# 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
Irvine, CA 92697-2625 USA
{jinfengl, chou, nader, kurdahi}@ece.uci.edu

Dept. of Electrical & Computer Engineering
University of California at Irvine

## 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 be able to 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 will compute a schedule that satisfies stringent min/max timing and max power constraints at all times. Furthermore, it will also make the best effort to satisfy its min power constraint in an attempt to fully utilize free solar power or to control power jitter. Experimental results show that our automated techniques yield designs that improve performance and reduce energy cost simultaneously compared to handcrafted designs used in previous missions. This tool forms the basis of a system-level framework that will enable designers to aggressively explore many more power-performance tradeoffs 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 life.

This paper investigates key issues in power management in mission-oriented systems. Our motivating example comes from the NASA Mars Pathfinder rover developed at JPL [7]. 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 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. 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 Pathfinder case, its designers constructed a low-power design. It incorporated some of the best low-power design techniques at all levels of abstraction. The Pathfinder rover itself has two power source: a solar panel and a 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 in 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 rover was that without a tool, the designer has no option 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 this larger design framework. The designer inputs a high-level

*All appropriate organizational approvals for the publication of this paper have been obtained. If accepted, the author(s) will prepare the final manuscript in time for inclusion in the Conference Proceedings and will present the paper at the Conference.

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 profiles to help predict the proper time to shutdown and powerup subsystems. [5, 3, 6]

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 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. [4]

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 extremely difficult to tune the voltage or frequency scale of such a processor. As a result, the risk of missing deadlines may be high, even if the 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]. Max power would track the budget imposed by the current power sources. Min power constraints, strictly speaking, may be a 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 the power-aware scheduling techniques, we choose 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 supply comes from a non-rechargeable battery array 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 among different target locations before experiments and imaging can be performed. Since the temperature on Mars surface is as low as $-80°$C, driving in low temperature requires more power consumption because the motors must be heated from time to time. 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. To start a single step of movement, it must detect any obstacles on the moving direction and choose a safe angle for the next step. Then the four steering motors are started to turn to the right direction. Finally, six wheel motors are driven to perform a single step of movement. Therefore, hazard detection - steering - driving must operate in sequence. The other set of timing constraint 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 suppose that the temperature is closely related to the sunlight density that can be measured by power output from solar panel. In order to examine how the power-aware scheduling techniques handle different constraints, we investigate three cases: best case, solar power output is 14.9W at noon time; typical case, when solar power output is 12W; and worst case, solar power output is 9W when the sun is to go down. The maximum supply power is limited by the threshold of battery power output. We assume the maximum battery power draw is 10W. Therefore, in all cases, the rover can be safely operated only

| Resource | Duration (s) | Power (W) | | |
|---|---|---|---|---|
| | | -40 degC | -60 degC | -80 degC |
| Solar power | | 14.9 | 12 | 9 |
| Battery | | 10 max | 10 max | 10 max |
| Heat one motor | 5 | 5.1 | 6.2 | 7.5 |
| Heat two motors | 5 | 7.6 | 9.5 | 11.3 |
| Drive | 10 | 7.5 | 10.9 | 13.8 |
| Steer | 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

if its instantaneous power consumption is less than available solar power plus 10W maximum battery power output, which constitutes the max power constraint. Table 2 illustrates the power sources and consumers.

The purpose 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 handcrafted. 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 cases.

By introducing power-aware scheduling, not only could we improve performance, but also save non-rechageable 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 at the cost of the 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 another time-driven scheduling problem. Section 4.1 reviews the base formulation and then describes our extensions to handling power constraints. Section 4.2 presents a way of viewing the time/power scheduling problem as a two-dimensional constraint problem by drawing analogies from the Gantt chart.

### 4.1 Input representation

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 relationships. In addition, the input includes a function $r(v)$ from the operations to the execution resources; a power consumption function $p(v)$ for the estimated power usage. The designer also inputs the minimum and maximum power constraints on the schedule.

#### *4.1.1 Timing constraint graph*

Each vertex has a non-negative weight $d(v)$ corresponding to the duration of execution. We assume all vertices represent tasks with bounded execution delay that are non-preemptible. The scheduler will generate a schedule by assigning a start time $s(v)$ to each vertex $v$.

The edges between pairs of vertices specify timing constraints conjunctively. Each edge $(u,v)$ has a weight $w(u,v)$ that constrains the start times of $u$ and $v$: it requires that the start time of $v$ must be scheduled as least $w(u,v)$ time units after $u$'s start time. More formally, $s(v) - s(u) \geq w(u,v)$. Depending on the weight, the edges are categorized as forward edges and backward edges. An edge $(u,v)$ with a nonnegative weight is called a forward edge, and it expresses a minimum time-separation constraint. Otherwise, an edge $(u,v)$ with a negative weight is called a backward edge, and it is a maximum time-separation constraint from $v$ to $u$ [1].

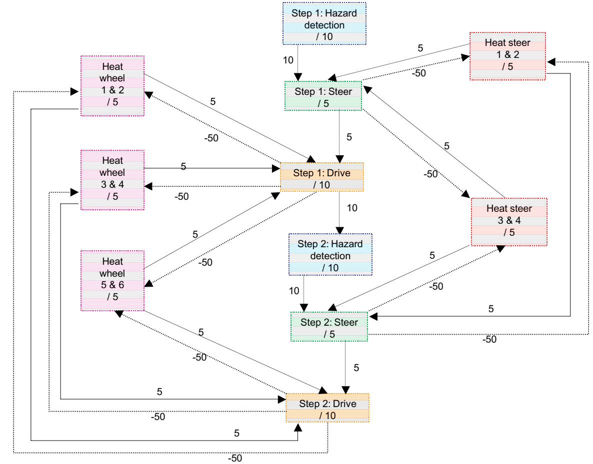


Figure 1: Constraint graph for Mars rover's operations

#### *4.1.2 Resource mapping*

To handle parallel execution resources that consume power, the function $r: V \to \mathbb{Z}$ maps each vertex to a resource ID. Examples of execution resources include not only computing resources such as a embedded microprocessor but also other consumers of power, including mechanical subsystems and heaters. We further assume that if two nodes $u$ and $v$ are mapped to the same resource (formally, $r(u) = r(v)$), then $u$ and $v$ must be serialized in the final schedule.

Figure 1 shows the timing constraint graph for the Mars rover example. It shows tasks that span digital, mechanical, and thermal domains. The CPU process is a constant background task without precedence relationship to any other operations, so it is not included in the graph. To allow more parallelism, there is no restriction on overlapping operations. Unless specified in the timing constraint, no serialization is necessary. Another example is available in Appendix.

#### *4.1.3 Power consumption and power constraints*

We also assume the availability of the power consumption function, $p: V \to \mathbb{R} > 0$, which returns the estimated power consumption by the tasks represented by all the vertices. In practice, the power consumption will be a range or in the form of (min, max, typical), rather than an exact number. Since our formulation can be extended trivially to handle intervals, we will assume a simple number $p(v)$ to simplify the discussion here.

The power usage of the system is constrained by two input parameters, max power and min power. The max power is a hard constraint: at any given moment, the total power consumption by all running tasks must not exceed the max power. The min power is a soft constraint: the scheduler should make the best effort to meet the min power goal. This will control the amount of jitter in power draw, as well as ensuring full utilization of renewable power such as solar.

In the Mars rover example, the amount of available solar power can translate to the min power constraint to ensure full utilization of free power. The max power constraint can be set to solar power plus maximum power draw from battery, although the designer can set a lower max power constraint to explore a more conservative design point.

### 4.2 Output representation

The scheduler computes the tasks that map to time slots as a *schedule*. A schedule $S$ is the set of tuple $\{(v, s(v))\}$, where $v \in V$ in graph $G$, and non-negative integer $s(v)$ is the start time assignment to vertex $v$ in time units. Although start time $s(v)$ could serve as an attribute of vertex $v$, we do intend to separate the results from the input set.

However, the connection between input problem set and results is somehow vague. It is not easy to justify the properties of the results by performance, power consumption, utilization and etc. from the definition of a schedule. We introduce the *power-aware Gantt chart* as a new representation for our power-aware schedule. It is proposed for at least two purposes: it is used by the scheduler as a representation to the results, and it also serves as the underlying model for a visualization tool.

Gantt charts are a common way of presenting schedules visually. The execution of a task is represented by a horizontal bin whose length corresponds to its duration. Timing constraints, and concepts such as scheduling slack in the time dimension, though normally not shown, can also be intuitively visualized by selectively attaching annotation on the bins.

We use the vertical size of the bins to denote power consumption. Note that we already have horizontal size of the bins for the weight of a node in time dimension, the vertical size now represents the weight in the power dimension. After scaling the vertical size of the bin with the a task's power consumption, the area of the bin indicates its energy expenditure, which is combined weight in energy domain. This investigation directly connects the schedule with power characteristics: by collapsing all bins to the lowest horizontal axis, the expected power surge of the schedule, how the power curve varies within or without the max-min range, and the make-up of the power contributors at each time can be clearly visualized.

## 5 Algorithm

The basic idea to use graph algorithms in scheduling problems is as follows. As the algorithm performs a traversal to the graph, the constraints are inspected to qualify the validity of the current partial ordering. When a complete ordering is verified, the algorithm successes with a feasible solution. Otherwise, different orders are attempted until a solution is found or all combinations have failed. This seems to be a straightforward approach to most problems.

As we use the power-aware Gantt chart as the representation of results, some new features arise as the alternative formulations to the problem set. Gantt charts suggest that the power-aware scheduling problem has an analogy to a two-dimensional bin-packing problem. It may be attempting to solve this problem in a bin-packing manner. However, the solution space of such NP-hard problems grows abruptly as the input problem set increases. We have not yet discovered an effective algorithm that can solve the problem by a single run.

One fundamental feature of the native scheduling problem hidden by the graphical representation is that, the constraints in the problem set should not be prioritized evenly as two-dimensional area boundaries in the bin-packing view. The timing constraint is the most critical one that must be taken care of first. If no valid time schedule exists, there is no point to consider power consumption. Thus, the horizontal dimension and vertical dimension in the graphical view are not equal in significance. Therefore, solving the two-dimensional bin-packing problem at the same time becomes a less promising scheme. This suggests an incremental approach.

First, based on precedence relationships in constraint graph, we try to find a schedule that is valid on time dimension. Power constraints and power weights are not used in this step. We extend the existing algorithm to discover a legal schedule from constraint graph. The algorithm is described in Section 5.1.

```
MaxPowerSchedule(Graph G, vertex anchor, MaxPower) {
    schedule := ParallelSchedule(G, anchor, anchor);
    if (schedule = FAIL) return FAIL;
    ts := execution time of schedule;
    for (t := 0; t ≤ ts ; t := t + 1) {
        S := set of active events at t;
        power := power consumption of all events in S;
        ExtendSchedule := FALSE;
        while (power > MaxPower or ExtendSchedule) {
B:          repeat
                v := most slack event in S;
                if (slack(v) = 0) ExtendSchedule := TRUE;
                delay v;
                power := power − p(v);
                S := S − {v};
            until (power ≤ MaxPower or S = ∅);
            if (S = ∅) return FAIL;
            lock start time of all vertices in S;
            schedule := MaxPowerSchedule(G, anchor, MaxPower);
            if (schedule ≠ FAIL) return schedule;
            ExtendSchedule := TRUE;
            Undo added edges since step B;
        } /* inner loop, while(...) */
    } /* outer loop, for(t ...) */
    return schedule;
}
```

Figure 2: Scheduling algorithm for max power constraint

Second, based on a valid schedule on time domain, power weights of vertices and max power constraint is applied to adjust the existing schedule. Section 5.2 explains the max power constraint scheduling algorithm which is close to a bin-packing approach based on power-aware Gantt chart representation. To avoid exhaustive search in the solution space, some heuristics are used to give hints so that more reasonable solutions are examined first.

Finally, given a schedule that meets both timing and max power constraint, we make further adjustments to match the min power constraint in Section 5.3. Similar to step two, some bin-packing heuristics are used. Since the min power constraint is a soft-constraint, the algorithm does not guarantee that power consumption will always exceed min power level, but arranges the power surge to reside within the min-max power range as much as possible.

### 5.1 Algorithm for parallel scheduling in time dimension

Time-constrained scheduling is a straightforward extension to a previous serialization algorithm [1]. Rather than serializing all vertices, the new algorithm only serializes tasks that share the same execution resource. Another modification is that virtual edges are added between vertices across resource boundaries to maintain a topological traversal to the graph. Similar to the previous study, the new algorithm can be proved to always find a valid schedule if one exists.

The algorithm and an example are illustrated in Appendix.

### 5.2 Algorithm for max power constraint scheduling

The approach to meeting max power constraint is similar to reducing the height of bins in a bin-packing method. The algorithm is shown in Figure 2. It has three parameters: graph $G$, vertex *anchor*, and scalar constraint *MaxPower*. The parallel scheduler is always called first to ensure a valid schedule in time domain. The algorithm scans the returned *schedule* to find the first time $t$ when the max power constraint is violated, which is referred to as a *power spike*. To eliminate the spike at this point, several simultaneous vertices are delayed so that the height of the power curve is below

*MaxPower*. To delay a vertex $v$, we add an edge from *anchor* to $v$, with desired positive weight as the start time. The method itself is called recursively after the spike is eliminated by delaying vertices.

A valid schedule is found if the algorithm cannot find any spike in the outer loop. The schedule is returned after the outer loop is completed. If no feasible solution can be found, a failure notice is returned suggesting that either more vertices need to be delayed or the already delayed ones have been incorrectly selected.

The key issue in this algorithm is to properly select vertices to be delayed. In the worst case where the wrong ones are delayed every time, the time complexity of exhaustive enumeration is $O(n! * exp(n))$.

The choice of vertices to delay for spike elimination is important, since badly chosen vertices could not only cause the algorithm to fail, but also extend the schedule time unnecessarily, leading to a poor performance. A "good" choice of vertex should (1) incur minimal additional execution time (2) necessitate minimum rescheduling to find a valid solution. We propose a heuristic for choosing vertices based on slack.

In our context, the *slack* of a vertex is the maximum amount of time by which a vertex can be delayed in a valid schedule without violating any timing constraints. The concept differs from the conventional meaning as the distance of an event to its deadline. Given forward and backward edges, the slack of a vertex can be categorized as forward slack and backward slack. If $(v,u)$ is a forward edge, the forward slack of $v$ referenced by $u$ is defined as $s(u) - s(v) - w(v,u)$. This means if $v$ is delayed by this amount of time unit, the schedule still remains valid since the min constraint still holds. If $v$ has multiple outgoing forward edges, its forward slack is the minimum slack in reference to all target vertices. The backward slack refers to the maximum delay allowed to the vertex without violating max constraints. If vertex $v$ has a backward edge to $u$ with weight $-w(v,u)$, the backward slack in reference to $u$ is $w(v,u) - s(v) + s(u)$, which signifies how far $v$ can be delayed without invalidating this max timing constraint. Similarly, if vertex $v$ has multiple outgoing backward edges, the minimum backward slack is selected. Note that slack is only meaningful with regard to outgoing edges, either forward or backward, of a vertex. Delaying vertex $v$ does not violate any timing constraints specified by incoming edges of $v$. If $v$ does not have any outgoing forward edges or backward edges, the corresponding slack is set to the positive infinite value that implies the vertex is not bounded by relating constraint at all. Finally, the slack of a vertex is the minimum value of its forward slack and backward slack.

At each point when max power is exceeded, the algorithm always delays the vertices with more slack until power consumption is lowered within the safe range. The illustration for such a greedy heuristic is that we preserved the least slack vertices to avoid longer execution time and hope to find a solution more quickly. There are cases where no vertex with non-zero slack is available, or, after having delayed enough vertices to lower down the power curve a valid schedule is not found. This suggests that max power constraint cannot be met without extending the schedule that leads to longer execution time. These cases refer to turning the Boolean variable *ExtendSchedule* to TRUE in the algorithm.

The slack-based heuristics most likely lead to the correct solutions. If there is no vertices with non-zero slack are available, the schedule must be extended. However, there are some undesirable effects in this case. When a non-slack vertex $v$ has to be delayed at time $t$, the schedule could be changed to the vertices starting after $t$, but also the ones before $t$. The vertices that are destinations of forward edges from $v$ may be delayed after $t$. In addition, the vertices with incoming backward edges from $v$ could be also delayed. Such behavior results in a loosened schedule by delaying vertices through backward edges. To prevent this, we could further extend our slack-based sorting procedure so that the vertices

```
MinPowerSchedule(Graph G, vertex anchor, MaxPower, MinPower) {
    schedule := MaxPowerSchedule(G, anchor, MaxPower);
    if (schedule = FAIL) return FAIL;
    ts := execution time of schedule;
    for (t1 := 0; t1 ≤ ts; t1 := t1 + 1) {
        S1 := set of simultaneous events at t1;
        p1:= power consumption of all events in S1;
        if (p1 < MinPower) {
            for (t2 := t1 - 1; t2 ≥ 0; t2 := t2 - 1) {
                S2 := set of simultaneous events at t2;
                p2 := power consumption of all events in S2;
                if (p2 > MinPower) {
                repeat
B:                  v := next candidate in S2;
                    if (v is qualified to be delayed to t1) delay v to t1;
                    S2 := S2 - {v};
                    schedule := MinPowerSchedule(G, anchor, MaxPower, MinPower);
                    if (schedule ≠ FAIL and execution time of schedule ≤ ts)
                        return schedule;
                    Undo added edges since step B;
                until (S2 = ∅);
                } /* if(p2...) */
            } /* inner loop, for(t2...) */
        } /* if(p1...) */
    } /* outer loop, for(t1...) */
    return schedule;
}
```

Figure 3: Scheduling algorithm for min power constraint

with backward edges will be considered last. But this still cannot avoid the problem completely when all candidates are zero-slack on backward. We use a much simpler heuristic. After enough vertices are selected to delay and max power constraint is satisfied, we lock the start time of remaining vertices in the candidate set. The start time of vertex $v$ is locked to $t$ by adding two edges, positive edge $(anchor, v)$ with weight $t$, and negative edge $(v, anchor)$ with weight $-t$. Therefore, these vertices are arbitrarily forced to start at certain time and no further delays can be performed to them unless the extra edges are undone. However, if such delay is mandatory to a feasible solution, the algorithm will return fail in later recursions and the algorithm will free one vertex from the set to make further delay and call the recursion again.

It is notable that in some extreme cases, the max power constraint scheduler may not be able to find a valid schedule even though there exists one. The reason is that when the recursion returns fail, the algorithm does not enumerate all possible combinations in selecting vertices from the candidate set. However, in practice, our heuristics perform very well in steering to a valid solution without unnecessary sacrifice on total schedule time. This is because our slack-based heuristics make strong hints toward the correct direction to a feasible solution. Also, the heuristic to lock the vertices in the candidate set before calling the algorithm recursively can help to reduce the possible ordering in both parallel scheduling algorithm and later recursions of max power constraint scheduler.

### 5.3 Algorithm for min power constraint scheduling

Similar to max power constraint scheduling, the method to satisfy min power constraint is analogous to a bin-packing problem with minimum height requirement. The algorithm is shown in Figure 3. Four parameters are passed to the algorithm: graph $G$, vertex *anchor*, scalar constraints *MaxPower* and *MinPower*. A valid schedule comes from calling max power scheduler at the beginning of the algorithm. The length of *schedule* is not changed due to satisfying min power constraint, which is a soft-constraint. The algorithm checks the schedule to find the first time $t1$ when power consumption is below *MinPower*, which we refer to as a *power*

*gap*. Then the schedule is inspected reversely in the time domain to find $t2$ where no power gap is present. If there is any appropriate vertex at $t2$ to be delayed to fill the gap at $t1$, the algorithm delays the candidate and calls itself recursively. Only delay is needed because the max power scheduler prefers to assign the earliest start time to vertices. Otherwise if no vertex at $t2$ is qualified to fill the gap at $t1$, the inner loop continues to check the schedule backward until it finishes. Then the algorithm proceeds forward to find the next gap.

After the whole schedule is inspected, the outer loop completes. The final schedule is returned as a feasible solution. However, if the recursive call to *MinPowerSchedule* returns a worse schedule, the algorithm will return the better one that has existed before the recursive call. The algorithm returns a failure notice only if max power scheduler cannot find a schedule, when the problem fails on hard-constraints.

The interesting part in this algorithm is again the candidate selection to delayed vertices. As we expect, delaying a vertex to meet min power constraint would (1) bring no extra execution time (2) result minimum change to the existing schedule (3) be beneficial to energy saving. Our heuristics on choosing delayed candidates are still based on slack, as described in the previous algorithm. However, due to the different nature of the max and min power constraints, we apply different policies in this algorithm. We perform more complex inspections to decide whether a vertex $v$ at time $t2$ is qualified to be delayed to fill a power gap at $t1$. First, if $v$ is a non-slack vertex, no delay should be performed since we do not expect a longer schedule time. Second, the slack of $v$ must be at least $t1-t2$. Otherwise moving $v$ to $t1$ will likely result in a longer schedule. Next, if any vertex $u$ in set $S1$, which is the set of simultaneous vertices at $t1$, shares the same resource with $v$, $v$ should not be the candidate, since delaying $v$ to $t1$ will at least force $u$ to be rescheduled, and could lead to an extended schedule. Finally, the delay should be beneficial to energy saving or more balanced power surge. That is, if delaying vertex $v$ at $t2$ to a power gap at time $t1$ is beneficial, there should not become a larger gap at $t2$ afterwards due to the absence of $v$. In the bin-packing perspective, vertex $v$ is to be placed in either $t2$ where the power consumption is $p2-p(v)$, or $t1$ where a power gap $p1$ is present. Vertex $v$ should be positioned to the place where the height of the bin is lower. That is, only if $p2-p(v)>p1$, is the delay a good move.

The running time of the algorithm may grow in some extreme cases. This is because the problem on min power constraint is potentially harder than the other two problems; and we do not try to limit the solution space by partially hardwiring the existing schedule. The extra running time is affordable to this soft-constraint problem. At least we already have a feasible solution that meets all critical constraints in terms of timing and max power restrictions. At this point we see the advantage to break the problem into several steps. We try to solve critical problems first, and if necessary, force some reasonable assumptions in order to find the solution quickly. As far as the hard-constraints are resolved and a feasible solution is provided, we do not mind allowing the algorithm to take more time on relatively non-critical adjustments. In fact, we do apply some other sets of restrictions to effectively limit the NP-complex solution space. Our heuristics only qualify vertices restricted by specific properties to be reordered. In practice, compared with max power constraint scheduler, the additional response time of this algorithm is insignificant in most cases.

Power-aware scheduling results of the Mars rover example can be found in Section 6. Figure 11, 12 and 13 in Appendix illustrate the steps in max and min power scheduling to another example. Two sets of max/min power constraints are examined: 20/10 and 15/5. Our scheduler gives feasible schedules in both cases.

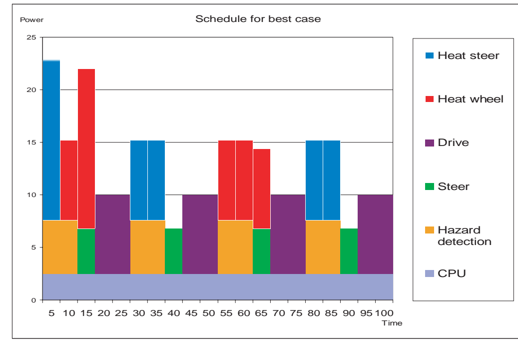


Figure 4: Schedule for the best case

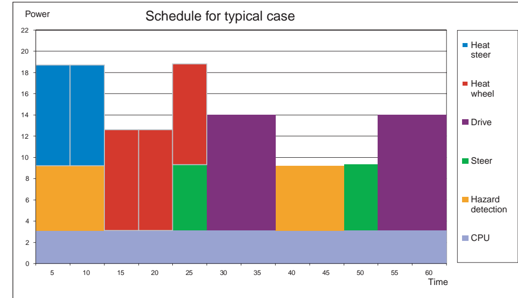


Figure 5: Schedule for the typical case

## 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.

Figure 4, 5 and 6 show the results for three cases after applying power-aware scheduling algorithms. Figure 4 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 processes to improve loop efficiency on better solar energy utilization. Therefore the second iteration can be repeated without too much energy cost. In other cases only one iteration is shown since loop unrolling is not necessary. In the best case, because 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 processes are serialized. In the worst case, a tight power budget forces all operations to be serialized, leading to a slow schedule.

It is necessary to look at the existing schedule in practice. To avoid exceeding max power supply, JPL gave a serialized schedule that is fixed in all situations, regardless of available solar power and

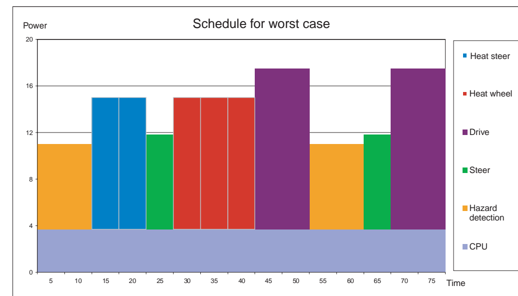


Figure 6: Schedule for the worst case

| Solar power (W) | Battery energy (J) | Solar energy (J) | % of solar energy | Time (s) | Moving distance |
|---|---|---|---|---|---|
| 14.9 | 0 | 672.5 | 60% | 75 | 2 steps - 14cm |
| 12 | 55 | 817 | 91% | 75 | 2 steps - 14cm |
| 9 | 388 | 675 | 100% | 75 | 2 steps - 14cm |

Table 3: Performance of the rover under existing schedule

| Solar power (W) | Battery energy (J) | Solar energy (J) | % of solar energy | Time (s) | Moving distance |
|---|---|---|---|---|---|
| 14.9 | 79.5 / 6 | 534 | 70% | 50 | 2 steps - 14cm |
| 12 | 147 | 679 | 94% | 60 | 2 steps - 14cm |
| 9 | 388 | 675 | 100% | 75 | 2 steps - 14cm |

Table 4: Performance of the rover under power-aware schedules

power consumption in different conditions. The existing schedule happens to be exactly the same as our schedule for the worst case. But the underlying distinction is that, our schedule is completely constraint-driven; the existing solution hardwires a serialized approach without awareness of unsteady power constraints. The performance and energy cost of our schedules and existing schedule are compared in Table 3 and Table 4.

We use execution time and non-rechargeable energy cost as the metrics to evaluate our schedules and compare with the existing solution. The existing scheme only schedules for the worst case; while in other cases, solar energy is under utilized and potential opportunities to performance improvement are overlooked. However, this leads to a seemingly "economic" approach since the energy cost is low. Our schedules, on the other hand, speedup the rover's movement at 50% in the best case and 25% in the typical case, while drawing more non-rechargeable 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 compare the performance and energy cost in this bigger picture.

We 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. During the period when 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 walking evenly in the best case, typical case, and worst case since its execution speed is not aware of power constraints. This results in a long execution time and considerable energy cost in the worst case. When our schedules are used, the rover finishes 50% of work in the best case, 42% of work in the typical case, leaving rest 8% to the worst case. Since our schedules speedup execution at the best case and typical case, 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 5. The analysis shows our schedules win both on performance and energy savings considerably.

Figure 7 highlights the property of the power-aware scheduler in a geometrical view. The top chart illustrates how our solution adjusts the execution speed adaptively with available power budget, while the existing scheme ignores the power constraint and always

| Time frame (s) | Solar power (W) | JPL | | | Power-aware | | |
|---|---|---|---|---|---|---|---|
| | | Travel distance | Time (s) | Energy cost (J) | Travel distance | 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 | 160 | 776 |
| Total | | 48 | 1800 | 3554 | 48 | 1360 | 2391.5 |
| | | | | | Improve ment | 24.4% | 32.7% |

Table 5: Comparison of existing schedule to power-aware schedules under a mission scenario

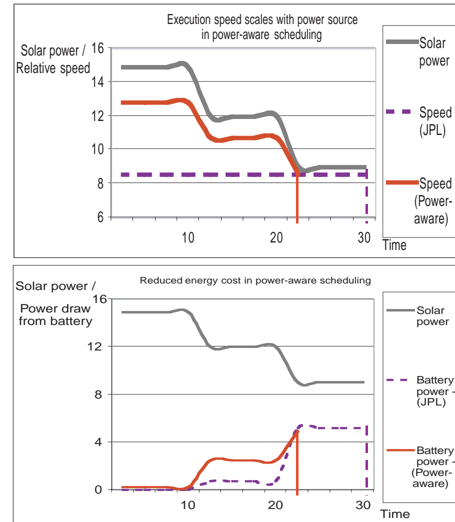


Figure 7: Adaptative speedup in power-aware scheduling

operates at the lowest speed. The workload is represented by the area (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 draw 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 speed curve and power curve stop. 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 longer execution time. Therefore, given the same workload, the power-aware scheduler wins both on performance and energy savings to non-renewable source.

## 7 Conclusion and Future Work

Power-aware design becomes a more important issue in mission critical systems that require best use of available power source and deliverable high performance at the same time. We target the scheduling algorithms to systems with various power constraints and different classes of power consumers, where power-aware techniques have potentials to 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 interpretation. We propose three core algorithms that break the power-aware scheduling problems into steps. Via an incremental approach, we distinguish the nature 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 non-renewable energy.

Several interesting issues in this dimension need further attention. To expand the applicability of our algorithms, more effective heuristics need to be discovered. As our example shows, automated loop scheduling techniques are necessary for the power-aware design to deliver high performance in a cost-effective manner. We would also like to incorporate more novel power management techniques including voltage/frequency scaling into this tool to support more effective power-aware design.

```
ParallelSchedule(Graph G, vertex anchor, vertex c) {
    La := Single source longest paths (G, anchor);
    if (positive cycle found) return FAIL ;
    C := set of topological successors of candidate c;
    if (C = ∅ ) return schedule with s(c) := La;
    D := C;
    while (D ≠ ∅) {
        v := SelectSuccessor(D);
B:      foreach u ∈ C − {v} {
            if (r(v) = r(u))
                add edge (v,u) to G, with weight w(v,u) := Max(d(v), La(u) − La(v));
            else if (edge (v,u) does not exist)
                add virtual edge (v,u) to G;
        }
        w := the most recently scheduled vertex, where (r(w) = r(v))
        if (w ≠ nil)
            add edge (w,v) to G, with weight w(w,v) := Max(d(w), La(v) − La(w));
        schedule = ParallelSchedule(G, anchor, v);
        if (schedule ≠ FAIL)
            return schedule with s(c) := La;
        Undo added edges since step B;
    }
    return FAIL ;
}
```

Figure 8: Parallel scheduling algorithm

| Vertex | Resource | Duration | Power |
|---|---|---|---|
| a | A | 4 | 8 |
| b | A | 3 | 2 |
| c | A | 3 | 8 |
| d | A | 2 | 4 |
| e | B | 2 | 7 |
| f | B | 3 | 8 |
| g | B | 2 | 6 |
| h | C | 2 | 6 |
| i | C | 3 | 8 |

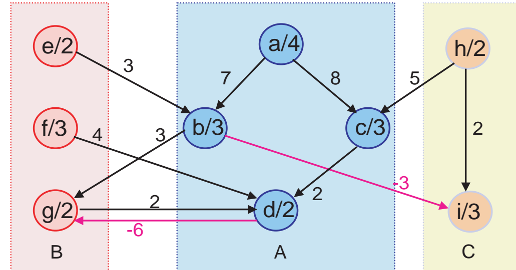


Figure 9: Constraint graph of the example

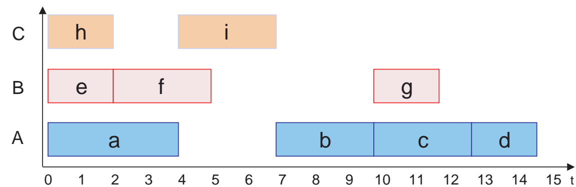


Figure 10: Parallel schedule result

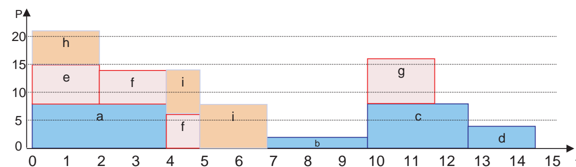


Figure 11: Power curve of the example before power-aware scheduling

## References

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] NASA/JPL's Mars Pathfinder Home Page *http://mars3.jpl.nasa.gov/MPF/index0.hml*

## APPENDIX: Detailed algorithm illustration

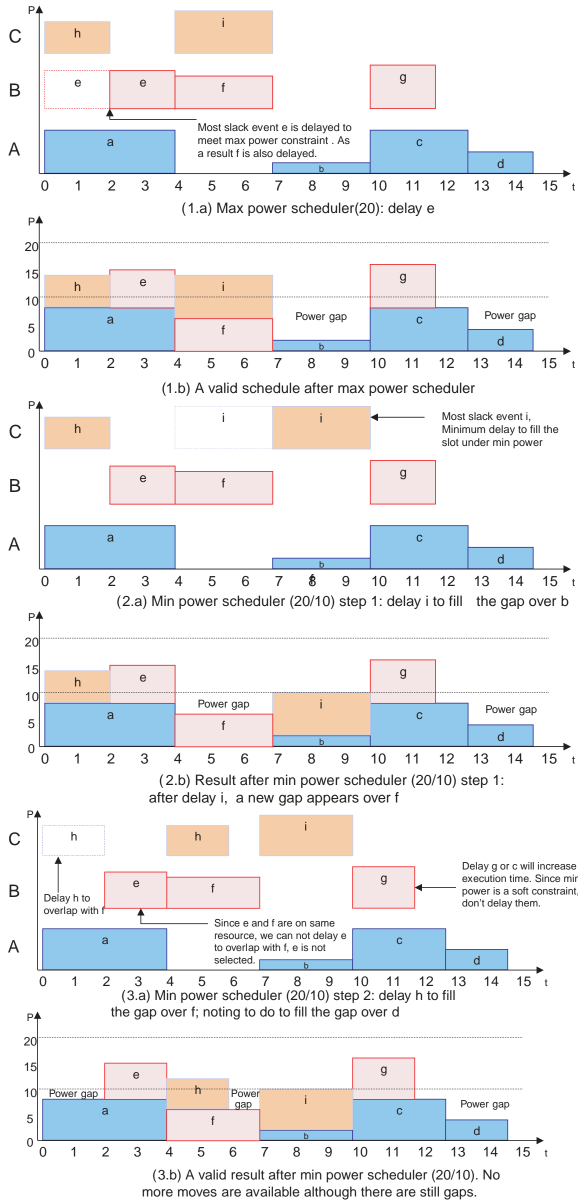

Figure 12: Steps of power-aware scheduling, max power = 20W, min power = 10W

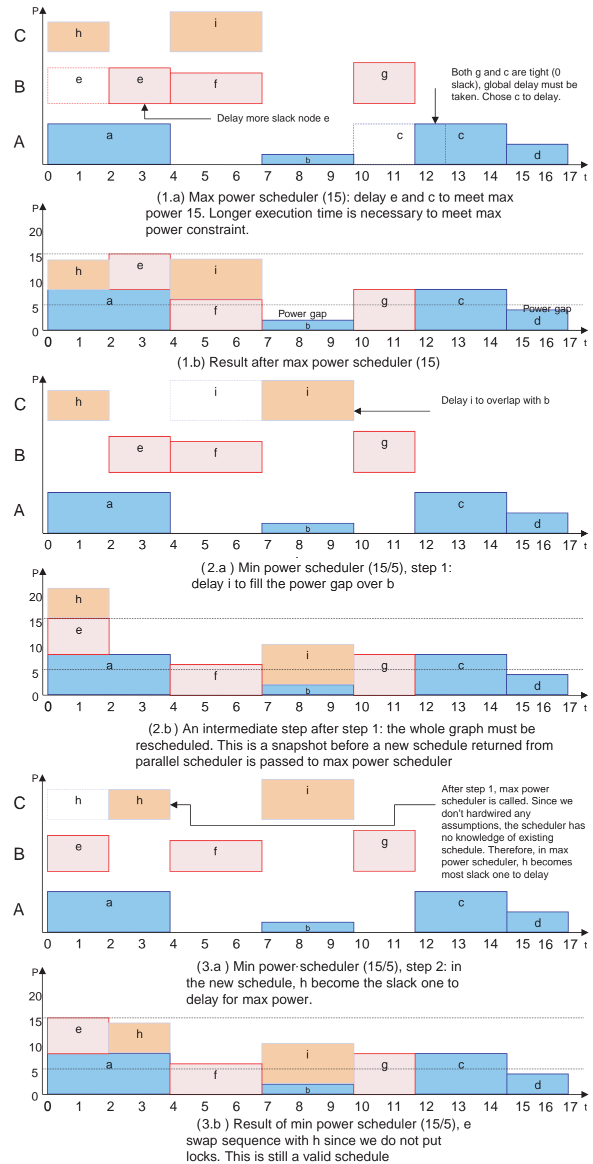

Figure 13: Steps of power-aware scheduling, max power = 15W, min power = 5W