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

Jinfeng Liu, Pai H. Chou, and Nader Bagherzadeh

Dept. of Electrical & Computer Engineering,
University of California, Irvine,
Irvine, CA 92697-2625 USA
{jinfengl, chou, nader}@ece.uci.edu
http://www.ece.uci.edu/impacct/

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 and A/C power are 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 are 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 for 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 multiple 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 explicitly 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. The new technique transforms tasks within their timing constraints and their precedence dependency 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 works 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 [13, 12, 3]. 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 [14], with follow-up by [4, 10] 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, a wider power/performance range can also be increased towards high performance by drawing from previous works on retiming or pipelining and applying them to the system level. Leiserson et al. first established the theoretical foundation for retiming synchronous circuits [7], and this has been extended to loop scheduling for VLIW processors [11, 2, 5]. 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 [6, 15].

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 *constraint*. 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 maximum output can vary, they must be strictly satisfied. This becomes especially important as we increase the range of power and performance trade-offs by tuning the parallelism. Second, the tasks to be scheduled are related to each other not only by precedence, data dependency or deadline, but there is a whole class of *timing constraints* between different components, such as *co-activation* dependency, which must be correctly modeled for system-level power management. 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 the problems with applying existing low-power and high-performance techniques to the system level.

3 Example of System-level Dependency

We present a simple example in Fig. 1 to illustrate effects of system-level dependencies that make existing techniques less than optimal. 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 is 10.
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 be co-activate with task x, and its resource Y is also not voltage-scalable (e.g. memory, I/O).

Note that task y does not necessarily start and finish at the same time as x, but it must 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.

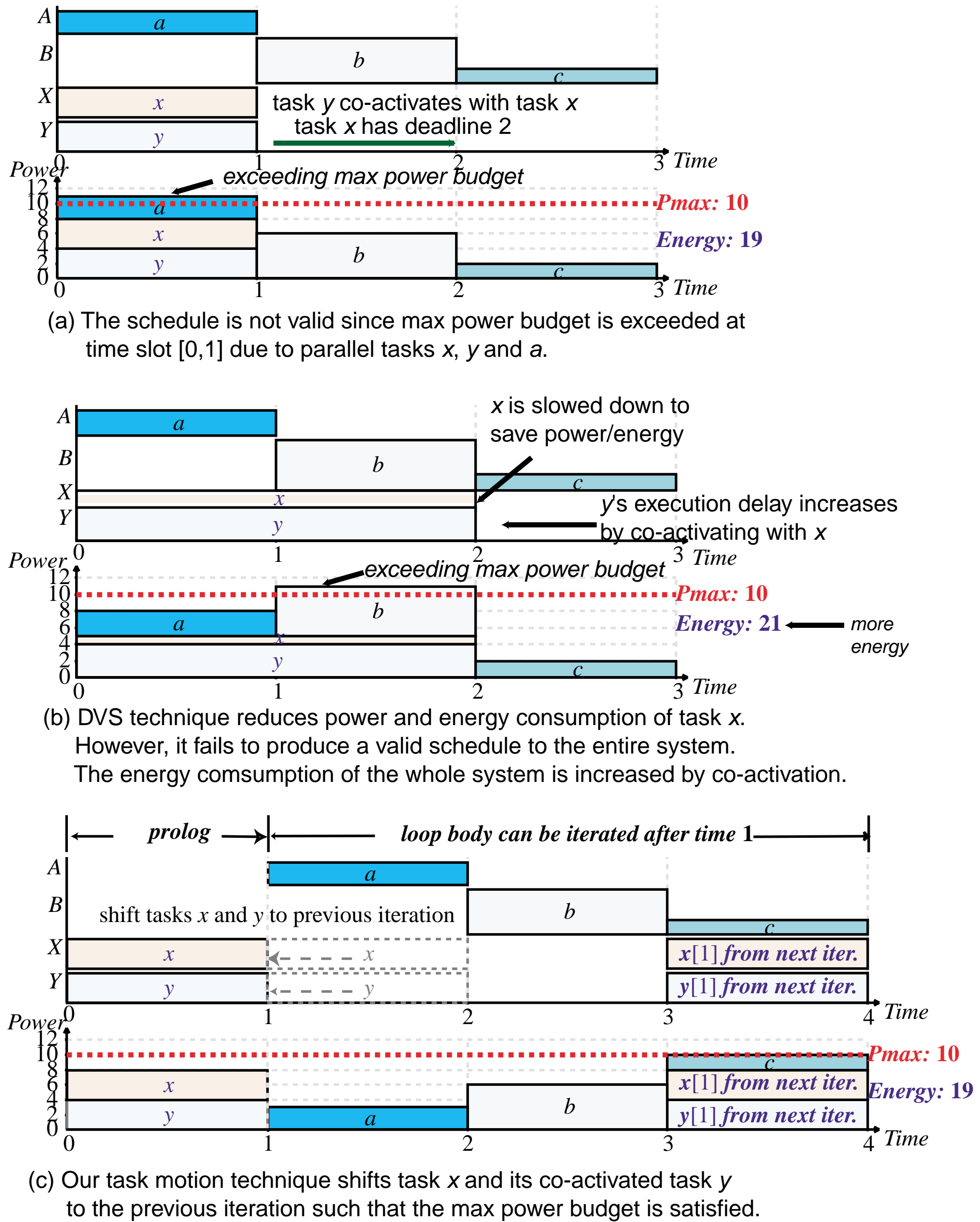

Fig. 1. An example where DVS fails to reduce power and energy at system level, while our new technique will succeed

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.

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, this should eliminate the max power violation by reducing both power and energy of task x. However, it does not reduce power but actually increases total energy at the system level, because not only is DVS limited to x on the processor only, its lengthened execution time forces itself to overlap with another task b, and its co-activated task y is also forced to consume power over a longer time. Thus, energy saved by slowing down x is more than offset by energy increased by y. This 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 present a new power-aware task motion technique to evaluate a wide range of power/performance trade-offs in embedded systems. We propose a constraint model in a system-level context such that the component-level power management techniques can synergistically contribute to the improvement in both power and performance to the entire system. Based on this model, we construct a timing constraint graph that expresses the tasks in an embedded system with pair-wise constraints. Previous studies [8, 9] proposed power-directed scheduling as a way to tune the system between low-power and high-performance. In this paper, we incorporate power-aware task motion as another knob to support more aggressive design space exploration by tuning the system into new variations and evaluate the power and performance properties.

This section is organized as follows. Section 4.1 defines the fundamental concepts in our constraint model and constructs a *constraint graph* G for a scheduling problem. Section 4.2 presents the new *task motion* technique. It constructs an *iteration graph* G' that reinterprets timing constraints during task motion and illustrates the transformation of both graphs G and G' in detail. Section 4.3 introduces a new class of timing constraints, called *utilization constraints* to support more aggressive but provably correct design space exploration. Section 4.4

sketches the algorithm that combines power-directed scheduling technique and task motion.

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*. For any function σ that assigns the start times to tasks u and v as $\sigma(u)$ and $\sigma(u)$, $\sigma(v) - \sigma(u) \geq \delta(u, v)$. If $\delta(u, v) \geq 0$, edge (u, v) is called a *forward edge* that specifies a *min timing constraint*. If $\delta(u, v) < 0$, 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 of each of the tasks 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 do not violate any 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 budget of supply power that can be provided 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

In this section we illustrate a new *task motion* technique. One feature of our model that distinguishes itself from previous scheduling techniques is that, existing techniques either do not consider timing constraints $\delta(u, v)$ in their data flow graphs (DFG), or the value of $\delta(u, v)$ is always 0 or 1 that only indicates precedence (data dependency). In comparison, we capture more general min/max timing constraints that are essential to correctly model the operation in new embedded systems, and our approach subsumes DFG as a special case.

Different versions of a scheduling problem can be obtained by task motion, when intra- and inter-iteration constraints are converted into each other. 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 the dependency depth λ, since it is always zero. The expected loop duration τ is obtained from the original schedule computed from the initial iteration graph G'. Different from existing scheduling techniques where one DFG suffices, we need two graphs because the iteration graph G' must change the values of its edges $\delta'(u, v)$ on each task motion in order to correctly reinterpret the timing constraints. Existing techniques do not handle timing constraints, and their $\delta(u, v)$ values never change.

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. The procedure for a promotion to a task v transforms both constraint graph G and iteration graph G' in the following steps.

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

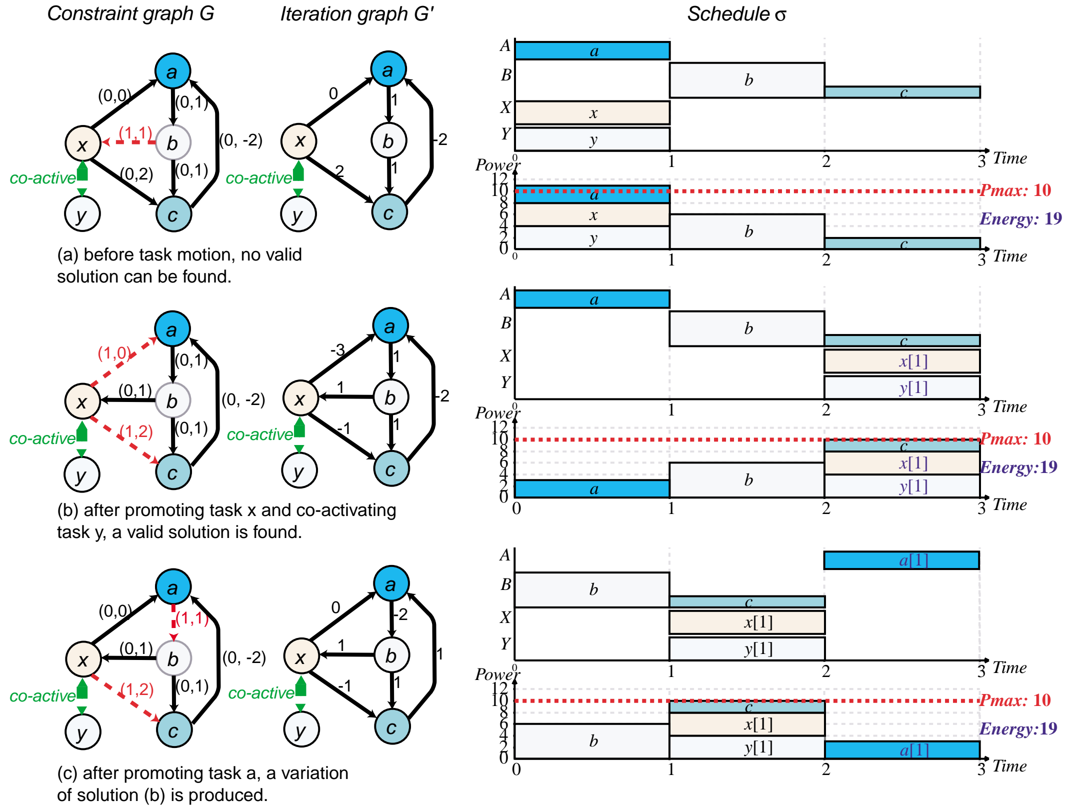

Fig. 2. Task motion under timing constraints

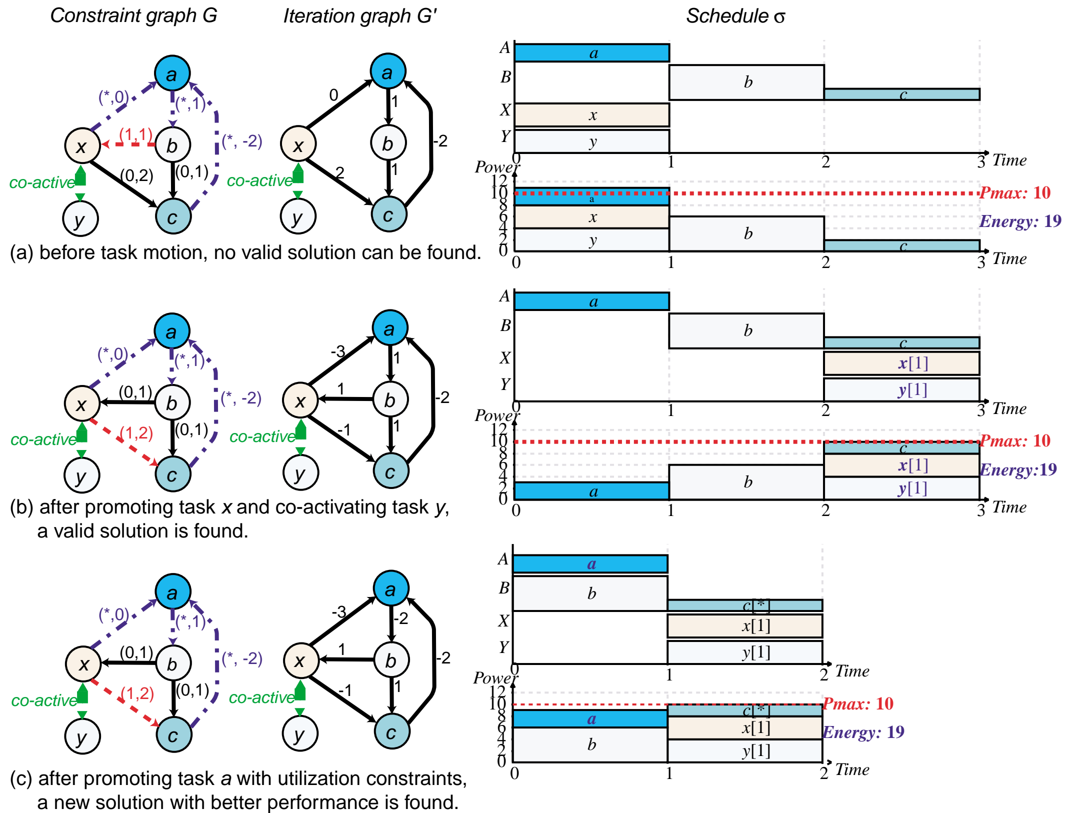

Fig. 3. Task motion under utilization constraints

constraints after promotion 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. Step (3) and (4) can be validated as follows.

When a task v is promoted by one iteration 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 (3); and vice versa, its corresponding max timing constraints will transform into new min timing constraints by step (4). Promotion effectively reduces the value of a constraint 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 our example previously shown in Fig. 1. Each edge in the constraint graph G is denoted as (λ, δ), while in iteration graph G' there is only δ. Inter-iteration constraints are marked as dashed arrows. Co-activation is denoted as a special pair of timing constraints. Fig. 2(a) shows the initial graphs G and G' and the schedule that violates the max power constraint. The original schedule has a finish time $\tau = 3$. Fig. 2(b) shows promotion to task x and its co-activated task y to produces a new valid schedule (same as Fig. 1(b), except that the prolog is not shown) which otherwise cannot be achieved without promotion. Task x can be rescheduled to time slot $[2, 3]$ because its 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)). 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. Fig. 2(c) further promotes task a. The changes to edges in G' in Fig. 2(b) are reversed, and some new edges corresponding to a are changed. It gives a variation of the solution in Fig. 2(b). If tasks b and c are promoted next, the initial constraint graph in Fig. 2(a) will be restored.

4.3 Utilization constraints

Task motion is based on the classification of inter-iteration and intra-iteration timing constraints. However, in some cases, it is difficult or unnecessary to decide whether a timing constraint should be inter-iteration or intra-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.

We define such type of constraints as *utilization-based timing constraints*, which means the constraints can be expressed as either inter-iteration or intra-iteration. 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; and it will be included in iteration graph G' initially.

Now we examine task motion under utilization constraints. It needs only minor modifications to the procedures we defined in the previous section. In step (3), the constraint $\delta'(u, v)$ is added by τ when v is promoted. 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. 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, these edges remains the same in the iteration graph G' during the promotion. For v's outgoing utilization-constraint edges, the values of constraints in G' are subtracted by the loop duration τ. As a result, utilization constraints will be always relaxed to make available more scheduling opportunities.

For example, if resource A is a heater, motors, or a CPU running a preemptible background task a with a guaranteed CPU share, then we can model it with utilization constraints. This is shown in Fig. 3(a) where utilization constraints are marked as a different type of dashed arrows. Fig. 3(b) performs promotion to tasks x and y, similar to Fig. 2(b). In Fig. 3(c), when task a with utilization constraints is promoted, the corresponding constraint values in graph G' are different from those in Fig. 2(c) in comparison. Specifically, we have the edge $\delta'(c, a) = -2$ as opposed to 1, and $\delta'(x, a) = -3$ instead of 0. Since the serialization chain formed by min constraints is broken, tasks a, b, c (after promotion to 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 with b such that an unexpected solution with shorter execution time ($\tau = 2$) is discovered, and it also satisfies max power constraint. This optimal solution could not have been obtained without using utilization constraints, which enabled more aggressive relaxation of the constraints while remaining safe.

4.4 Scheduling algorithms for power-aware task motion

In this paper we combine the efforts of power-directed scheduling [9] with system-level task motion as a way to discover a wider range of power/performance trade-offs. Our core scheduling algorithms consist of two sets of algorithms: (a)

```
ITERATION GRAPH(graph G)
    create graph G'(V, E), with G'.V := G.V, G'.E := ∅
    for each edge (u, v) ∈ G.E loop
        if (λ(u, v) = 0 or λ(u, v) = ∗) then
            add edge (u, v) to G'.E, with δ'(u, v) := δ(u, v)
        end if
    end loop
    return G'
```

Fig. 4. Algorithm to construct the iteration graph

```
PROMOTABLE(graph G, vertex v)
    for each v's incoming forward edge (u, v) ∈ G.E loop
        if (λ(u, v) = 0) then
            return FALSE
        end if
    end loop
    return TRUE
```

Fig. 5. Algorithm to decide whether a task v is promotable

transforming the problem into its new versions by task motion, and (b) power-directed scheduling for each version. From the illustration in Section 4.2 and 4.3, the implementation of (a) is straightforward from the task motion technique presented in Section 4.4. In Section 4.4 we present Algorithm (b), which is directly derived from [9] by applying power-directed scheduling to the iteration graph G' after each task promotion.

Task motion algorithms We present three algorithms for task promotion and the corresponding graph transformation to both constraint graph G and the iteration graph G'. The first algorithm in Fig. 4 constructs an iteration graph G' based on a constraint graph G with intra-iteration, inter-iteration and utilization constraints. The second algorithm in Fig 5 decides whether a task v is promotable by checking v's incoming forward edges. If they consist of only inter-iteration and utilization constraints, or if v does not have any incoming forward edges, then v is promotable. The third algorithm in Fig 6 promotes a task v by transforming both graphs G and G' with an expected loop duration τ.

Algorithm for power-aware task motion/scheduling The algorithm is shown in Fig. 7. It first constructs a iteration graph G' from the constraint graph G. Then G' is scheduled by a power-aware scheduler, which is derived from [9]. The returned schedule σ is kept as a temporarily best schedule and whose duration τ_σ is taken as the expected loop duration τ. Then the algorithm traverses all vertices in $G.V$ in a topological order by extracting one promotable task v at each step. When a task v is promoted, both graphs G and G' are

```
PROMOTE(graph G, graph G', vertex v, time τ)
    for each v's incoming forward edge (u, v) ∈ G.E loop # step 1
        if (λ(u, v) ≠ ∗) then
            λ(u, v) := λ(u, v) − 1
        end if
        if (λ(u, v) = 0) then
            add edge (u, v) to G'.E with δ'(u, v) := δ(u, v)
        end if
    end loop
    for each v's outgoing forward edge (v, u) ∈ G.E loop # step 2
        if (λ(v, u) ≠ ∗) then
            λ(v, u) := λ(v, u) + 1
        end if
    end loop
    for each v's incoming edge (u, v) ∈ G'.E loop # step 3
        if (λ(u, v) ≠ ∗ and δ(u, v) < 0) then
            δ'(u, v) := δ'(u, v) + τ
        end if
    end loop
    for each v's outgoing edge (v, u) ∈ G'.E loop # step 4
        δ'(v, u) := δ'(v, u) − τ
    end loop
    return
```

Fig. 6. Task promotion algorithm

```
POWER AWARE TASK PROMOTION(graph G, P_max, P_min)
    G0 := G; Alt := ∅
    G' := ITERATION GRAPH(G)
    σ := POWER AWARE SCHEDULING(G', P_max, P_min) # [9]
    Ec := Ec_σ(P_min); τ := τ_σ
    V' := G.V; V_prolog := ∅
    for each v ∈ V' loop
        if PROMOTABLE(G, v) then
            V' = V' − {v}
M:          PROMOTE(G, G', v, τ)
            if (G ∈ Alt) then
                break
            end if
            σ' := POWER AWARE SCHEDULING(G', P_max, P_min)
            if (τ_σ' ≠ τ) then
                Alt := Alt + {G}; V' = V' + {v}
                undo step M
            else
                if (Ec_σ' < Ec) then
                    σ := σ'; V_prolog := V'
                end if
            end if
        end if
    end loop
    if (V_prolog = G.V or V_prolog = ∅) then
        return σ, ∅, ∅, Alt
    end if
    V_epilog := G.V - V_prolog
    σ_prolog = PROLOG(G0, V_prolog, σ)
    σ_epilog = EPILOG(G0, V_epilog, σ)
    return σ, σ_prolog, σ_epilog, Alt
```

Fig. 7. Power-aware task motion algorithm

updated. Then the power-aware schedule is invoked again to examine whether an improved schedule with the same execution time and less energy cost can be found, and the better schedule is stored. In case a schedule with a different finish time is found, it indicates another version of the loop with different performance (and power as well). It is not appropriate to simply discard the slower schedule, because it could represent a different power/performance trade-off. Instead, the graphs leading to the incompatible versions are stored in set *Alt*, and the algorithm cancels the last promotion and attempts another topological ordering. The algorithm completes if all tasks are promoted, or the topological traversal cannot proceed since the next promotion always generates an incompatible version. Finally, it computes two additional schedules, one for the prolog and one for the epilog to start and finish the partial loop iteration (algorithms not shown). Once the algorithm finds the best schedule σ, it returns the full set of schedules that includes σ_{prolog}, σ, and σ_{epilog}. The algorithm also returns the set *Alt* that contains graphs leading to alternative solutions. These graphs will be examined by the same algorithm to evaluate different power/performance trade-offs.

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 the 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 as metrics:

(0) the existing manual solution (fully serialized),
(I) power-aware scheduling [9],
(II) task motion without utilization constraints,
(III) task motion with utilization constraints.

- For scenario 1 (14.9W solar power), all schedulers except JPL's (0) computed fast schedules (i.e., short τ), but, these three solutions vary in energy cost. Solutions by schedulers I and II must draw from the battery in addition to the solar power in order to achieve the same performance as scheduler III. Scheduler III could not have achieved this 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).

This result shows 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.

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

Suppose the rover is traveling to a target location in a distance of 48 steps. 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 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 very 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 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 33% speedup and 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 first showed the pitfalls of applying existing power management techniques without considering system-level dependency like co-activation, and this resulted in counterintuitive, incorrect schedules that violated max power constraints. We then showed our constraint formulation and task motion to safely transform the tasks while respecting these system-level dependencies. We further enhanced task motion by introducing a class of 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 under contract F33615-00-1-1719. It represents a collaboration between the University of California at Irvine and the NASA/Cal Tech Jet Propulsion Laboratory. Special thanks to Dr. N. Aranki, Dr. B. Toomarian, Dr. M. Mojarradi and Dr. J. U. Patel at JPL and Kerry Hill at AFRL for their discussion and assistance.

References

1. NASA/JPL's Mars Pathfinder home page. http://mars3.jpl.nasa.gov/MPF/-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. 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.
4. 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.
5. 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.
6. 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.
7. C. Leiserson and J. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1990.
8. 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.
9. 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.
10. 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.
11. F. Sanchez and J. Cortadella. Time-constrained loop pipelining. In *Proc. International Conference on Computer-Aided Design*, pages 592–596, November 1995.
12. 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.
13. 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.
14. 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.
15. 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.