A Hybrid Framework for Application Allocation and Scheduling in Multicore Systems with Energy Harvesting

Yi Xiang and Sudeep Pasricha
Department of Electrical and Computer Engineering,
Colorado State University, Fort Collins, CO, USA
E-mail: {yix, sudeep}@colostate.edu

Abstract - In this paper, we propose a novel hybrid design-time and run-time framework for allocating and scheduling applications in multi-core embedded systems with solar energy harvesting. Due to limited energy availability at run-time, our framework offloads scheduling complexity to design time by creating energy-efficient schedule templates for varying energy budget levels, which are selected at run-time in a manner that is contingent on the available harvested energy and executed with a lightweight slack reclamation scheme that extracts additional energy savings. Our experimental results show that the proposed framework produces energy-efficient and dependency-aware schedules to execute applications under varying and stringent energy constraints, with 23-40% lower miss rates than in prior works on harvesting energy-aware scheduling.

Categories and Subject Descriptors
C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems -- *Real-time and embedded systems*. B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

Keywords
Mapping and scheduling, energy harvesting, DVFS

1. INTRODUCTION
Recent years have witnessed a significant increase in the use of multi-core processors in low-power embedded devices [1]. With advances in parallel programming and power management techniques, embedded devices with multi-core processors are outperforming single-core platforms in terms of both performance and energy efficiency [2]. But as core counts increase to cope with rising application complexity, techniques for efficient run-time workload distribution and energy management are becoming extremely vital to achieving energy savings in emerging multi-core embedded systems.

Energy autonomous systems are an important class of embedded systems that utilize ambient energy to perform computations without relying on an external power supply or frequent battery charges. As the most widely available energy source, solar energy harvesting has attracted a lot of attention and is rapidly gaining momentum [3][4][5]. Due to the variable nature of solar energy harvesting, deployment of an intelligent run-time energy management strategy is not only beneficial but also essential for meeting system performance and reliability goals. Such a strategy must possess low overhead, so as to not stress the limited available energy budget at run-time. A few prior efforts have explored workload scheduling for embedded systems with energy harvesting, e.g., [6][7][8]. However, all of these efforts are aimed at idealized independent task models, and cannot be easily extended to more complex application sets that possess inter-node data dependencies, such as the workloads represented by direct acyclic graphs (DAGs).

In this paper, we propose a low-overhead hybrid workload management framework (HyWM) to address the problem of allocating and scheduling multiple applications on multicore embedded systems powered by energy harvesting. Compared to related prior work, the novelty and main contributions of our work can be summarized as follows:

—A novel hybrid workload management framework (HyWM) is proposed that integrates a comprehensive design-time analysis methodology with lightweight run-time components for low-overhead energy management;
—A mixed integer linear programming (MILP) optimization formulation is proposed to solve DAG scheduling problems at design time, generating schedule templates composed of optimized execution schedules for various energy budgets;
—A fast run-time scheduling scheme is proposed that selects the best-fit schedule template based on the available energy budget, which is applied with a lightweight slack reclaiming heuristic for additional energy saving by exploiting slack time of tasks finishing before their worst case execution time (WCET).

2. RELATED WORK
Many prior works have focused on the problem of run-time management and scheduling for embedded systems with energy harvesting. Moser et al. [6] proposed the lazy scheduling algorithm (LSA) that executed tasks as late as possible, reducing task deadline miss rates when compared to the classical earliest deadline first (EDF) algorithm. However, LSA does not consider frequency scaling for energy saving and thus is not suitable for emerging power-constrained environments. Liu et al. [7] exploited scaling capability of processors by slowing down execution speed of arriving tasks as evenly as possible, which saves energy because a processor's dynamic power dissipation is generally a convex function of frequency. Xiang et al. [8] proposed a battery-supercapacitor hybrid energy harvester that helps reduce energy supply variability, allowing tasks to execute more uniformly to save energy. *However, none of these prior works take inter-task dependency into consideration*.

Several other efforts have explored mapping and scheduling for task graph based workloads. Luo et al. [9] proposed a hybrid technique to find a static schedule for known periodic task graphs at design time with the flexibility to accommodate aperiodic tasks dynamically at run-time. Sakellariou et al. [10] proposed hybrid heuristics for DAG scheduling on heterogeneous processor platforms. Coskun et al. [11] proposed a hybrid scheduling framework that adjusts task execution schedule dynamically to reduced thermal hotspots and gradients for MPSoCs. *However, all these prior efforts cannot maintain performance when applied to energy harvesting systems that possess a fluctuating energy supply at run-time. Some of these efforts also do not focus on energy as a design constraint*. Our work is the first hybrid framework that targets the problem of energy-aware allocation/scheduling of multiple co-executing task graphs in energy harvesting based multicore platforms.

GLSVLSI'14, May 21–23, 2014, Houston, Texas, USA.

ACM 978-1-4503-2816-6/14/05…$15.00.
http://dx.doi.org/10.1145/2591513.2591527

3. PROBLEM FORMULATION

3.1 System Model

This paper focuses on hybrid allocation and scheduling of multiple task-graph applications with real-time deadlines on multi-core embedded systems with solar energy harvesting, as shown in Figure 1. The key components and assumptions of our system model are described in the following sub-sections.

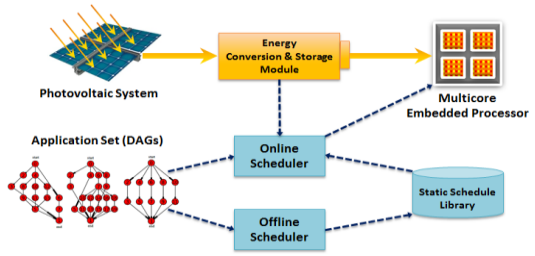

Figure 1. DAG workload based multicore embedded system platform with solar energy harvesting

3.1.1 Energy Harvesting and Energy Storage

A photovoltaic (PV) system is used as the power source for our multicore embedded system, converting ambient solar energy into electric power. Naturally, the amount of harvested power varies over time due to changing environment conditions. To cope with the unstable nature of the solar energy source, an energy conversion and storage module is required to bridge the photovoltaic system with the embedded processor efficiently [21]. We assume that our run-time scheduler can cooperate with this module to inquire about the amount of energy available in storage. We adapt the hybrid battery-supercapacitor energy storage module proposed by Xiang et al. [8] in our work.

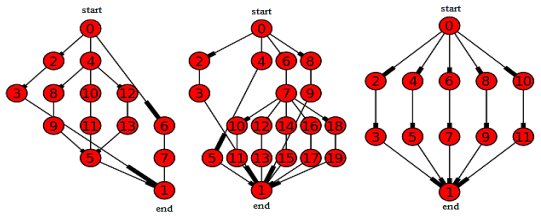

Figure 2. Examples of applications modeled as directed acyclic graphs

3.1.2 Application Workload Model

In this work, we consider multi-core systems hosting multiple recursive real-time applications modeled as periodic task graphs, ψ: $\{G_1, \ldots, G_{Ng}\}$, such as the examples shown in Figure 2. Each of the N_g applications is represented by a weighted directed acyclic graph (DAG), denoted as G_i: (t_i, e_i, T_i, D_i), $i \in \{1, \ldots, N_g\}$, which contains a set of task nodes, t_i: $\{\tau_1, \ldots, \tau_j\}$ with worst-case execution cycles, $WCEC_i$, (number of CPU clock cycles needed to finish a task i in the worst case); and a set of directed edges, e_i: $\{\varepsilon_1, \ldots, \varepsilon_j\}$, used to represent inter-task dependences with communication (inter-core data transfer) delay from source to destination nodes represented as $COMM_{src,dst}$. A task node can have multiple dependences to/from other nodes, forking/rejoining execution paths in the task graph. We assume that every task graph's execution paths rejoin at its last task node, which accumulate results and conclude execution of the task graph.

Apart from task nodes and edges, every periodic task graph has its unique period, T_i and relative deadline, D_i. At the beginning of each period, a new instance of a task graph will be dispatched to the system for execution. The task graph's relative deadline, D_i, is the time interval between the task graph instance's arrival time and deadline. A task graph instance misses its deadline if it cannot finish executing all task nodes before its deadline. In this work, we assume that D_i equals T_i, i.e., for a periodic task graph, its instance has to finish execution or be dropped before the arrival of the next instance.

3.1.3 Multicore Platform with DVFS Capability

We consider a homogeneous multi-core embedded processing platform with dynamic voltage and frequency scaling (DVFS) capability at the core level. For inter-core communication, a network-on-chip (NoC) architecture is used with a 2D mesh topology and dimension order packet routing over conflict-free TDMA virtual channels. Each core on the processor has N_l discrete frequency levels: φ: $\{L_0, \ldots, L_{Nl}\}$. Each level is characterized by L_j: (v_j, p_j, f_j), $j \in \{1, \ldots, N_l\}$, which represents voltage, average power, and frequency, respectively.

Table 1. Processor power and frequency levels [19]

Level	1	2	3	4	5
Power(mW)	80	170	400	900	1600
Frequency(MHz)	150	400	600	800	1000
Energy Efficiency	1.875	2.353	1.5	0.889	0.625

We consider power-frequency levels for each processor core as shown in Table 1. Typically, the dynamic power-frequency function is convex. Thus, a processor running at lower frequency can execute the same number of cycles with lower energy consumption. However, this is not always the case when static power is considered. To find an energy optimal frequency, we represent energy efficiency of a frequency level L_j by δ_j = *cycles executed / energy consumed* = f_j / p_j. From Table 1 we can conclude that level 2 is the most energy efficient because executing at this level consumes the least energy for a given number of cycles. Besides, we assume idle power of 40mW when no workload is available for execution.

The computation utilization of a periodic task graph (U_{comp}) is defined as the sum of execution times of all its task nodes for the highest processor clock frequency divided by its period:

$$U_{comp\,i} = \frac{\sum_j WCET_{i,j}/f_{max}}{T_i}, i \in \{1, \ldots, N_g\} \quad (1)$$

Similarly, we define communication utilization of a periodic task graph (U_{comm}) as the sum of the communication times for all of its edges divided by the task graph's period:

$$U_{comm\,i} = \frac{\sum_k COMM_{i,k}}{T_i}, i \in \{1, \ldots, N_g\} \quad (2)$$

The computation/communication utilization of the entire multi-application workload is simply the accumulation of utilizations for all task graphs, which provides an indication of the overall workload intensity of a given DAG application set.

3.1.4 Run-Time Scheduler

This module is an important component of the system for run-time information gathering and dynamic application execution control. The online scheduler gathers information by monitoring the energy storage medium and the multi-core processor (Figure 1). The gathered information, together with preloaded schedule template library generated by the offline scheduler for the given workload, allows the run-time scheduler to coordinate operation of the multi-core platform at run-time.

3.2 Problem Objective

Our primary objective is to allocate and schedule the execution of a workload composed of multiple application task graphs (DAGs) running in parallel simultaneously at run-time such that total task graph miss rate is minimized. Our workload management framework must react to changing harvested energy dynamics to schedule as many of the task graph instances as possible without overloading the system with complex re-scheduling calculations at run-time.

4. HYBRID SCHEDULING FRAMEWORK

4.1 Motivation and System Overview

The problem of scheduling weighted directed acyclic graphs (DAGs) on a set of homogeneous cores under optimization goals and constraints is known to be NP-complete [12]. This paper addresses the even more difficult problem of scheduling on systems that rely entirely on limited and fluctuating solar energy harvesting. *The limited energy supply prevents the deployment of complex scheduling algorithms at run-time.* Moreover, *execution of applications that will not have enough energy or computation resources to complete due to shortages in harvested solar energy can lead to significant wasted energy with no beneficial outcome.* To address these challenges, we propose a hybrid workload management framework (HyWM) that combines *template-based hybrid scheduling* with a novel *energy budget window-shifting* strategy to decouple run-time application execution from the complexity of DAG scheduling in the presence of fluctuations in energy harvesting.

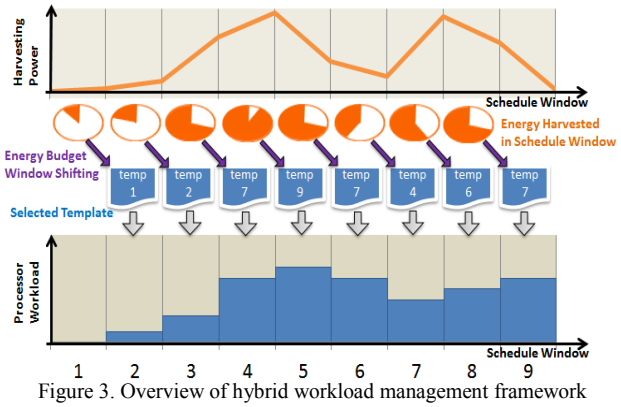

Figure 3. Overview of hybrid workload management framework

An important underlying idea in our framework, as shown in Figure 3, is time-segmentation during run-time workload control that creates an independent stable energy environment for run-time scheduling within each segment. The time of system execution is divided into *schedule windows* with identical length, which is referred to as the *hyper-period* of the DAGs. An energy budget is assigned to a schedule window at its beginning, based on the amount of harvested and unused energy from the previous window. This conservative budget assignment scheme, called *energy budget window shifting*, can delay utilization of harvested energy slightly to ensure that dynamic variations in energy harvesting do not halt executing applications in subsequent windows. The run-time scheduler knows the amount of energy that is available at the beginning of each window, and selects the best-fit schedule template generated at design time based on this known energy budget.

In the rest of this section, we describe our mixed integer linear programing (MILP) formulation for design-time template generation (section 4.2) and the run-time scheduler with a lightweight slack reclamation heuristic (section 4.3).

4.2 MILP for Offline Template Generation

We formulated a mixed integer linear program (MILP) to aid with the generation of optimal task scheduling templates at design time. In this section, we give an overview of our MILP formulation that aims to minimize miss rate for DAG instances in a schedule window under a given energy budget constraint. The constructed formulation is solved multiple times offline with different energy budget constraints to generate a set of schedule templates for the run-time scheduler to select. As our formulation focuses on workload management within an independent schedule window, periodic task graphs in set ψ are unrolled into a set of all task graph instances that arrive within a schedule window, ψ^+: $\{G_1, \ldots, G_{Ni}\}$. Our target processor is set to have N_c number of cores with N_l discrete frequency levels.

4.2.1 Inputs and Decision Variables

For our MILP formulation, we provide several inputs that represent the energy budget and characteristics of the target workload and platform, as shown in Table 2. The energy budget parameter (*ENGY_BGT*) allows different schedule template outcomes, such that each of them can best match the energy budget available. The $WCET_{j,l}$ and $ENGY_{j,l}$ paramters are calculated based on worst case execution cycles (*WCEC*) of every task node for every frequency level supported by the processor (as per the discussion in section 3.1.3).

Table 2. Inputs for MILP formulation

Inputs	Description
EGY_BGT	energy budget of the schedule template to generate
$ARRIVAL_i$	arrival time of task graph instance *i*
$DDLINE_i$	deadline of task graph instance *i*
$WCET_{j,l}$	worst-cast execution time of task node *j* at frequency level *l* , $l \neq 0$
$ENGY_{j,l}$	energy consumption of task node *j* at frequency level *l*, when $l = 0$, $ENGY_{j,0} = 0$
$COMM_{src,dst}$	communication delay when preceding node *src* and descendent node *dst* are allocated to separate cores
Ni, Nt, Nl, and Nc	number of task graph instances, number of task nodes, number of frequency levels, and number of cores

* In our formulation, task nodes can be indexed in two different ways:
(1) Local ID: tuple *(i, j)* for task node *j* of task graph *i*
(2) Global ID: single variable *j* for task node *j* in the entire set

For our MILP problem, there are two major requirements for decision variables: firstly, they must form a complete representation of a feasible execution schedule; secondly, they should make it possible to represent all constraints and objective as linear formulations. The decision variables used in our formulation are shown in Table 3. The binary indicators of task graph miss, $miss_i$, are used to construct the major part of the objective function. For $freq_{j,l}$, when $l = 0$, it indicates that the task node *j* is not scheduled for execution and is thus to be dropped. The two indicators $dec_{j,j'}$ and $bef_{j,j'}$ are used to construct constraints that arrange timing of the task nodes without direct dependencies.

Table 3. Design variables of MILP formulation

Variables	Description
$miss_i$	binary variable to indicate if task graph instance *i* is missed
$start_{(i,j)}$	Execution start time of task graph *i* on node *j*. Note that we also use variable $end_{i,j}$ as the end time of execution. Our schedule does not consider task preemption so that $end_{i,j} = start_{i,j} + WCET_{i,j}$
$freq_{j,l}$	binary variable which indicates if task node *j* is assigned with frequency level *l*
$alloc_{j,k}$	binary variable which indicates if task node *j* is mapped to core *k*, $k \neq 0$
$dec_{j,j'}$	binary variable which indicates if task nodes *j* and *j'* are NOT mapped to the same core (decoupled)
$bef_{j,j'}$	binary variable which indicates if task node *j* is scheduled before *j'*

4.2.2 Optimization Objective

In our formulation, the major objective is to minimize the number of misses of task graph instances in a schedule window. Additionally, we include an auxiliary objective: the percentage of energy budget used, so that the MILP optimization also searches for a schedule with the least energy consumption possible. Note that this auxiliary objective does not sacrifice minimization of miss rate for less energy consumption, as the energy minimization term in the objective function always has less impact on the objective function value than any single task graph instance miss. The objective formulation is shown below:

$$\text{Min:} \quad \sum_{i=1}^{Ni} miss_i + \sum_{j=1}^{Nt}\sum_{l=0}^{Nl}(ENGY_{j,l} \times freq_{j,l}) \Big/ EGY\_BGT \qquad (3)$$

4.2.3 Constraints

The constraints in our formulation guarantee the satisfaction of the energy budget constraint and correctness of the execution schedule for the target workload and platform. The key constraints are described as follows:

1) Energy constraint for a schedule window: Total energy consumption of all task nodes at their assigned frequency levels should be less or equal to energy budget:

$$\sum_{j=1}^{Nt}\sum_{l=0}^{Nl}(ENGY_{j,l} \times freq_{j,l}) \le EGY\_BGT \qquad (4)$$

2) Timing constraints for task graph scheduling: We formulate multiple constraints, which in combination form a complete timing constraint for all task graph instances and their task nodes, as illustrated in Figure 4.

(2.a) Timing constraints for graph instances: The two constraints below confine start time of the first task node and end time of the last task node to ensure that timing requirements of their task graph instances are satisfied, as illustrated in Figure 4 (a.1, a.2).

$$start_{(i,1)} \ge ARRIVAL_i - M \times miss_i \qquad i \in [1, N_i] \qquad (5)$$

$$end_{(i,-1)} = start_{(i,-1)} + \sum_{l=1}^{Nl}(WCET_{(i,-1),l} \times freq_{(i,-1),l}) \qquad (6)$$
$$end_{(i,-1)} \le DDLINE_i + M \times miss_i \qquad i \in [1, N_i]$$

We use a sufficiently large constant, *M*, in the formulation to equivalently represent "if" statements that cancel out constraints when $miss_i = 1$ *(graph instance dropped)*. The constraints can be canceled out when $miss_i = 1$ because large values of *M* ensure that the inequality is satisfied for any variable values in range. In the rest of this paper, we use the same approach for "if" statements. However, for the purpose of intuitive representation, the following sections show "if" statements explicitly.

(2.b) Timing constraints for task nodes with dependencies: The type of constraints shown below model dependencies by forcing destination task nodes to start only after their predecessor nodes have finished. Also it takes communication cost into consideration when two dependent nodes are decoupled (not allocated to the same core), as illustrated in Figure 4 (b.1, b.2):

$$\textbf{if } miss_i = 0: \qquad (7)$$
$$end_{(i,src)} + COMM_{src,dst} \times dec_{src,dst} \le start_{(i,dst)}$$
$$\in [1, N_i], \quad (src, \text{dst}) \in edges\ of\ G_i\ , G_i \in \psi^+$$

(2.c) Timing constraints for task nodes without dependencies: The type of constraints shown below address the fact that task nodes allocated to the same core cannot overlap their execution times, as each core executes only one task at a time without preemption, as shown in Figure 4 (c).

$$dec_{j,j'} \le 2 - allocs_{j,k} - allocs_{j',k} \qquad (8)$$
$$j \in [1, Nt], j' \in [1, Nt], j \ne j', k \in [1, Nc]$$

$$dec_{j,j'} \ge allocs_{j,k} + allocs_{j',k'} - 1 \qquad (9)$$
$$j \in [1, Nt], j' \in [1, Nt], j \ne j'$$
$$k \in [0, Nc], k \in [1, Nc], k \ne k'$$

These constraints represent relations between task node allocation variables, $alloc_{i,k}$, and node pair decoupling variables, $dec_{j,j'}$. The constraint in (8) ensures that the pair decoupling variable is equal to 0 when task nodes are on a same core. The constraint in (9) forces the decoupling variable to be 1 when two task nodes are found to be allocated to any two different cores.

With the value of $dec_{j,j'}$ available, the following constraints are used to avoid timing conflicts for every pair of task nodes:

$$bef_{j,j'} + bef_{j',j} - dec_{j,j'} = 1 \qquad (10)$$

$$\textbf{if } bef_{j,j'} = 1: \quad end_j < start_{j'} \qquad (11)$$
$$\textbf{if } bef_{j',j} = 1: \quad end_{j'} < start_j$$
$$j \in [1, Nt], j' \in [1, Nt], j \ne j' \text{ for (10) and (11)}$$

The constraint in (10) implies that the task node *j* should be scheduled either before or after task node *j'* when they are allocated on the same core. Based on the scheduled order of these two tasks, the constraint in (11) ensures that the task node only starts when earlier scheduled task nodes are finished. When two task nodes are decoupled to two different cores, the constraints in (11) cancel out.

3) Constraints for target platform: The type of constraints shown below guarantee that only one frequency level and at most one core are selected for execution of each task node:

$$\sum_{l=0}^{Nl} freq_{j,l} = 1, \quad j \in [1, N_t] \qquad (12)$$
$$\sum_{k=1}^{Nc} alloc_{j,k} \le 1, \quad j \in [1, N_t] \qquad (13)$$
$$\textbf{if } freq_{j,0} = 0: \quad \sum_{k=1}^{Nc} alloc_{j,k} = 1, \quad j \in [1, N_t] \qquad (14)$$

A task is indicated as dropped in the generated schedule when its frequency level is set to 0. The constraint in (14) ensures that all tasks that are not dropped will be allocated to a core; otherwise they may end up being executed on a "ghost core" to escape timing constraints with other tasks.

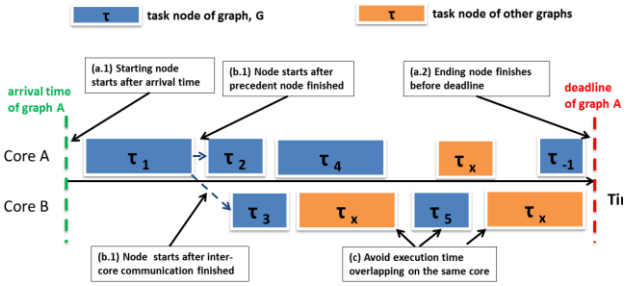

Figure 4. Illustration of timing constraints for periodic task graph set

All of the above constraints are necessary to create a correct, feasible and optimal set of schedule templates. We also establish additional constraints (not shown for brevity) to eliminate obviously sub-optimal solutions and reduce the search space for the MILP solver.

4.3 Online Template and Slack Management

The main goal of our run-time scheduler is to monitor harvested solar energy and select the best-fit template for an upcoming schedule. With schedule templates generated at design time and energy budgets provided at beginning of each schedule window, this is a low-overhead operation, done by selecting the schedule template that finishes the most task graph instances, contingent on the energy budget. Most importantly however, the run-time scheduler must be able to adjust schedule templates as necessary to best cope with system uncertainty at run-time. A typical example of such uncertainty is the actual execution times taken by tasks, which are often shorter than WCET of tasks due to variances in system status and input parameters. [22] In this work, we propose a lightweight run-time slack reclamation heuristic, as shown in Algorithm 1.

Algorithm 1 Run-time slack reclamation

For task τ that finishes at time end'$_\tau$ and has designated finish time end$_\tau$:

```
1  if end'_τ < end_τ
2      find next task scheduled to execute on the core, τ'
3      if task τ' is ready to execute
4          slack_time = start_τ' – end'_τ
5          start_τ' = end'_τ
6          if slack_time > WCET_τ',Lτ' – WCET_τ',Lτ'-1
7              Lτ' = Lτ' – 1
```

The heuristic monitors each core separately to find slack

time for additional energy saving. The heuristic is triggered when the actual finish time of a task node is earlier than its schedule finished time based on its WCET. It then finds the next task schedule to execute on the same core and inquires about its precedencies from the run-time scheduler. If the next task is ready to execute, the scheduler will find out if it is possible to execute this task with lower frequency while finishing this task no later than its designated finish time as per the schedule template, reclaiming the slack time for energy savings. Otherwise, this task will execute with its designated frequency immediately, earlier than its designated start time so that it will finish early and pass on the slack time for upcoming tasks.

5. EXPERIMENTAL RESULTS

5.1 Experiment Setup

We developed a simulator in C++ to evaluate our proposed hybrid workload management scheme (HyWM). For offline schedule template generation, we wrote a python script that constructs the data structure of task graphs using the NetworkX package and formulates the MILP problem using a GNU linear programming kit (GLPK) [15]. We chose the Gurobi Optimizer [16] as our MILP solver to generate the optimal schedule templates. We used task graphs for free (TGFF) [18] for pseudo-random task graph generation for most experiments and the distribution of actual execution time of task nodes is obtained from [22]. In addition, we also utilized task graphs of real applications (FFT, Gaussian Elimination, and MPEG) [23]. In the rest of this section, we first analyze characteristics of the generated schedule templates and then study the overall system performance of our proposed hybrid workload management scheme in comparison with prior work.

5.2 Analysis of Generated Schedule Templates

In this set of experiments, we check the quality and optimality of the schedule templates generated using our MILP approach. We randomly generated four periodic task graphs with computation utilization set to 0.8×4 and communication utilization set to 0.15×4, i.e., a total workload utilization of 0.95×4. Based on the periods of the generated task graphs, we set the length of schedule window to be 1 minute, within which 9 task graph instances arrive in the system for execution. We generated 11 schedule templates with energy budgets evenly distributed from 0 to E_{peak}, which is the peak energy budget available from solar energy harvesting (240 Joules).

Table 4. Results of MILP based schedule template generation

Schedule template ID	Energy budget	Objective value	Energy budget usage	Energy usage	Number of misses
0	0J	9.000	NA	0J	9
1	24J	7.788	78.8%	18.9J	7
2	48J	5.838	83.8%	40.2J	5
3	72J	4.867	86.7%	62.4J	4
4	96J	3.882	88.2%	84.7J	3
5	120J	2.891	89.1%	106.9J	2
6	144J	2.743	74.3%	106.9J	2
7	168J	1.940	94.0%	157.9J	1
8	192J	1.823	82.3%	157.9J	1
9	216J	1.739	73.9%	157.9J	1
10	240J	0.957	95.7%	229.6J	0

The results of the schedule template generation for a system with four cores are shown in Table 4. We can observe that schedule template 10, with a peak energy budget can finish all task instances in time, showing the competence of our MILP optimization to deal with stringent timing constraints even for heavy workloads with per-core utilization as high as 0.95. Note also that while 95.7% of E_{peak} is required to finish all instances, template 3 with energy budget less than 1/3rd of E_{peak} managed to successfully schedule more than half of the instances. *Thus our approach can be seen to create superior schedules even under highly constrained energy budget requirements.* The notable schedule performance is a reflection of our MILP optimization approach that finds the optimal schedule by sacrificing more energy-hungry task graph instances, reserving energy for less energy-hungry ones, and scaling down execution frequency whenever possible for optimal energy efficiency, thereby minimizing miss rate of task graphs.

To study the quality of schedule templates from another perspective, we show how our MILP optimization approach selects frequencies for task nodes under different energy budget constraints, as shown in Figure 5. We can observe from the figure that templates with higher energy budgets utilize higher frequency levels more frequently than templates with lower budgets. On the other hand, templates with lower energy budget drop more tasks and slow down execution for better energy efficiency. Note that the 150MHz frequency is never used by any schedule; this is due to the fact that the frequency level of 150MHz has lower efficiency and lower speed than the 400MHz level (see Table 1). Therefore our MILP optimization approach rules out this sub-optimal frequency choice as it is always better to schedule at 400MHz instead.

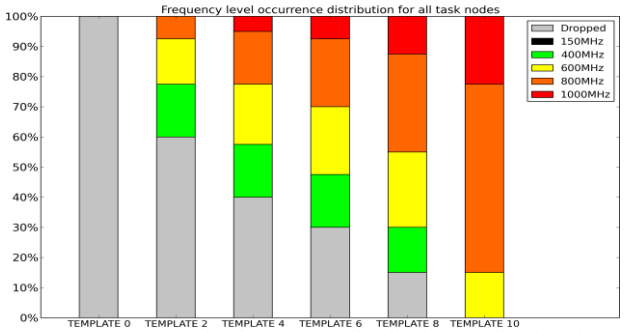

Figure 5. Frequency level occurrence distribution for all task nodes

5.3 Evaluation of System Performance

In this section, we compare the overall system task graph miss rate performance of our hybrid workload management framework (HyWM) against workload management approaches proposed in prior work. Our simulation uses realistic energy harvesting profiles based on historical weather data from Golden, Colorado, USA, provided by the Measurement and Instrumentation Data Center (MIDC) of the National Renewable Energy Laboratory (NREL) [17]. We evaluate the system's performance over a span of 750 minutes, from 6:00 AM to 6:30 PM when solar radiation is available.

To compare our approach with state-of-the-art approaches, we implemented three recent works: 1) UTA [24], which first executes schedulable workload at full speed and then drop tasks when energy is insufficient; 2) SDA [8], which divides system execution time into segments and selects a stable frequency to execute a subset of the workload that can be supported by the assigned energy budget; and 3) LP+SA [20], which finds a feasible but non-optimal schedule using MILP, and uses this schedule as an initial solution to a simulated annealing (SA) based heuristic that finds a near-optimal solution. To compare HyWM with these approaches, we adapt the techniques to our environment and problem formulation. As UTA and SDA are designed for energy-constrained scheduling of independent periodic tasks while our workload consists of multiple task graphs, we enhance these techniques so that our scheduler module analyzes inter-task dependency and provides ready task nodes for the techniques to schedule. In LP+SA, the original approach focuses on task graph scheduling while minimizing

energy but without awareness of energy harvesting and not considering task dropping. We enhanced LP+SA by dropping tasks iteratively till the remaining task sets meet the energy budget, and these task sets are then sent as inputs to LP+SA.

The results of our comparison study on random task graphs are shown in Figure 6. The figure shows the total task graph miss rate for three different platform complexities (with 4, 8, and 16 cores). For the platform with 4 cores, it can be observed that UTA has the highest miss rate. This is because UTA executes tasks at full speed and task dependencies in the workload make its slack reclamation techniques practically unusable. The SDA technique generates better results by considering an energy budget for each scheduling window and assigning lower frequencies to avoid violating the budget. However SDA does not consider task dependencies and thus all nodes in the task graph are assigned the same frequencies, resulting in a less efficient schedule. LP+SA outperforms UTA and SDA as it can generate task dependency-aware offline schedules after comprehensive design space exploration unlike in UTA or SDA. However, the superior offline schedules obtained using our MILP formulation in the HyWM framework coupled with its intelligent run-time template selection and slack reclamation techniques allow HyWM to outperform all of these efforts. HyWM reduces absolute miss rate by 9.2% compared to LP+SA, 11.4 % compared to SDA, and 20.2% compared to UTA. In terms of relative performance improvement, HyWM accomplishes an improvement of 23.4%, 27.4%, and 40.1% over LP+SA, SDA, and UTA, respectively.

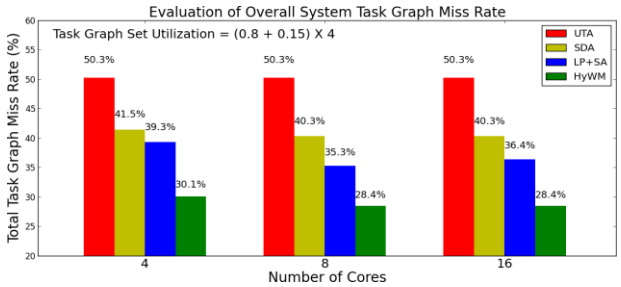

Figure 6. Comparison between proposed HyWM framework and prior work ([8], [20], [24]) in terms of overall system task graph miss rate

Figure 6 also shows the scheduling performance of these frameworks for platforms with a greater number of available cores while keeping the workload and energy budget the same. When the core count doubles from 4 to 8, our *HyWM* framework achieves a lower miss rate compared to other techniques, as it can better distribute workload across more cores, directing these cores to operate at a lower execution frequency and with better energy efficiency. However, the system with 16 cores shows no further improvements because there is no additional parallelism available in our workload to make use of the 16 cores. Note that LP+SA shows a deteriorated result on 16 cores because even though there is no more parallelism to exploit, the search space of its SA heuristic enlarges, leading to slightly worse near-optimal solutions.

We also conducted additional experiments on a real application DAG workload composed of FFT, Gaussian transformation, and MPEG encoder applications running simultaneously. Table 5 shows results for these applications on a 4-core platform configuration. Once again, our proposed HyWM framework can be seen to outperform other frameworks proposed in prior work to generate better quality results.

Table 5. Miss rate on real application DAG set with 4 cores

UTA	SDA	LP+SA	HyWM
52.87%	48.49%	43.02%	35.13%

6. CONCLUSIONS

In this paper, we proposed a novel workload management framework (HyWM) for allocating and scheduling on multi-core embedded systems powered by solar energy harvesting with workload consists of multiple task-graphs that must be executed in parallel in an energy constrained environment. Our experimental results show that the proposed framework produces energy-efficient and dependency-aware schedules to execute task graphs under varying and stringent energy constraints, with 23-40% lower miss rates than the best known recent prior works on energy-harvesting aware scheduling.

REFERENCES

[1] Arm Cortex-A9 Processor, http://www.arm.com/products/processors/cortex-a/cortex-a9.php

[2] "The benefits of multiple CPU cores in mobile devices", http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf

[3] C. Li et al., "SolarCore: Solar energy driven multi-core architecture power management", in HPCA 2011, pp. 205-216

[4] X. Lin et al., "Online fault detection and tolerance for photovoltaic energy harvesting systems", in ICCAD 2012, pp. 1-6

[5] Y. Zhang, Y. Ge, and Q. Qiu, "Improving charging efficiency with workload scheduling in energy harvesting embedded systems", in DAC 2013,. article 57

[6] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Lazy scheduling for energy-harvesting sensor nodes", in DIPES, 2006, pp. 125-134

[7] S. Liu, J. Lu, Q. Wu, and Q. Qiu, "Harvesting-aware power management for real-time systems with renewable energy", IEEE Trans. VLSI Syst., vol. 20, no. 8, pp. 1473-1486, Aug. 2012

[8] Y. Xiang, S. Pasricha, "Harvesting-Aware Energy Management for Multicore Platforms with Hybrid Energy Storage", Proc. GLSVLSI 2013

[9] J. Luo and N.K. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems", in ICCAD 2000, pp. 357-364

[10] R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems", in IPDPS 2004, pp. 111

[11] A.K. Coskun et al., "Temperature-aware MPSoC scheduling for reducing hot spots and gradients", in ASPDAC 2008, pp. 49-54

[12] Y. Kwok and I. Ahmad, "Benchmarking the Task Graph Scheduling Algorithms", in IPPS 1998, pp. 531-537

[13] F. Ongaro, S. Saggini, and P. Mattavelli, "Li-Ion battery-supercapacitor hybrid storage system for a Long Lifetime, Photovoltaic-Based Wireless Sensor Network", IEEE Trans. Power Electron., vol. 27, issue 9, pp. 3944-3952

[14] V. Suhendra, C. Raghavan, and T. Mitra. "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures", in CASES 2006, pp. 401-410

[15] A. Makhorin, "GLPK—GNU Linear Programming Kit," http://www.gnu.org/software/glpk/

[16] Gurobi Optimization. (2009) Gurobi Optimizer Reference Manual, 2nd edn, http://www.gurobi.com/html/doc/refman/

[17] NREL Measurement and Instrumentation Data Center (MIDC), http://www.nrel.gov/midc/

[18] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free", in CODES/CASHE 1998, pp. 97-101

[19] Intel XScale, http://download.intel.com/design/intelxscale

[20] R. Wtanabe et al., "Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor SoC", in DATE 2007

[21] I. Veerachary, T. Senjyu, and K. Uezato, "Maximum power point tracking of coupled inductor interleaved boost converter supplied PV system", IEE Proc. EPA, 2004, vol. 150, no. 1, pp. 71-80

[22] H.F. Sheikh and I. Ahmad, "Dynamic task graph scheduling on multicore processors for performance, energy, and temperature optimization", in IGCC 2013, pp. 1-6

[23] I. Ahmad et al., "CASCH: a tool for computer-aided scheduling," IEEE Concurrency, vol.8, no.4, pp. 21-33, Oct-Dec 2000

[24] J. Lu, Q. Qiu, "Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting," Proc. IGCC, 2011.