) requirements. Qu and Potkonjak proposed a voltage-scaling scheme for system-level communication pipelines [38, 39, 40], it can reduce energy consumption compared to shutting down idle component while satisfying all QoS requirement. Qiu, Wu and Pedram proposed generalized-stochastic-Petri-nets (GSPN) as a model for communication systems [35, 34, 36, 37]. They derived power consumption from a set of QoS constraints such as delay and jitter, and proposed a linear programming solution to finding the optimal power management policy.

Although these techniques manage power for the entire communication system, they have a few limitations that cannot be applied to power-aware systems. First, QoS-directed power management can improve performance on average, but do not guarantee the timing constraints. Second, they do not handle power budget as a hard constraint. Although the average power is minimized, the instantaneous power can still exceed the budget. Also, such techniques do not distinguish between different types of energy sources.

2.5 Battery-oriented power management in portable systems

Extending the battery life becomes another goal of power management in portable systems. One practical concern is that the battery life varies with the discharge patterns. Thus, the effort to minimize total energy consumption of average power does not guarantee the optimal use of the non-ideal batteries.

Pedram et al. studied the the variety of effective battery capacity on different load patterns [32, 31]. Their results showed that the battery capacity decreases with the level of discharge current as well as the variance of the load; therefore the load with minimum variance maximizes the battery life. They proposed a power management policy that uses the product of battery life and circuit delay as the optimization goal [33], and a case study of a dual-battery powered portable system is examined [49]. Martin and Siewiorek analyzed the non-linear trade-off between performance and battery life in mobile systems [23]. They proposed a few case studies on a real portable embedded system with non-ideal batteries and a variable-voltage processor [24, 25, 26]. Luo and Jha proposed an instruction scheduling technique based on certain battery discharge patterns [22]. The idea is to smooth the power profile curve as a way to maximize battery life. Benini et al. proposed a discrete-time model for batteries as a way to estimate battery life. They noticed that if the battery can be disconnected from the load after a certain period of discharge, it can regain some of the lost capacity by taking some "rest". They presented a multi-battery powered portable system, which alters the supply power source from a set of batteries by several power management heuristics [5, 4, 6]. Their results showed an extended battery life by a synergy of different power management techniques when taking battery properties into account.

Battery life is a very important issue in portable embedded systems. These techniques represented the first step to incorporate battery characteristics into power management. Such techniques could be further extended to handle power budget as well as the remaining battery capacity as hard constraints. Also, timing constraints should be taken into account for a successful power manager in portable systems.

2.6 Power-aware scheduling at system-level

We believe that the first step towards full power-awareness is constraint-driven system-level design [19, 20]. Timing and power must be treated as hard constraints, not just desirable by-products or hardwired goals. Then, domain-specific knowledge about the power source, battery model, and other operating conditions must then be expressible in terms of the these constraints. The types of constraints that are sufficiently expressive for our application are (1) min and max timing constraints between pairs of tasks, and (2) min and max power constraints on the system. Min/max timing constraints subsume deadlines and precedence dependencies and can express dependencies across subsystems [7, 8]. Max power is a hard constraint in real systems in that it should track the budget imposed by the current power sources. Min power constraints, on the other hand, is not a strict constraint on a system's correct operation. However, we support it as a "knob" for steering the minimum level of activity. For example, in cases where the power from solar panels or other free sources cannot be stored, min power can be set to make the power manager more fully utilize the free power greedily. Another use of min power is to control the power jitter: the power manager will keep the system-level power profile sandwiched between the min and max curves, and this can help maximize battery life. If necessary, we assume that min power can be met by scheduling background tasks.

3 Motivating Example

We use the NASA/JPL Mars rover as our motivating example to demonstrate the effectiveness and applicability of our power-aware scheduling techniques. It is designed to perform scientific experiments and imaging on Mars surface. The rover is deployed to operate for at least 7 sols (days on Mars) initially, and if it keeps performing well at the end of the designated period, then an extended mission follows. The rover has two power sources: it uses the solar panel for most operations, and it also has a non-rechargeable battery pack for sleep mode.

The rover travels to several target locations to perform experiments and imaging. Since the temperature on Mars surface can be as low as $-80°$C, the motors must be heated periodically. As a result, the mechanical and thermal

Operation	Duration (s)	Timing constraints
Heating steering motors	5	At least 5s, at most 50s before steering
Heating wheel motors	5	At least 5s, at most 50s before driving
Hazard detection	10	At least 10s before steering
Steering	5	At least 5s before driving
Driving	10	At least 10s before next hazard detection

Table 1: Timing constraints in Mars rover's operations

Power sources & tasks	Duration (s)	Power (W)		
		Best case @-40°C	Typical case @-60°C	Worst case @-80°C
Solar panel		14.9	12	9
Battery pack		10 max	10 max	10 max
CPU	constant	2.5	3.1	3.7
Heating two motors	5	7.6	9.5	11.3
Driving	10	7.5	10.9	13.8
Steering	5	4.3	6.2	8.1
Hazard detection	10	5.1	6.1	7.3

Table 2: Power consumption of Mars rover's operations

subsystems are the major power consumers, and it is therefore fruitful to target the mechanical and thermal subsystem for power management opportunities when the rover is moving to the next location. Every 7cm in distance traveled by the rover is called a *step*. During each step, it must first detect any obstacles in the moving direction and choose a safe angle to move. Then the four steering motors are then turned in the correct direction. Finally, the six wheel motors are driven to perform one step of movement. Therefore, hazard detection, steering, and driving must operate in sequence. The other set of timing constraints comes from the requirement that all four steering motors and six wheel motors must be heated prior to mechanical operations. The timing constraints are summarized in Table 1.

The power consumption of each operation varies with environmental temperature. We assume that the temperature is closely related to the sunlight density that can be measured by power output from the solar panel. In order to examine how the power-aware scheduling techniques handle different constraints, we investigate three cases of solar power output: best case is 14.9W at noon time; the typical case is 12W; and the worst case is at dusk. The maximum supply power is limited by the threshold of battery power output, which we assume to be 10W. Therefore, in all cases, the rover can be safely operated only if its instantaneous power consumption is less than available solar power plus 10W maximum battery power output, which constitutes the max power constraint. We also extract the solar power level as the min power constraint to distinguish such free power from the costly power. Table 2 illustrates the power sources and consumers in three cases.

The goal of a scheduler is to assign tasks to time slots such that all timing and power constraints are satisfied. Without an automated tool, the JPL scientists had to hand-craft a fully serialized schedule that is very low-power, but is also very slow and can possibly incur additional energy cost. In contrast to the conventional trade-off between energy and performance, where improvement on one is done at the expense of each other, a power-aware approach can win both at the same time. Section 6 provides a detailed analysis to a case study on the Mars rover example.

4 Problem Formulation

Our problem formulation is based on an extension to a constraint graph used in a previous time-driven scheduling problem [7]. One distinction we make in this paper is that we represent one task by its start event and end event.

This separation enables a new constraint classification that exposes some key characteristics of this constraint graph representation. Section 4.1 formulates tasks, timing constraints, resources, modes and schedules in mathematical definitions. Section 4.2 defines a scheduling problem as a constraint graph and illustrates some key characteristics of this graph representation. Section 4.3 introduces the slack properties of events and tasks in a schedule. Section 4.4 defines power characteristics of the scheduling problem including the power profile of a schedule and new properties by applying the max and min power constraints. Section 4.5 presents a new way of viewing the time/power scheduling problem as a two-dimensional constraint problem by drawing analogies from the Gantt chart.

4.1 Preliminaries

We formally define the concepts in our model, including tasks, constraints, resources, operational modes, and schedules.

Definition 1 (Task $x \in T$) A task x is characterized by a tuple $x = (s_x, e_x, \gamma_x, \chi_x, M_x, m_x, d_x, p_x)$, where s_x and e_x are the start and end events, respectively; γ_x is a generalized workload of the task; χ_x is the power characteristic of the task; M_x is the execution resource to which the task is mapped; $m_x \in M_x$ is the selected operational mode of resource M_x; d_x is the execution delay; p_x is the power profile function of the task over time in its execution duration.

Among these attributes, s_x, e_x, γ_x and χ_x are intrinsic to the task itself and will remain immutable over partitions and mode switches. We use a pair of events s_x, e_x to capture the timing requirements of the task itself, e.g., the variable execution delay on different operational modes. Such timing relationships are partially related to workload γ_x, in that execution time d_x depends on γ_x. Power characteristic χ_x is a general description of power behavior, e.g., constant, linear, exponential, etc. It must be specified explicitly before the power profile p_x is given based on mode selection m_x. This power property can be adapted to various forms that either characterize different subsystems or are application-specific.

The other attributes M_x, m_x, d_x, and p_x of task x are determined by the design tool, or by the designer with tool aid. We assume resource mapping M_x is done in the allocation stage prior to scheduling, although the scheduler may override it if task migration is allowed. Examples of execution resources include not only computing resources such as an embedded microprocessor, but also other consumers of power, e.g., mechanical subsystems and heaters. Operational mode m_x can be arranged by the scheduler either statically or dynamically. The execution delay d_x is a function of time t, resource M_x and mode m_x. The power profile p_x is a function of time t, d_x, M_x and m_x, where $p_x \geq 0, 0 \leq t \leq d_x$.

Definition 2 (Resource M with operational modes m) An execution resource M is a set of valid operational modes, $M = \{m_i : 1 \leq i \leq n$. An operational mode m is a collection $m = (\alpha_m, f_m)$, where its characteristic parameters are included in α_m, $f_m = (d_m, p_m)$ is a series of functions including delay function $d_m(x)$ and power function $p_m(x)$ to describe the behaviors of a task x that is executed in this mode.

The parameter set α_m consists of attributes of mode m such as voltage, clock rate, bandwidth, etc. The function set f_m includes several functions that characterize tasks executed in this mode. For a task x executed in mode m, the delay function $d_m(x)$ calculates the execution delay d_x of task x based on mode parameters α_m and workload γ_x; the power function $p_m(x)$ gives task x's power profile $p_x(t)$, which is a function of mode parameters α_m, delay d_x, and power characteristic χ_x.

Definition 3 (Timing constraints) A timing constraint δ specifies the timing relationship between two events u and v, in one of the two forms:

(1) A *min timing constraint*, $u \rightarrow v : \delta$, where $\delta \geq 0$, indicates that v must occur at least δ time units after u happens, formally $t_v - t_u \geq \delta$.

(2) A *max timing constraint* $u \leftarrow v : \delta$, where $\delta > 0$, indicates that v must happen at most δ time units after u happens, formally $t_v - t_u \leq \delta$.

A min timing constraint $u \rightarrow v : \delta$ implies precedence from u to v, since $t_v - t_u \geq \delta \geq 0$; on the other hand, a max constraint $u \leftarrow v : \delta$ does not imply precedence between u and v. In fact, in the absence of any other precedence constraint, such a max constraint may be satisfied trivially by ordering v before u. This min/max timing separation handles more general timing relationships between events. For example, an event u with a deadline τ is a special case of a max timing constraint: $anchor \leftarrow u : \tau$, where *anchor* is a virtual task that starts the schedule, $t_{anchor} = 0$.

Definition 4 (Schedule σ, Finish time τ_σ) Given a task set T and a resource set R,

(1) A schedule σ maps a task $x \in T$ to an operational mode $m_x \in M_x \in R$, and assigns start and end times t_{s_x}, t_{e_x} to events s_x and e_x. Without ambiguity, we further overload the σ notation to map any event u to its assigned time according to σ, that is, $t_u = \sigma(u)$. The time assignment to events s_x and e_x must be consistent with the execution delay d_x, where $\sigma(e_x) - \sigma(s_x) = d_x \geq 0$.

(2) The *finish time* of a schedule σ is the time when all tasks in T finish their execution. It is defined as

$$\tau_\sigma = max(\sigma(e_x)), \forall x \in T. \tag{1}$$

4.2 Constraint graph and properties

We construct a constraint graph based on the tasks, their resource mapping and the corresponding timing constraints among the tasks in a scheduling problem. A schedule as the solution to the problem can be computed based on the constraint graph and its properties.

Definition 5 (Constraint graph $G(V,E)$) Given a task set T, a set of timing constraints C, a constraint graph $G(V,E)$ can be constructed as follows. The vertices V represent events $\{anchor\} \cup \{s_x, e_x\}, \forall x \in T$, where *anchor* represents the virtual start-event that precedes all other events. The edges $E \subseteq V \times V$ represent timing relationships between events. For two vertices $u, v \in V$, an edge (u,v) with weight $w_{(u,v)}$ is denoted as $(u,v) : w_{(u,v)}$. It specifies the timing relationships between events u and v in any schedule σ, such that $\sigma(v) - \sigma(u) \geq w_{(u,v)}$.

Three types of edges represent three different types of timing relationships between two events. They are defined as follows.

Definition 6 (Constraint edges $(u,v) : \pm\delta$) Each timing constraint in C is represented by a constraint edge in the constraint graph G.
(1) A min timing constraint $u \rightarrow v : \delta$ is represented by a *forward edge* $(u,v) : \delta$ with weight $w_{(u,v)} = \delta \geq 0$.
(2) A max timing constraint $u \leftarrow v : \delta$ is represented by a *backward edge* $(v,u) : -\delta$ with weight $w_{(v,u)} = -\delta < 0$.

Definition 7 (Duration edges $(s_x, e_x) : d_x$ and $(e_x, s_x) : -d_x$) The execution delay $d_x \geq 0$ of a task $x \in T$ is represented by a pair of duration edges in the constraint graph G: edge $(s_x, e_x) : d_x$, and edge $(e_x, s_x) : -d_x$.

Definition 8 (Serialization edges $(e_x, s_y) : 0$) A serialization edge $(e_x, s_y) : 0$ is an added edge to serialize task x before y.

Among all three types of edges, constraint edges represent timing constraints between tasks and always remain constant. The weights on duration edges, which represent execution delays of tasks, can be changed according to operational modes. Serialization edges can be added and removed by the scheduler. Tasks that share the same resource must be serialized to prevent resource conflict. The scheduler can also serialize tasks on different resources to avoid exceeding maximum supply power.

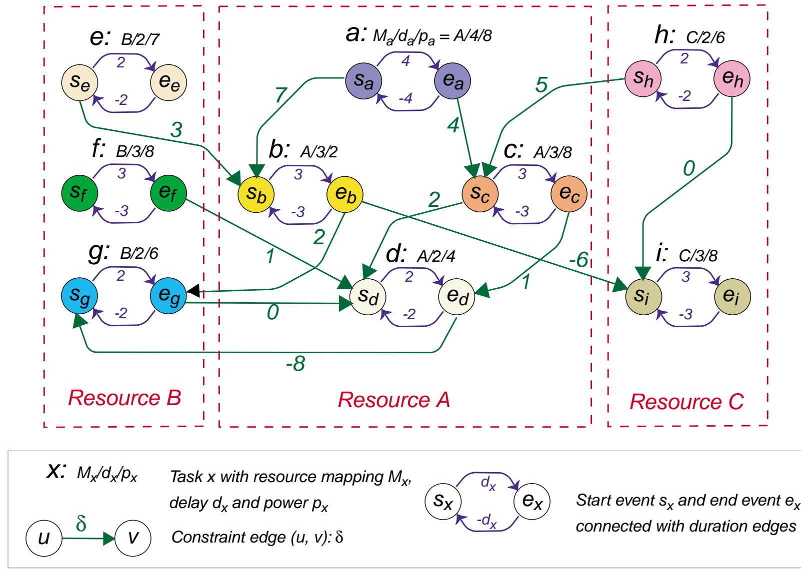

Figure 1: Constraint graph for a scheduling problem.

An example of a constraint graph is illustrated in Fig. 1. Nine tasks named $a \dots i$ are mapped into three resources, A, B and C. Each task x is denoted with a name, its attributes including resource M_x, delay d_x and power p_x in the form of $M_x/d_x/p_x$ and the pair of start and end events connected by two duration edges. Every constraint edge connects two different tasks. For simplicity, we assume that there is one mode of operation per resource and that the tasks have constant power profiles.

Lemma 1 (Schedulability) Given a scheduling problem formulated as a constraint graph, the time assignments by a schedule σ can be computed as the single source longest path lengths from the anchor vertex on the constraint graph. A positive cycle in the graph indicates a conflicting set of timing requirements that cannot be satisfied.

Corollary 1 (Extension to schedulability properties) After scheduling, if a positive cycle consists of
(1) constraint edges only: then the problem is not schedulable.
(2) some duration edges: the problem may be schedulable by changing modes of those tasks.
(3) some serialization edges: the problem may be schedulable by an alternative serialization.

Lemma 1 and Corollary 1 can be used to discover design points that are implied in the problem. Fig. 2 shows an example of how some extra constraints can be extracted by preprocessing the graph. Since the delays are functions of modes, the inequalities involving delays can be viewed as rules to selections of operational modes on related tasks.

Lemma 2 (Transitivity property) If there exist two edges $(u,v) : d_1$, and $(v,w) : d_2$ in the constraint graph, then an edge $(u,w) : d_1 + d_2$ is implied, regardless of the edge type.

Proof By definition of edges, for any schedule σ, $\sigma(v) - \sigma(u) \geq d_1$, $\sigma(w) - \sigma(v) \geq d_2$, thus $\sigma(w) - \sigma(u) \geq d_1 + d_2$

Corollary 2 (Edge transformation by transitivity)
(1) An edge $(u, s_x) : \delta$ is equivalent to edge $(u, e_x) : \delta + d_x$.
(2) An edge $(u, e_x) : \delta$ is equivalent to edge $(u, s_x) : \delta - d_x$.
(3) An edge $(s_x, u) : \delta$ is equivalent to edge $(e_x, u) : \delta + d_x$.
(4) An edge $(e_x, u) : \delta$ is equivalent to edge $(s_x, u) : \delta - d_x$.

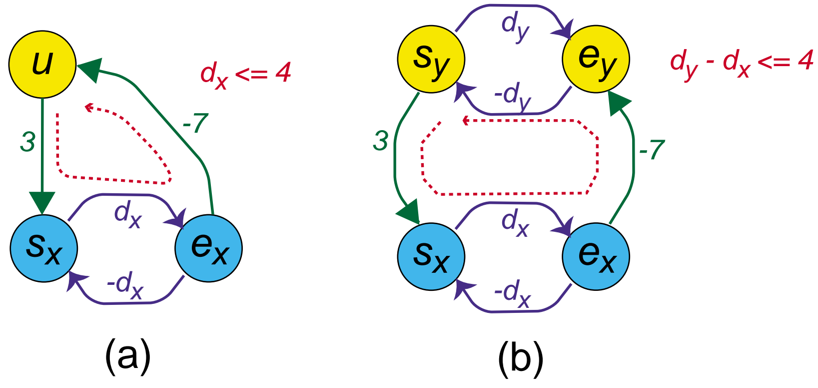

Figure 2: Extra constraints extracted by preprocessing

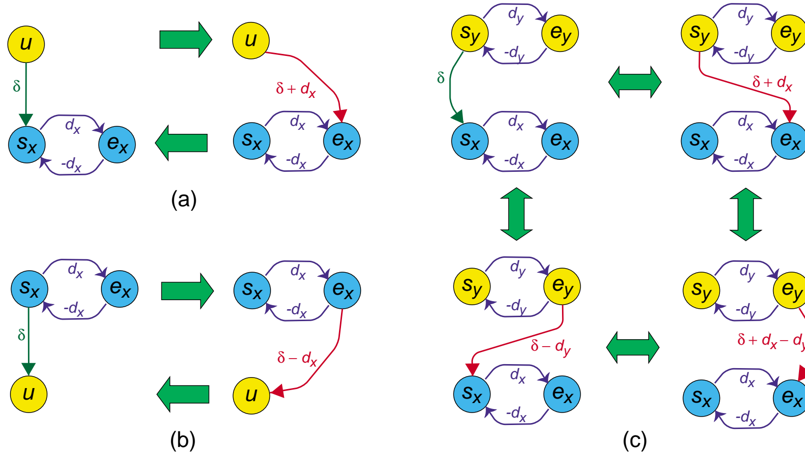

Figure 3: Transformations to constraint edges

Proof By Lemma 2, the edge $(u, s_x) : \delta$ and duration edge $(s_x, e_x) : d_x$ implies edge $(u, e_x) : \delta + d_x$; while this new edge $(u, e_x) : \delta + d_x$ and duration edge $(e_x, s_x) : -d_x$ implies the edge $(u, s_x) : \delta$. Therefore, they are equivalent, as illustrated in Figure 3.

The difference between the original constraint edges and deducted edges is that the deducted ones have d_x in their weights, which will vary with different operational modes; while the original edge remains constant. The transitivity property allows transformation to constraint edges in four forms of equivalent edges, as shown in Figure 3.

A schedule computed by Lemma 1 must satisfy all timing constraints. In addition, a feasible schedule must not have any resource conflict, that is, tasks that share the same resource must be serialized.

Definition 9 (Time-validity of a schedule) Given a constraint graph G constructed from a task set T and a constraint set C, and a resource set R to which all tasks in T are mapped, a schedule σ is *time-valid* if
(1) G is schedulable by Lemma 1, and
(2) $\forall x, y \in T$ such that $M_x = M_y \in R$, x and y must be serialized, i.e., either $\sigma(e_x) \le \sigma(s_y)$ or $\sigma(e_y) \le \sigma(s_x)$ holds.

Definition 9 indicates a way to use graph algorithms to solve scheduling problems with timing constraints and resource sharing between tasks. The scheduler can add serialization edges to the constraint graph to eliminate resource conflict as it computes the longest path lengths for the vertices. For example, to serialize task y after x, a serialization edge $(e_x, s_y) : 0$ can be added to G, such that $\sigma(s_y) \ge \sigma(e_x)$ is guaranteed.

4.3 Slack properties of a time-valid schedule

Given a time-valid schedule σ, there may be alternative valid time assignments $\sigma'(u)$ to an event u. We extract these available time slots as slacks of events. *Slack* is a measure of time interval by which an event can be rescheduled while keeping the resulting schedule time-valid.

Definition 10 ($\Delta^c_\sigma(u,v)$, the <u>Slack</u> of a <u>timing constraint</u> (u,v)) Given is a given time-valid schedule σ computed on a constraint graph $G(V,E)$, and $\overline{(u,v):\delta} \in E$, the *slack of a timing constraint* (u,v) is defined as

$$\Delta^c_\sigma(u,v) = \sigma(v) - \sigma(u) - \delta \tag{2}$$

The slack is necessarily non-negative, because otherwise the schedule σ is not time-valid. The slack also exposes the bounds on reassigning times to events u and v, by delaying u or moving v earlier, without violating the timing constraint $(u,v):\delta$. Note that slacks are defined for constraint edges, but not for serialization edges and duration edges. This is because serialization edges are not intrinsic to the schedulability of the problem, and the duration edges are not controlled by the scheduler. Without ambiguity, we now overload term slack for *events* in terms of slacks for timing constraints.

Definition 11 ($\Delta^{c_f}_\sigma(u)$ and $\Delta^{c_b}_\sigma(u)$, the Forward/backward <u>slacks</u> of an <u>event</u> u) Given a time-valid schedule σ for a constraint graph $G(V,E)$,

- The *forward slack* of an event vertex $u \in V$ is the minimum among all slacks of u's outgoing constraint edges,

$$\Delta^{c_f}_\sigma(u) = \begin{cases} min_v(\Delta^c_\sigma(u,v)) & \forall \text{constraint edges } (u,v), \\ \tau_\sigma - \sigma(u) & \text{if } \nexists \text{ outgoing constraint edges}(u,v). \end{cases} \tag{3}$$

- The *backward slack* of an event vertex $u \in V$ is the minimum among all slacks of u's incoming constraint edges,

$$\Delta^{c_b}_\sigma(u) = \begin{cases} min_v(\Delta^c_\sigma(v,u)) & \forall \text{constraint edges}(u,v), \\ \sigma(u) & \text{if } \nexists \text{ incoming constraint edges } (v,u). \end{cases} \tag{4}$$

The forward slack of an event u indicates the amount of time by which u can be delayed without violating any timing constraints corresponding to u. If u is not constrained by any other events, u can be delayed to the finish time of the whole schedule without violating any timing constraints. Similarly, the backward slack of u specifies the amount of time by which u can be scheduled earlier, without violating any corresponding timing constraints. This pair of forward/backward slacks forms an interval, defined as following.

Definition 12 ($\Delta^c_\sigma(u)$, the <u>Slack interval</u> of an event u) Given a schedule σ computer on a constraint graph $G(V,E)$, the *slack interval* of an event $u \in V$ is a time interval

$$\Delta^c_\sigma(u) = [\sigma(u) - \Delta^{c_b}_\sigma(u), \sigma(u) + \Delta^{c_f}_\sigma(u)] \tag{5}$$

Lemma 3 If a schedule σ is time-valid, then a modified schedule σ' does not violate any timing constraints if it is identical to σ except $\sigma(u) \neq \sigma'(u)$, and $\sigma'(u) \in \Delta^c_\sigma(u)$, for a specific event u. That is, rescheduling event u within its constraint slack interval $\Delta^c_\sigma(u)$ does not violate any timing constraints.

Definition 13 (Reschedule distance $\lambda_{\sigma'-\sigma}(u)$) Given a valid schedule σ and a modified schedule σ' to the same scheduling problem, the *reschedule distance* of u is the difference between the two schedules for an event u:

$$\lambda_{\sigma'-\sigma}(u) = \sigma'(u) - \sigma(u) \tag{6}$$

Corollary 3 Given a valid schedule σ, a modified schedule σ' is also time-valid if the reschedule distance $\lambda_{\sigma'-\sigma}(u) \neq 0$ for only one event u and $-\Delta_{\sigma}^{c_b} \leq \lambda_{\sigma'-\sigma}(u) \leq \Delta_{\sigma}^{c_f}$.

Proof Directly derived from Lemma 3.

Lemma 3 exposes the available time space for an alternative time assignment to an event. However, such adjustment must not result in any resource conflicts. The following definition indicates the vacant time slots to schedule a task without resource conflicts. The start and end events of a task can have different constraint slack intervals, while their resource slack intervals are defined to be identical.

Definition 14 (Resource slots $\Delta_{\sigma}^{r}(x)_i$, resource slack intervals $\Delta_{\sigma}^{r}(x)$) Given a valid schedule σ and a task x,

- The ith *resource slot* of task x $\Delta_{\sigma}^{r}(x)_i$ is a closed interval in time dimension during which resource M_x is not occupied by any tasks, except possibly by x itself.
- Task x's *resource slack intervals* $\Delta_{\sigma}^{r}(x)$ are a collection of all resource slots of x,

$$\Delta_{\sigma}^{r}(x) = \bigcup_{i} \Delta_{\sigma}^{r}(x)_i \tag{7}$$

- Events s_x and e_x have identical resource slots and resource slack intervals as those of task x:

$$\Delta_{\sigma}^{r}(s_x)_i = \Delta_{\sigma}^{r}(e_x)_i = \Delta_{\sigma}^{r}(x)_i \tag{8}$$

$$\Delta_{\sigma}^{r}(s_x) = \Delta_{\sigma}^{r}(e_x) = \Delta_{\sigma}^{r}(x) \tag{9}$$

Lemma 4 If a schedule σ is time-valid, then a modified schedule σ' does not have any resource conflicts if it is identical to σ, except that both start and end events of a task x are rescheduled into the same resource slot $\Delta_{\sigma}^{r}(x)_i \in \Delta_{\sigma}^{r}(x)$, their resource slack intervals. That is, $\exists i$ such that both $\sigma'(s_x), \sigma'(e_x) \in \Delta_{\sigma}^{r}(x)_i$.

The resource slack intervals of a task x represent the vacant time slots on resource M_x that could be assigned to x without any conflicts with other tasks that share the resource M_x. The resource slack intervals can consist of a disjoint interval as a collection of a few closed intervals (resource slots), while the constraint slack is a single closed interval. Based on these two type of intervals, we define their conjunction as the overall slack intervals, within which the event can be rescheduled without either timing or resource violation.

Definition 15 ($\Delta_{\sigma}(u)$, the <u>Overall slack intervals</u> of an event u) Given a valid schedule σ and an event u, the *overall slack intervals of an event u* are defined as the conjunction of u's constraint slack interval $\Delta_{\sigma}^{c}(u)$ and resource slack intervals $\Delta_{\sigma}^{r}(u)$. Formally,

$$\Delta_{\sigma}(u) = \Delta_{\sigma}^{c}(u) \cap \Delta_{\sigma}^{r}(u) \tag{10}$$

Lemma 5 Given a valid schedule σ, a modified schedule σ' neither violates any timing constraint nor causes any resource conflict, if it is identical to σ except $\sigma(u) \neq \sigma'(u)$, and $\sigma'(u) \in \Delta_{\sigma}(u)$, for a specific event u.

Lemma 5 allows an event to be assigned new time slots within its overall slack intervals to avoid timing and resource violations. It is a necessary but not sufficient condition, since the start and end events of x must satisfy extra conditions for σ' to be time-valid.

Lemma 6 (<u>Slack-bounded time-validity</u>) Given a time-valid schedule σ, an alternative schedule σ' derived by rescheduling task x is time-valid if and only if:

(1) Both events s_x and e_x are rescheduled into their overall constraint intervals, i.e., $\sigma'(s_x) \in \Delta(\sigma, s_x)$ and $\sigma'(e_x) \in \Delta(\sigma, e_x)$ (by Lemma 5), and

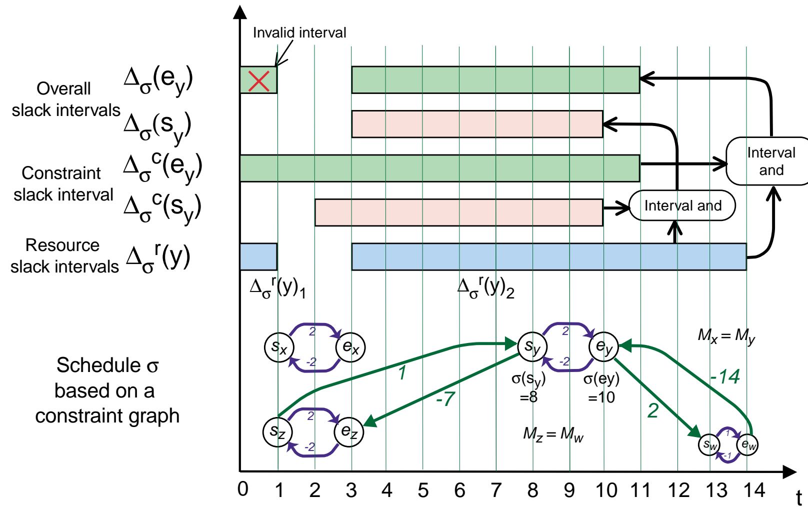

Figure 4: Slack intervals of task b

(2) Both events s_x and e_x are rescheduled into the same resource slot, that is, $\exists i$ such that both $\sigma'(s_x), \sigma'(e_x) \in \Delta^r_\sigma(x)_i$ (by Lemma 4), and

(3) Event s_x precedes e_x by task x's execution delay d_x, that is, $\sigma'(e_x) - \sigma'(s_x) = d_x$, where d_x is a function of mode m_x on resource M_x to execute task x (Definitions 1 and 4).

Proof By satisfying (1), the new schedule does not have violate any timing constraints (Lemma 5). By satisfying (2), it does not have any resource conflict (Lemma 4). Condition (3) is to enforce the definition of a task in a schedule (Definitions 1 and 4) after the start or end event of a task has been given a new time slot.

Given a time-valid schedule, Lemma 6 allows some tasks to be rescheduled while yielding new time-valid schedules. The slack properties of tasks form the basis of our power-aware scheduling algorithms for power/performance trade-offs. Such properties are especially meaningful to support different power management decisions. For example, in voltage/frequence scaling on processors, the execution delay of a task can change due to the voltage or frequency settings. It is applicable to task y in Fig. 4, where events s_y and e_y can be arranged into different time slots with various execution delay d_y. Fig. 4 also exemplifies how to eliminates illegal combinations of s_y and e_y's slack intervals by using Lemma 6

If one event is rescheduled, the slacks of other events will be changed accordingly. If the number of events in a scheduling problem is large, the time to recompute all slacks can be very expensive. The following lemmas tell that only a subset of all the events need to recompute their slacks.

Lemma 7 (Constraint slacks of the new schedule) Given a time-valid schedule σ based on a constraint graph G, and an alternative time-valid schedule σ' identical to σ except $\sigma'(u) \neq \sigma(u)$ for an event u,

(1) the forward and backward slacks of u will be changed according to reschedule distance $\lambda_{\sigma'-\sigma}(u)$:

(1.a) Forward slack $\Delta^{c_f}_{\sigma'}(u) = \Delta^{c_f}_{\sigma'}(u) - \lambda_{\sigma'-\sigma}(u)$, Backward slack $\Delta^{c_b}_{\sigma'}(u) = \Delta^{c_b}_{\sigma'}(u) + \lambda_{\sigma'-\sigma}(u)$

(1.b) while $\Delta^c_\sigma(u)$, the slack interval of u, remains unchanged, $\Delta^c_{\sigma'}(u) = \Delta^c_{\sigma'}(u)$

(2) only those events that are *immediate predecessors* to u via constraint edges in G will change their *forward slacks*. For any u's immediate predecessor v via a constraint edge $(v,u) : \delta$,

(2.a) if u's reschedule distance $\lambda_{\sigma'-\sigma}(u) > 0$, v's forward slack is increased by an amount no more than u's reschedule distance $\lambda_{\sigma'-\sigma}(u)$, that is, $0 \le \Delta_{\sigma'}^{cf} - \Delta_{\sigma}^{cf} \le \lambda_{\sigma'-\sigma}(u)$

(2.b) if u's reschedule distance $\lambda_{\sigma'-\sigma}(u) < 0$, v's forward slack is decreased by an amount no more than the absolute value of u's reschedule distance $-\lambda_{\sigma'-\sigma}(u)$, that is, $0 \ge \Delta_{\sigma'}^{cf} - \Delta_{\sigma}^{cf} \ge \lambda_{\sigma'-\sigma}(u)$

(3) only those events that are *immediate successors* from u via constraint edges in G will increase their *backward slacks*. For any u's immediate successor v via a constraint edge $(u,v) : \delta$,

(3.a) if u's reschedule distance $\lambda_{\sigma'-\sigma}(u) > 0$, v's backward slack is decreased by an amount no more than u's reschedule distance $\lambda_{\sigma'-\sigma}(u)$, that is, $0 \ge \Delta_{\sigma'}^{cb} - \Delta_{\sigma}^{cb} \ge -\lambda_{\sigma'-\sigma}(u)$

(3.b) if u's reschedule distance $\lambda_{\sigma'-\sigma}(u) < 0$, v's backward slack is increased by an amount no more than the absolute value of u's reschedule distance $-\lambda_{\sigma'-\sigma}(u)$, that is, $0 \le \Delta_{\sigma'}^{cf} - \Delta_{\sigma}^{cf} \le -\lambda_{\sigma'-\sigma}(u)$.

Proof

(1) By definition of forward slacks,
$\Delta_{\sigma}^{cf}(u) = \min_v(\Delta_{\sigma}^{c}(u,v)) = \min_v(\sigma(v) - \sigma(u) - w_{(u,v)}) = \min_v(\sigma(v) - w_{(u,v)}) - \sigma(u)$,
for all u's successor v via a constraint edge $(u,v) : w_{(u,v)}$.
Similarly, $\Delta_{\sigma'}^{cf}(u) = \min_v(\sigma'(v) - w_{(u,v)}) - \sigma'(u)$. Since σ and σ' are only different at event u, $\sigma(v) = \sigma'(v)$.
Therefore, $\Delta_{\sigma'}^{cf}(u) - \Delta_{\sigma'}^{cf} = \sigma(u) - \sigma'(u) = -\lambda_{\sigma'-\sigma}(u)$
For backward slack of u,
$\Delta_{\sigma}^{cb}(u) = \sigma(u) - \min_v(\sigma(v) - w_{(u,v)})$, $\Delta_{\sigma'}^{cb}(u) = \sigma'(u) - \min_v(\sigma'(v) - w_{(u,v)})$,
for all u's predecessor v via a constraint edge $(v,u) : w_{(v,u)}$.
Since $\sigma(v) = \sigma'(v)$, $\Delta_{\sigma'}^{cb}(u) - \Delta_{\sigma'}^{cb} = \sigma'(u) - \sigma(u) = \lambda_{\sigma'-\sigma}(u)$
(1.a) is proved.
By definition of slack of u, $\Delta_{\sigma'}^{c}(u) = [\sigma'(u) + \Delta_{\sigma'}^{cf}(u), \sigma'(u) - \Delta_{\sigma'}^{cb}(u)]$,
from (1.a) it is equal to
$[\sigma'(u) + \Delta_{\sigma}^{cf}(u) - (\sigma'(u) - \sigma(u)), \sigma'(u) - \Delta_{\sigma}^{cb}(u) - (\sigma'(u) - \sigma(u))] = [\sigma(u) + \Delta_{\sigma}^{cf}(u), \sigma(u) - \Delta_{\sigma}^{cb}(u)] = \Delta_{\sigma'}^{c}(u)$.
This proves (1.b).

(2) If $(v,u) : w_{(v,u)}$ is a constraint edge, for schedules σ and σ', u's forward slacks are
$\Delta_{\sigma}^{cf}(v) = \min_z(\Delta_{\sigma}^{c}(v,z))$ and $\Delta_{\sigma'}^{cf}(v) = \min_z(\Delta_{\sigma'}^{c}(v,z))$, respectively,
for all v's immediate successors z (including u) via a constraint edge (v,z).
Therefore, $\Delta_{\sigma'}^{cf}(v) - \Delta_{\sigma}^{cf}(v) = \min_z(\Delta_{\sigma'}^{c}(v,z)) - \min_z(\Delta_{\sigma}^{c}(v,z))$.
Since σ and σ' differ only for event u, all the slacks for edges (v,z) are the same for σ and σ', except for edge (v,u). There are four cases:

(i) $\min_z(\Delta_{\sigma'}^{c}(v,z)) \ne \Delta_{\sigma'}^{c}(v,u)$ and $\min_z(\Delta_{\sigma}^{c}(v,z)) \ne \Delta_{\sigma}^{c}(v,u)$,
therefore $\min_z(\Delta_{\sigma'}^{c}(v,z)) = \min_z(\Delta_{\sigma}^{c}(v,z))$, since for all the rest edges (v,z) except (v,u) their slacks do not change, the minimum will be the same.
In this case, $\Delta_{\sigma'}^{cf}(v) - \Delta_{\sigma}^{cf}(v) = 0$.

(ii) $\min_z(\Delta_{\sigma'}^{c}(v,z)) = \Delta_{\sigma'}^{c}(v,u)$ and $\min_z(\Delta_{\sigma}^{c}(v,z)) = \Delta_{\sigma}^{c}(v,u)$,
then $\Delta_{\sigma'}^{cf}(v) - \Delta_{\sigma}^{cf}(v) = \Delta_{\sigma'}^{cf}(v,u) - \Delta_{\sigma}^{cf}(v,u) = \sigma'(u) - \sigma(u) = \lambda_{\sigma'-\sigma}(u)$.
(i) and (ii) are the extreme cases of (2.a) and (2.b)

(iii) $\min_z(\Delta_{\sigma'}^{c}(v,z)) \ne \Delta_{\sigma'}^{c}(v,u)$ and $\min_z(\Delta_{\sigma}^{c}(v,z)) = \Delta_{\sigma}^{c}(v,u)$,
then $\Delta_{\sigma'}^{c}(v,u) > \min_z(\Delta_{\sigma'}^{c}(v,z)) \ge \Delta_{\sigma}^{c}(v,u)$, since $\Delta_{\sigma'}^{c}(v,z) = \Delta_{\sigma}^{c}(v,z), \forall z \ne u$.
Therefore, $\Delta_{\sigma'}^{cf}(v) - \Delta_{\sigma}^{cf}(v) < \Delta_{\sigma'}^{cf}(v,u) - \Delta_{\sigma}^{cf}(v,u) = \lambda_{\sigma'-\sigma}(u)$ and $\lambda_{\sigma'-\sigma}(u) > 0$.

(iv) $\min_z(\Delta^c_{\sigma'}(v,z)) = \Delta^c_{\sigma'}(v,u)$ and $\min_z(\Delta^c_\sigma(v,z)) \neq \Delta^c_\sigma(v,u)$,
then $\Delta^c_{\sigma'}(v,u) \leq \min_z(\Delta^c_\sigma(v,z)) < \Delta^c_\sigma(v,u)$, since $\Delta^c_{\sigma'}(v,z) = \Delta^c_\sigma(v,z), \forall z \neq u$.
Therefore, $\Delta^{cf}_{\sigma'}(v) - \Delta^{cf}_\sigma(v) > \Delta^{cf}_{\sigma'}(v,u) - \Delta^{cf}_\sigma(v,u) = \lambda_{\sigma'-\sigma}(u)$, and $\lambda_{\sigma'-\sigma}(u) < 0$.

Cases (i, ii, iii) prove (2.a), since case (iv) will not happen if $\lambda_{\sigma'-\sigma}(u) > 0$.
Cases (i, ii, iv) prove (2.b).

(3) The proof is similar to (2).

Lemma 8 (Resource slack intervals of the new schedule) Given a time-valid schedule σ based on a constraint graph G, and an alternative time-valid schedule σ' which is identical to σ except $\sigma'(u) \neq \sigma(u)$ for an event $u \in \{s_x, e_x\}$ for a given task x,
(1) Task x's resource slack intervals does not change, $\Delta^r_{\sigma'}(x) = \Delta^r_\sigma(x)$.
(2) Only those tasks that are mapped onto the same resource M_x will change their resource slack intervals.

Lemma 7 and 8 indicate a the subset of events/tasks that will change the values of their slacks upon rescheduling one single event u. The scheduler will only need to update the slacks of the events in this subset without extensively recomputing slack values for all events.

4.4 Power characteristics of a schedule

We extend the power properties to schedules based on the constraint graph formulation. A schedule has a power profile representing the power consumption of task execution. We introduce max and min power constraints and extract some new properties by applying power constraints to a schedule.

Definition 16 (Power profile P_σ, Total energy E_σ) Given a time-valid schedule σ,

(1) The *power profile* of σ is defined as a function of time. At any given time t, its value is the total power consumption of all tasks that are being executed at t. That is,

$$P_\sigma(t) = \sum_{x \in T} p_x(t - \sigma(s_x)), 0 \leq t \leq \tau_\sigma. \tag{11}$$

(2) The *total energy* of σ is the integral of its power profile over time, that is,

$$E_\sigma = \int_0^{\tau_\sigma} P_\sigma(t)dt \tag{12}$$

Definition 17 (Max and min power constraints P_{max} and P_{min}) The power profile P_σ is constrained by two parameters, $P_{max}, P_{min} \in \mathbb{R}$, $P_{max} \geq P_{min} \geq 0$.

(1) The *max power constraint* P_{max} specifies the maximum level of supply power that can be provided to support task execution.

(2) The *min power constraint* P_{min} specifies the level of power consumption to maintain a preferred magnitude of activity.

We treat the max power constraint as a hard constraint. At any given moment, the total power consumption by all running tasks must not exceed P_{max}. The min power constraint is a soft constraint in this paper.

Definition 18 (Power spike, power gap) Given a schedule σ with its power profile $P_\sigma(t)$, and power constraints P_{max} and P_{min},

(1) In a given time interval $[t_1, t_2]$, if the power profile exceeds max power constraint, that is, $P_\sigma(t) > P_{max}$ ($t_1 \le t \le t_2$), the interval $[t_1, t_2]$ is called a *power spike*.

(2) In a given time interval $[t_1, t_2]$, if the power profile is below the min power level, that is, $P_\sigma(t) < P_{min}$ ($t_1 \le t \le t_2$), the interval $[t_1, t_2]$ is called a *power gap*.

Power spikes and power gaps are the times slots where the power constraints are violated. Since only the max power constraint is treated as a hard constraint, a schedule with any power spikes must not be considered as a valid one. However, power gaps will not invalidate a schedule. Accordingly, a valid schedule is defined as follows.

Definition 19 (Power-validity of a schedule) Given a time-valid schedule σ computed from a constraint graph G with the task set T, constraint set C, and resource set R, for a max power constraint P_{max}, schedule σ is *power-valid* if,
(1) σ is time-valid by Definition 9, and
(2) Its power profile does not exceed max power constraint, that is, $P_\sigma(t) \le P_{max}$, for $0 \le t \le \tau_\sigma$.

Definition 19 incorporates the power usage of a schedule as a constraint in addition to the existing constraints on the time dimension. Only max power constraint is used to qualify the validity of a schedule. (In the ensuing text, if not explicitly specified, a "valid" schedule means it is power-valid, which implies its time-validity.) Min power usage, which refers to the utilization to free power sources, is not enforced. Such separation distinguishes different power sources as expensive power and free power. It forms some new perspectives on power/performance trade-offs in a power-aware system, as described in the following definitions.

Definition 20 (Power cost $Pc_\sigma(P_{min})$, Energy cost $Ec_\sigma(P_{min})$) Given a time-valid schedule σ with a min power constraint P_{min} representing free power level,

- The *power cost* of σ is the power usage above the min power level P_{min}. It is defined as a function of time and P_{min},

$$Pc_\sigma(P_{min}, t) = \left\{ \begin{array}{ll} P_\sigma(t) - P_{min} & \text{when } P_\sigma(t) > P_{min} \\ 0 & \text{when } P_\sigma(t) \le P_{min} \end{array} \right\} \text{for } 0 \le t \le \tau_\sigma \tag{13}$$

- The *energy cost* is the integral of the power cost function,

$$Ec_\sigma(P_{min}) = \int_0^{\tau_\sigma} Pc_\sigma(P_{min}, t)dt \tag{14}$$

Definition 21 (Min power utilization $\rho_\sigma(P_{min})$) Given a time-valid schedule σ with a min power constraint $P_{min} > 0$ representing free power level, its *min power utilization* is defined as the ratio of its energy drawn from free power source over the total available free energy,

$$\rho_\sigma(P_{min}) = \frac{E_\sigma - Ec_\sigma(P_{min})}{P_{min} \times \tau_\sigma} \tag{15}$$

We do not limit the power and energy costs and min power utilization in Definitions 20 and 21 to only a power-valid schedule, since these properties are also meaningful to schedules that are not power-valid. They further highlight the difference between costly power and free power. Any power consumption below the min power level does not contribute to the energy consumption from costly energy sources. In fact, the free power should be utilized greedily to preserve the costly power. This new perspective subsumes the conventional power or energy minimization techniques as a special case, where $P_{min} = 0$. A power-aware design should explore different trade-offs between *performance vs. costly power*, while making the best effort to fully utilize the free energy for performance speedup. This forms the basis of our power-aware scheduling techniques presented in Section 5.

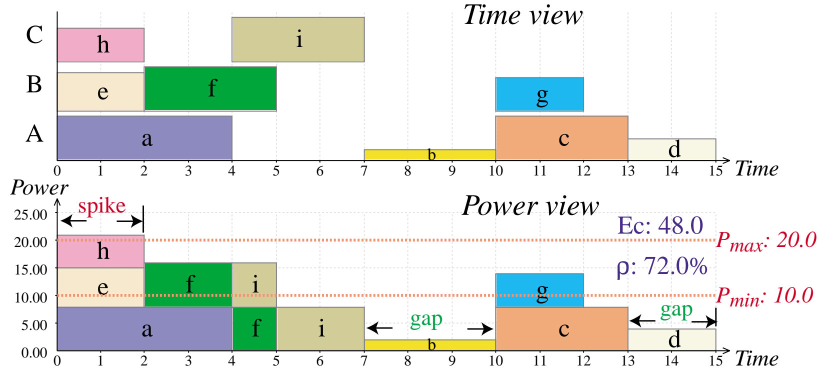

Figure 5: Power-aware Gantt chart of a time-valid schedule

4.5 Power-aware Gantt chart

There exist various visual representations for real-time scheduling problems, e.g. Gantt chart. However, very few of them have the capability to express power properties of a schedule, regardless of any power constraints. We introduce the *power-aware Gantt chart* as a new visual representation for power-aware scheduling problems. It presents a schedule in two different views: *time view* and *power view*. Each view is a two dimensional diagram whose horizontal axis represents time and vertical axis represents power. In the time view, tasks are displayed as bins placed in several rows that denote parallel execution resources. The power view shows the power profile of the schedule with min and max power constraints and some corresponding power properties.

In the time view for a schedule σ computed from a constraint graph G with task set T and resource set R, the execution of a task $u \in T$ is represented by a horizontal bin beginning with its start time $\sigma(u)$ and whose length corresponds to its duration $d(u)$. We scale the vertical size of the bin to denote power consumption $p(u)$. As a result, the area of the bin indicates its energy expenditure. Each execution resource $R_i \in R$ takes one row denoted by R_i. All tasks that are mapped on this resource, that is, $\forall u \in T$ such that $r(u) = R_i$, are displayed in row R_i in timing order. The empty time slots between adjacent bins represent the resource slacks. Timing constraints, and slacks in the time dimension, though normally not shown, can also be intuitively visualized by selectively attaching annotation on the bins.

By collapsing all bins in the time view to the lowest horizontal axis, the expected power profile $P_\sigma(t)$ can be shown in the power view of the power-aware Gantt chart. It also illustrates the composition of the power profile from every power consumer's contribution at each time. With annotation of max and min power level, power spikes and power gaps can be directly observed; the power/energy cost vs. free power usage are clearly separated; and power properties such as power/energy cost, $Pc_\sigma(P_{min}, t), Ec_\sigma(P_{min})$ and min power utilization $\rho_\sigma(P_{min})$ can be visualized with the corresponding annotations. Fig. 5 shows the power-aware Gantt chart of a time-valid schedule to the example problem in Fig. 1.

In addition to a graphical representation to schedules, the power-aware Gantt chart also serves as the underlying model for a power-aware design tool that allows the designers to evaluate different power/performance trade-offs visually. The designers can manually intervene with the automated scheduling process by dragging and locking the bins to alternative time slots in the time view, while observing the results in the power view interactively.

5 Algorithm

Based on the constraint graph formulation, we develop graph algorithms for power-aware scheduling. Given a scheduling problem, the goal of the power-aware scheduler is to find a valid schedule σ with following properties:

(1) σ must be time-valid, that is, it satisfies all timing constraints and can arrange all tasks to corresponding execution resources without any resource conflicts.

(2) σ must satisfy the max power constraint, that is, no power spikes can be found in the schedule.
By qualifying (1) and (2) the schedule is a valid one that meets all hard constraints.

(3) σ could have power gaps according to the min power constraint, but the scheduler should make its best efforts to remove power gaps, by reducing power/energy cost or improving min power utilization.

Power-aware scheduling is a multi-constraint solving problem. Our approach is to first examine different constraints in our model defined in Section 4. We find that the constraints on timing and resource sharing are the most critical ones that must be considered first as necessary conditions. Next, we consider max power constraints after after a time-valid schedule is found. The scheduler must eliminate all power spikes while keeping the schedule time-valid to generate a valid schedule. Finally, the min power constraint can be applied after a valid schedule is given. The analysis suggests an incremental approach by solving one type of constraint at a time in the following three steps.

First, based on the constraint graph of the problem, we try to find a schedule that is time-valid. Power constraints and power consumption of tasks are not considered in this step. The algorithm is presented in Section 5.1. It extends previous work on a time-driven serial scheduling for a single execution resource to handling parallel execution on multiple resources.

Second, after a time-valid schedule is computed from the first step, the max power constraint is applied to constrain its power profile. Section 5.2 explains the algorithm to remove power spikes by using heuristics based on slack properties of the schedule. Tasks that contribute to a power spike are partially reordered by a slack-based ordering function. To avoid exhaustive search in the solution space, we apply heuristics to examine more reasonable solutions first.

Finally, given a valid schedule provided by the previous step, we apply the min power constraint and reorder tasks within their slacks to reduce power gaps and improve the min power utilization. The algorithm is illustrated in Section 5.3. It does not guarantee full utilization of the min power level. Also, the final schedule should not have a longer finish time with a loss of performance, since min power is a soft constraint that is not critical to the applicability of the schedule.

5.1 Algorithm for timing scheduling

The time-constrained scheduling algorithm is shown in Fig. 6. It is a extension to a previous serialization algorithm [7]. G is the constraint graph for the scheduling problem. *anchor* is the source vertex that is used in the SINGLE SOURCE LONGEST PATH algorithm. It represents a virtual task that starts at time 0. c is called the candidate vertex that is being visited at each step as the algorithm traverses graph G topologically. The schedule for the event c $\sigma(c)$ is assigned as the distance from the *anchor* to c in the longest path. The next candidate v is selected from c's successors. Tasks that share the same resources are serialized by adding serialization edges between vertices. If these additional edges for serialization produce any positive loops in the graph, they are then removed by the algorithm and another topological ordering is attempted. The first invocation of the algorithm starts from *anchor* as the first candidate. Then the algorithm is recursively invoked at each step when a new candidate is selected. One task is scheduled by visiting its start and end events in consecutive two steps. A time-valid schedule is returned when all vertices are reordered without any positive loops and resource conflicts.

This algorithm can be proved to always find a time-valid schedule if one exists, since it will traverse all possible topological orderings of the graph before it terminates with a failure.

Based on the problem shown in Fig. 1, its time-valid schedule is illustrated in Fig. 5 in the form of a power-aware Gantt chart. There are one power spike and several power gaps left for the remaining steps of our power-aware scheduler.

```
TimingScheduler(Graph G, vertex anchor, vertex c)
    La := SINGLE SOURCE LONGEST PATH(G, anchor)
    if (positive cycle found) then
        return FAIL
    C := set of topological successors of candidate c
    if (C = ∅) then
        return σ with σ(u) = La[u] ∀u ∈ G.V
    x_c := task of event c
    while (C ≠ ∅) do
        if (c = s_{x_c}) then
            v := e_{x_c}
        else
            v := one topological successor of C
            C := C − {v}
            x_v = task of event v
            if (v = e_{x_v}) then
                v := s_{x_v}
B:          foreach u ∈ C do
                if u ∉ v's successors then
                    add u to v's successors
                x_u := task of event u
                if (M_{x_c} = M_{x_u}) then
                    add serialization edge (e_{x_c}, s_{x_u}) : 0 to G
            y := the most recently scheduled task, such that (M_y = M_{x_v})
            if (y ≠ nil) then
                add serialization edge (e_y, s_{x_v}) : 0 to G
        σ := TimingScheduler(G, anchor, v)
        if (σ ≠ FAIL) then
            return σ
        undo added edges to G since step B
        if (c = s_{x_c}) then
            return FAIL
    return FAIL
```

Figure 6: Algorithm for timing scheduling.

```
MaxPowerScheduler(Graph G, vertex anchor, P_max)
    σ := TimingScheduler(G, anchor, anchor)
    if (σ = FAIL) then
        return FAIL
    for (t := 0; t ≤ τ_σ; t := t + 1) do
        S := set of all active tasks at t, ordered a heuristic slack function
        power := P_σ(t)
        reschedule := FALSE
        while (power > P_max or reschedule = TRUE) do
B:          repeat
                x := EXTRACT MAX(S)
                if (reschedule = FALSE and Δ_σ(v) = 0) then
                    reorder tasks in S by forward constraint slack Δ_σ^{cf}
                    reschedule := TRUE
                delay x by some time units (heuristically determined)
                power := power − p_x
                S := S − {x}
            until (power ≤ P_max or S = ∅)
            if (S = ∅) then
                return FAIL
            if (reschedule = TRUE) then
                lock start time of all tasks in S
                σ := MaxPowerScheduler(G, anchor, P_max)
                if (σ ≠ FAIL) then
                    return σ
                undo added edges to G since step B
    return σ
```

Figure 7: Algorithm for max power scheduling.

5.2 Algorithm for max power scheduling

Our approach to meeting max power constraint is to eliminate the power spikes of a time-valid schedule computed by the previous step. In this algorithm we make two assumptions. First, we only consider one operational mode on each execution resource. Therefore the execution delay of a task x is fixed at a single value d_x and we simplify the problem by scheduling both start and end events of a task together. Second, we assume task x's power consumption is a constant value p_x, instead of a function varying over time. These assumptions simplify the problem without losing our focus on power-constrained scheduling, although our application model can support more sophisticated power management techniques that incorporate different operational modes on resources and non-constant power consumption functions.

The algorithm is shown in Fig. 7. It has three parameters: graph G, vertex *anchor*, and max power constraint P_{max}. The timing scheduler is always called first to obtain a time-valid schedule. The algorithm examines the power profile P_σ of the returned schedule σ to find the first power spike at time t. To eliminate the spike, several simultaneous tasks at t are partially reordered so that the height of the power curve is less than P_{max}. The algorithm itself is called recursively after the spike at t is eliminated by delaying tasks. A valid schedule σ is found if there is no power spike in σ; and the time-validity of σ is always guaranteed. If no solution can be found after the recursive call, a failure notice is returned suggesting that either additional tasks at t need to be delayed, or one or more tasks already delayed have been incorrectly chosen.

The key issues in this algorithm are properly selecting and delaying tasks for spike elimination. We do not attempt

exhaustive enumeration to all possible partial orderings of tasks since it would be exponential. Instead, we apply heuristics to avoid several pitfalls. First, the total execution time τ_σ may be extended unnecessarily, leading to a loss of performance. Second, the algorithm may evaluate some invalid schedules repeatedly before approaching a valid one, so that the scheduler requires extra computation time needlessly. Finally, the algorithm may fail to find a valid schedule even if one exists.

We propose slack-based heuristics for selecting and delaying tasks. First, a slack-based heuristic function is used to order simultaneous tasks. When a power spike is detected at time t, the algorithm orders tasks that are active at t by their a heuristic slack function, which is the quantized value of the length of slack intervals. This slack-based ordering function indicates which task has more available time slot to be rescheduled while the new schedule is still time-valid. Then, the algorithm selects tasks to delay based on the following conditions.

(1) If there are tasks with non-zero slacks, the task with the largest slack is always selected first. The algorithm continues selecting tasks to delay until the power spike at t is removed.

(2) If no tasks with non-zero slack is available while the power spike at t is still present, the remaining tasks are reordered by their forward constraint slacks (which is the minimum of the forward constraint slacks of the start and end events, since we fixed the execution delay d_x). Tasks with larger forward constraint slacks will be delayed.

(3) If the power spike cannot be removed until all the remaining tasks have zero constraint slack, tasks are randomly selected to be delayed.

After a task x is selected to be delayed, the second question is what its new start time $\sigma'(s_x)$ should be. To delay a task x based on an existing schedule σ, we add an edge from *anchor* to s_x, with positive weight t' as the lower bound on its new start time. Therefore, $\sigma'(s_x) \geq t'$ and the reschedule distance $\lambda_{\sigma'-\sigma}(s_x) \geq t' - \sigma(u)$. Clearly, making a small reschedule distance is not efficient. On the other hand, we do not expect the reschedule distance to be too large such that the finish time of the schedule may be unnecessarily increased. We currently heuristically set the upper bound of the reschedule distance to the execution time of the task. In addition, in case (1) where the selected task x has some slack, we apply slack-bounded time-validity (Lemma 6) to ensure the new schedule is still time-valid. This is called *local reordering*. Therefore, in case (1), we put this extra bound to reduce the effort for rescheduling for time-validity. The algorithm can still proceed with a time-valid schedule. While in cases (2) and (3), since the new schedule after the delay is no longer time-valid, the timing scheduler must be invoked to make the schedule time-valid again by asserting the Boolean variable *reschedule*, which is called *global reordering*. In case (2), the selected task u has some constraint slack but no resource slack, the delay distance is further bounded by the forward constraint slack $min(\Delta^c_\sigma(s_x, \Delta^c_\sigma(e_x)$, so that all timing constraints are preserved thus the scheduler only needs to eliminate the resource conflict caused by the delay. All of these constraints can actually serve the purpose of pruning out the search space tremendously. Finally, in case (3), which eliminates a power spike at the cost of introducing new timing violations, some significant timing adjustment to the schedule is expected.

After enough tasks are delayed and the power spike at t disappears, we lock the start time of the remaining tasks. The start time of a task y is locked by adding two edges to graph G, a forward edge $(anchor, s_y) : \sigma(s_y)$, and a backward edge $(s_y, anchor) : -\sigma(s_y)$. As a result, task y is forced to start at time $\sigma(s_y)$ by the SINGLE SOURCE LONGEST PATH algorithm. These locks are especially meaningful to case (3). When the scheduler delays a task x to eliminate a power spike at time t, it is desirable to keep all tasks that are scheduled before t intact. While in case (3), the attempt to remove a power spike starting at time t by delaying a task x may cause some partial reorderings on tasks before t due to backward edges. Such type of reordering may produce a new power spike before time t and will certainly complicate the scheduler. The algorithm could spend much more time dealing with the unexpected spikes before it converges to a valid schedule. By locking the tasks that do not form a power spike at t, no further delays can be applied to these tasks. However, if delays to these tasks are necessary for a valid schedule, the algorithm will fail in its next recursions and these locks will be undone. Then the algorithm will choose one task from them to make further delay and continue recursion.

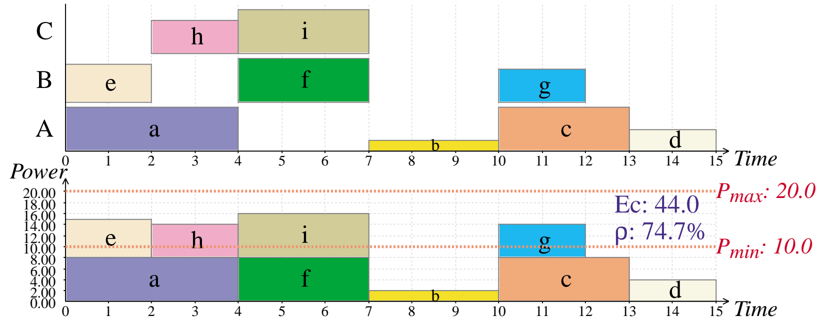

Figure 8: A valid schedule after max power scheduling.

It is notable that in some extreme cases, the max power constraint scheduler may not be able to find a valid schedule even though one exists . The reason is that the algorithm does not enumerate all possible combinations in partially ordered tasks. However, in practice, our heuristics perform very well in finding a valid solution without sacrificing performance. Our slack-based heuristics tend to examine more reasonable schedules first. Also, the heuristic to lock the tasks before the recursion can help reduce the computation of the scheduler.

The schedule shown in Fig. 5 does not satisfy the max power constraint. Fig. 8 show the valid schedule after applying the max power scheduler. Tasks h and f are delayed to remove the power spike.

5.3 Algorithm for min power scheduling

The goal of the min power constraint scheduler is to reduce the energy cost by improving min power utilization for a given valid schedule. The algorithm is shown in Fig. 9. Four parameters are passed to the algorithm: graph G, vertex *anchor*, power constraints P_{max} and P_{min}. A valid schedule σ is obtained from the power-valid scheduler at the beginning of the algorithm. If σ already has full min power utilization, then no further improvement is necessary, and the algorithm completes. Otherwise, it tries to find a power gap at time t and to delay some tasks scheduled before t to fill this power gap. These tasks are can be delayed until t such that the new schedule is time-valid. The algorithm also checks whether the new schedule has any power spikes, and whether its min power utilization is better than the existing schedule. If so, it is a better schedule and the algorithm continues searching for further improvement. Otherwise, the delay is canceled and the previous schedule is restored. The algorithm keeps locally reordering tasks to produce valid schedules with improved min power utilization, until full utilization is achieved or a maximum number of iteration is reached.

In order to find an "optimal" schedule whose energy cost is the minimized, the algorithm should examine all valid partial orderings of tasks, which will increase the complexity of computation to an exponential order of tasks. Therefore, we apply heuristics based on following observations. First, the scheduler may need to scan the schedule multiple times. This is because delaying tasks to fill a power gap at time t may create new power gaps before t. Also, since delaying one task x will change the slacks of other tasks that are constrained by x, there may be new opportunities for reordering those tasks that are not eligible for delay previously. As a result, either new power gaps or new tasks to fill other power gaps can be found after the algorithm scans the schedule again. Moreover, the order in which to visit the power gaps will lead to different final schedules because different partial reorderings of tasks are applied. This suggests that better schedules could be found if we scan the schedule in various orders in time dimension, e.g. incremental order, reverse order, or random order. Finally, when a task u is selected to fill a power gap at t, we consider alternative time slots to reschedule x, rather than just starting x at t. It is difficult to determine the "best" time slot for rescheduling x since it alters not only the power profile but also the slacks of some other tasks. We also address this issue by heuristics. Some available heuristics are: starting x at t, finishing x at the end of the power gap including t, or a randomly chosen time slot. In practice, we can scan the schedule multiple times while altering some of the heuristics during each scan and take the best results.

```
MinPowerScheduler(Graph G, vertex anchor, P_max, P_min)
    σ := MaxPowerScheduler(G, anchor, P_max)
    if (σ = FAIL) then
        return FAIL
    if (ρ_σ(P_min) = 1) then
        return σ
    improvement := TRUE
    while (improvement = TRUE) do
        improvement := FALSE
        for (t in a heuristic order of range (0, τ_σ)) do
            if (P_σ(t) < P_min) then
                S := set of tasks that start before t
                foreach x ∈ S such that delaying x to t − d_x is time-valid do
                    σ' := σ
B:                  delay x some time units such that x is active at t
                    if (σ is valid and ρ_σ(P_min) > ρ_σ'(P_min)) then
                        improvement := TRUE
                        if (ρ_σ(P_min) = 1) then
                            return σ
                    else
                        undo added edges to G in step B
                        σ := σ'
    return σ
```

Figure 9: Algorithm for min power scheduling.

The Boolean variable *improvement* refers to whether the scheduler finds a better schedule during one scan to the existing schedule. If no further delay can improve the schedule, the algorithm terminates successfully. Each scan (except the last one) will improve the schedule by delivering the same performance with a reduced energy cost. Since min power constraint is a soft constraint, the schedule tolerates the existence of power gaps after it makes the best efforts to remove them.

In this algorithm, tasks are locally reordered within their slacks during schedule improvement. This guarantees that the delays always result in time-valid schedules. The algorithm also never introduces delays that either create power spikes or incur a higher energy cost. Therefore, no additional rescheduling will be necessary after delaying these tasks. Furthermore, the min power scheduler can possibly reduce the height of the power profile. This indicates the same schedule can be applied to different power constraints without any extra effort to reschedule the problem.

Fig. 10 shows a better schedule that improves on the valid schedule in Fig. 8. Energy cost is reduced while the height of the power profile curve is also reduced. In fact, the same schedule can be directly applied to all cases with a range of constraints where $15 \leq P_{max} \leq 20, 2 \leq P_{min} \leq 15$, without recomputing a schedule for each case. This feature makes our statically computed power-aware schedules directly adaptable to a run-time scheduler that schedules tasks according to the dynamically changing constraints imposed by the environment.

6 Experimental Results

This section presents scheduling results for the Mars rover operations and a case study for evaluating our power-aware scheduling algorithms in a mission scenario.

The constraint graph for the Mars rover is shown in Fig. 11. Since the power consumption varies in three different cases, the power attributes of tasks are not shown in the graph. To simplify the problem, we assume all heaters are independent resources and one heater can heat two motors at a time. Therefore there are a total of five thermal

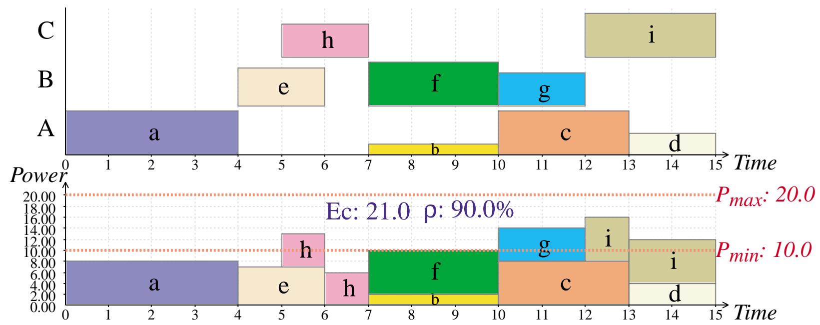

Figure 10: The improved schedule after min power scheduling.

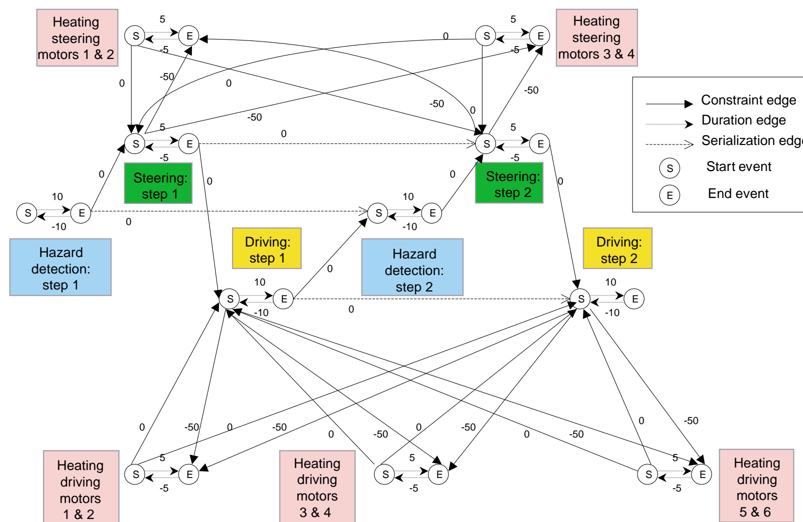

Figure 11: Constraint graph of the Mars rover

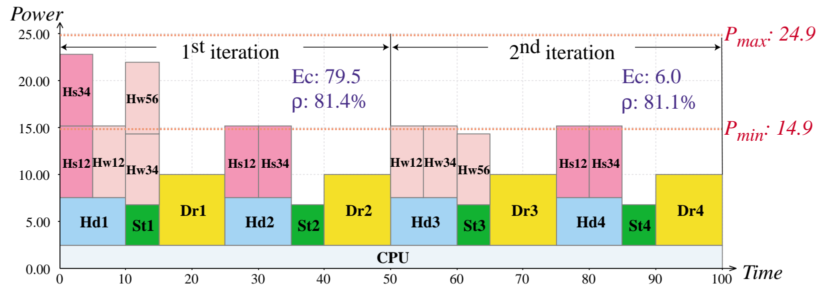

Figure 12: Schedule for the best case

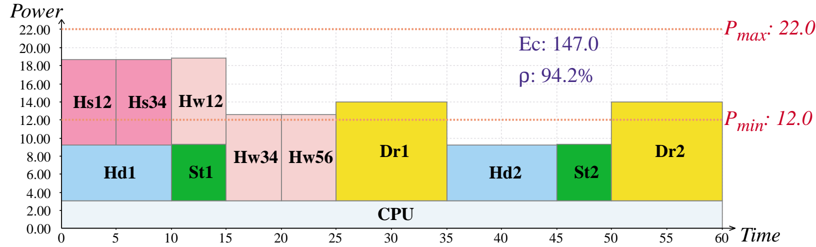

Figure 13: Schedule for the typical case

heaters. Four steering motors are considered a single steering mechanical resource. The six wheel motors are modeled as one mechanical unit for driving. There is also a laser guided digital component for hazard detection.

Fig. 12, 13 and 14 show the results for three cases after applying power-aware scheduling algorithms. Fig. 12 gives first two iterations of the loop in the best case. To utilize the available free energy, we manually unroll the loop and insert two heating tasks to improve loop efficiency through better solar energy utilization. Therefore the second iteration can be repeated without too much energy cost. In other cases only one iteration is shown since loop unrolling is not necessary. In the best case, because the power budget is sufficient, a fast schedule is given by allowing operations to overlap. In the typical case, parallel operations are still possible while some heating tasks are serialized. In the worst case, a tight power budget forces all operations to be serialized, leading to a slow schedule.

The existing schedule used in the past mission was designed to be low-power. To avoid exceeding max power supply, JPL uses a serialized schedule that is fixed in all situations, regardless of available solar power and power consumption in different conditions. The existing schedule is identical to our power-aware schedule computed with the lowest min and max power constraints. The fundamental difference is that, our schedule is completely constraint-driven; whereas the existing solution is hardwired and does not track the power availability. The performance and energy cost of our schedules and the existing schedule are compared in Table 3.

We use finish time τ_σ and energy cost $Ec_\sigma(P_{min})$ to the non-rechargeable battery as the metrics. The existing

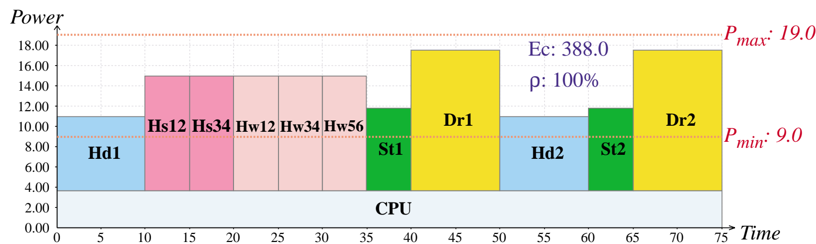

Figure 14: Schedule for the worst case

Solar power P_{min} (W)	JPL			Power-aware		
	Energy cost $Ec_\sigma(P_{min})$ (J)	Utilization $\rho_\sigma(P_{min})$	Time τ_σ (s)	Energy cost $Ec_\sigma(P_{min})$ (J)	Utilization $\rho_\sigma(P_{min})$	Time τ_σ (s)
14.9	0	60%	75	79.5(1st) 6(2nd)	81%	50
12	55	91%	75	147	94%	60
9	388	100%	75	388	100%	75

Table 3: Comparison on performance and energy cost

Time frame (s)	Solar power (W)	JPL			Power-aware		
		Distance (step)	Time (s)	Energy cost (J)	Distance (step)	Time (s)	Energy cost (J)
0 - 599	14.9	16	600	0	24	600	145.5
600 - 1199	12	16	600	440	20	600	1470
1200 -	9	16	600	3114	4	150	776
Total		***48***	***1800***	***3554***	***48***	***1350***	***2391.5***
					Improve-ment	***33.3%***	***32.7%***

Table 4: Comparison to schedules under a mission scenario

scheme only schedules for the worst case; while in other cases, solar energy is under-utilized and opportunities to performance improvement are overlooked. However, JPL's low-power schedule appears "economic" since its energy cost is low. Our schedules, on the other hand, speeds up the rover's movement by up to 50% in the best case and 25% in the typical case, while drawing more costly energy from the battery. To evaluate this trade-off, we apply our schedules and the existing schedule to a mission scenario when the available solar power varies over time, and then evaluate the performance vs. energy cost in this bigger picture.

Suppose the mission is to travel to the next target location, which is 48 steps away from the current location. The mission starts around noon when maximum solar power is present. While the mission is in progress, the power output from the solar panel drops from 14.9W to 12W after 10 minutes, then falls to the worst case at 9W 10 minutes later. If the existing schedule is applied, the rover will spend 10 minutes evenly in the best case, typical case, and worst case since it has a fixed moving speed (16 steps per 10 minutes). This results in a long execution time (30 minutes) and considerable energy cost in the worst case. When our schedules are used, the rover finishes 50% of its work (24 steps) in the first 10 minutes, 42% of work (20 steps) in the next 10 minutes, leaving the remaining 8% (4 steps) in the worst case for less than 3 minutes. Since our schedules accelerate execution at the best and typical cases, the rover can finish the mission earlier before having to work in the costly worst case. The results of this case study are shown in Table 4. The analysis shows our schedules win both on performance and energy savings considerably.

Fig. 15 highlights the property of the power-aware scheduler in a geometrical view. The top chart illustrates how the power-aware scheduler adjusts the execution speed adaptively with available power budget, while the existing scheme ignores the power constraint and always operates at the lowest speed. The workload is represented by the integral of the speed curve over time. Therefore our curve reaches the given workload earlier because of higher execution speed before operating in the worst case. The bottom chart shows the power cost from battery over time and how it alters as power constraint varies. The energy expenditure is symbolized by the integral of power curve over time. When the mission is completed, both the speed curve and power curve also end. Although our power curve is higher in most time during the mission, by completing earlier we avoid further energy cost from integrating a high power curve with a longer execution time. Therefore, given the same workload, the power-aware scheduler is capable of achieving performance speedup less energy cost simultaneously.

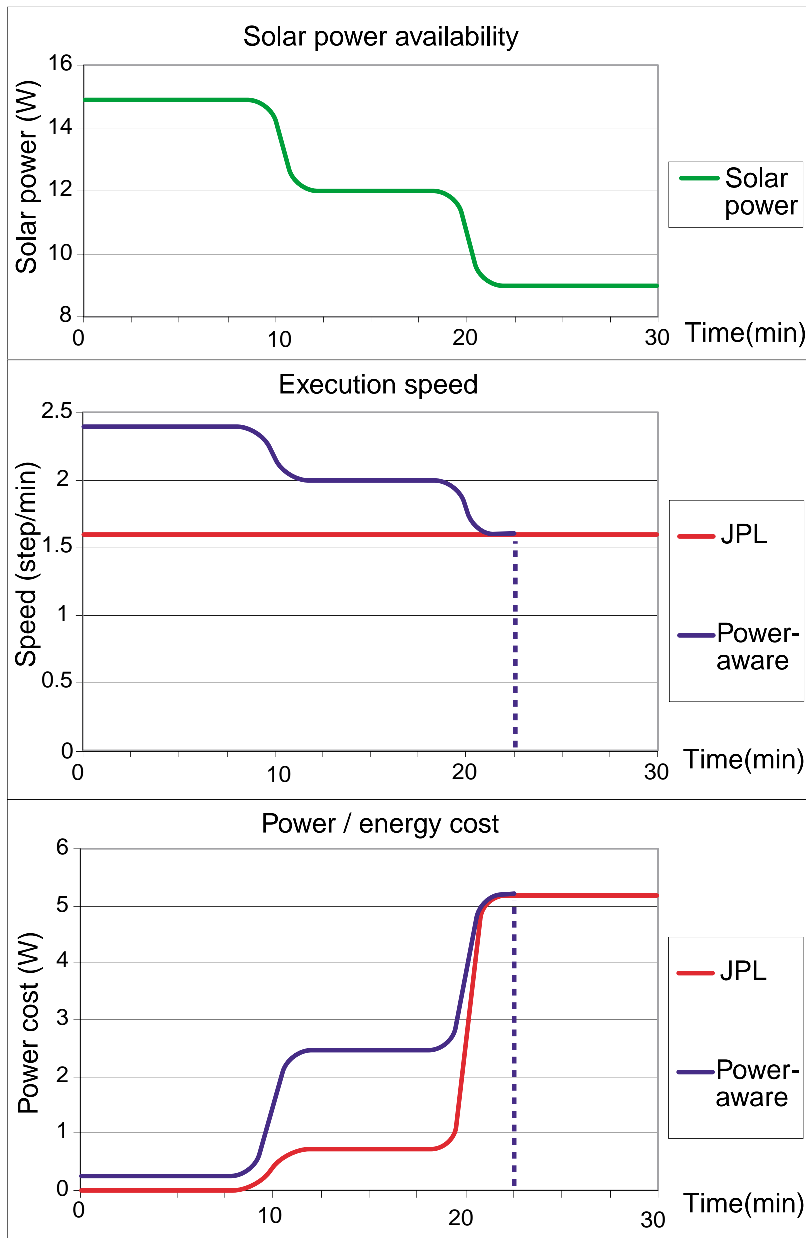

Figure 15: Adaptive speedup in power-aware scheduling

7 Conclusion and Future Work

Power-aware design becomes a more important issue in mission-critical systems that require best use of available power sources and deliver high performance at the same time. We target the scheduling algorithms to embedded systems with variable power constraints and various types of power consumers, as well as different energy sources that are classified as costly power vs. free power. In these systems, power-aware techniques have potentials for both performance improvement and energy savings.

In this paper, we present a constraint-driven model that incorporates power and timing constraints in a system-level context. We propose three core algorithms that decompose the power-aware scheduling problems into steps. Via this incremental approach, we distinguish the properties of each sub-problem and apply heuristics to solve the constraints by different methods. The case study to a real application demonstrates that our power-aware method is capable of improving performance while saving expensive energy.

Several interesting issues in this dimension need further attention. To expand the applicability of our algorithms, more effective heuristics need to be discovered. We would also like to incorporate more novel power management techniques including voltage/frequency scaling into this tool to support more effective power-aware designs.

Acknowledgment

This research represents a collaboration between the University of California at Irvine and the NASA/CalTech Jet Propulsion Laboratory. Special thanks to Dr. M. Mojarradi and Dr. J. U. Patel at JPL and Kerry Hill at AFRL for their discussion and assistance.

References

[1] NASA/JPL's Mars Pathfinder home page. http://mars3.jpl.nasa.gov/MPF/index0.html.

[2] A. Acquaviva, L. Benini, and B. Ricco. Processor frequency setting for energy minimization of streaming multimedia application. In *Proc. International Symposium on Hardware/Software Codesign*, pages 249–253, April 2001.

[3] N. K. Bambha, S. S. Bhattacharyya, J. Teich, and E. Zitzler. Hybrid global/local search strategies for dynamic voltage scaling in embedded multiprocessors. In *Proc. International Symposium on Hardware/Software Codesign*, pages 243–248, April 2001.

[4] L. Benini, G. Castelli, A. Macii, E. Macii, M. Poncino, and R. Scarsi. Extending lifetime of portable systems by battery scheduling. In *Proc. Design, Automation and Test in Europe*, pages 197–201, March 2001.

[5] L. Benini, G. Castelli, A. Macii, and R. Scarsi. Battery-driven dynamic power management of portable systems. In *Proc. International Symposium on System Synthesis*, pages 25–30, September 2000.

[6] L. Benini, G. Castelli, A. Macii, and R. Scarsi. Battery-driven dynamic power management. *IEEE Design and Test of Computers*, 18(2):53–60, March 2001.

[7] P. Chou and G. Borriello. Software scheduling in the co-synthesis of reactive real-time systems. In *Proc. Design Automation Conference*, pages 1–4, June 1994.

[8] P. Chou and G. Borriello. Interval scheduling: Fine grained code scheduling for embedded systems. In *Proc. Design Automation Conference*, pages 462–467, June 1995.

[9] E.-Y. Chung, L. Benini, and G. De Micheli. Dynamic power management using adaptive learning tree. In *Proc. International Conference on Computer-Aided Design*, pages 274–279, 1999.

[10] I. Hong, D. Kirovski, G. Qu, and M. Potkonjak. Power optimization of variable-voltage core-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1702–1714, 1999.

[11] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastavas. Power optimization of variable voltage core-based systems. In *Proc. Design Automation Conference*, pages 176–181, June 1998.

[12] I. Hong, M. Potkonjak, and M. B. Srivastava. On-line scheduling of hard real-time tasks on variable voltage processor. In *Proc. International Conference on Computer-Aided Design*, pages 653–656, November 1998.

[13] I. Hong, G. Qu, M. Potkonjak, and M. Srivastavas. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proc. IEEE Real-Time Systems Symposium*, pages 178–187, December 1998.

[14] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. International Symposium on Low Power Electronics and Design*, pages 197–202, August 1998.

[15] S. Lee and T. Sakurai. Run-time power control scheme using software feedback loop for low-power real-time applications. In *Proc. Asian and South Pacific Design Automation Conference*, pages 381–386, January 2000.

[16] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *Proc. Design Automation Conference*, pages 806–809, June 2000.

[17] Y.-H. Lee and C. Krishna. Voltage-clock scaling for low energy consumption in real-time embedded systems. In *Proc. International Conference on Real-Time Computing Systems and Applications*, pages 272–279, December 1999.

[18] Y.-H. Lee and C. Krishna. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *Proc. IEEE Real-Time Technology and Applications Symposium*, pages 156–165, June 2000.

[19] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *Proc. International Symposium on Hardware/Software Codesign*, pages 153–158, April 2001.

[20] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints for mission-critical embedded systems. In *Proc. Design Automation Conference*, pages 840–845, June 2001.

[21] J. Luo and N. Jha. Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems. In *Proc. International Conference on Computer-Aided Design*, pages 357–364, November 2000.

[22] J. Luo and N. K. Jha. Battery-aware static scheduling for distributed real-time embedded systems. In *Proc. Design Automation Conference*, pages 444–449, June 2001.

[23] T. Martin and D. Siewiorek. A power metric for mobile systems. In *Proc. International Symposium on Low Power Electronics and Design*, pages 37–42, August 1996.

[24] T. Martin and D. Siewiorek. The impact of battery capacity and memory bandwidth on cpu speed-setting: a case study. In *Proc. International Symposium on Low Power Electronics and Design*, pages 200–205, August 1999.

[25] T. Martin and D. Siewiorek. Non-ideal battery properties and low power operation in wearable computing. In *Proc. Third International Symposium on Wearable Computers*, pages 101–106, October 1999.

[26] T. Martin and D. Siewiorek. Nonideal battery and main memory effects on cpu speed-setting for low power. *IEEE Transactions on VLSI Systems*, 9(1):29–34, February 2001.

[27] T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *Proc. International Symposium on System Synthesis*, pages 24–29, November 1999.

[28] T. Okuma, T. Ishihara, and H. Yasuura. Software energy reduction techniques for variable-voltage processors. *IEEE Design and Test of Computers*, 18(2):31–41, March 2001.

[29] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Energy-aware instruction scheduling. In *Proc. International Conference on High Performance Computing*, pages 335–344, December 2000.

[30] A. Parikh, M. Kandemir, N. Vijaykrishnan, and M. Irwin. Instruction scheduling based on energy and performance constraints. In *Proc. IEEE Computer Society Workshop on VLSI 2000. System Design for a System-on-Chip Era*, pages 37–42, April 2000.

[31] M. Pedram, C.-Y. Tsui, and Q. Wu. An integrated battery-hardware model for portable electronics. In *Proc. Asian and South Pacific Design Automation Conference*, pages 109–112, January 1999.

[32] M. Pedram and Q. Wu. Battery-powered digital CMOS design. In *Proc. Design, Automation and Test in Europe*, pages 72–76, March 1999.

[33] M. Pedram and Q. Wu. Design considerations for battery-powered electronics. In *Proc. Design Automation Conference*, pages 861–866, June 1999.

[34] Q. Qiu and M. Pedram. Dynamic power management based on continuous-time markov decision processes. In *Proc. Design Automation Conference*, pages 555–61, June 1999.

[35] Q. Qiu, Q. Wu, and M. Pedram. Stochastic modeling of a power-managed system: construction and optimization. In *Proc. International Symposium on Low Power Electronics and Design*, pages 194–199, August 1999.

[36] Q. Qiu, Q. Wu, and M. Pedram. Dynamic power management of complex systems using generalized stochastic petri nets. In *Proc. Design Automation Conference*, pages 352–356, June 2000.

[37] Q. Qiu, Q. Wu, and M. Pedram. Dynamic power management in a mibile multimedia system with guaranteed quality-of service. In *Proc. Design Automation Conference*, pages 834–839, June 2001.

[38] G. Qu, D. Kirovski, M. Potkonjak, and M. Srivastavas. Energy minimization of system pipelines using multiple voltages. In *Proc. IEEE International Symposium on Circuits and Systems*, pages 362–365, May 1999.

[39] G. Qu and M. Potkonjak. Power minimization using system-level partitioning of applications with quality of service requirements. In *Proc. International Conference on Computer-Aided Design*, pages 343–346, November 1999.

[40] G. Qu and M. Potkonjak. Energy minimization with guaranteed quality of service. In *Proc. International Symposium on Low Power Electronics and Design*, pages 43–48, July 2000.

[41] G. Quan and X. S. Hu. Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors. In *Proc. Design Automation Conference*, pages 828–835, June 2001.

[42] D. Shin, J. Kim, and S. Lee. Low-energy intra-task voltage scheduling using static timing analysis. In *Proc. Design Automation Conference*, pages 438–443, June 2001.

[43] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proc. Design Automation Conference*, pages 134–139, June 1999.

[44] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proc. International Conference on Computer-Aided Design*, pages 365–368, November 2000.

[45] T. Simunic, L. Benini, and G. De Micheli. Event-driven power management of portable systems. In *Proc. International Symposium on System Synthesis*, pages 18–23, 1999.

[46] M. Srivastava, A. Chandrakasan, and R. Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on VLSI Systems*, 4(1):42–55, March 1996.

[47] V. Swanminathan, K. Chakrabarty, and S. S. Iyengar. Dynamic i/o power management for hard real-time systems. In *Proc. International Symposium on Hardware/Software Codesign*, pages 237–242, April 2001.

[48] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 13–23, 1994.

[49] Q. Wu, Q. Qiu, and M. Pedram. An interleaved dual-battery power supply for battery-operated electronics. In *Proc. Asian and South Pacific Design Automation Conference*, pages 387–390, January 2000.

[50] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995.