Power-Aware Scheduling under Timing Constraints and Slack Analysis for Mission-Critical Embedded Systems [1]

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

15 March 2001

[1] This research was sponsored by DARPA under contract F33615-00-1-1719

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. This paper presents a new scheduling technique for supporting the design and evaluation to a class of power-aware systems in mission critical applications. It 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 one of the central issues in embedded systems. They are particularly critical to systems that must carry their own power source and cannot rely on a power outlet on the wall. Without power, the system is useless. In the consumer space, the consequence may mean not being able to make an emergency call or other minor inconveniences; but in mission-critical systems, such a failure can cost millions and even human lives.

This paper investigates key issues in power management for mission-oriented systems. Our motivating example comes from the NASA Mars Pathfinder rover developed at JPL [1]. It features several interesting properties 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

Traditionally, many components and systems have been designed to be low-power. However, we believe there is a critical difference between power-aware and low-power systems. Power-aware systems must make the best use of their available power, and they subsume low-power as a special case.

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, the 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 pictures per day.

A power-aware design can greatly improve the utility of the rover. We observe that the battery is non-rechargeable, and thus solar power would be wasted if not used while it is 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. A rover with more parallelism in its schedule can perform better (more tasks, more quickly) while saving even more battery energy than the existing low-power design if it can take advantage of the free power, as validated by our experiments in the results section.

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 only draws 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 biggest consumers are not even in the digital computer, but they also 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.

1.3 Approach: design tools

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.

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, either interactively or semi-automatically by the design tool. 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 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 help predict the proper time to shutdown and power up subsystems [6, 4, 7].

While it is important to manage the power of subsystems, unfortunately these techniques have several limitations. First, they do not handle timing constraints, including deadlines and min/max separation. Second, they are not power-aware in the sense that they do not distinguish between free power (such as solar sources) vs. expensive power (non-rechargeable battery). These power managers do not control their workload; instead, they make the best effort to minimize power consumption by treating the workload as a given.

2.2 Real-time scheduling

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 [5].

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, in practice, it is difficult to tune the voltage or frequency scale to such a fine precision. As a result, the risk of missing deadlines may be high, even if context switching overhead is taken into account. Also, while these schedulers meet timing constraints, they do not handle constraints on power usage.

2.3 Power awareness

We believe power-aware scheduling must have several key features. First, they must handle both timing and power stringently as hard constraints. This is unlike previous work that treats them as desirable by-products but cannot always make strong guarantees. Second, domain-specific knowledge about the power source, battery model, and other operating conditions must be expressible in terms of supported types of constraints on the timing and power. The types of constraints that are sufficiently expressive for our application are min and max timing constraints on tasks, as well as min and max power constraints on the system. Min/max timing constraints subsume deadlines and precedence dependencies and can express dependencies across subsystems [2, 3]. Max power would track the budget imposed by the current power sources. Min power constraints, strictly speaking, may be counter-intuitive in that it forces the power manager to maintain a certain level of activity. The primary motivation is that power from solar panels or other free sources that cannot be stored should be fully utilized greedily, or else they will be wasted. Another motivation is to control the jitter in the system-level power curve in an attempt to optimize battery usage. However, min power constraints are not imperatively enforced, and we assume that they may be violated occasionally or be met by scheduling background tasks.

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

3 Motivating Example

To demonstrate the effectiveness and applicability of our power-aware scheduling techniques, we choose the NASA/JPL Mars rover as our motivating example. Its mission is to perform scientific experiments and imaging on Mars surface. The rover is deployed and operated for at least 7 sols (days on Mars). If it keeps performing well at the end of the designated period, an extended mission may continue. The rover's power sources consist of a non-rechargeable battery pack and a solar panel. Clearly, the duration of a mission is limited by the amount of remaining battery energy. Thus, a careful management of power usage may yield potential energy savings, as well as performance speedup.

The rover travels between different target locations before experiments and imaging can be performed. Since the temperature on Mars surface can be as low as $-80^{\circ}C$, driving in low temperature requires more power and energy consumption because the motors must be heated periodically. This fact indicates that mechanical and thermal subsystems are the major power consumers. Therefore, our model targets the mechanical and thermal subsystem under a typical mission scenario when the rover is moving to the next location.

We give a high-level description of the rover's operations. The rover drives about 7cm in distance in one single step of movement. 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 started to turn to the right direction. Finally, the six wheel motors are driven to perform one step of movement. Therefore, hazard detection - steering - driving must operate in sequence. The other set of timing constraints comes from the requirement to heat the motors before steering and driving. All four steering motors and six wheel motors must be heated within a certain period 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

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

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 existing solution by JPL had to be hand-crafted. It serializes all operations to minimize power draw from the non-rechargeable battery. The existing design is very low-power, but is also very slow and can possibly incur additional energy cost in some bad cases.

By introducing power-aware scheduling, not only could we improve performance, but also save non-rechargeable energy by better utilization of solar energy. This is 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 [2]. Section 4.1 reviews the base formulation and our extension on parallel execution and slack properties. Section 4.2 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 4.3 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 Constraint graph and properties

We formally define the concepts in our model as we construct the constraint graph formulation for a scheduling problem. These concepts include tasks, timing constrains, schedules and the slack properties of a schedule.

Definition 1 (Tasks $u \in T$) Given T as the set of all tasks and a set of execution resource R, a task $u \in T$ is characterized by a set of functions, $u = \{r(u), d(u), p(u)\}$, where $r(u) \in R$ is the execution resource onto which the task is mapped, $d(u)$ is its execution delay, and $p(u)$ is its power consumption.

To handle parallel execution resources that consume power, the function $r : T \to R$ maps each task to a resource set R. 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. We further assume that if two tasks u and v are mapped to the same resource ($r(u) = r(v)$), then u and v must be serialized in the final schedule to eliminate resource conflicts.

The execution of task u takes $d(u)$ time units. We also assume the availability of the power consumption function, $p : T \to \mathbb{R} > 0$, which returns the estimated power consumption by all tasks. As a result, the energy consumption of task u is $d(u) \times p(u)$. In practice, the power consumption can be either in the form of (min, typical, max), or a function over time, rather than an exact value. Since our formulation can be extended to handling these cases, we will assume a single value to simplify the discussion.

Definition 2 (Timing constraints) A timing constraint specifies the timing relationship between two tasks $u, v \in T$, in one of the two forms:
(1) A *min timing constraint* $u \to v : \delta, \delta \geq 0$ indicates that v must start at least δ time units after u starts, formally $t_v - t_u \geq \delta$.
(2) A *max timing constraint* $u \leftarrow v : \delta, \delta > 0$ indicates that v must start at most δ time units after u starts, formally $t_v - t_u \leq \delta$.

A min timing constraint $u \to v : \delta$ implies task u precedes v, since $t_v - t_u \geq \delta \geq 0$; while a max constraint $u \leftarrow v : \delta$ does not imply any precedence relationship between u and v. This min-max timing separation handles more general timing relationships between tasks. For example, a task u with a deadline τ to finish its execution ($\tau \geq d(u)$) is a special case of a max timing constraint: $anchor \leftarrow u : \tau - d(u)$, where *anchor* is a virtual task that starts the schedule, $t_{anchor} = 0$.

Definition 3 (Schedule σ, Finish time τ_σ) Given a task set T,
(1) A *schedule* σ assigns start time t_u to every task $u \in T$. Without ambiguity, we further overload the σ notation to map any task u to its assigned start time according to σ, that is, $\sigma(u) = t_u$.
(2) The *finish time* of a schedule σ is the time when all tasks in T finish their execution. It is defined as $\tau_\sigma = max(\sigma(u) + d(u)), \forall u \in T$.

We construct a constraint graph based on the tasks, their resource mapping and the corresponding constraints among the tasks in a scheduling problem. A schedule

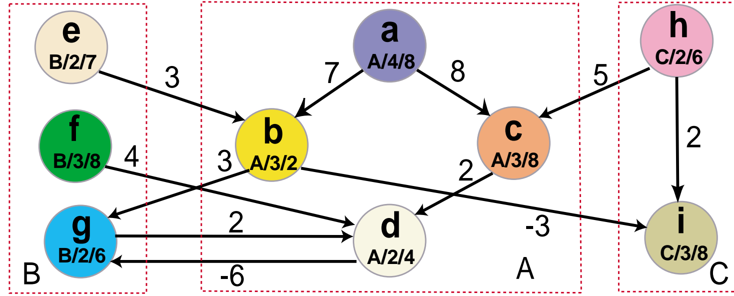

Figure 1: Constraint graph of a scheduling problem

as the solution to the problem can be computed based on the constraint graph and its properties.

Definition 4 (Constraint graph $G(V,E)$) Given a task set T, a resource set R to which all tasks in T are mapped, and a timing constraint set C specifying the timing constraints between tasks in T, a *constraint graph* $G(V,E)$ can be constructed as follows.
(1) The vertices V represent all tasks, $V = \{anchor\} \cup \{u\}, \forall u \in T$. Each vertex $u \in V$ has three attributes: $r(u), d(u), p(u)$ representing its resource mapping, execution delay and power consumption of task u, respectively. $r(anchor) = nil, d(anchor) = 0, p(anchor) = 0$.
(2) The edges $E \subseteq V \times V$ represent timing constraints between tasks. 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 a timing constraint between task u and v, such that $t_v - t_u \geq w(u,v)$.
(2.a) A min timing constraint $u \rightarrow v : \delta, \delta \geq 0$ is represented by an edge $(u,v) : \delta$ with non-negative weight $\delta \geq 0$, called a *forward edge*.
(2.b) A max timing constraint $u \leftarrow v : \delta, \delta > 0$ is represented by an edge $(v,u) : -\delta$ with negative weight $-\delta < 0$, called a *backward edge*.

An example of a constraint graph is illustrated in Fig. 1. Nine tasks named $a \ldots i$ are mapped into three resources, A, B and C. Each vertex u is denoted with a name and its attributes in the form of $r(u)/d(u)/p(u)$.

Lemma 1 (Schedulability property) Given a scheduling problem with a task set T, a resource set R and a timing constraint set C formulated as a constraint graph $G(V,E)$, a schedule σ that satisfies all timing constraints can be computed as the SINGLE SOURCE LONGEST PATH lengths from $anchor \in V$. A positive cycle in the graph indicates a conflicting set of timing constraints that cannot be satisfied.

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 5 (Time-validity of a schedule) Given a scheduling problem with a task set T, a resource set R and a timing constraint set C, a schedule σ is *time-valid* if

(1) σ satisfies all timing constraints in C, and
(2) $\forall$ tasks $u, v \in T$ such that $r(u) = r(v) \in R$, u and v must be serialized, that is, either $\sigma(u) + d(u) \leq \sigma(v)$ or $\sigma(v) + d(v) \leq \sigma(u)$ holds.

Lemma 1 indicates the way to use graph algorithms to solve scheduling problems with timing constraint. The constraint graph can also be used to serialize tasks on shared resources, as required by Definition 5 to obtain a time-valid schedule. Serialization can be performed by adding extra edges to constraint graph G. For example, to serialize task v after u, an edge $(u, v) : d(u)$ can be added to G, such that $\sigma(u) + d(u) \leq \sigma(v)$ is guaranteed.

Given a time-valid schedule σ, there are alternative choices for start time assignment $\sigma(u)$ to a task u. We extract these available time slots as slacks of tasks. Slack is a measure of how much a task can be delayed without invalidating the schedule.

Definition 6 (Constraint slack Δ^c_σ) Given a time-valid schedule σ computed from a constraint graph $G(V, E)$,
(1) $\forall$ edge $(u, v) : \delta \in E$, the *edge slack* of edge (u, v) is defined as $\Delta^c_\sigma(u, v) = \sigma(v) - \sigma(u) - \delta$.
(2) For any task u represented by a vertex $u \in V$, the *constraint slack* of u is the minimum among all edge slacks of u's outgoing edges, that is, $\Delta^c_\sigma(u) = min(\Delta^c_\sigma(u, v))$, $\forall$ vertex v such that $(u, v) \in E$.
(3) If u does not have any outgoing edges, $\Delta^c_\sigma(u) = \tau_\sigma - \sigma(u) - d(u)$.

Lemma 2 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 that the start time of task u is delayed until another time $\sigma'(u)$ within its constraint slack $\Delta^c_\sigma(u)$, that is $0 < \sigma'(u) - \sigma(u) \leq \Delta^c_\sigma(u)$, for a specific task u.

The constraint slack of a task u defines the maximum time unit by which it can be delayed without violating any timing constraint. It is calculated by the outgoing edges from u. If there is no outgoing edges, u can be delayed all the way until it completes at the finish time of the schedule. Such a delay will maintain the satisfaction of all timing constraints, but it may introduce new resource conflicts.

Definition 7 (Resource slack Δ^r_σ) Given a time-valid schedule σ computed from a constraint graph $G(V, E)$, for any task u represented by a vertex $u \in V$,
(1) If $\exists$ a task v that is mapped to resource $r(u)$ and v is scheduled after u, task u's *resource slack* is defined as $\Delta^r_\sigma(u) = min(\sigma(v)) - \sigma(u) - d(u)$, $\forall v$ such that $r(v) = r(u)$ and $\sigma(v) > \sigma(u)$.
(2) If such v does not exist, $\Delta^r_\sigma(u) = \tau_\sigma - \sigma(u) - d(u)$.

Lemma 3 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 the start time of task u is delayed until another time $\sigma'(u)$ within its resource slack $\Delta^r_\sigma(u)$, that is $0 < \sigma'(u) - \sigma(u) \leq \Delta^r_\sigma(u)$, for a specific task u.

The resource slack of a task u represents the vacant time slots between u's completion and the start of the next task on resource $r(u)$. If u is the last task scheduled on $r(u)$, then it can be delayed all the way until it completes at the finish time of the schedule. The new schedule remains time-valid if the delay on u does not exceed both its constraint slack and resource slack.

Definition 8 (Slack Δ_σ) Given a time-valid schedule σ computed from a constraint graph $G(V,E)$, for any task u represented by a vertex $u \in V$, its *slack* is defined as the minimum of its constraint slack and its resource slack, $\Delta_\sigma(u) = min(\Delta^c_\sigma(u), \Delta^r_\sigma(u))$.

Lemma 4 (Slack-bounded time-validity) If a schedule σ is time-valid, then a modified schedule σ' is also time-valid if it is identical to σ, except that the start time of task u is delayed until another time $\sigma'(u)$ within its slack $\Delta_\sigma(u)$, that is $0 < \sigma'(u) - \sigma(u) \leq \Delta_\sigma(u)$, for a specific task u.

Given a time-valid schedule, Lemma 4 allows some tasks to be delayed while yielding new schedules that are also time-valid. The slack properties of tasks form the basis of our power-aware scheduling algorithms for power/performance trade-offs.

4.2 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 9 (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, $P_\sigma(t) = \sum p(u), \forall$ task $u \in T$ such that $\sigma(u) \leq t \leq \sigma(u) + d(u)$.
(2) The *total energy* of σ is the integral of its power profile over time, that is, $E_\sigma = \int_0^{\tau_\sigma} P_\sigma(t)dt$.

Definition 10 (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. The scheduler should make the best effort to meet the min power goal, in order to fully utilize free power such as solar, as well as to control the amount of jitter in power profile.

Definition 11 (Power spike, power gap) Given a schedule σ with its power profile $P_\sigma(t)$, and power constraints P_{max} and P_{min},
(1) At any given time t_1, if the power profile $P_\sigma(t_1)$ exceeds max power constraint, that is, $P_\sigma(t_1) > P_{max}$, then the power profile at time $t1$ is called a *power spike*.
(2) At any given time t_2, if the power profile $P_\sigma(t_2)$ is below min power level, that is, $P_\sigma(t_2) < P_{min}$, then the power profile at time $t2$ 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 12 (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 5, and
(2) Its power profile does not exceed max power constraint, that is, $P_\sigma(t) \leq P_{max}$, for $0 \leq t \leq \tau_\sigma$.

Definition 12 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 13 (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} reprsenting free power level,
(1) 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},

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

(2) The *energy cost* is the integral of the power cost function,
$Ec_\sigma(P_{min}) = \int_0^{\tau_\sigma} Pc_\sigma(P_{min}, t)dt$.

Definition 14 (Min power utilization $\rho_\sigma(P_{min})$) Given a time-valid schedule σ with a min power constraint $P_{min} > 0$ reprenting 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, $\rho_\sigma(P_{min}) = \frac{E_\sigma - Ec_\sigma(P_{min})}{P_{min} \times \tau_\sigma}$.

We do not limit the power and energy costs and min power utilization in Definitions 13 and 14 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 non-renewable 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.

4.3 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 our *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 on 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. 2 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.

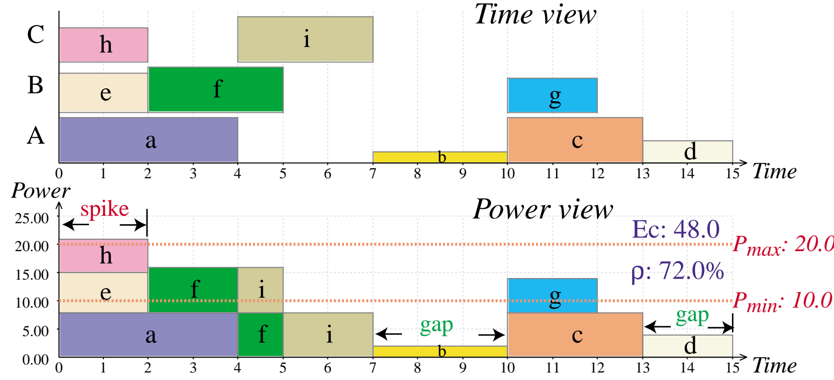

Figure 2: Power-aware Gantt chart of a time-valid schedule

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 order-

ing 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. 3. It is a extension to a previous serialization algorithm [2]. G is the constraint graph for the scheduling problem. *anchor* is the source vertex that is used in 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 start time of candidate 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 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 to the algorithm starts from *anchor* as the first candidate. Then the algorithm is recursively invoked at each step when a new candidate is selected. A time-valid schedule is returned when all vertices are scheduled.

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. 2 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.

5.2 Algorithm for max power scheduling

The approach to meeting max power constraint is to eliminate the power spikes of a time-valid schedule computed by the previous step. The algorithm is shown in Fig. 4. 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 delayed 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.

```
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(c) := La
    while (C ≠ ∅) do
        v := one topological successor of C
        C := C − {v}
B:      foreach u ∈ C do
            if u ∉ v's successors
                then add u to v's successors
            if (r(c) = r(u)) then
                serialize u after c
        w := the most recently scheduled task, such that (r(w) = r(v))
        if (w ≠ nil) then
            serialize v after w
        σ = TimingScheduler(G, anchor, v)
        if (σ ≠ FAIL) then
            return σ with σ(c) := La
        undo added edges to G since step B
    return FAIL
```

Figure 3: 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 by slack Δ_σ
        power := P_σ(t)
        reschedule := FALSE
        while (power > P_max or reschedule = TRUE) do
B:          repeat
                v := EXTRACT MAX(S)
                if (reschedule = FALSE and Δ_σ(v) = 0) then
                    reorder tasks in S by constraint slack Δ_σ^c
                    reschedule := TRUE
                delay v by some time units (heuristically determined)
                power := power − p(v)
                S := S − {v}
            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 4: Algorithm for max power scheduling

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 orders of tasks which would take exponential orders of total number of tasks. Therefore, some heuristics must be applied. The badly chosen tasks could have several impacts. 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 slacks Δ_σ(Definition 8), and then 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 constraint slacks Δ^c_σ(Definition 6). Tasks with larger 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 is selected to be delayed, the second question is by how long it should be delayed, which is referred to as the *delay distance*. To delay a task u based on an existing schedule σ, we add an edge from *anchor* to u, with positive weight t' as the lower bound on its new start time. Therefore, the delay distance is $t' - \sigma(u)$. Clearly, making a small delay distance is not efficient. On the other hand, we do not expect the delay 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 delay distance to the execution time of the task. In addition, in case (1) where the selected task u has some slack, the delay distance is further bounded by its slack $\Delta_\sigma(u)$. According to the slack-bounded time-validity (Lemma 4), if the delay distance of u is less than its slack, the new schedule is still time-valid. 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* . In case (2), the selected task u has some constraint slack but no resource slack, the delay distance is further bounded by its constraint slack $\Delta^c_\sigma(u)$, 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 u is locked by adding two edges to graph G, a forward edge $(anchor, u) : \sigma(u)$, and a backward edge $(u, anchor) : -\sigma(u)$. As a result, task u is forced to start at time $\sigma(u)$ by the SINGLE SOURCE LONGEST PATH algorithm. These locks are especially meaningful to case (3). When the scheduler

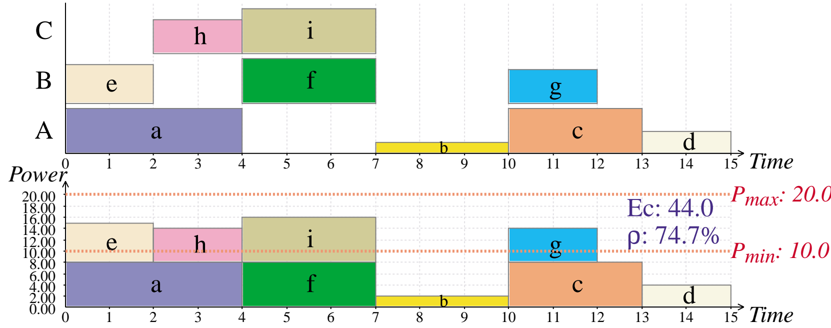

Figure 5: A valid schedule after max power scheduling

delays a task u 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), if the delayed task u has an outgoing backward edge (u,v) such that task v is scheduled before t, the delay to u will force v to be also delayed. In fact, an attempt to remove a power spike starting at time t by delaying a task u may cause a new power spike before t. Such a result 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.

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. 2 does not satisfy the max power constraint. Fig. 5 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. 6. 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 to and delay some tasks scheduled before t to fill this power gap. These tasks must have enough slacks to be delayed until t such that the new sched-

```
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 u ∈ S such that Δ_σ(u) ≥ t − σ(u) − d(u) do
                    σ' := σ
B:                  delay u some time units such that u 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 6: Algorithm for min power scheduling

ule 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 cancelled and the previous schedule is restored.

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 u will change the slacks of other tasks that are constrained by u, 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 u, rather than just starting u at t. It is difficult to determine the "best" time slot for rescheduling u 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 u at t, finishing u at the end of the power gap starting from 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.

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 delayed 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. 7 shows a better schedule that improves on the valid schedule in Fig. 5. 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.

Figure 7: The improved schedule after min power scheduling

Figure 8: Constraint graph of the Mars rover

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. 8. 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 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. 9, 10 and 11 show the results for three cases after applying power-aware scheduling algorithms. Fig. 9 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

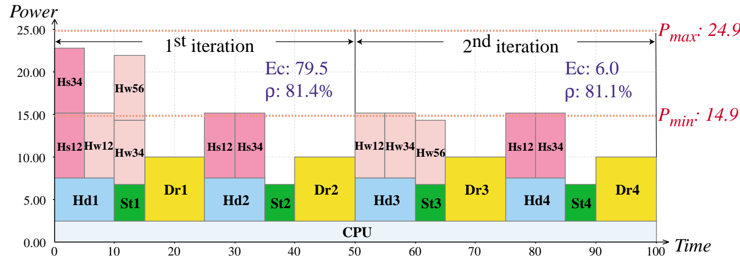

Figure 9: Schedule for the best case

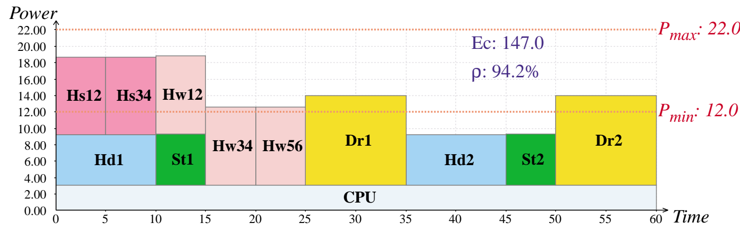

Figure 10: Schedule for the typical case

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

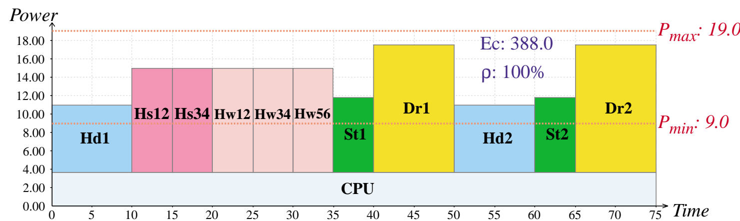

Figure 11: Schedule for the worst case

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 to the schedules

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 of existing schedule to power-aware schedules under a mission scenario

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 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. 12 highlights the property of the power-aware scheduler in a geometrical view.

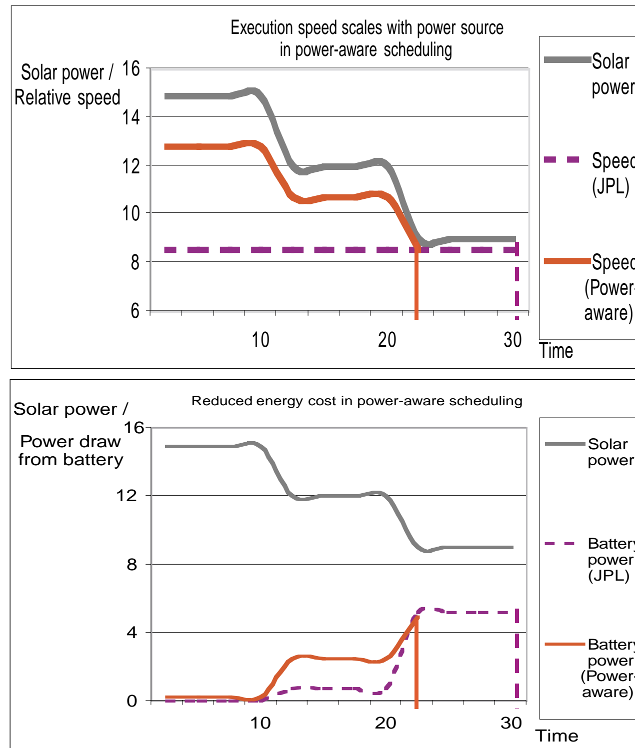

Figure 12: Adaptative speedup in power-aware scheduling

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.

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.

Bibliography

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

[2] 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.

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

[4] 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.

[5] 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, 1999.

[6] 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.

[7] 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.

Acknowledgement

This work represents a collaboration between the University of California at Irvine and the NASA/Cal Tech Jet Propulsion Laboratory. Special thanks to Dr. N. Aranki, Dr. B. Toomarian, Dr. M. Mojarradi and Dr. J. U. Patel at JPL and Kerry Hill at AFRL for their discussion and assistance.