Power-Aware Scheduling under Timing Constraints for Mission-Critical Embedded Systems

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

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 high performance when required. This paper presents a new scheduling technique for supporting the design and evaluation of a class of power-aware systems in mission-critical applications. It satisfies stringent min/max timing and max power constraints. It also makes the best effort to satisfy the min power constraint in an attempt to fully utilize free power or to control power jitter. Experimental results show that our scheduler can improve performance and reduce energy cost simultaneously compared to hand-crafted designs for previous missions. This tool forms the basis of the IMPACCT system-level framework that will enable designers to explore many power-performance trade-offs with confidence.

1. INTRODUCTION

This paper investigates key issues in power management for mission-critical systems. We use the NASA/JPL Mars Pathfinder rover [1] as our motivating example. It features several interesting properties that were not adequately handled by previous work. First, such a system must be designed to be *power-aware*, rather than *low-power*. Second, the power management decisions must 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, rather than power-aware. We believe that the crucial difference between power-aware and low-power systems is that power-aware systems must make the best use of the available power, and they subsume low-power as a special case.

In the Mars rover case, its designers exploited some of the best low-power design techniques. The rover has two power sources: a solar panel and a non-rechargeable battery. By serializing all tasks, this low-power design enables 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.

DAC 2001, June 18-22, 2001, Las Vegas, Nevada, USA.

A power-aware design can greatly improve the utility of the rover. We observe that the battery is non-rechargeable, and thus the excess solar power would be wasted if not used. In the existing design, the rover follows the same serial schedule regardless of the solar power level. By taking advantage of the free solar power, a rover with more parallelism in its schedule can perform better (more tasks, less time) while saving even more battery energy than the existing low-power design, as validated by our experiments.

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 as performance. That is, the power saving of a given component must be scaled by its percentage contribution in an entire system. Thus, it is critical to identify where power is being consumed in the context of a system.

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 designs with a system-level design tool. A lesson learned from the Mars rover was that, without a tool, designers 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 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 the IMPACCT design framework. The designers input a system-level behavioral specification of the design in terms of communicating processes running on specific execution resources, and constraints on processes and the system. Our scheduler constructs a constraint graph and outputs a power-aware schedule that is in turn fed to another tool that performs architecture-level power optimizations.

Section 2 reviews related work, and Section 3 describes the application example. We present the problem formulation in Section 4 and graph-based scheduling algorithms in Section 5. We discuss experimental results in Section 6.

2. RELATED WORK

Prior works have addressed system-level power minimization while maintaining either satisfactory performance or meeting real-

time constraints. However, these low-power techniques often cannot be directly adapted in power-aware systems.

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 by making the timeout adaptive to the actual usage pattern, or using profiling to help predict the proper time to shutdown and power up subsystems [7, 4, 8]. 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) and 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.

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 [6]. Such techniques also have limitations. First, they are CPU schedulers that minimize CPU power, whereas our power managers 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, these schedulers do not handle constraints on power.

We present *power-aware scheduling* with several new features. First, the scheduler must handle both timing and power stringently as hard constraints. It is unlike previous work that treats them as desirable by-products but cannot always make strong guarantees. Second, domain-specific knowledge about power sources, battery models, and other operating conditions must be expressible in terms of supported types of constraints on both 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]. The max power constraint tracks the budget imposed by the power sources. It constrains the system-level power curve under a budgeted level. The min power constraint, 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 energy from free sources that cannot be stored should be utilized greedily. Another motivation is to control the jitter in the system-level power curve to improve battery usage. A power-aware scheduler should satisfy the rigorous min/max timing constraints and the max power budget, while making the best effort to meet the min power goal [5].

3. MOTIVATING EXAMPLE

The NASA/JPL Mars rover [1] is our motivating example for demonstrating the effectiveness and applicability of our power-aware scheduler. The rover travels between different target locations on Mars surface to perform scientific experiments and shoot images. Its power sources consist of a non-rechargeable battery and a solar panel. The life-time of its mission is limited by the amount of remaining battery energy. Since the temperature on Mars surface can be as low as $-80°$C, the rover must heat its motors periodically as it drives them to move. Thus, mechanical and thermal subsystems are the major power consumers.

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 of the Mars rover

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 sources and consumers of the Mars rover

We construct a high-level representation that includes the mechanical and thermal subsystems, as well as different energy sources. We also focus on a typical scenario when the rover is traveling. Each time the rover drives its six wheels for a full rotation to move one step, which is about 7cm in distance. Before driving the wheels, it must first detect any obstacles on its way and choose a safe angle to turn. Then it turns itself to the right direction using the four steering motors. Finally, the six wheel motors are driven. Therefore, hazard detection, steering, and driving must operate in sequence. Other constraints on the mechanical operations are summarized in Table 1. We assume the power consumption of tasks varies with environmental temperature that tracks the sunlight intensity, and 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 9W at dusk. The max power constraint is equal to the available solar power plus 10W maximum battery power output. We also extract the solar power level as the min power constraint to distinguish the free power from the costly power. Table 2 illustrates the power sources and consumers in three cases.

4. PROBLEM FORMULATION

Our problem formulation is based on an extension to a constraint graph used in a previous time-driven scheduling problem [2]. We sketch the key concepts in our model as follows. Section 4.1 reviews the constraint graph formulation and slack properties. Section 4.2 defines power properties of the scheduling problem by applying 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. A detailed discussion and formal definitions to all terms and concepts can be found in [5].

4.1 Constraint graph and slack properties

The input to the power-aware scheduling tool is a *constraint graph* $G(V,E)$, where the vertices V represent tasks, and the edges $E \subseteq V \times V$ represent timing constraints in a form of min and max timing separation [2].

Each vertex $v \in V$ has three attributes, $d(v)$, $p(v)$ and $r(v)$, where $d(v)$ corresponds to the *execution delay* of task v, $p(v)$ is its *power consumption*, $r(v)$ is the *execution resource* onto which task v is mapped. We assume the scheduler is non-preemptive so that each task v has a bounded execution delay $d(v)$. We also assume its

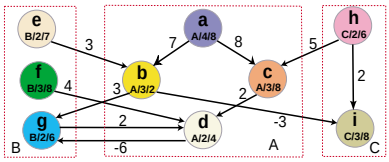

Figure 1: Constraint graph of a scheduling problem

power consumption is an exact value $p(v) \geq 0$. As a result, its energy consumption is $d(v) \times p(v)$. In practice, the power consumption can be either in the form of (min, typical, max), or a function over time. Since our formulation can be extended to handling these cases, we will assume a single value to simplify the discussion. The resource mapping function $r : V \rightarrow R$ maps all tasks 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. If two tasks u and v are mapped to the same resource ($r(u) = r(v)$), then u and v must be *serialized* by the scheduler to eliminate resource conflicts.

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

A *schedule* σ assigns a start time $\sigma(v)$ to each task $v \in V$. It has a *finish time* τ_σ when all tasks complete their execution. Schedule σ is called *time-valid* if all the start time assignments do not violate any timing constraints, and tasks that share the same resource are serialized. Given a time-valid schedule σ, there are alternative time slots for start time assignment $\sigma(v)$ to a task v, which is defined as slacks. Task v's *slack* $\Delta_\sigma(v)$ can be computed from σ and vertex v's outgoing edges in constraint graph G [5]. If the scheduler delays task v until another times slot within its slack $\Delta_\sigma(v)$, the new schedule remains time-valid.

4.2 Power properties of a schedule

A schedule σ has a *power profile* function of time $P_\sigma(t), 0 \leq t \leq \tau_\sigma$ representing the instantaneous power consumption during the execution of σ. The power profile is constrained by two parameters: P_{max}, P_{min}, such that $P_{max} \geq P_\sigma(t) \geq P_{min} \geq 0$. The *max power constraint* P_{max} specifies the maximum budget of supply power that can be provided to support task execution. The *min power constraint* P_{min} specifies the level of power consumption to maintain a preferred level of activity.

The max power constraint is a hard constraint. At any given time t, the value of the power profile function $P_\sigma(t)$ must not exceed P_{max}. Schedule σ is called *power-valid* (or simply, *valid*) if it is time-valid and its power profile does not exceed the max power constraint. Otherwise, the time interval $[t_1, t_2]$ where $P_\sigma(t) > P_{max}$ ($t_1 \leq t \leq t_2$) is called a *power spike*. The min power constraint violation is called a *power gap* at a time interval $[t_1, t_2]$ such that $P_\sigma(t) < P_{min}$ ($t_1 \leq t \leq t_2$). However, we treat the min power constraint as a soft constraint.

The max and min power constraints form some new properties on power/performance trade-offs in power-aware systems. In cases where the min power constraint P_{min} represents free power level, the energy drawn from the non-renewable energy sources is defined as the *energy cost* $Ec_\sigma(P_{min})$ of a schedule σ. We also define the *energy utilization of min power level* $\rho_\sigma(P_{min})$ as the ratio of energy drawn from free source over total available free energy [5]. These new properties distinguish between costly power and free power. Any power consumption below the free power level does not contribute to the energy cost on non-renewable energy sources, and therefore should be utilized maximally. The conventional power or energy minimization is simply a special case, where $P_{min} = 0$.

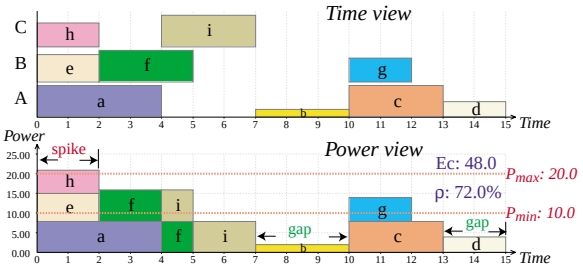

Figure 2: Power-aware Gantt chart of a time-valid schedule

4.3 Power-aware Gantt chart

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 corresponding power properties.

In the time view for a schedule σ computed from a constraint graph $G(V,E)$, the execution of a task $v \in V$ is represented by a bin beginning at start time $\sigma(v)$ horizontally and whose length corresponds to its execution delay $d(v)$. We scale the vertical size of the bin to denote power consumption $p(v)$. As a result, the area of the bin indicates the energy expenditure of task v. Each execution resource $R_i \in R$ takes one row denoted by R_i. All tasks that are mapped to this resource, that is, $\forall v \in V$ such that $r(v) = R_i$, are displayed in row R_i in timing order. Timing constraints and slacks can also be intuitively visualized by selectively annotating the bins.

The power view is obtained by collapsing all bins in the time view to the lowest horizontal axis, which yields the power profile $P_\sigma(t)$. 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 energy cost vs. free power usage are clearly separated; and power properties such as energy cost $Ec_\sigma(P_{min})$ 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 for exploring 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. SCHEDULING ALGORITHMS

Given a scheduling problem, the goal of the power-aware scheduler is to find a schedule σ that is both time-valid and power-valid with minimum power gaps. We use 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 (Section 5.1). Second, the max power

```
TimingScheduler(Graph G, vertex anchor, vertex c)
    L(c) := SINGLE SOURCE LONGEST PATH(G, anchor)
    if (positive cycle found) then return FAIL
    if (Succ[c] = ∅) then return σ with σ(c) := L(c)
    while (Succ[c] ≠ ∅) do
        v := extract one vertex from Succ[c]
B:      Succ[v] := Succ[v] ∪ Succ[c]
        foreach (vertex u that is not traversed) do
            if (r(u) = r(c)) then serialize u after c
        σ = TimingScheduler(G, anchor, v)
        if (σ ≠ FAIL) then return σ with σ(c) := L(c)
        undo changes to G since step B
    return FAIL
```

Figure 3: Algorithm for timing scheduling

constraint is applied to the time-valid schedule to remove power spikes using slack-based heuristics (Section 5.2). 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 in Section 5.3.

5.1 Algorithm for timing scheduling

The time-constrained scheduling algorithm is shown in Fig. 3. It is an extension to a previous serialization algorithm [2]. G is the constraint graph. *anchor* 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 topological successors $Succ[c]$. Tasks that share the same resources are serialized by adding edges between vertices. If these additional edges produce any positive cycles in the graph, they are 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. There are one power spike and several power gaps left for the remaining steps.

5.2 Algorithm for max power scheduling

The algorithm shown in Fig. 4 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 by adding extra edges to G so that the power curve is below P_{max}. A valid schedule σ is found if there is no power spike in σ. The time-validity of σ is always guaranteed recursively. 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 some tasks already delayed have been incorrectly chosen.

We propose slack-based heuristics to select and delay tasks for spike elimination. First, a slack-based ordering function is used to order simultaneous tasks. When a power spike is detected at time t, the algorithm orders active tasks at t by their slacks Δ_σ, 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 selected first. The algorithm continues selecting tasks to delay until the power spike at t is removed. (2) If the power spike cannot be removed until all remaining tasks have zero slacks, tasks are randomly selected. After a task is selected, the second question is by how long it should be delayed, which is referred to as the *delay distance*. We 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 v has some slack, the delay distance is further bounded by its slack $\Delta_\sigma(v)$ so that the new schedule is still time-valid. In cases (2), which eliminates a power spike at the cost of introducing new timing violations, some significant timing adjustment to the schedule is required by asserting the Boolean variable *reschedule*. Finally, after enough tasks are delayed and the power spike at t disappears, we lock the start time of the remaining tasks by adding extra edges to G. These locks are especially meaningful to case (2). Since all remaining tasks have zero slacks, they should not be delayed subsequently. However, if delays to these tasks are necessary for a valid schedule, the algorithm will fail in its next recursion 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 scheduler may not 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 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

Min power scheduling reduces the energy cost by improving min power utilization for a valid schedule. The algorithm is shown in Fig. 6. It has four parameters: graph G, vertex *anchor*, power constraints P_{max} and P_{min}. A valid schedule σ is obtained from the max power scheduler. If σ already has full min power utilization, then no further improvement is necessary. Otherwise, the algorithm finds a power gap at time t and delays some tasks started before t to fill this power gap. These tasks must have enough slacks to be delayed until t such that the new schedule is time-valid. The algorithm also checks whether the new schedule is power-valid, and whether it has a higher min power utilization. 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 scanning the schedule until no further improvement can be found. 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 scheduler tolerates the existence of power gaps after it makes the best efforts to remove them.

To find an "optimal" schedule whose energy cost is 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 should scan the schedule multiple times. This is because, after delaying some tasks in the previous scan, either new power gaps or new tasks to fill other power gaps can be found if the schedule is scanned again. Moreover, the order to visit the power gaps will lead to different final schedules due to different partial orders. This suggests that better

```
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
        reschedule := FALSE
        while (P_σ(t) > P_max or reschedule = TRUE) do
B:          repeat
                v := EXTRACT MAX(S)
                if (Δ_σ(v) = 0) then reschedule := TRUE
                delay v by some time units (heuristically determined)
            until (P_σ(t) ≤ P_max)
            if (reschedule = TRUE) then
                lock start times of remaining 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

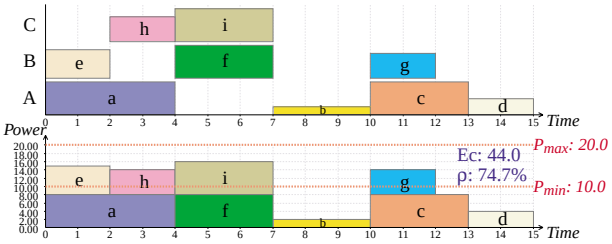

Figure 5: A valid schedule after max power scheduling

```
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
    σ_old := σ
    while (TRUE) do
        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 (v ∈ S such that Δ_σ(v) ≥ t − σ(v) − d(v)) do
B:                  σ_new := delay v some time units such that v is active at t
                    if (σ_new is valid and ρ_σnew(P_min) > ρ_σ(P_min)) then
                        σ = σ_new
                        if (ρ_σ(P_min) = 1) then return σ
                    else
                        undo added edges to G in step B
        if (σ = σ_old) then return σ
    return σ
```

Figure 6: Algorithm for min power scheduling

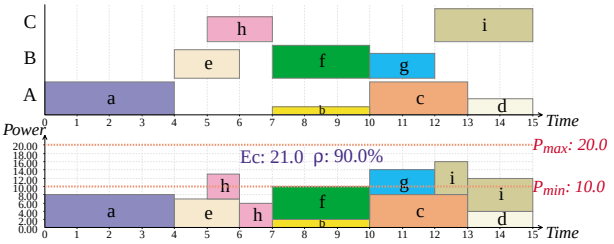

Figure 7: The improved schedule after min power scheduling

schedules could be found if the schedule can be scanned in various orders in time dimension, e.g. incremental order, reverse order, or random order. Finally, when a task v is selected to fill a power gap at t, we consider alternative time slots to reschedule v by heuristics. It is difficult to determine the "best" time slot since the choice alters not only the power profile but also the slacks of some other tasks. Some available heuristics are: starting v at t, finishing v at the end of the power gap beginning at t, or a randomly chosen time slot. In practice, we scan the schedule multiple times while altering some of the heuristics during each scan and take the best results.

Fig. 7 shows a better schedule that improves on the valid schedule in Fig. 5. In fact, the same schedule can be directly applied to all cases with a range of constraints where $P_{max} \geq 16, P_{min} \leq 14$, without recomputing a schedule for each case. This feature makes our statically computed power-aware schedules adaptable to a runtime scheduler that schedules tasks according to the dynamically changing constraints imposed by the environment.

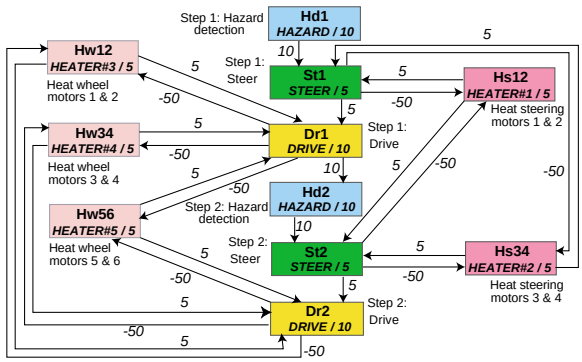

Figure 8: Constraint graph of the Mars rover

6. EXPERIMENTAL RESULTS

This section presents scheduling results for the Mars rover's 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. 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. Each task is denoted with the resource mapping and the execution delay. The power consumption is not shown since it varies in different cases. During each iteration of the schedule, the rover moves two steps (14cm). Fig. 9, 10 and 11 show the schedules (power views only) for three cases after applying power-aware scheduling algorithms. Fig. 9 gives first two iterations in the best case. To utilize the available free energy, we manually unroll the loop and insert two heating tasks to improve solar energy utilization. Therefore the second iteration can be repeated with less energy cost. Since 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

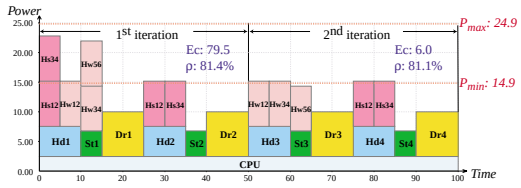

Figure 9: Schedule for the best case

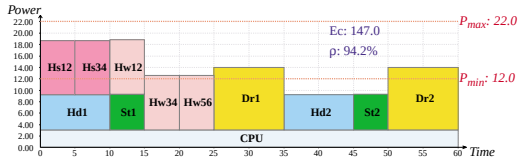

Figure 10: Schedule for the typical case

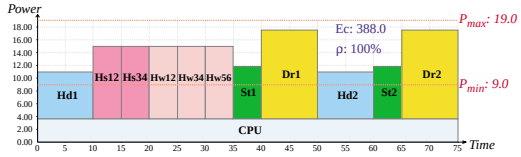

Figure 11: Schedule for the worst case

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: Performance and energy cost of 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***
					Improvement	***33.3%***	***32.7%***

Table 4: Comparison to schedules under a mission scenario

a fixed, fully serialized schedule, without tracking available solar power and power consumption in different conditions. The existing schedule is identical to our power-aware schedule in the worst case with the lowest power budget. The fundamental difference is that, our schedule is completely constraint-driven; whereas the existing solution is hardwired manually.

We use the performance of the schedule (the inverse of the finish time τ_σ) and the energy cost $Ec_\sigma(P_{min})$ to the non-rechargeable battery as the metrics, and compare JPL's schedule and our power-aware schedules in Table 3. The existing scheme only schedules for the worst case; while in other cases, solar energy is underutilized 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 between performance and energy cost, we apply our schedules to a scenario where the available solar power varies over time.

Suppose the mission is to travel to a target location in a distance of 48 steps. The mission starts with maximum solar power at 14.9W. Then, it drops 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% (20 steps) in the next 10 minutes, leaving the remaining 8% (4 steps) in the worst case for only 2.5 minutes. Since our schedules accelerate the execution at the best and typical cases, the rover can finish the mission earlier before having to work in the costly worst case. The analysis to this case study in Table 4 shows our schedules win both on performance (33.3% speed-up) and energy savings (32.7% reduction) considerably.

7. CONCLUSION

Power-aware design has become an important issue in mission-critical systems that require the best use of available power sources while delivering high performance. We target the scheduling algorithms to embedded systems with variable power constraints and heterogeneous power consumers, as well as different energy sources with different costs, from costly to free. In these systems, power-aware techniques have potentials for both performance improvement and energy savings.

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. This incremental approach applies heuristics to solve the constraints with different properties. The case study with a real application shows that our power-aware method is capable of improving performance while saving expensive energy.

Acknowledgement

This research was sponsored by DARPA under contract F33615-00-1-1719. It 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.

REFERENCES

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

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

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

[4] E.-Y. Chung, L. Benini, and G. De Micheli. Dynamic power management using adaptive learning tree. In *Proc. ICCAD*, pages 274–279, 1999.

[5] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints and slack analysis for mission-critical embedded systems. Technical Report IMPACCT-01-03-01, University of California, Irvine, March 2001.

[6] T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *Proc. ISSS*, pages 24–29, 1999.

[7] T. Simunic, L. Benini, and G. De Micheli. Event-driven power management of portable systems. In *Proc. ISSS*, pages 18–23, 1999.

[8] 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.