Power-Aware Task Motion for Enhancing Dynamic Range of Embedded Systems with Renewable Energy Sources

Jinfeng Liu, Pai H. Chou, and Nader Bagherzadeh

Department of Electrical and Computer Engineering,
University of California, Irvine, CA 92697-2625, USA
{jinfengl, chou, nader}@ece.uci.edu

Abstract. New embedded systems are being built with new types of energy sources, including solar panels and energy scavenging devices, in order to maximize their utility when battery or A/C power is unavailable. The large dynamic range of these unsteady energy sources is giving rise to a new class of *power-aware* systems. They are similar to *low-power* systems when energy is scarce; but when energy is abundant, they must be able to deliver high performance and fully exploit the available power. To achieve the wide dynamic range of power/performance trade-offs, we propose a new *task motion* technique, which tunes the system-level parallelism to the power/timing constraints as an effective way to optimize power utility. Results on real-life examples show an energy reduction of 24% with a 49% speedup over best previous results on the entire system.

Keywords: power-aware scheduling/task motion, timing/power constraint modeling, power/performance range, system-level design

1 Introduction

Recent years have seen the emergence of *power-aware* embedded systems. They are characterized by not only low power consumption, but more generally by their ability to support a wide range of power/performance trade-offs. That is, these systems can be viewed as providing "knobs" that can be turned one direction to reduce power consumption, or the other direction to increase performance. The ability to adapt the range of power/performance trade-offs is driven by new applications that demand very high performance while under stringent timing and power constraints.

One example that fits this description is the Mars rover by NASA/JPL [1]. It was designed to roam on Mars to take digital photographs and perform scientific experiments over several hundred days. Its energy sources consist of a battery pack and a solar panel, and future versions are expected to incorporate nuclear generators, thermal batteries, and energy scavenging devices. Besides the Mars rover, many new emerging embedded systems are also following this trend towards new types of heterogeneous, renewable energy sources. Future personal digital assistants (PDAs) will likely include solar panels as found in many calculators today. Yet another example is the distributed sensors. They are being built today to draw energy from solar power, wind power, or even ocean waves. They represent a great improvement because they enable the system's continued operation for useful or critical tasks when the traditional energy sources like battery and A/C become unavailable.

These new types of energy sources are posing new challenges to designers of power-aware systems. What they all have in common is that many of these new energy sources are far from being ideal power supplies. For example, the output of a portable solar panel today can be up to 15W under direct sunlight, or down to 1mW under incandescent light. Similarly, other sources will be determined by the wind or ocean wave, which can also cause the available power to vary by several orders of magnitude. Embedded systems powered by such sources must be designed to operate in as wide a range as possible. Indeed, new emerging components such as the Intel XScale are able to scale their power/performance over 20×, and this dynamic range will likely to increase.

While low power operation is clearly important, the ability to fully exploit the available power when energy is abundant is equally important. However, today's systems let much free energy go to waste, because they are designed for fixed budgets. For example, a system with an XScale draws approximately 1W of power, but when the solar panel outputs 15W in direct sunlight, up to 1400% of the power will be wasted. Even if there is a rechargeable battery, when it becomes fully charged, the extra power turns into waste heat. This is also the case with the Mars rover, which accomplishes its low-power property by serializing all tasks, including mechanical and heating as well as computation. However, it also discards excess power as waste heat.

One way to take advantage of the excess power is to increase parallelism. In fact, parallelism is in general an effective way for both high performance and low power. By operating additional processors at their peak rate, they will be able to take advantage of the abundant energy. Parallelism can also enable a set of processors to operate at a lower power level than a single processor with the same performance. Although it is difficult to parallelize algorithms in general, systems with many concurrent activities present many opportunities for parallelism-based trade-offs.

Peak-power poses new challenges to such a power-aware architecture with multiple processors. Today's systems satisfy the peak-power constraint by construction, that is, each component is given a budget that is guaranteed never to be exceeded according to their data sheet. However, by using multiple processors to fully utilize the available power when abundant, a multi-processor architecture would risk exceeding the total budget when the supply power is low, if it is not designed carefully. Therefore, it is of utmost importance that the proposed scheme be able to fully respect the maximum power as a hard constraint.

In this paper, we propose to enhance the dynamic range of these embedded systems by means of *task motion* and power-aware scheduling. It transforms tasks within their timing constraints and their precedence dependencies in order to match the parallelism to the available power level. Furthermore, we exploit domain-specific knowledge about the power-consuming tasks to achieve additional significant power/performance improvements over existing schedulers. The enhanced dynamic range and power-awareness enable the system to accomplish more tasks in a shorter amount of time while respecting all timing constraints. The benefits must ultimately be translated into application-specific metrics, but as power-aware systems are deployed in more mission-critical applications, the saving from reduced mission time or enhanced quality may translate into a saving of millions of dollars.

Section 2 reviews related work. Section 3 uses an example showing a counter-intuitive result when some of the well-known techniques will fail at the system level. However, this problem can be successfully addressed by our new technique, which is presented in Section 4. We discuss experimental results in Section 5.

2 Related Work

To explore the power/performance range in power-aware embedded systems, we can draw from many techniques developed for low power and high performance. This section surveys related work in these areas with a discussion on their integration at the system level.

Low power can be achieved by many ways. For system-level designs, where the components are largely off-the-shelf or already designed, the applicable techniques include subsystem shutdown and dynamic voltage scaling (DVS). In the first case, subsystem shutdown decision can be based on fixed idle times, adaptive timeout, or predictive based on a mix of profile and runtime history [15, 14, 4]. Similarly, power-up may be either event-driven or predictive in an attempt to minimize timing or power penalty. In the second case, DVS techniques have been developed for variable-voltage processors (introduced by [16], with follow-up by [5, 12] and more). Because energy is a quadratic function of voltage, lowering the voltage can result in significant saving while still enabling the processor to continue making progress, unlike the shutdown case. Lowering the voltage will also require reduction in frequency, which has the effect of reducing dynamic switching power.

In addition to low power, the power/performance range can also be increased towards high performance by drawing from previous work on retiming or pipelining and applying them to the system level. Leiserson et al. first established the theoretical foundation for retiming synchronous circuits [8], and this has been extended to loop scheduling for VLIW processors [13, 2, 6]. Shifting tasks in a data flow graph (DFG) across the iteration boundary can result in a shorter execution time or alleviate the resource pressure (e.g. number of registers and functional units). Such techniques are also used in power minimization by reducing switching activities [7, 17].

Existing techniques need significant enhancements before they can be correctly applied to a system-level power management problem. First, most techniques to date treat either power or timing as an *objective*, rather than a *con-*

straint. In real systems, the max power budget is a real, hard constraint, whose violation can lead to malfunction. Max power was not of central concern previously, but as we consider additional power sources such as solar whose power output can vary, max power constraints must be strictly enforced. This becomes especially important as we increase the range of power and performance tradeoffs by tuning the parallelism. Second, the tasks to be scheduled are related to each other not only by precedence, data dependency or deadline, but also related across different components by dependencies like *co-activation*, which must be correctly modeled for system-level power management, or else anomalies can occur. Co-activation means the execution of one task requires the power consumption of other dependent services or tasks. A simple example is that when the CPU is running, it imposes a co-activation dependency on the memory. Techniques such as DVS are designed mainly for minimizing CPU power, but they have not considered other components that have dependencies on the CPU. In fact, energy saved on the CPU may be more than offset by the increased energy consumed by the rest of the system. The following section presents a simple example to illustrate such an anomaly with applying DVS without system-level considerations.

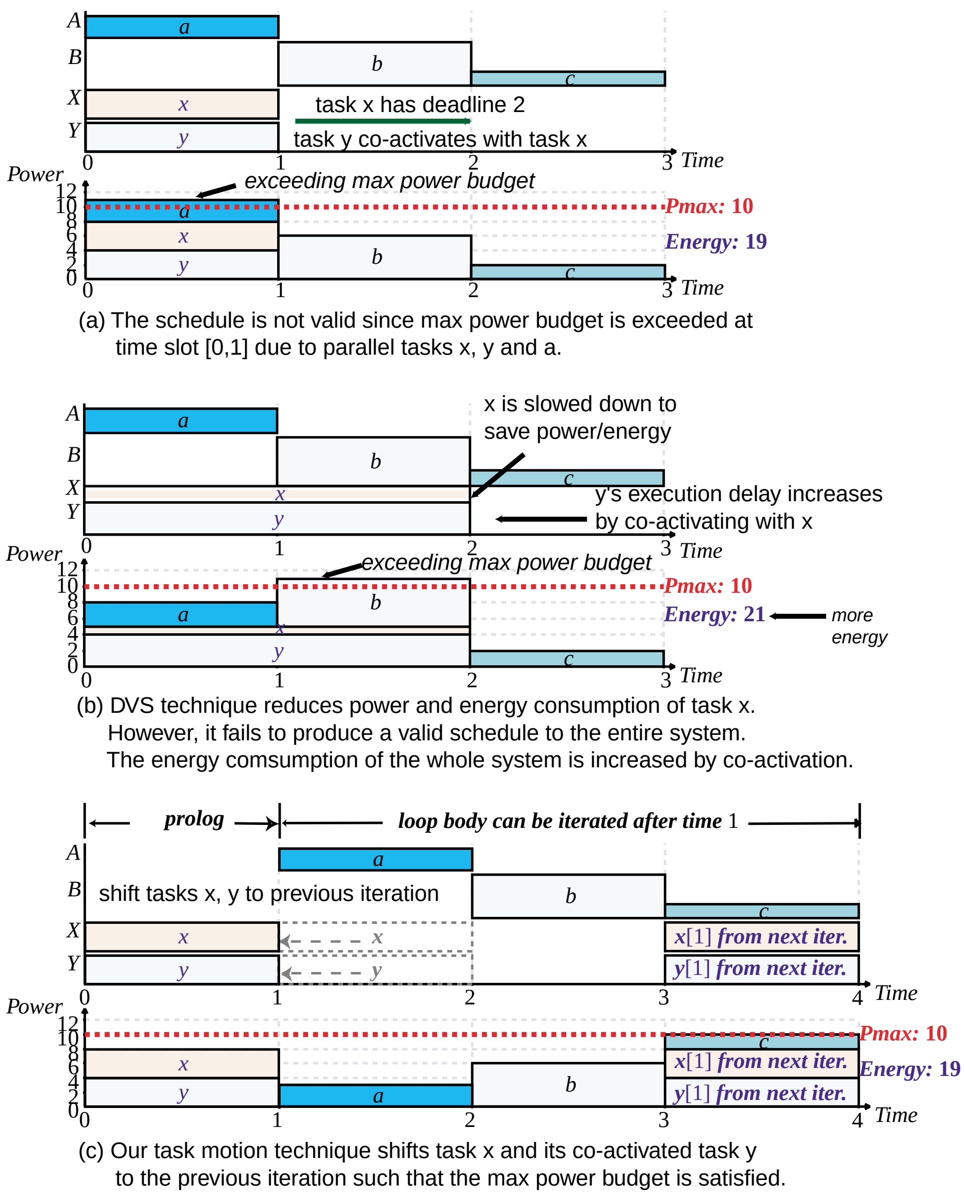
3 DVS Anomaly

We present a simple example in Fig. 1 to illustrate an anomaly with applying DVS without considering system-level dependencies, resulting in an incorrect system. It will be further used to explain our new system model and scheduling technique in the ensuing text. In this example, five tasks a, b, c, x, y are to be scheduled on four execution resources A, B, X, Y. The constraints are:

1. The overall deadline is at time 3.
2. The max power budget is 10W.
3. Tasks a, b and c must be serialized.
4. The execution resources A, B are not voltage-scalable.
5. Only task x can be voltage-scaled on resource X (e.g. a processor), and it has some slack time to finish before time 2.
6. Task y must co-activate with task x, and its resource Y is also not voltage-scalable (e.g. memory, I/O).

Note that task y need not start and finish at the same time as x, but it must *envelop* x, i.e., start no later than x starts and finish no sooner than x finishes. For simplicity, this example assumes x and y start and finish together.

We present schedules as power-aware Gantt charts, where the horizontal and vertical axes represent time and power, respectively. Each chart also consists of a pair of views: *time view* organizes tasks by horizontal tracks that correspond to power consuming resources (processors, peripherals), and *power view* stacks the tasks over time to show the power breakdown by tasks. The curve that traces the height of the power view is the *power profile* for the entire system.



(a) The schedule is not valid since max power budget is exceeded at time slot [0,1] due to parallel tasks x, y and a.

(b) DVS technique reduces power and energy consumption of task x. However, it fails to produce a valid schedule to the entire system. The energy comsumption of the whole system is increased by co-activation.

(c) Our task motion technique shifts task x and its co-activated task y to the previous iteration such that the max power budget is satisfied.

Fig. 1. An example where DVS fails to reduce power and energy at system level, while our new technique will succeed

Fig. 1(a) shows a time-valid schedule with a max power violation during time $[0, 1]$. Rescheduling x and y in $[1, 2]$ will be time-valid but still violates max power. Fig. 1(b) shows the case when DVS was used to slow down task x until its deadline of time 2. Intuitively, reducing both power and energy of task x should eliminate the max power violation, but instead it not only does not reduce max power, but actually increases total energy at the system level. Because x runs more slowly, its co-activated task y must also consume power for longer on a device that is not voltage scalable. As a result, the execution of x and y overlaps that of task b, thereby leading to higher system-level power. Furthermore, energy saved by slowing down x is more than offset by the additional energy consumed by the lengthened y. This anomaly is an example where DVS should not be applied in isolation.

Fig. 1(c) shows a feasible solution obtained by our new *power-aware task motion* technique on iterative tasks. Task x and y are shifted (or *promoted*) to the previous iteration to overlap task c instead of a or b. As a result, both the max power and the deadline are satisfied. However, the optimal solution cannot be obtained unless we exploit domain-specific knowledge about the task set by eliminating a precedence dependency and replacing it with a *utilization constraint*. The details will be explained in later sections.

4 Task Motion under Timing and Power Constraints

We propose power-aware task motion for exploring power/performance trade-offs in embedded systems. We first define our constraint model and introduce our representations based on a timing constraint graph, where we capture two classes of constraints: intra-iteration and inter-iteration timing constraints. Task motion shifts tasks across iteration boundaries and relaxes timing constraints to achieve more scheduling opportunities. We also define *utilization constraints* to support more aggressive but provably correct design space exploration. We close this section by sketching an algorithm that combines power-aware scheduling [9, 10] and task motion as a new "knob" for power-aware designs.

4.1 Constraint graph and schedule

The input to the scheduler is a *(timing) constraint graph* $G(V, E)$, where the vertices V represent tasks, and the edges $E \subseteq V \times V$ represent timing constraints between tasks. Each vertex $v \in V$ has three attributes, $d(v)$, $p(v)$ and $r(v)$, representing task v's *execution delay*, *power consumption* and *resource mapping*, respectively. Each edge $(u, v) \in E$ has two attributes, $\delta(u, v)$ and $\lambda(u, v)$. $\delta(u, v)$ specifies the *min/max timing constraints* [3]. For any function σ that assigns the start times to tasks u and v as $\sigma(u)$ and $\sigma(v)$, $\sigma(v) - \sigma(u) \geq \delta(u, v)$. If $\delta(u, v) \geq 0$, then the edge (u, v) is called a *forward edge*, and it specifies a *min timing constraint*. If $\delta(u, v) < 0$, then it is a *backward edge* indicating a *max timing constraint*. $\lambda(u, v)$ is called the *dependency depth*, which specifies constraints across iterations. An *iteration* is a full pass of executing each task once in a valid order. $\delta(u, v)$ and $\lambda(u, v)$ indicate that the execution of task u in iteration i must precede task v in iteration $i + \lambda(u, v)$ by $\delta(u, v)$ time units. If $\lambda(u, v) = 0$, edge (u, v) specifies an *intra-iteration constraint*. Otherwise, it is an *inter-iteration constraint*. We assume that inter-iteration constraints are only precedence dependencies (forward edges) and their dependency depths are positive integers. For backward edges, their dependency depths are always zero.

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 satisfy all timing constraints, and tasks that share the same resource are serialized. If G represents an iteration of a loop, σ must also satisfy inter-iteration constraints such that they must hold across iterations when multiple instances of σ are concatenated.

A schedule σ has a *power profile* function of time $P_\sigma(t), 0 \leq t \leq \tau_\sigma$ representing the instantaneous power consumption of all tasks during the execution of σ (illustrated by the power view of the Gantt-chart in Fig. 1). 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 level of power that can be supplied by the power sources. 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. However, we treat the min power constraint as a soft constraint that could be violated occasionally in a valid schedule.

In cases where the min power constraint P_{min} represents the free power level (e.g. solar), the energy drawn from the non-renewable energy sources is defined as the *energy cost* $Ec_\sigma(P_{min})$ of a schedule σ. It distinguishes between costly power and free power in such a way that 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.

4.2 Task motion under timing constraints

Task motion obtains different versions of a scheduling problem by converting between intra-iteration and inter-iteration constraints. We first construct an *iteration graph* $G'(V, E')$: it has the same vertices as those of the constraint graph $G(V, E)$, but edges E' consist of only intra-iteration constraints. Formally, $E' = \{(u, v) : (u, v) \in E \text{ such that } \lambda(u, v) = 0, \delta'(u, v) = \delta(u, v)\}$. The edges in E' will not have dependency depths λ, since they are always zero. The expected loop duration τ is obtained from the original schedule computed from the initial iteration graph G'.

Without loss of generality, we focus our discussion on task *promotion* by which the execution of a task is shifted to the previous iteration of the loop, and the instance of the same task in the next iteration is promoted into the new loop body. The inverse procedure for task *demotion* can be similarly defined.

A task v is *promotable* if either vertex $v \in V$ does not have any incoming forward edges, or all of v's incoming forward edges in G have at least one dependency depth. If σ is a valid schedule of one iteration, we can *promote* a task v according to the *expected loop duration*, which is the finish time τ_σ of σ. Given $\tau = \tau_\sigma$, promoting a task v entails the following transformations on G and G':

1. For each of v's *incoming forward edges* (u, v) in graph G, decrease $\lambda(u, v)$ by one. If (u, v) becomes an intra-iteration constraint, ($\lambda(u, v) = 0$), edge (u, v) is added to graph G' if it is not present in G'.
2. For each v's *outgoing forward edge* (v, u) in graph G, increase $\lambda(v, u)$ by one.
3. For each v's *incoming backward edge* (u, v) in graph G', increase $\delta'(u, v)$ by τ, that is, $\delta'(u, v) = \delta'(u, v) + \tau$.
4. For each v's *outgoing edge* (v, u) in graph G', decrease $\delta'(v, u)$ by τ, that is, $\delta'(v, u) = \delta'(v, u) - \tau$.

Steps 1 and 2 push one dependency depth from v's incoming forward edges to its outgoing forward edges. Step 1 also adds any new intra-iteration edges to graph G', which tracks only intra-iteration constraints. Step 3 transforms the incoming backward edges of v for the promotion (its incoming forward edges are managed in step 1). Step 4 transforms the outgoing edges of v, for both forward and backward edges. Steps 3 and 4 can be validated as follows.

When a task v is promoted in graph G', vertex v represents the execution of task v in the next iteration. Therefore, the new start time assignment $\sigma'(v) = \sigma(v)+\tau$. In step 3, before promoting v, edge (u, v) indicates $\sigma(v)-\sigma(u) \geq \delta'(u, v)$. Thus after the promotion, $\sigma'(v) - \sigma(u) = (\sigma(v) + \tau) - \sigma(u) \geq \delta'(u, v) + \tau$. Therefore, the new constraint in G' is $\delta'(u, v) + \tau$. Similarly in step 4, edge (v, u) means $\sigma(u) - \sigma(v) \geq \delta'(v, u)$ before promotion. Thus, $\sigma(u) - \sigma'(v) = \sigma(u) - (\sigma(v) + \tau) \geq \delta'(u, v) - \tau$. The constraint becomes $\delta'(u, v) - \tau$ after the promotion.

When a task v is being promoted, its corresponding min timing constraints (zero or positive values) will become max timing constraints (negative values) by step 4; and vice versa, its corresponding max timing constraints will transform into new min timing constraints by step 3. Promotion effectively reduces the values of min constraints and makes the problem easier to solve by exposing more scheduling opportunities. We say that the constraint is *relaxed*, and this is a key technique for increasing the system's dynamic range.

Fig. 2 illustrates task promotion on the example previously shown in Fig. 1. Fig. 2(a) shows the initial constraint graph G consisting of five vertices representing five tasks a, b, c, x, y. They all have the same execution delay of one time unit, and their power consumption is $p(a) = 3W, p(b) = 6W, p(c) = 2W, p(x) = p(y) = 4W$. Therefore the most power consuming task is b and the least power consuming one is c. Tasks a, x, y have dedicated execution resource A, X, Y ($r(a) = A, r(x) = X, r(y) = Y$), respectively; while tasks b and c share the execution resource B ($r(b) = r(c) = B$). For brevity, these task attributes are not shown in the graph. The edges in the constraint graph G represent timing constraints. They are denoted as (λ, δ) corresponding to the dependency depths and the values of the timing constraints.

For example, the forward edge (a, b) represents an intra-iteration constraint with dependency depth $\lambda(a, b) = 0$, and it is a min constraint with $\delta(a, b) = 1$ indicating $\sigma(b) - \sigma(a) \geq 1$. Since task a's delay $d(a) = 1$, this constraint can be paraphrased as "task b cannot start until task a completes," that is, tasks a and b must be serialized. Similarly tasks b and c are also serialized by edge (b, c). Edge (x, a) with $\delta(x, a) = 0$ indicates that task a cannot start before task x starts, because $\sigma(a) - \sigma(x) \geq 0$. Edge (x, c) with $\delta(x, c) = 2$ specifies a min separation between task x and task c, that is, $\sigma(c) - \sigma(x) \geq 2$. Therefore, task c must wait until task x has started for two time units. Edge (c, a) with $\delta(c, a) = -2$ is a backward edge representing a max constraint: $\sigma(c) - \sigma(a) \leq 2$. It defines the deadline to start task c relative to the start time of task a. This deadline is equal to the start time of task a plus two time units. In addition to these intra-iteration timing constraints, there is an inter-iteration timing constraint (b, x), indicating that the start time of task b precedes task x in the *next iteration* $(\lambda(b, x) = 1)$ by one time unit $(\delta(b, x) = 1)$. Inter-iteration constraints are marked as dashed arrows. There is a co-activation dependency between task x and task y. This is denoted as a pair of special timing constraints. As mentioned previously, we assume each iteration must finish within three time units.

The initial iteration graph G' has the same set of vertices representing tasks a, b, c, x, y. The edges in G' only represent intra-iteration constraints. Therefore only the constraint value δ' is shown on each edge. Dependency depth λ is not shown since it is always zero in graph G'. For example, the inter-iteration edge (b, x) does not appear in the initial G'. The co-activation dependency is still denoted as a special constraint in G'.

The initial schedule σ computed from the iteration graph G' is also shown in Fig. 2(a). It is the same as Fig. 1(a). Although all timing constraints are satisfied, the schedule σ is not valid since during time [0, 1] the power consumption of the whole system is 11W, exceeding the max power constraint $P_{max} = 10W$. No valid solution is possible even if we try voltage scaling for tasks x.

In Fig. 2(b) task x and its co-activated task y are promoted to produce a valid schedule (same as Fig. 1(c), except that the prolog is not shown). Tasks x and y are promoted together due to co-activation, but they are scheduled as separate tasks because they may not start and finish at the same time. The constraint graph G will only update dependency depths λ of the timing constraints corresponding to x. Since the original schedule finishes at time 3, the timing constraints δ' in G' will be transformed using $\tau = 3$. By step 1, edge $(b, x) \in G$ becomes an intra-iteration edge (solid arrow) and is inserted to G'. By step 2, edges (x, a) and $(x, c) \in G$ become inter-iteration edges (dashed arrows). By step 4, edges (x, a) and $(x, c) \in G'$ reduce their constraint values by $\tau = 3$. Accordingly, task x's outgoing min constraints are transformed into more relaxed max constraints $(\delta'(x, a) = -3, \delta'(x, c) = -1$, compared to 0 and 2 in Fig 2(a)). As a result, tasks x can be rescheduled in time slot $[2, 3]$ without violating any timing constraints, and the max power constraint is also satisfied. Without task motion, this valid solution cannot be achieved.

4.3 Utilization constraints

Task motion is based on the classification of intra-iteration and inter-iteration timing constraints. However, in some cases, it is difficult or unnecessary to decide whether a timing constraint should be intra-iteration or inter-iteration. Such cases are present in the Mars rover. For example, for timing constraints between a heater and a motor by which the motor is heated periodically, whether to model these constraints as intra-iteration or inter-iteration is not clear. In fact, whether the heaters and the motors stay in the same iteration does not matter. In the computation domain, these correspond to background, preemptible tasks that need not synchronize with the main control loop but must be given a share of the CPU time to avoid starvation.

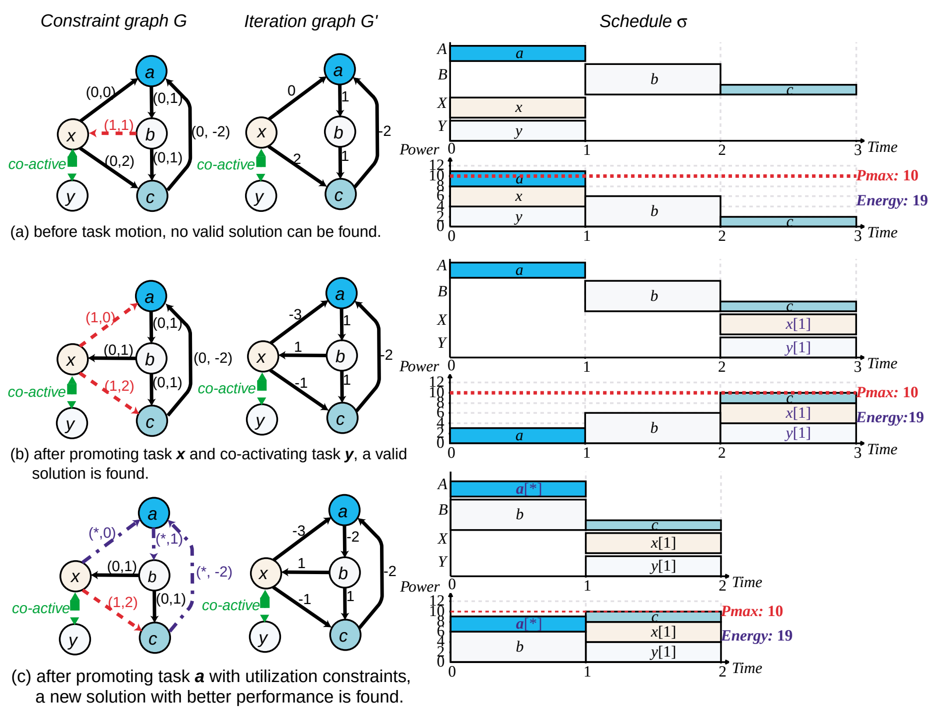


Fig. 2. Task motion under timing constraints

We call such constraints *utilization*-based timing constraints. They can be expressed as either intra-iteration or inter-iteration ones. A utilization constraint between two tasks u and v is also represented as an edge $(u, v) \in E$ in constraint graph G with its dependency depth denoted as $\lambda(u, v) = *$, indicating that it can be either zero or non-zero.

Now we examine task motion under utilization constraints. It needs only minor modifications to the procedure we defined in Section 4.2.

(a) The initial iteration graph G' will include both intra-iteration constraints and *utilization constraints* in its edges. (*Treat utilization constraints as intra-iteration*).
(b) A task v is promotable if either vertex $v \in V$ does not have any incoming forward edges, or the dependency depths λ of all v's incoming forward edges are positive values *or* $*$. (*Treat utilization constraints as inter-iteration*).
(c) The modified procedure for promoting a task v is as follows.
   1. For each of v's incoming forward edges (u, v) in graph G, decrease $\lambda(u, v)$ by one, *if* $\lambda(u, v) \neq *$. If $\lambda(u, v)$ becomes 0, add edge (u, v) to graph G' if it is not present in G'. (*No update for utilization constraints in step 1*).
   2. For each v's outgoing forward edge (v, u) in graph G, increase $\lambda(v, u)$ by one, *if* $\lambda(u, v) \neq *$. (*No update for utilization constraints in step 2*).
   3. For each v's incoming backward edge (u, v) in graph G', $\delta'(u, v) = \delta'(u, v) + \tau$, *if* $\lambda(u, v) \neq *$. *Otherwise,* $\delta'(u, v)$ *remains unchanged.* (*No update for utilization constraints in step 3*).
   4. For each v's outgoing edge (v, u) in graph G', $\delta'(v, u) = \delta'(v, u) - \tau$. (*Same as the previous step 4*).

Since utilization constraints can be either intra-iteration or inter-iteration, by giving them some special treatments, the modified procedure is straightforward except steps 3 and 4 need more explanation. In step 3, if edge (u, v) represents a utilization constraint, $\delta'(u, v)$ can be transformed into either one of the two forms: $\delta'(u, v)$ or $\delta'(u, v) + \tau$, since it can be either intra-iteration or inter-iteration. That is, the transformation is valid either $\sigma'(v) - \sigma(u) \geq \delta'(u, v)$ or $\sigma'(v) - \sigma(u) \geq \delta'(u, v) + \tau$ holds. Obviously, the solution to these two inequalities with an *OR* relation is $\sigma'(v) - \sigma(u) \geq \delta'(u, v)$, which means the constraint with the smaller value applies. Therefore, the value of a utilization constraint will not increase by τ in step 3. Likewise, in step 4, the value of the new constraint is the smaller one between $\delta'(v, u) - \tau$ and $\delta'(v, u)$, which is $\delta'(v, u) - \tau$. In summary, if the promoted task v has any incoming utilization-constraint edges, then these edges remain the same in the iteration graph G' during the promotion. For v's outgoing utilization-constraint edges, the values of constraints in G' are decreased by the loop duration τ. As a result, utilization constraints will always be relaxed to produce more scheduling opportunities.

For example, if resource A is a heater, a motor, or a CPU running a preemptible background tasks, then we can model task a with utilization constraints (x, a), (a, b) and (c, a). The initial graphs G, G' and schedule σ look very similar to Fig. 2(a), except utilization constraints (x, a), (a, b) and (c, a) in G will be denoted as a new type of arrows, and their dependency depths $\lambda = *$ (as seen in Fig. 2(c)). After promoting tasks x and y, graphs G, G' and schedule σ will also look similar to Fig. 2(b) except that the utilization constraints (x, a), (a, b) and (c, a) in G will not be changed by task motion.

Fig. 2(c) shows the resulting graphs G, G' and schedule σ after promoting task a with utilization constraints, which are marked as a different type of dashed arrows in graph G. By the modified step 3, the value of constraint $\delta'(c, a)$ in G' will remain -2; otherwise it will be resumed to 1 if it is not a utilization constraint. The same rule also applies to utilization constraint (x, a) such that $\delta'(x, a) = -3$ instead of 0. Since the serialization chain formed by min constraints is broken, tasks a, b, c (after promoting a, the chain becomes b, c, a in Fig. 2(c)) no longer need to be serialized. Now task a, a small power consumer, can overlap b such that an unexpected solution with a shorter execution time ($\tau_\sigma = 2$) is discovered, and it also satisfies the max power constraint. This optimal solution could not have been obtained without using utilization constraints, which enable more aggressive, provably correct relaxation of the time constraints.

4.4 Scheduling algorithms for power-aware task motion

We combine power-aware scheduling with system-level task motion as a way to discover a wider range of power/performance trade-offs. Our core scheduling algorithms consist of (a) transforming the problem into its new versions by task motion, and (b) power-aware scheduling for each version. From the illustration in Sections 4.2 and 4.3, the implementation of (a) is straightforward. Algorithm (b) is derived from [10] by applying the power-aware scheduler to the iteration graph G' after each task motion. For brevity, details of the scheduling algorithms are omitted in this paper but can be found in [11].

5 Experimental Results

We use the NASA/JPL Mars rover [1] to evaluate the effectiveness our power-aware task motion technique. We construct a system-level representation that includes the computational, mechanical and thermal subsystems. The timing constraints on the heaters and preemptible background computation tasks can be modeled with utilization constraints. We also consider dual energy sources: a solar panel and a non-rechargeable battery. We consider three scenarios with different solar power output levels: 14.9W (noon time), 12W, and 9W (dusk). The min power constraints are set to the respective solar output levels, while the max power constraints are set to the solar power plus 10W, which is the maximum battery power rating.

Table 1 compares the results of four techniques by using the energy cost to the non-rechargeable battery and the execution time of each iteration as metrics:

(0) the existing manual solution (fully serialized),
(I) power-aware scheduling [10],
(II) power-aware task motion without utilization constraints,
(III) power-aware task motion with utilization constraints.

– For scenario 1 (14.9W solar power), all schedulers except JPL's (0) compute fast schedules (i.e., short τ), but these three solutions vary in energy cost.

Solutions by schedulers I and II are eliminated, because they must draw more energy from the battery in addition to the solar panel in order to achieve the same performance as solution III. Scheduler III could not have achieved this solution without exploiting utilization constraints.

– For scenario 2 (12W solar power), schedulers I and II produce the same solution that is slower than in scenario 1 due to the limited power budget. Scheduler III produces a fast schedule at a higher energy cost than I and II, but it is still within the max power constraint. No one solution is strictly better than the other, and they represent different trade-off points.
– In scenario 3 (9W solar power), the low power budget rules out all but the fully serialized solution, and all schedulers produce the same solution as JPL's manual schedule (0).

Scenario	(0) JPL's Low-power (hand-craft)	(I) Power-aware	(II) Power-aware + Task motion	(III) Power-aware + Task motion + Utilization constraint
1	τ = 75s Ec = 0J √	τ = 50s Ec = 79.5J ×	τ = 50s Ec = 16.5J ×	τ = 50s Ec = 4.5J √
2	τ = 75s Ec = 55J √	τ = 60s Ec = 147J √	same as (I)	τ = 50s Ec = 208J √
3	τ = 75s Ec = 388J √	same as (0)	same as (0)	same as (0)

√ = keep × = drop

Table 1. Comparison in three scenarios

Time frame (s)	Scenario	JPL (0-0-0)			Task motion A (III-I-0)			Task Motion B (III-III-0)		
		Distance (step)	Time (s)	Energy cost (J)	Distance (step)	Time (s)	Energy cost (J)	Distance (step)	Time (s)	Energy cost (J)
0 - 599	1	16	600	0	24	600	129	24	600	129
600 - 1199	2	16	600	440	20	600	1470	23	600	2482
1200 -	3	16	600	3114	4	150	776	1	10	85
Total		***48***	***1800***	***3554***	***48***	***1350***	***2375***	***48***	***1210***	***2696***
Improve-ment						***33%***	***33%***		***49%***	***24%***

Table 2. Comparison in a comprehensive scenario

The results show that our technique not only yields a larger dynamic range by being able to operate at different power levels, but more importantly it uses the available energy more effectively for actual useful work. This is not easy due to complex timing constraints, but the improvement can translate into significant savings in application-specific metrics, as shown in Table 2.

Suppose the rover is traveling to a target location in a distance of 48 steps. Since the rover moves two steps during each iteration, it needs 24 iterations to reach the destination. The mission starts with maximum solar power at 14.9W (Scenario 1). Then, it drops to 12W (Scenario 2) after 10 minutes, and falls to 9W (Scenario 3) 10 minutes later. If the existing low-power, serial schedule is applied, the rover will spend 10 minutes evenly in all three scenarios at a fixed slow moving speed. This results in a long execution time and a high energy cost in Scenario 3. On the other hand, our technique can produce two schemes. Both schemes use more free solar energy to speed up in scenarios 1 and 2 (while satisfying timing and power constraints) so that they can finish the mission earlier to avoid the costly scenario 3. Schemes A and B differ only in scenario 2 where A uses solution I while B uses the faster but more expensive solution III. As a result, scheme A achieves a 33% speedup and a 33% energy saving; and scheme B further speeds up by 49% with a 24% energy reduction. These two alternative designs with different energy/performance trade-offs are discovered by our power-aware task motion technique. They could not have been found by the existing techniques.

6 Conclusion

We have presented a power-aware task motion technique for enhancing the dynamic range of embedded systems powered by heterogeneous energy sources that include renewable, unsteady ones like solar panels. They must be able to not only operate as low-power devices when the supply power is low, but equally importantly use the free abundant energy for useful work while respecting power and timing constraints. We used a DVS Anomaly example to show the pitfalls of applying existing power management techniques without considering system-level dependencies like co-activation, and this has resulted in not only higher energy consumption but also violation of max power constraints. We then showed our constraint formulation and task motion technique to safely transform the tasks while respecting these system-level dependencies. We further enhanced task motion by exploiting utilization-based constraints that exposed additional scheduling opportunities for preemptible, background tasks or even non-computational power consumers such as heaters. These all served to enhance the dynamic range while ensuring all transformations are safe and provably correct. Experimental results on the Mars rover demonstrated the effectiveness of our approach for the solar- and battery-powered system. We expect the benefits to transfer to a whole emerging class of new embedded systems that must draw energy from many renewable but unsteady sources.

Acknowledgement

This research was sponsored by DARPA grant F33615-00-1-1719 and Printronix Fellowship. 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/-index0.html.
2. L.-F. Chao, A. LaPough, and E. H.-M. Sha. Rotation scheduling: A loop pipelining algorithm. *IEEE Transactions on Computer Aided Design*, 16(3):229–239, March 1997.
3. 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.
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. I. Hong, D. Kirovski, G. Qu, and M. Potkonjak. Power optimization of variable-voltage core-based systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(12):1702–1714, 1999.
6. M. Jacome, G. de Veciana, and C. Akturan. Resource constrained dataflow retiming heuristics for VLIW ASIPs. In *Proc. International Symposium on Hardware/Software Codesign*, pages 12–16, May 1999.
7. K. Lalgudi and M. Papaefthymiou. Fixed-phase retiming for low power design. In *Proc. International Symposium on Low Power Electronics and Design*, pages 259–264, August 1996.
8. C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1990.
9. J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *Proc. International Symposium on Hardware/Software Codesign*, pages 153–158, April 2001.
10. J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware scheduling under timing constraints for mission-critical embedded systems. In *Proc. Design Automation Conference*, pages 840–845, June 2001.
11. J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. Power-aware task motion: Dynamic range enhancement for power-aware embedded systems. Technical Report IMPACCT-01-09-01, University of California, Irvine, September 2001.
12. T. Okuma, T. Ishihara, and H. Yasuura. Real-time task scheduling for a variable voltage processor. In *Proc. International Symposium on System Synthesis*, pages 24–29, November 1999.
13. F. Sanchez and J. Cortadella. Time-constrained loop pipelining. In *Proc. International Conference on Computer-Aided Design*, pages 592–596, November 1995.
14. 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.
15. 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.
16. F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *IEEE Annual Foundations of Computer Science*, pages 374–382, 1995.
17. T. Z. Yu, F. Chen, and E. H.-M. Sha. Loop scheduling algorithms for power reduction. In *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3073–6, May 1998.