Fault-Aware Application Scheduling in Low-Power Embedded 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 hybrid design-time and run-time framework for reliable resource allocation, i.e., mapping and scheduling of applications, in multi-core embedded systems with solar energy harvesting. Our framework is designed to cope with the complexity of an application model with data dependencies and run-time variations in solar radiance, execution time, and transient faults, with support for flexible schedule templates at design-time, and lightweight online adjustment mechanisms to monitor run-time dynamics and make adjustments to task execution strategy. Our experimental results indicate improvements in performance and adaptivity using our framework, with up to 29.5% miss rate reduction compared to prior work and 55% performance benefits from adaptive run-time workload management, under stringent energy constraints and varying system conditions at run-time.

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

General Terms
Algorithms, Design, Performance

Keywords
Task Scheduling, Energy Harvesting, Soft Errors

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][3]. 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 [4]-[6]. 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., [7]-[11]. However, all of these efforts are aimed at independent task execution 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).

Due to aggressive scaling in CMOS technology, emerging multi-core processors are also facing ever-increasing likelihoods of transient faults (or soft errors) caused by a variety of factors, e.g., high-energy cosmic neutron or alpha particle strikes, and capacitive and inductive crosstalk [12]. For low-power embedded systems that scale down voltage and frequency for energy saving, the rate of transient fault occurrence is more severe as lower supply voltage leads to drastically increased rate of transient faults [13]. Co-optimization of reliability and energy efficiency have thus become a critical design concern in recent work on task scheduling [14]-[19]. However none of these efforts focus on energy harvesting based systems.

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

—A hybrid application mapping and scheduling framework is proposed that integrates a comprehensive design-time analysis methodology with lightweight run-time components for low-overhead energy management in solar energy harvesting based multi-core embedded systems for the first time;
—We propose two different approaches to solve the DAG scheduling problems at design time, generating schedule templates composed of energy efficient execution schedules for various energy budgets that can be encountered at run-time;
—Our run-time scheduler utilizes a novel lightweight run-time heuristic that co-manages run-time *slack reclamation* and *soft error handling* in a multi-core environment without diminishing the benefits of schedule templates generated at design time.

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. [7] 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. [8] 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. [9] 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*.

ESWEEK'14, October 12 - 17 2014, New Delhi, India

ACM 978-1-4503-3051-0/14/10…$15.00.
http://dx.doi.org/10.1145/2656075.2656084

Several other efforts have explored mapping and scheduling for task graph based workloads. Luo et al. [20] 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. [21] proposed hybrid heuristics for DAG scheduling on heterogeneous processor platforms. Coskun et al. [22] proposed a hybrid scheduling framework that adjusts the task execution schedule dynamically to reduced thermal hotspots and gradients for MPSoCs. *However, all of 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 specifically targets the problem of energy-aware allocation and scheduling of multiple co-executing task graphs in energy harvesting based multi-core platforms.

A few efforts have addressed the problem of reliability and energy co-optimization. Zhu et al. proposed an approach to insert a recovery task during slack time for multiple tasks [14]. To address the conservative nature of individual-recovery based approaches, Zhao et al. [15] proposed a share recovery (SHR) technique that shares a small number of recovery nodes among all nodes executing tasks, to meet system wide reliability target. This SHR technique also has been applied to address reliability during scheduling of DAG-based workloads [16]. *None of these works are applicable to environments with energy harvesting.* In our work, unlike prior efforts on integrating reliability during scheduling, we do not aim to satisfy a target reliability. Instead, our focus is on alleviating the impact of soft errors to finish as many applications correctly as possible for a time varying and stringent energy budget in the presence of energy harvesting.

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 and presence of soft errors, as shown in Figure 1. The key components and assumptions of our system model are described in the following sub-sections.

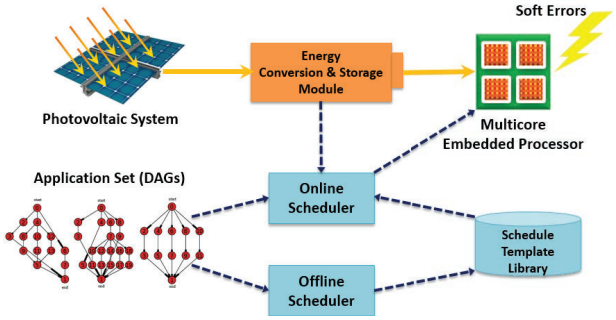

Figure 1. DAG workload based multi-core embedded system platform with solar energy harvesting and soft errors

3.1.1 Energy Harvesting and Energy Storage

A photovoltaic (PV) system is used as the power source for our multi-core 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 [32]. 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. [9] in our work.

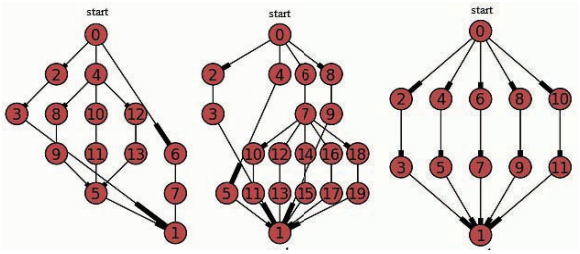

Figure 2. Examples of applications modeled as DAGs

3.1.2 Application Workload Model

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 accumulates results and concludes execution.

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.

The actual clock cycles consumed by a task node may vary at run-time due to variations such as changing data readings from sensors, varying memory system behavior and randomness in application procedures. We use probability distributions to model variations in execution time [33] and assume that clock cycles consumed by a task never exceed its WCEC.

3.1.3 Multi-core 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 (XY) 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 core power dissipation and frequency levels [30]

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 calculate energy efficiency of a frequency level L_j as: 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 40mW idle core power dissipation when no workload is available for execution.

To assess the computation intensity of an application relative to a processor's full capability, 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:

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

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 Soft Error Model

In this paper, we assume that task nodes can produce incorrect output due to transient faults occurring during execution and such incorrect outputs can be detected by verification logic executed at the end of regular task execution. To recover from a soft error, the task node with a faulty output must be re-executed, otherwise the output of the entire task graph will become invalid, which is counted as a task graph miss. We apply the exponential model proposed in [13] to simulate soft error rates, as shown in (3):

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}} \quad (2)$$

where λ_0 is the average error rate corresponding to the maximum frequency, d is a constant that indicates the sensitivity of error rate to voltage scaling, f_{min} is the normalized minimum core frequency, and f is the normalized core frequency. Thus we can observe that lower power execution at lower supply voltage (and thus frequency) to save energy can result in an exponential increase in soft error rate. [16]

3.1.5 Run-Time (Online) 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 run-time scenarios, such as varying energy budgets, variation in task execution time, and random transient faults, to schedule as many of the task graph instances as possible without overloading the system with complex re-scheduling calculations at run-time. Also our run-time workload management scheme should consider slack reclamation to aggressively save energy and support soft error handling to avoid finishing task graphs with incorrect output (which is counted as a task graph miss).

4. HYBRID SCHEDULING FRAMEWORK: MOTIVATION AND 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 [23]. 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 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.

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 partitioned into *schedule windows* of 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.

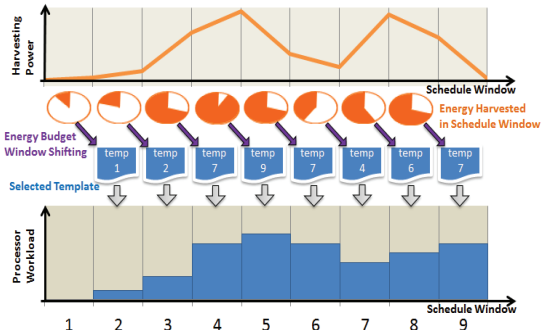

Figure 3. Overview of hybrid workload management framework

In the following sections, we describe our proposed framework in detail. Section 5 describes two design-time scheduling template generation approaches. Section 6 presents a run-time scheduler with a lightweight slack reclamation support and an integrated soft error handling heuristic. Experimental results to validate our framework are discussed in Section 7.

5. OFFLINE TEMPLATE GENERATION

In this section we propose and discuss two different approaches to solve the DAG scheduling problem at design time. Both approaches generate scheduling templates composed of energy efficient execution schedules for various energy budgets. The first approach is based on a mixed integer linear programming (MILP) method that ensures schedule optimality for maximum online performance. The second approach is an analysis-based template generation (ATG) heuristic that is faster and more scalable than MILP, to accommodate larger problem sizes with acceptable compromise in schedule optimality.

5.1 MILP-Based 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 cores, each with N_l discrete frequency levels.

5.1.1 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 available energy budget. The $WCET_{j,l}$ and $ENGY_{j,l}$ parameters 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

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'*

There are two major requirements for decision variables in our MILP problem,: firstly, they must form a complete representation of a feasible execution schedule; secondly, they should make it possible to represent all constraints and objectives 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 timings of the task nodes without direct dependencies.

5.1.2 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:

$$\text{Min:} \quad \sum_{i=1}^{Ni} miss_i + \Big(\sum_{j=1}^{Nt} \sum_{l=0}^{Nl} (ENGY_{j,l} \times freq_{j,l}) \Big) \Big/ EGY\_BGT \qquad (4)$$

5.1.3 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:

$$\sum_{j=1}^{Nt} \sum_{l=0}^{Nl} (ENGY_{j,l} \times freq_{j,l}) \leq EGY\_BGT \qquad (5)$$

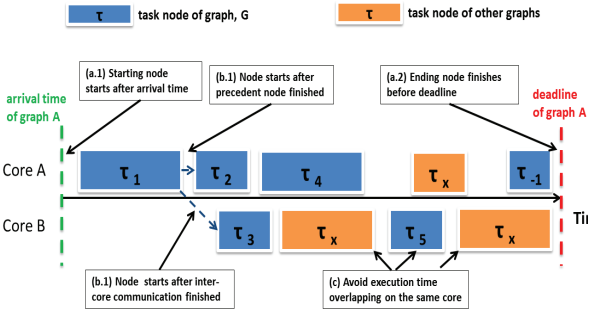

Figure 4. Timing constraints for periodic task graph set

2) Timing constraints for task graph scheduling: We formulate multiple constraints, which when combined together 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 corresponding task graph instances are satisfied, as illustrated in Figure 4 (a.1, a.2).

$$start_{(i,1)} \geq ARRIVAL_i - M \times miss_i \qquad i \in [1, N_i] \qquad (6)$$

$$end_{(i,-1)} = start_{(i,-1)} + \sum_{l=1}^{Nl} (WCET_{(i,-1),l} \times freq_{(i,-1),l}) \qquad (7)$$
$$end_{(i,-1)} \leq DDLINE_i + M \times miss_i \qquad i \in [1, N_i]$$

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):

$$\textbf{if } miss_i = 0: \quad end_{(i,src)} + COMM_{src,dst} \times dec_{src,dst} \le start_{(i,dst)} \quad (8)$$
$$\in [1, N_i], \ (src, \mathrm{dst}) \in edges\ of\ G_i\ , G_i \in \Psi^+$$

(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).

$$dec_{j,j'} \le 2 - allocs_{j,k} - allocs_{j',k} \quad (9)$$
$$j \in [1, Nt], j' \in [1, Nt], j \ne j', k \in [1, Nc]$$

$$dec_{j,j'} \ge allocs_{j,k} + allocs_{j',k'} - 1 \quad (10)$$
$$j \in [1, Nt], j' \in [1, Nt], j \ne j'$$
$$k \in [0, Nc], k \in [1, Nc], k \ne k'$$

These constraints represent relations between task node allocation variables, $alloc_{i,k}$, and node pair decoupling variables, $dec_{j,j'}$. The constraint in (9) ensures that the pair decoupling variable is equal to 0 when task nodes are on the same core. The constraint in (10) forces the decoupling variable to be 1 when two task nodes are found to be allocated to 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:

$$bef_{j,j'} + bef_{j',j} - dec_{j,j'} = 1 \quad (11)$$
$$\textbf{if } bef_{j,j'} = 1: \quad end_j < start_{j'} \quad (12)$$
$$\textbf{if } bef_{j',j} = 1: \quad end_{j'} < start_j$$
$$j \in [1, Nt], j' \in [1, Nt], j \ne j' \ \text{ for (11) and (12)}$$

The constraint in (11) 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 (12) 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 (12) cancel out [24].

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:

$$\sum_{l=0}^{Nl} freq_{j,l} = 1, \quad j \in [1, N_t] \quad (13)$$
$$\sum_{k=1}^{Nc} alloc_{j,k} \le 1, \quad j \in [1, N_t] \quad (14)$$
$$\textbf{if } freq_{j,0} = 0: \quad \sum_{k=1}^{Nc} alloc_{j,k} = 1, \quad j \in [1, N_t] \quad (15)$$

A task is indicated as dropped in the generated schedule when its frequency level is set to 0. The constraint in (15) 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.

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

5.2 Fast Heuristic-Based Offline Template Generation

The MILP optimization approach can provide optimal static schedule templates when online performance is the primary goal and the complexity of the workload is not excessive. For problems with larger sizes, however, the complexity of MILP optimization will increase dramatically such that the execution time of the MILP solver becomes impractical, even for design time exploration. Thus we propose another novel analysis-based template generation (ATG) heuristic that emphasizes scalability and fast solution generation with an acceptable compromise on the optimality of generated schedule templates.

The outline of our proposed analysis-based template generation (ATG) method is illustrated in Figure 5. The main idea in ATG is to iteratively analyze and improve performance of tentative execution schedules based on feedback from step-by-step simulation, which detects energy inefficient events to help make informed updates to the tentative schedule that is evaluated in another round of analysis. ATG also has an in-built checkpoint mechanism to save system status so that a new round of analysis after a rewind event (discussed later) saves time before a modification on a tentative schedule takes effect. The three main components of ATG are outlined below:

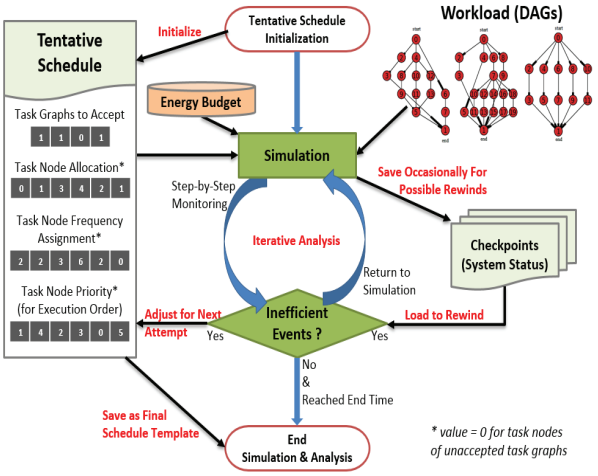

Figure 5. Analysis-based schedule template generation heuristic

1) Firstly, Algorithm 1 shows the steps to generate an initial tentative schedule for ATG based on a specified energy budget level. The algorithm starts out by finding the workload utilization that can be supported by a given energy budget level (step 1~3). Then the schedule accepts a subset of task graphs for execution and drops the remaining task graphs (step 4~11), while ensuring that task graphs with lower WCECs are more likely to be accepted and the total utilization of the task graphs satisfies the supportable workload utilization for the given energy budget. The generated initial schedule conservatively rules out some obviously sub-optimal portions of the solution space during scheduling and reserves enough headroom for upcoming iterative analysis and scheduling. The resulting initial schedule does not include core allocation and priority assignment of task nodes yet, which will be decided by the list scheduling algorithm used in a later iterative analysis stage.

Algorithm 1 Initializing of tentative schedule template

Inputs:
ψ, task graph set to be scheduled
EGY_BGT, specified energy budget for one schedule window
T_{win}, duration of a schedule window
num_cores, number of cores in system
f_{max}, maximum frequency of processors

Outputs:
$miss_i$, binary variables to indicate is task graph G_i is missed/dropped in schedule
$freq_j$, assigned frequency level of task node τ_j, value range $[0, N_t]$

1 $avg\_power \leftarrow (EGY\_BGT/T_{win})/num\_cores$
2 find f_{ref}, the highest frequency that can be supported by avg_power
3 $U_{ref} \leftarrow f_{ref}/f_{max}$
4 $U_{accepted} \leftarrow 0$
5 sort ψ according to WCEC of each task graph
6 **while** $U_{accepted} < U_{ref}$:
7 find the task graph with lowest WCEC, G_i
8 $miss_i \leftarrow FALSE$*
9 **for** τ_j **in** all task nodes of G_i:
10 $freq_j \leftarrow f_{ref}$
11 $U_{accepted} \leftarrow U_{accepted} + U_{Gi}$

* Default values of all elements in $miss_i$ for all task graphs is *TRUE*

2) Secondly, a list scheduling based algorithm is adapted and applied during iterative analysis, as shown in Algorithm 2. The algorithm is divided into two parts: Part I is concerned with

task priority assignment, while Part II deals with allocation and execution order scheduling of task nodes.

First, let us discuss the priority assignment in Part I. As each task graph in our application model only has one deadline specified at the end, every task node except the last one does not have an individual deadline. However, there exists a latest time to finish a task node so that it is possible to finish all remaining task nodes in the task graph before the global deadline, which is referred to as a task node's *implicit deadline*. We use implicit deadline to represent priority of a task node, as the earlier the implicit deadline is, the more urgent it is to finish the task node to avoid a deadline miss for the entire task graph. Algorithm 2 shows the heuristic in Part I that calculates implicit deadlines of all task nodes by using a nested function to traverse the entire task graph starting from the end task node, which has its implicit deadline equal to the deadline of the task graph it belongs to (step 2~3). Then in step 5~9, the nested function is called to back-traverse from the end node to predecessor nodes, calculating implicit deadlines of other task nodes in a depth-first manner. As a task node can have multiple successor nodes in a task graph, multiple values of implicit deadline can be derived from different calculation paths. To address this issue, steps 7 and 8 ensure that only the earliest value among all derived ones is kept as a task node's implicit deadline. An illustrative example of this priority (implicit deadline) assignment heuristic is shown in figure 6.

Algorithm 2 List scheduling based approach for task scheduling

Part I Task node priority (implicit deadline) assignment; called every time tentative schedule is changed

Inputs:

ψ, task graph set to be scheduled
DD_LINE_i, deadline of task graph instance i
$WCET_j$, worst cast execution time of each task node in task graph
$COMM_{src,dst}$, communication delay between node *src* and node *dst*

Output:

implicit_priority$_j$, implicit deadlines as priority indicators of task node τ_j

***priority_assign*():**

1 **for** G_i **in** ψ:
2 $\tau_j \leftarrow$ end task node of G_i
3 $dead\_priority_j \leftarrow DD\_LINE_i$
4 **call** *nested_priority*(τ_j)

***nested_priority*(τ_j):**

5 **for** τ_j' **in** all predecessor nodes of τ_j:
6 $implicit\_deadline \leftarrow implicit\_priority_j - WCET_j - COMM_{j',j}$
7 **if** $implicit\_priority_{j'} > implicit\_deadline$:
8 $implicit\_priority_{j'} \leftarrow implicit\_deadline$
9 **call** *nested_priority*(τ_j')

Part II List scheduling method; called in every simulation step

Inputs:

sys_pool, system task pool, containing task nodes that are ready to allocate
$core\_pool_k$, task pool for core k, containing allocated task nodes that are ready to execute
$CORE\_WCET_k$, remaining WCET of all task nodes assigned to core k
$implicit\_priority_j$, implicit deadlines as priority indicators of task node τ_j

Outputs:

$alloc_j$, allocation results of task node τ_j, value range [0, num_cores]
selected task node to execute in current simulation step

list_schedule():

10 sort sys_pool according to WCET of each node
11 **for** all task nodes **in** *sys_pool*
12 find τ_j in *sys_pool* with highest $WCET_j$
13 find core k, with lowest $CORE\_WCET_k$
14 allocation τ_j to core k, $alloc_j \leftarrow k$ *
15 $CORE\_WCET_k \leftarrow CORE\_WCET_k + WCET_{\tau j}$
16 **for** all cores **in** system:
17 sort $core\_pool_k$ according to implicit deadline of each tasks
18 select task node with earliest implicit deadline to execute

* Allocated task is not ready to execute until preceding dependencies are resolved

Part II of Algorithm 2 shows the steps for allocating and scheduling task nodes during each simulation step. For task node allocation, a task pool is used to collect task nodes that are ready to be allocated and each core has a record of WCET required to finish all task nodes already assigned to it. A good allocation scheme should distribute task nodes to cores so that their workloads are as evenly balanced as possible. In steps 10~15, we use a heuristic that is similar to a first-fit decreasing algorithm for the bin-packing problem [25], which sorts task nodes in decreasing order based on their WCETs and then iteratively allocates the task node with highest WCETs to cores with lowest WCETs accumulated for execution. The scheduling of task nodes on each core is performed based on the earliest implicit dead line first (EiDF) algorithm (steps 16~18), which is essentially EDF that uses implicit deadlines generated in part I. With multiple task graphs to be scheduled at the same time, EiDF gives priority to task nodes in the critical path of different task graphs, after comparing their implicit deadlines.

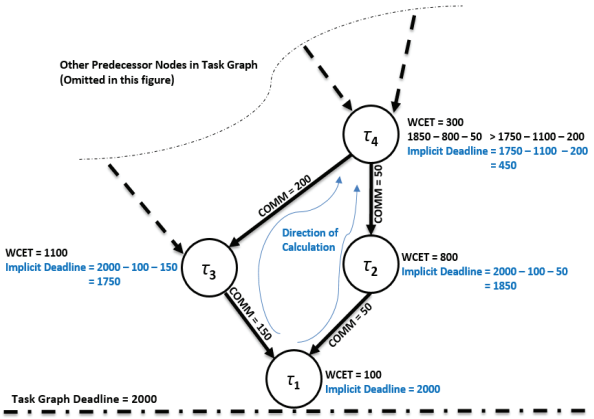

Figure 6. An example of priority (implicit deadline) assignment

3) Lastly, at the core of the ATG heuristic is a checkpoint-based iterative analysis method, as shown in Algorithm 3. At the beginning of each simulation step, the ATG heuristic saves the current system status as a checkpoint for newly arriving task graphs, so that the simulation can rewind to this checkpoint saved before the schedule for the new task graph takes effect (step 2~3). Subsequently, a list scheduler is invoked and the system executed for one simulation step with the tentative schedule (step 4~5). As illustrated in Figure 7, when energy inefficient events are detected during execution, the ATG heuristic will update the execution schedule accordingly and rewind to a previous checkpoint for another round of evaluation with an updated schedule (step 6~16). If ATG detects depletion of the energy budget before finishing all accepted task graphs in the current schedule (energy violation event), one accepted task graph with highest WCEC will be dropped in the updated schedule and simulation rewinds for re-analysis (step 6~9). If ATG detects a task node that missed its implicit deadline (timing violation event), which implies that a deadline miss for the task graph it belongs to is inevitable, the tentative schedule will be updated to boost execution frequency of related task nodes: the task node in the critical path with the lowest frequency assigned will get a frequency boost (step 11~13); and if there exists a task node from another task graph allocated to the same core that finished just before the nodes with timing violation, it will also get a frequency boost (step 14~15). Note that WCETs of selected task nodes change with their boosted frequencies, thus we call a nested priority assignment function starting from these nodes to recalculate implicit deadlines of their predecessors. Then simulation rewinds for re-analysis with the new schedule (step 16). If the current simulation step detects no energy inefficient events, the simulation continues to the next step (step 17~18). When the entire schedule window is analyzed without energy inefficient events, the analysis process ends and the

updated schedule is saved as a schedule template for the specified energy budget (step 19).

Algorithm 3 Checkpoint-based iterative analysis

Inputs:
EGY_BGT, specified energy budget for one schedule window
T_{win}, duration of a schedule window
$implicit\_priority_i$, implicit deadlines as priority indicators of task node τ_i
initial tentative schedule from Algorithm 1

Output:
Static schedule template for energy budget of *EGY_BGT*

```
1  while T_cur < T_win:
2      if new task graph G_i arrives:
3          checkpoint_i ← all system status (include T_cur)
4      alloc ←list_schedule()
5      execute for one step using tentative schedule
6      if EGY_BGT depleted during execution:
7          find arrived task graph with highest WCEC, G_i
8          miss_i ← TRUE
9          all system status ← checkpoint_i
10     else if node τ_j of task graph G_i missed its implicit deadline:
11         find the critical path in G_i that ends at τ_j
12         find τ_j', the task node with lowest frequency assigned
13         freq_j' ← freq_j' + 1,    nested_priority(τ_j')
14         find τ_j", the task finished just before τ_j on the same core
15         freq_j" ← freq_j" + 1,    nested_priority(τ_j")
16         all system status ← checkpoint_i
17     else:
18         T_cur = T_cur + T_step
19 save final tentative schedule as schedule template
```

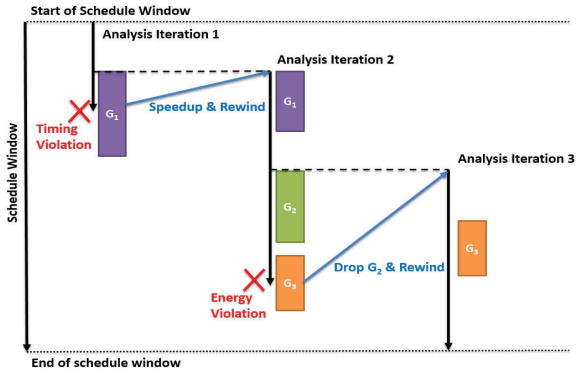

Figure 7. Illustration of checkpoint-based rewind and analysis

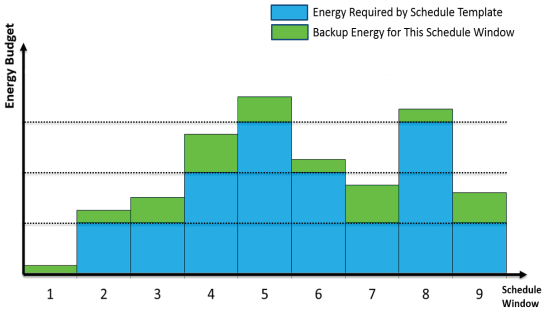

Figure 8. Residual energy availability over time

At design time, the ATG heuristic is executed multiple times with different energy budget levels (similar to the MILP approach) to generate a set of schedule templates for the run-time scheduler to select from, based on the harvested and available energy in the target multi-core computing platform.

6 ADAPTIVE ONLINE MANAGEMNT

6.1 Run-Time Template Selection

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 the 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. The amount of residual energy that exceeds the energy requirement of the selected schedule template is used as backup energy (Figure 8) for possible task re-execution to recover from detected faults caused by soft errors during execution.

6.2 Dynamic Adjustment for Slack Reclamation and Soft Error Handling at Run-Time

Utilizing static schedule templates for run-time workload management shifts the burden associated with the complex task graph scheduling problem to design time. However, embedded systems in the real-world encounter various unpredictable variations at run-time such as those due to fluctuations in harvested solar energy, slight variations in task execution time on the same core, and randomness of soft error occurrences. Among these factors, the fluctuations in harvested solar energy are already dealt with in our framework by using the energy budget window-shifting technique and the schedule template set prepared for different energy budget levels. In this section, we introduce a lightweight run-time management scheme that provides an integrated solution to address slack reclamation and soft error handling without diminishing the benefits of schedule templates generated at design time. This scheme is described in Algorithm 4.

Algorithm 4 Dynamic slack reclamation and soft error handling

Inputs:
T_{win}, duration of a schedule window
ψ, task graph set to be scheduled
$start_i$, designated time to start execution of τ_i in slected schedule template
$bkup\_energy_i$, amount backup energy for a schedule window

Output:
Static schedule template for energy budget of *EGY_BGT*

```
1  while T_cur < T_win:
2      load schedule in template
3      for τ_j in taskpool:
4          if τ_j is about to start execution and T_cur < start_j
5              slack_time ← start_j – T_cur
6              while slack_time > WCET increased at freq_j – 1:
7                  freq_j ← freq_j – 1
8                  bkup_energy ← bkup_energy + energy saved
9      execute task nodes based on schedule template
10     for τ_j in just finished tasks:
11         if error detected on τ_j:
12             if T_cur ≤ start_j:
13                 schedule another instance of τ_j to re-execute
14             else if ∃ a freq that has reduced WCET > T_cur - start_j
               and can be supported by bkup_energy:
15                 freq_j ← freq,
16                 bkup_energy ← bkup_energy – energy used
17                 schedule another instance of τ_j to re-execute
18             else:
19                 find next node to execute on the same core, τ_j'
20                 if τ_j ∈ G_i, τ_j' ∈ G_i', and G_i ≠ G_i':
21                     update remain WCEC of both graphs
22                     if G_j' has more WCEC:
23                         drop G_j'
24                         schedule τ_j to re-execute
25                     else:
26                         drop G_j
27                 else:
28                     drop G_j
```

Our run-time management scheme can reclaim slack time that becomes available when a task node finishes before its worst case finishing time. This slack time can be used to slow down execution of upcoming tasks, to save energy. The offline generated schedule templates have a designated start time recorded for all task nodes, to help identify any instances of

slack time. Whenever a new task node is about to start execution, the amount of slack time is calculated by subtracting the node's designated start time with the current time (steps 4~5). If the amount of slack time is usable, slower execution frequency is assigned to the task node for the purpose of saving energy (step 6~7). Even if the amount is not sufficient to step down a frequency level, the task node will start execution earlier than the designated time and thus the slack time can be passed on to upcoming tasks, as shown in Figure 9. The estimated amount of energy saved is added to the backup energy for use during possible task re-execution in the presence of soft errors (step 8).

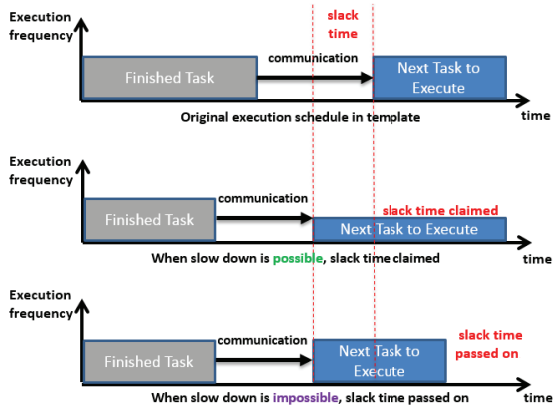

Figure 9. Illustrative example of slack time reclamation

Our run-time management scheme is also capable of reacting to soft errors with node-to-node soft error detection. Whenever a task node finishes execution, the correctness of the result is verified to trigger an error handling heuristic if errors are detected during task node execution (steps 10~11). If there is slack time directly available, the system reclaims it to execute a new instance of the faulty task node (step 12~13). If sufficient slack time is not available, the error handling heuristic checks to determine if there exists a higher frequency supportable by the available backup energy to finish re-execution of the fault-affected task node before its implicit deadline (steps 14~17). If both options are not viable for the faulty task node, the heuristic will attempt to drop other task graphs with higher WCEC so that the faulty node can be rescheduled. This process involves checking if the next node scheduled to execute on the same core is from another task graph (step 18~20). If true, both task nodes have the WCEC of their unfinished nodes updated and the task graph with the higher WCEC is dropped (step 21~26). The three error handling stages described above attempt to exploit slack time, backup energy and relatively less important task graphs to save the computation efforts invested into all predecessor nodes of the faulty task node, for better overall energy efficiency. During slack reclamation and error handling, all task nodes that do not belong to faulty or dropped task graphs will not have their template-designated finish time compromised, thus a chosen schedule template remains effective during run-time workload management.

7. EXPERIMENTAL RESULTS

7.1 Experiment Setup

We developed a simulator in C++ to evaluate our proposed fault-aware 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. We formulated the MILP problem using a GNU linear programming kit (GLPK) [26]. We chose the Gurobi Optimizer [27] as our MILP solver to generate the optimal schedule templates. We used task graphs for free (TGFF) [29] for pseudo-random task graph generation for most experiments and the distribution of actual execution times of task nodes is obtained from [33]. In addition, we also utilized task graphs of real applications (FFT, Gaussian Elimination, and MPEG) [35]. 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.

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

7.2 Template Generation Analysis

In this first 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).

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 that while 95.7% of E_{peak} is required to finish all task instances, template 3 with energy budget less than $1/3^{rd}$ of E_{peak} managed to successfully schedule more than half of the instances. *The results demonstrate how our approach can create efficient schedules even under highly constrained energy budget requirements.* The 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 the 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 10. We can observe from the figure that templates with higher energy budgets utilize higher frequency levels more frequently than templates with lower budgets. Templates with lower energy budget end up dropping 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.

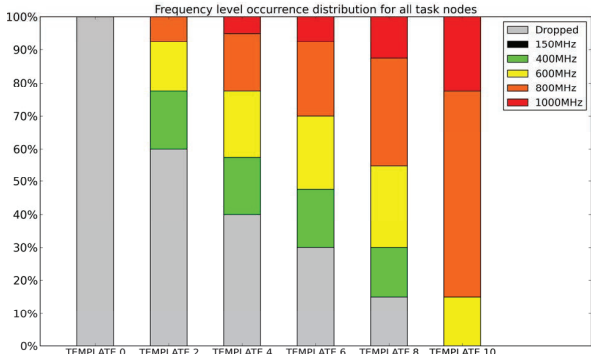

Figure 10. Frequency level occurrence distribution for all task nodes

Table 5. Computation resource requirement of MILP and ATG

Method	Complexity		Memory footprint	Execution time
	Number of nodes	Number of edges		
ATG	40	46	40 MB	0.1hr
MILP			255 MB	6.25hr
ATG	140	185	50 MB	1hr
MILP			5000 MB	> 100 hr

While the MILP approach generates optimized schedule templates, we found that the approach is not scalable for larger problem sizes. Table 5 shows a comparison between the MILP and ATG heuristics, in terms of execution time and memory footprint, for two problem instances. It can be observed that the MILP approach requires significant computation resources for large problem sizes, which may not be practical even at design time. The ATG heuristic is much faster, but this speedup comes at the cost of lower performance due to sub-optimal schedule templates generated (see next section).

7.3 Evaluation of System Performance without Error Injection and Execution Time Variance

In this section, we compare the overall system task graph miss rate for the two variants of our hybrid workload management framework: HyWM-LP and HyWM-ATG, 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) [28]. 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 [11], which first executes schedulable workload at full speed and then drop tasks when energy is insufficient; 2) SDA [9], 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 [31], 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.

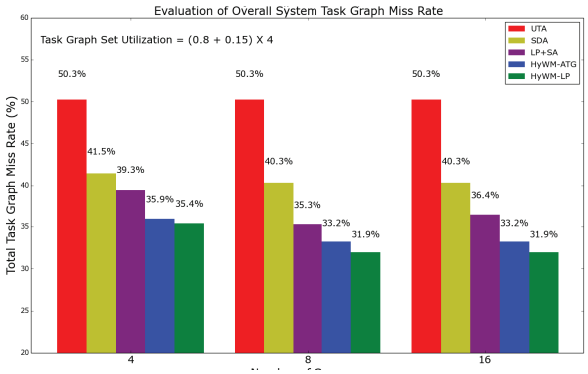

Figure 11. Comparison between HyWM framework and prior work ([9], [11], [31]) in terms of overall system task graph miss rate

The results of our comparison study on random task graphs are shown in Figure 11. 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 frequency slowdown 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-LP reduces absolute miss rate by 3.8%, 6%, and 14.8% over LP+SA, SDA, and UTA, respectively. In terms of relative performance improvement, HyWM-LP accomplishes an improvement of 9.7%, 15.2%, and 29.5% over LP+SA, SDA, and UTA, respectively. HyWM-ATG ends up with higher miss rates than HyWM-LP, however it still outperforms the other three techniques from prior work. The HyWM-ATG can however serve as an alternative approach when scalability is an issue, e.g., for larger problem sets.

Figure 11 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 two *HyWM* methods achieve lower miss rates compared to other techniques, as they can better distribute the 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 slightly 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.

Table 6. Miss rate on real application DAG set with 4 cores

UTA	SDA	LP+SA	HyWM-ATG	HyWM-LP
52.87%	48.49%	43.02%	41.77%	39.07%

We also conducted additional experiments on a real application DAG workload composed of FFT, Gaussian transformation, and MPEG encoder applications running simultaneously [34]. Table 6 shows results for these applications

running on a 4-core platform configuration. Once again, our proposed HyWM-ATG and HyWM-LP techniques can be seen to outperform other frameworks proposed in prior work to generate better quality results.

7.4 Evaluation of System Performance with Error Injection and Execution Time Variance

In this section, we show the performance improvements due to our run-time slack reclamation and error handling heuristic. In the experiment, we assume an average error rate of 1e-5 soft errors per second per core at maximum frequency. The results are shown in Table 7. The base case uses HyWM-ATG without enabling any run-time adjustment technique, and has a miss rate of 40.11%. This miss rate is greatly improved when the slack reclamation capability in run-time heuristic is activated (SR-only). With the addition of basic soft error-awareness that causes faulty task graphs to be dropped as soon as an error is detected (to avoid unnecessary energy consumption), the miss rate reduces further to 27.19%. When the heuristic adds support for dropping other task graphs with high WCET to allow re-execution of the faulty task node (SR+drop+compare), the system sees a drop in miss rate to 24.7%. Finally, when the fully-enabled heuristic is utilized that adds further support for utilizing backup energy to speed up faulty node re-execution (SR+drop+compare+back), we end up with the lowest miss rate of 22.01%. These results highlight the significance of slack reclamation and soft error handling in our framework.

Table 7. Miss rate with error injection and execution time variance

Base case	SR-only	SR+drop	SR+drop +compare	SR+drop+com pare+backup
40.11%	29.19%	27.19%	24.7%	22.01%

8. CONCLUSIONS

In this paper, we proposed a hybrid design-time and run-time framework for reliable resource allocation in multi-core embedded systems with solar energy harvesting. Our framework was shown to cope with the complexity of an application model with data dependencies and run-time variations in solar radiance, execution time, and transient faults. Our experimental results indicated improvements in performance and adaptivity using our framework, with up to 29.5% miss rate reduction compared to prior work and 55% performance benefits from adaptive run-time workload management, under stringent energy constraints and varying system conditions at run-time. With the increasing prevalence of energy-constrained computing, energy scavenging, execution time variability, and the rise in soft errors with technology scaling, our proposed framework provides a comprehensive and practical solution that considers all of these factors to perform efficient resource management that improves upon prior efforts in both scope and performance, for emerging multi-core embedded computing platforms.

ACKNOWLEDGEMENTS

This research is sponsored in part by grants from NSF (CCF-1252500, CCF-1302693) and SRC.

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] N. Kapadia, S. Pasricha, "VERVE: A framework for variation-aware energy efficient synthesis of NoC-based MPSoCs with voltage islands", in ISQED 2013, pp. 603-610
[4] C. Li et al., "SolarCore: Solar energy driven multi-core architecture power management", in HPCA 2011, pp. 205-216
[5] X. Lin et al., "Online fault detection and tolerance for photovoltaic energy harvesting systems", in ICCAD 2012, pp. 1-6
[6] Y. Zhang, et al., "Improving charging efficiency with workload scheduling in energy harvesting embedded systems", in DAC 2013
[7] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Lazy scheduling for energy-harvesting sensor nodes", in DIPES, 2006, pp. 125-134
[8] 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
[9] Y. Xiang, S. Pasricha, "Harvesting-aware energy management for multicore platforms with hybrid energy storage", GLSVLSI 2013
[10] Y. Xiang, S. Pasricha, "Thermal-aware semi-dynamic power management for multicore systems with energy harvesting", in ISQED 2013, pp. 619-626
[11] J. Lu, Q. Qiu, "Scheduling and mapping of periodic tasks on multi-core embedded systems with energy harvesting", in IGCC 2011.
[12] S. S. Mukherjee, J. Emer, S K. Reinhardt, "The soft error problem: An architectural perspective", in HPCA 2005, pp. 243-247.
[13] D. Zhu, R. Melhem and D. Mossé, "The effects of energy management on reliability in real-time embedded systems", in ICCAD 2004, pp. 35-40
[14] D. Zhu, and H. Aydin, "Energy management for real-time embedded systems with reliability requirements", in ICCAD 2006, pp. 528-534
[15] B. Zhao, H. Aydin and D. Zhu, "Generalized reliability-oriented energy management for real-time embedded applications", in DAC 2-11, pp. 381-386
[16] B. Zhao, H. Aydin and D. Zhu, "Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints", ACM Trans. Des. Autom. Electron. Syst. Vol. 18, no. 2, article 23, Mar. 2013
[17] Y. Zou, S. Pasricha, "Reliability-aware and energy-efficient synthesis of NoC based MPSoCs", in ISQED 2013, pp. 643-650
[18] Y. Zou, Y. Xiang and S. Pasricha, "Analysis of on-chip interconnection network interface reliability in multicore systems", in ICCD 2011, pp. 427-428.
[19] Y. Zou, Y. Xiang and S. Pasricha, "Characterizing vulnerability of network interfaces in embedded chip multiprocessors", IEEE Embe. Syst. Lett. Vol. 4, no. 2, pp. 41-44, June. 2012
[20] 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
[21] R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems", in IPDPS 2004, pp. 111
[22] A.K. Coskun et al., "Temperature-aware MPSoC scheduling for reducing hot spots and gradients", in ASPDAC 2008, pp. 49-54
[23] Y. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms", in IPPS 1998, pp. 531-537
[24] V. Suhendra, C. Raghavan, and T. Mitra. "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures", in CASES 2006, pp. 401-410
[25] B. Xia, and Z. Tan, "Tighter bounds of the First Fit algorithm for the bin-packing problem", Discrete Applied Mathematics vol.158, no.15, pp. 1668–1675, Aug, 2010
[26] A. Makhorin, "GLPK—GNU linear programming kit", http://www.gnu.org/software/glpk/
[27] Gurobi Optimization. (2009) Gurobi optimizer reference manual, 2nd edn, http://www.gurobi.com/html/doc/refman/
[28] NREL Measurement and Instrumentation Data Center (MIDC), http://www.nrel.gov/midc/
[29] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: task graphs for free", in CODES/CASHE 1998, pp. 97-101
[30] Intel XScale, http://download.intel.com/design/intelxscale
[31] R. Wtanabe et al., "Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor SoC", in DATE 2007
[32] 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
[33] 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
[34] I. Ahmad et al., "CASCH: a tool for computer-aided scheduling", IEEE Concurrency, vol.8, no.4, pp. 21-33, Oct-Dec 2000