Soft and Hard Reliability-Aware Scheduling for Multicore Embedded Systems with Energy Harvesting

Yi Xiang, *Student Member, IEEE* and Sudeep Pasricha, *Senior Member, IEEE*

Abstract—For multicore embedded systems powered by energy harvesting, it is necessary to develop intelligent resource allocation techniques that adjust the application execution strategy on-the-fly to adapt to changing energy supply from the harvesting system. To cope with the complexity of managing applications with data dependencies on such systems, we propose a hybrid design-time/run-time framework for resource allocation that takes into consideration variations in solar radiance and execution time, transient faults, and permanent faults due to aging effects. Our framework generates schedule templates at design-time with an emphasis on energy efficiency and uses lightweight online management schemes to react to run-time system dynamics. Experimental results indicate that our framework presents improvements in performance and adaptivity, with up to 23.2 percent miss rate reduction compared to prior work, 43.6 percent performance benefits from adaptive run-time workload management, and up to 24.5 percent expected system lifetime improvement with aging-aware allocation of workload partitions.

Index Terms—Energy-aware systems, performance and reliability, real-time and embedded systems

1 Introduction

Recent years have witnessed a significant increase in the use of multicore processors in low-power embedded devices [1]. With advances in parallel programming and power management techniques, embedded devices with multicore processors have improved performance and energy efficiency over single-core platforms [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 multicore 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], [5], [6]. Due to the variable nature of solar radiation intensity, the most suitable role of embedded systems with limited-scale solar energy harvesting as the only energy source is to host non-critical applications that allow for imperfect operation. Thus it may not be desirable to consider such systems for real-time applications with hard deadlines, such as life-support mechanisms and powertrain controllers, for which any deadline miss is a critical system failure that may have catastrophic consequences. Instead, it is more practical to deploy such systems without energy guarantees for best-effort execution of soft or firm real-time applications where a deadline miss is not considered a failure of the entire system but a degradation of performance.

Consider an example of such a best-effort embedded system powered by energy harvesting that is deployed for continuous structural integrity sensing at a remote location on a bridge. For each operation interval, a usable raw data point can be collected from sensor modules by executing certain real-time control tasks such as data accessing, data post-processing and data-transmission. In the event of an energy shortage, the system stays operational with certain data collection intervals ignored such that overall sensing quality is sacrificed in favor of ensuring system continuity.

To achieve best-effort operation with limited resources, the deployment of an intelligent run-time resource management strategy is not only beneficial but also essential. Such a strategy must possess low overhead, so as to not stress the limited energy resources at run-time. A few prior efforts have explored workload scheduling for such real-time embedded systems with energy harvesting, e.g., [7], [8], [9], [10], [11], [12]. 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 workloads represented by direct acyclic graphs (DAGs).

Due to aggressive scaling in CMOS technology, emerging multicore processors are also facing ever-increasing likelihoods of transient faults (i.e., soft errors) and permanent faults (i.e., hard errors). Co-optimization of reliability and energy-efficiency have thus become a critical design concern in recent work on task scheduling [15], [16], [17], [18], [19], [20], [21], [22], [23]. However none of these efforts focus on energy harvesting based systems. For low-power embedded systems that scale down voltage and frequency for energy savings, the rate of transient fault occurrences, caused by a

• *The authors are with the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO, 80523-1373. E-mail: {yix, sudeep}@colostate.edu.*

Manuscript received 10 May 2015; revised 11 Aug. 2015; accepted 20 Sept. 2015. Date of publication 13 Oct. 2015; date of current version 27 Jan. 2016. Recommended for acceptance by N. Jha.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMSCS.2015.2487983

variety of factors, e.g., high-energy cosmic neutron or alpha particle strikes, and capacitive and inductive crosstalk [13], is more severe as lower supply voltage leads to drastically increased susceptibility to transient faults [14]. Additionally, embedded systems with energy harvesting must also consider the impact of hard errors because a major incentive of deploying such systems is long-term system autonomy, which requires an extended system lifetime. For these reasons, we believe it is necessary to study workload management schemes that consider both transient errors and aging effects to enhance system reliability and lifetime for low-power systems with energy harvesting.

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

- A hybrid application mapping and scheduling framework is proposed that integrates a rigorous design-time analysis methodology with lightweight run-time components for low-overhead energy management in solar energy harvesting based multicore embedded systems for the first time;
- We propose two different approaches to solve the DAG scheduling problem at design time, generating schedule templates composed of energy-efficient application execution schedules for various energy budgets possible at run-time;
- Our allocation scheme for workload partitions considers different wear-out profiles of cores and adjusts workload distribution to maximize lifetime of the entire system;
- Our run-time scheduler utilizes a novel lightweight run-time heuristic that co-manages run-time *slack reclamation* and *soft/hard error handling* in a multicore computing 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. We provide a brief overview of a few key prior works in this section.

Moser et al. [7] proposed the lazy scheduling algorithm (LSA) for energy harvesting platforms. Their approach executes 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 to save energy and thus is not suitable for emerging power-constrained environments. Liu et al. [8] exploited the scaling capability of processors to slow 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. The UTB approach was proposed in [9] to better address periodic task scheduling in energy-harvesting embedded systems. UTB takes advantage of the predictability provided by the periodic task information for more efficient task allocation than in prior work. Moreover, UTB was extended to support multicore platforms by allocating a subset of tasks to each core and executing the single-core UTB algorithm separately on each core. Xiang and Pasricha [12] proposed using a battery-supercapacitor hybrid energy harvester that helps reduce energy supply variability, allowing tasks to be scheduled and executed 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 and Jha [24] 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 and Zhao [25] proposed hybrid heuristics for DAG scheduling on heterogeneous processor platforms. Coskun et al. [26] proposed a hybrid scheduling framework that adjusts the task execution schedule dynamically to reduce 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 scheduling of multiple co-executing task graphs in energy harvesting based multicore platforms.

A few efforts have addressed the problem of reliability and energy co-optimization during scheduling. For soft-error reliability, Zhu and Aydin [15] proposed an approach to insert a recovery task during slack time obtained from executing multiple tasks. To address the conservative nature of individual-recovery based approaches, Zhao et al. [16] proposed a shared recovery (SHR) technique that shares a small number of recovery nodes among all nodes executing tasks, to meet a system wide reliability target. This SHR technique also has been applied to address reliability during scheduling of DAG-based workloads [17]. For hard failures, prior work has studied aging effects that lead to permanent system failure, such as electromigration (EM), negative bias temperature instability (NBTI), and time dependent dielectric breakdown (TDDB). Coskun et al. [21] proposed a framework to evaluate architecture-level effects of task scheduling and power management on lifetime of multi-processors. An analytical model to estimate lifetime reliability of multi-processors with a periodic workload was proposed in [22]. Basoglu et al. [23] quantitatively evaluated the long- term impact of NBTI-aware task-to-core mapping for multi- processors. *None of these works target systems with unstable supply from 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 and hard errors to finish as many applications correctly as possible and extending expected lifetime for a system with a time varying and stringent energy budget from 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

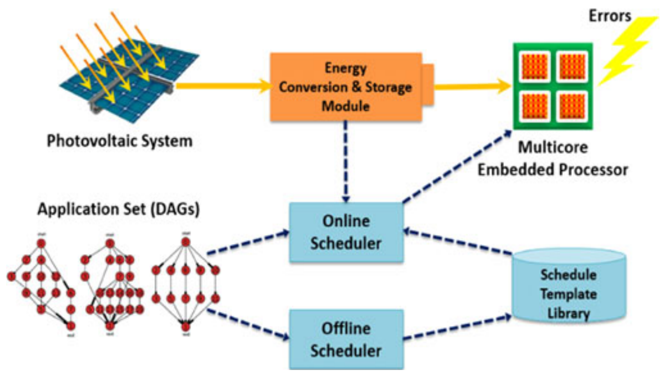

Errors
Photovoltaic System
Energy Conversion & Storage Module
Multicore Embedded Processor
Application Set (DAGs)
Online Scheduler
Offline Scheduler
Schedule Template Library

Fig. 1. DAG workload based multicore embedded system platform with solar energy harvesting and soft/hard errors.

on multicore embedded systems with solar energy harvesting, in the presence of soft and hard errors, as shown in Fig. 1. The key components and assumptions of our system model are described in the following sub-sections.

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 (e.g., changing cloud cover, time of day). To cope with the unstable nature of solar energy, an energy conversion and storage module is required to bridge the photovoltaic system with the embedded processor efficiently [27]. We assume that our run-time scheduler can cooperate with this module to determine the energy available in storage.

We adapt the hybrid supercapacitor-battery storage design proposed in [10] that combines the advantages of supercapacitors and batteries to support high-capacity energy storage (battery) and high energy-efficiency (supercapacitor). The energy storage system consists of one Li-Ion battery and two separate supercapacitors connected by a dc bus. During each schedule window, one capacitor is used to collect energy extracted from the PV array, while the other one is used as a power source for system operation or battery charging. At each reschedule point, the two supercapacitors switch their roles. Supercapacitors charge the battery only when their saved energy exceeds peak requirements of processors running at full speed. The PV array, battery, and supercapacitors are coupled with bidirectional dc-dc converters to serve the purpose of voltage conversions between components with maximum power point tracking (MPPT) and voltage level compatibility.

3.1.2 Application Workload Model

We consider multicore systems hosting multiple recursive real-time applications modeled as periodic task graphs, $\psi : \{G_1, \ldots, G_{Ng}\}$, such as the examples shown in Fig. 2. Each of the N_g applications is represented by a weighted directed acyclic graph (DAG), denoted as G_i:$(t_i, e_i, T_i, D_{i,j}), i \in \{1, \ldots, N_g\}$, 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

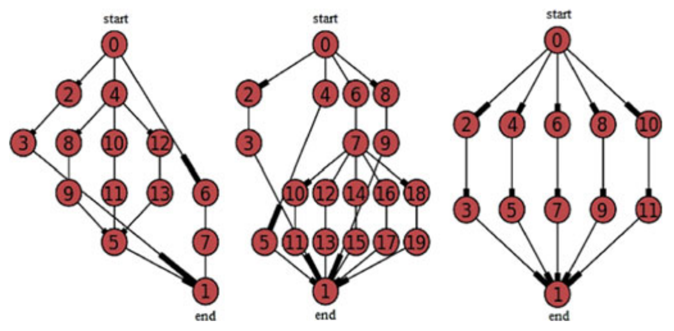


Fig. 2. Examples of applications modeled as DAGs.

(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.

Every periodic task graph has a unique period, T_i and nodes in the graph are assigned a relative deadline, $D_{i,j}$. At the beginning of each period, a new instance of a task graph will be dispatched to the system for execution. A task node's relative deadline, $D_{i,j}$, is the time interval between the task graph instance's arrival time and node deadline. A task graph instance misses its deadline if it cannot finish executing any nodes before their deadlines. In this work, we assume that the deadline of each task graph's last node $D_{i,-1}$ 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 time (clock cycles) required to execute a task node may vary at run-time due to variations in memory system behavior and randomness in application procedures. We therefore use probability distributions to model variations in task node execution time [36] and assume that clock cycles consumed by a task node never exceed its WCEC.

3.1.3 Multicore Platform with DVFS Capability

We consider a homogeneous multicore 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. Each core on the processor has N_l discrete frequency levels: φ: $\{L_0, \ldots, L_{Nl}\}$. Each level is characterized by a tuple L_j:$(v_j, p_j, f_j), j \in \{1, \ldots, N_l\}$, which includes voltage, average power, and frequency, respectively.

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

TABLE 1
Processor Core Power Dissipation and Frequency Levels [29]

Level	1	2	3	4	5
Power(mW)	80	170	400	900	1,600
Frequency(MHz)	150	400	600	800	1,000
Energy Efficiency	1.875	2.353	1.5	0.889	0.625

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: $\delta_j = cycles\ executed/energy\ consumed = f_j/p_j$. 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 40 mW 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:

$$U_{comm\ i} = \frac{\sum_k \mathrm{COMM}_i^k}{T_i}, i \in \{1, \ldots, N_g\}. \quad (2)$$

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 DAG application set.

3.1.4 Soft Error Model

In this paper, we assume that task nodes can produce incorrect output due to soft errors 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 [14] to simulate soft error rates, as shown in (3):

$$\lambda(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}}, \quad (3)$$

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. It can be observed that lower power execution at lower supply voltage (and thus frequency) to save energy can result in an exponential increase in soft error rate [17].

3.1.5 Aging Model

In addition to soft errors, we also consider aging effects that eventually lead to hard errors (permanent failure) in electronic systems. We adapt an analytical method to capture system-level lifetime reliability in embedded systems with multiple cores. In the rest of this section we first introduce how aging effects are modeled in our work and then describe a method to calculate reliability of a multicore embedded system according to a specified level of failure tolerance.

Many prior works model hard reliability characteristics of systems using mean-time-to-failure (MTTF) prediction [21]. However, for aging effects, it is more intuitive to model the changing of reliability over time due to progressive wear-out [28]. In our work, we estimate instantaneous hard reliability of a core, which reflects the possibility of core's avoidance of permanent failure within a time epoch. We utilize a Weibull distribution, which is one of the most widely used and versatile lifetime distributions in reliability engineering, to characterize per-core wear-out over time [40]. The instantaneous hard reliability of a single core at time t, R(t), can be expressed as:

$$R(t) = e^{-\left(\frac{t}{\alpha}\right)^{\beta}}, \quad (4)$$

where α and β represent the scale parameter and slope parameter in the Weibull distribution, respectively. While β is a constant that reflects architectural characteristics of core, α is highly dependent on the operating history of the core. Thus in our reliability model we need to deduce the relationship between the scale parameter α and operating history of the processing core. Firstly, by the definition of a Weibull distribution, MTTF of a core can be calculated as

$$MTTF = \alpha \times \Gamma\left(1 + \frac{1}{\beta}\right). \quad (5)$$

Then we can represent the scale parameter α as:

$$\alpha = \frac{MTTF}{\Gamma\left(1 + \frac{1}{\beta}\right)}. \quad (6)$$

This representation makes it possible to calculate the scale parameter for a core's instantaneous hard reliability model by adapting various MTTF-based hard error models, such as electromigration, time dependent dielectric breakdown, and negative bias temperature instability [21], [22], [23]. In this work we focus on EM-based aging, the MTTF model for which can be expressed as:

$$MTTF = A_0 (J - J_{crit})^{-n} e^{\frac{E_a}{kT}}, \quad (7)$$

where A_0 is a material-related constant, $J = V_{dd} \times f \times \mathrm{p_i}$ [21], and J_{crit} is the critical current density. Then we have

$$\alpha = \frac{A_0 (V_{dd} \times f \times p_i - J_{crit})^{-n} e^{\frac{E_a}{kT}}}{\Gamma\left(1 + \frac{1}{\beta}\right)}, \quad (8)$$

where $\mathrm{V_{dd}}$, f, and T can be controlled by our workload management framework.

To approximate aging effects over time, we use a fixed time epoch of length Δt as the basic time unit, for which averaged core frequency, supply voltage and temperature are applied to the above model for hard reliability calculation. According to [22], the reliability of a core at time epoch t_w, as the result of accumulated wear-out effects in previous time epochs from $\mathrm{t_0}$ to $\mathrm{t_{w-1}}$, can be approximately calculated as:

$$R(t_w) = e^{-\left(\sum_{i=0}^{w-1} \frac{\Delta t}{\alpha(t_i)}\right)^{\beta}}. \tag{9}$$

Also, MTTF of a core can then be represented as

$$MTTF = \sum_{i=0}^{\infty} \Delta t \times R(t_i). \tag{10}$$

For multicore systems, it is essential to consider not just reliability of each core individually, but rather the impact of aging on the entire system. We define a system-level failure threshold (h) as the maximum number of core failures allowed before the entire system is considered to have failed. For example, if $h = 0$, the system fails as soon as one core fails, i.e., all cores must maintain their functionality to keep system up. The hard reliability of a system for this case is:

$$R_{sys}(t_w)_{h=0} = \prod_{k=1}^{N} R_k(t_w), \tag{11}$$

where N is number of cores in a system. For general cases, where failure threshold h has a non-zero value, the hard reliability of system can be calculated as shown below:

$$\begin{gathered} R_{sys}(t_w)_h = R_{sys}(t_w)_{h-1} + \sum_{\substack{F\subset\{1,\ldots,n\}\\ |F|=h}} \\ \left(\prod\nolimits_{k\in F}(1 - R_k(t_w)) \times \prod\nolimits_{k\in\{1,\ldots,n\}\setminus F} R_k(t_w)\right) \\ h \in [1, N-1]. \end{gathered} \tag{12}$$

In the above equation, hard reliability of the system is calculated recursively, such that reliability of the system with failure threshold h equals reliability of the system with threshold of h–1 plus the probability of the system to have exactly h cores failed. Different cores usually have different hard reliabilities due to uneven workload distribution among them, therefore when calculating probability of a certain number of cores failed, it is essential to enumerate all cases in combination and sum up their probabilities.

3.1.6 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 multicore processor (Fig. 1). The gathered information, together with preloaded schedule template library generated by the offline scheduler for the given workload (discussed further in Section 5), allows the run-time scheduler to coordinate operation of the multicore platform at run-time.

3.2 Problem Objective

The primary objective of our workload management framework is to allocate and schedule the execution of a workload composed of multiple application task graphs (DAGs) arriving periodically and running in parallel simultaneously at run-time, such that *total task graph miss rate is minimized*. Our framework must react to changing run-time scenarios, such as varying harvested energy budgets, variations 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. The framework must also 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). As a secondary objective, the framework must take aging effects into consideration to *maximize overall system lifetime*.

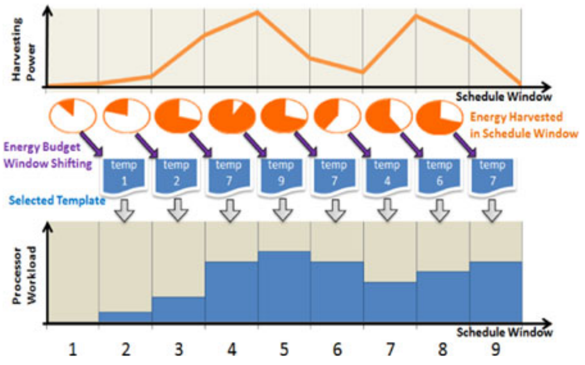


Fig. 3. Overview of hybrid workload management framework.

4 Hybrid Scheduling Framework: Motivation and Overview

The problem of scheduling weighted directed acyclic graphs on a set of homogeneous cores under some optimization goals and constraints is known to be NP-complete [30]. This paper addresses the even more difficult problem of scheduling on systems that rely entirely on limited and fluctuating harvested energy. *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 solar energy harvesting.

An important underlying idea in our framework, as shown in Fig. 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 DAG workload. 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 the execution of 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 energy budget.

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*
$DDLINE_{i,j}$	deadline of task graph instance *i node j*
$WCET_{j,l}$	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'*

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 aging-aware allocation of workload partitions, lightweight slack reclamation, and integrated soft error handling heuristics. Experimental results to validate our framework are presented 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 schedule templates composed of energy-efficient execution schedules for various energy budgets. The first approach is based on mixed integer linear programming (MILP) that ensures schedule optimality for maximum 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 to aid with the generation of optimal task scheduling templates at design time. The MILP formulation 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, in this section we assume periodic task graphs in set ψ are unrolled into a set of all task graph instances that arrive within a schedule window, $\psi^+ : \{GI_1, \ldots, GI_{Ni}\}$. Our target processor has 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 processing cores (as per the discussion in Section 3.1.3).

There are two major requirements for decision variables in our MILP problem: *(i)* they must form a complete representation of a feasible execution schedule; and *(ii)* they should make it possible to represent all constraints and objectives as linear formulations. Table 3 shows decision variables used in our formulation. 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 task node *j* is not scheduled for execution and is thus to be dropped. The indicators $dec_{j,j'}$ and $bef_{j,j'}$ are used to construct constraints that arrange timings of task nodes without direct dependencies.

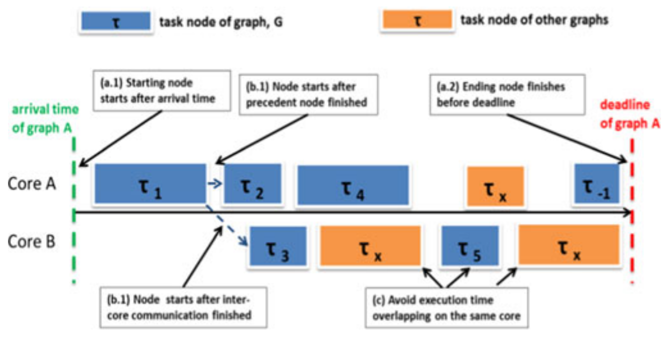
5.1.2 Optimization Objective

The major objective of the MILP formulation 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 number of task graph misses for less energy consumption, as the energy usage percentage, with value no greater than 1, always has less impact on the objective function value than any single task graph instance miss. The objective formulation is shown below:

$$\text{Min}: \sum_{i=1}^{Ni} miss_i + \sum_{j=1}^{Nt} \sum_{l=0}^{Nl} (ENGY_{j,l} \times freq_{j,l}) /_{EGY\_BGT}. \quad (13)$$

5.1.3 Constraints

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



task node of graph, G
task node of other graphs
(a.1) Starting node starts after arrival time
(b.1) Node starts after precedent node finished
(a.2) Ending node finishes before deadline
arrival time of graph A
deadline of graph A
Core A
Core B
τ1
τ2
τ4
τx
τ-1
τ3
τx
τ5
τx
(b.1) Node starts after inter-core communication finished
(c) Avoid execution time overlapping on the same core

Fig. 4. Timing constraints for periodic task graph set.

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 the energy budget:

$$\sum_{j=1}^{Nt}\sum_{l=0}^{Nl}(ENGY_{j,l} \times freq_{j,l}) \leqslant EGY\_BGT. \quad (14)$$

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 Fig. 4.

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

$$start_{(i,1)} \geq ARRIVAL_i - M \times miss_i \quad i \in [1, N_i] \quad (15)$$

$$end_{(i,j)} = start_{(i,j)} + \sum_{l=1}^{Nl}(WCET_{(i,j),l} \times freq_{(i,j),l}) \quad (16)$$

$$\begin{gathered} end_{(i,j)} \leq DDLINE_{i,j} + M \times miss_i \\ i \in [1, N_i], \quad j \in [1, Nt]. \end{gathered} \quad (17)$$

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 the constraints take communication cost into consideration when two dependent nodes are decoupled (not allocated to the same core), as illustrated in Fig. 4 (b.1, b.2):

$$\begin{gathered} \textbf{\textit{if}}\ miss_i = 0: \\ end_{(i,src)} + COMM_{src,dst} \times dec_{src,dst} \leq start_{(i,dst)} \\ i \in [1, N_i], \quad (src, \mathrm{dst}) \in edges\ of\ G_i, G_i \in \psi^+. \end{gathered} \quad (18)$$

(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 Fig. 4c:

$$dec_{j,j'} \leq 2 - allocs_{j,k} - allocs_{j',k} \quad (19)$$

$$\begin{gathered} j \in [1, Nt], j' \in [1, Nt], j \neq j', k \in [1, Nc] \\ dec_{j,j'} \geq allocs_{j,k} + allocs_{j',k'} - 1 \end{gathered} \quad (20)$$

$$\begin{gathered} j \in [1, Nt], \ j' \in [1, Nt], j \neq j' \\ k \in [0, Nc], \ k \in [1, Nc], k \neq k'. \end{gathered}$$

These constraints represent relations between task node allocation variables, $alloc_{i,k}$, and node pair decoupling variables, $dec_{j,j'}$. The constraint in (19) ensures that the pair decoupling variable is equal to 0 when task nodes are on the same core. The constraint in (20) 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:

$$bef_{j,j'} + bef_{j',j} - dec_{j,j'} = 1 \quad (21)$$

$$\textbf{\textit{if}}\ bef_{j,j'} = 0: \quad end_{j'} < start_j \quad (22)$$

$$\textbf{\textit{if}}\ bef_{j',j} = 0: \quad end_j < start_{j'} \quad (23)$$

$$j \in [1, Nt], j' \in [1, Nt], j \neq j' \qquad \text{for } (21-23).$$

The constraint in (21) 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 (22 and 23) 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 (22 and 23) 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:

$$\sum_{l=0}^{Nl} freq_{j,l} = 1, \quad j \in [1, N_t] \quad (24)$$

$$\sum_{k=1}^{Nc} alloc_{j,k} \leqslant 1, \quad j \in [1, N_t] \quad (25)$$

$$\begin{gathered} \textbf{\textit{if}}\ freq_{j,0} = 0: \quad \sum_{k=1}^{Nc} alloc_{j,k} = 1, \\ j \in [1, N_t]. \end{gathered} \quad (26)$$

A task is indicated as dropped in the generated schedule when its frequency level is set to 0. The constraint in (26) 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 heuristic that emphasizes scalability and fast solution generation with an acceptable compromise on the optimality of generated schedule templates.

Algorithm 1. Initializing of Tentative Schedule Template

```
Inputs:
  $\psi$, 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
  $U_{Gi}$, utilization of periodic task graph $G_i$
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 $\tau_j$, value range [0, $N_l$]
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 $\psi$ 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 $\tau_j$ 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.*

The outline of our proposed analysis-based template generation method is illustrated in Fig. 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:

1) First, 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

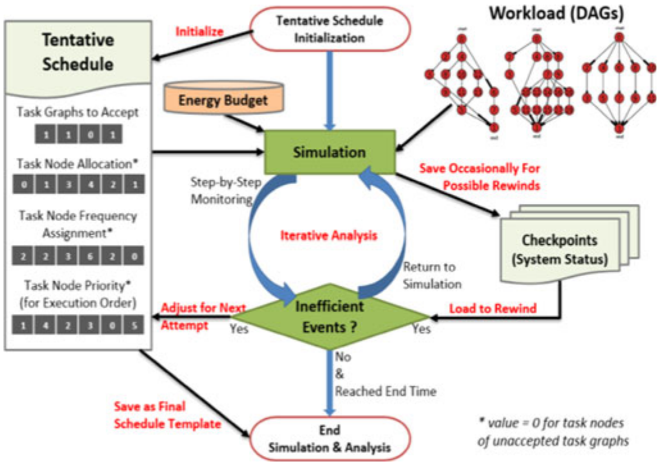


Fig. 5. Analysis-based schedule template generation heuristic.

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 stage.

2) Second, a list scheduling based algorithm is adapted to our problem 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, we discuss the priority assignment in Part I. In our application model, not all task nodes in a task graph will have deadlines assigned to represent timing requirements of the corresponding real-time application (see Section 3.1.2). For task nodes with deadlines assigned, we refer to their associated deadlines as *explicit deadlines*. On the other hand, for tasks nodes without explicitly assigned deadlines, there still exists a latest-time-to-finish for each of them to allow all remaining task nodes with explicit deadlines to finish. Thus tasks without explicitly assigned deadlines can be said to have *implicit deadlines*. We use implicit or explicit deadline to represent priority of a task node, as the earlier the 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 task nodes with explicit deadlines assigned (step 1~4). Then in step 5~9, the nested function is called to back-traverse from nodes with explicit deadlines to predecessor nodes, calculating implicit deadlines of other task nodes in a depth-first manner. As a task node can have multiple successor nodes

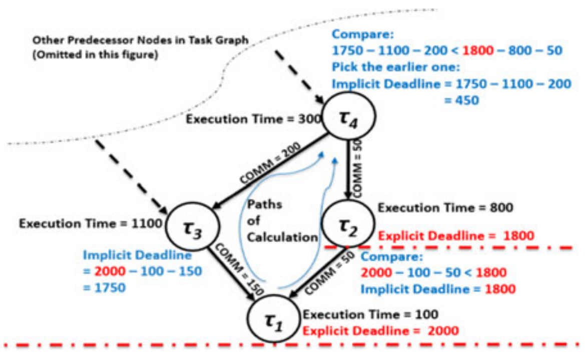


Fig. 6. An illustration example of implicit deadline calculation.

in a task graph, multiple values of implicit deadline can be derived from different calculation paths or different explicit deadlines of nodes. 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 Fig. 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:
  $\psi$, task graph set to be scheduled
  $DD\_LINE_{i,j}$, deadline of task graph instance $i$ node $j$
  $WCET_j$, 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 $\tau_j$
priority_assign():
1    for $G_i$ in $\psi$:
2      for $\tau_j$ in task nodes of Gi with deadline constraints:
3        dead_priority$_j \leftarrow DD\_LINE_{i,j}$
4        call nested_priority($\tau_j$)
nested_priority($\tau_j$):
5    for $\tau_{j''}$ in all predecessor nodes of $\tau_j$:
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($\tau'_j$)
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 $\tau_j$
Outputs:
  alloc$_j$, allocation results of task node $\tau_j$, 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 $\tau_j$ in sys_pool with highest $WCET_j$
13     find core $k$, with lowest $CORE\_WCET_k$
14     allocation $\tau_j$ 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 [32], 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.

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). 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 or explicit 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).

Algorithm 3. Checkpoint-Based Iterative Analysis

```
Inputs:
  $EGY\_BGT$, specified energy budget for one schedule window
  $T_{win}$, duration of a schedule window
  $implicit\_priority_j$, implicit deadlines as priority indicators of task node $\tau_j$
  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           $checkpoint_i \leftarrow$ all system status (include $T_{cur}$)
4       $alloc \leftarrow$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           $miss_i \leftarrow TRUE$
9           all system status $\leftarrow checkpoint_i$
10      else if node $\tau_j$ of task graph $G_i$ missed its implicit deadline:
11          find the critical path in $G_i$ that ends at $\tau_j$
12          find $\tau_{j'}$, the task node with lowest frequency assigned
13          $freq_{j'} \leftarrow freq_{j'} + 1, nested\_priority(\tau_j')$
14          find $\tau_{j''}$, the task finished just before $\tau_j$ on the same core
15          $freq_{j''} \leftarrow freq_{j''} + 1, nested\_priority(\tau_j'')$
16          all system status $\leftarrow checkpoint_i$
17      else:
18          $T_{cur} = T_{cur} + T_{step}$
19  save final tentative schedule as schedule template
```

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).

At design time, the ATG heuristic is executed multiple times for 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 multicore computing platform.

6 Adaptive Online Management

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

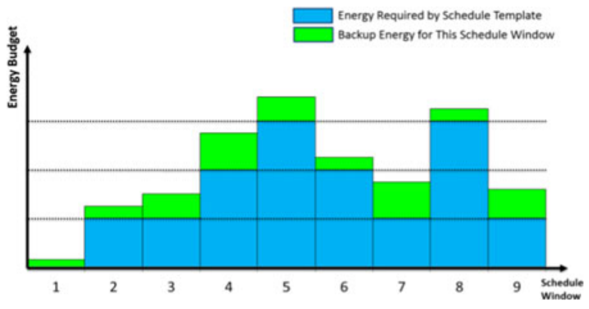


Fig. 7. Residual energy availability over time.

most task graph instances, contingent on the energy budget. Each selected template provides a schedule with task-to-core allocation, execution order, and frequency assignment for every task node. As the offline schedule template assumes all cores to be identical, each task node is actually only assigned to a virtual core id. We call a set of task nodes assigned offline to a core id as a *workload partition*, each of which should be allocated to a dedicated physical core for execution in the upcoming schedule window. This partition-to-core allocation can be adjusted dynamically to mitigate aging effect that leads to hard failures (see Section 6.2). On the other hand, the amount of residual energy that exceeds the energy requirement of the selected schedule template is used as backup energy (Fig. 7) for possible task re-execution to recover from detected faults caused by soft errors during execution (see Section 6.3).

Algorithm 4. Dynamic Workload Distribution in Awareness of Core Aging

```
Inputs:
  work_partition_set, set of workload chunks in schedule template, each chunk should be executed on an individual core
  R_set, reliability of cores
Output:
  Allocation of workload partitions to cores.
1   for each reliability detection interval:
2       update R_set
3       sort cores in non_decreasing order of hard reliabilities
4   for each schedule window:
5       get work_partition_set from selected schedule template
6       sort work_partition_set in non-decreasing order of workload
        partitions' total task execution cycles
7       for all cores in system:
8           allocate workload partition with lowest execution cycles to unassigned core with lowest hard reliability
```

6.2 Aging-Aware Allocation of Workload Partitions

After a schedule template is selected based on the energy budget for a schedule window, our framework can trigger a scheme to allocate workload to cores with awareness of core aging to enhance system lifetime. Although schedule templates set fixed execution strategies for all task nodes, there still exists some flexibility as the allocation of *workload partitions* to cores can still be altered from the default provided by the schedule template, for a homogeneous multicore platform.

The outline of our aging-aware dynamic workload allocations scheme is shown in Algorithm 4. We assume that our scheduler can fetch hard reliability information of cores from aging detection circuitry [39] or execution history tracking mechanisms at certain interval (much longer interval than schedule windows) [14] (steps 1~3). Besides, at the beginning of each schedule window, workload partitions are fetched from the selected schedule template (step 4~6). Then recursively our heuristic allocates unassigned workload partitions with the lowest workload intensity to idle cores with the lowest hard reliability. As a result, cores with faster wear-out during previous system operation are more likely to receive less workload than others so that aging processing on the entire multicore system can be rebalanced. Otherwise, some cores may be utilized more intensively than others and detrimentally impact system lifetime of the multicore chip.

Algorithm 5. Dynamic Slack Reclamation and Soft Error Handling

```
Inputs:
  $T_{win}$, duration of a schedule window
  $\psi$, task graph set to be scheduled
  $start_j$, designated time to start execution of $\tau_j$ in selected schedule template
  $bkup\_energy_j$, 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 $\tau_j$ in taskpool:
4          if $\tau_j$ 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              $freq_j \leftarrow freq_j - 1$
8              bkup_energy ← bkup_energy + energy saved
9      execute task nodes based on schedule template
10     for $\tau_j$ in just finished tasks:
11         if error detected on $\tau_j$:
12           if $T_{cur} \le start_j$:
13             schedule another instance of $\tau_j$ to re-execute
14        else if ∃ a freq that has reduced WCET > $T_{cur} - start_j$ and can be supported by bkup_energy:
15             $freq_j \leftarrow freq$,
16             bkup_energy ← bkup_energy – energy used
17             schedule another instance of $\tau_j$ to re-execute
18         else:
19             find next node to execute on the same core, $\tau_{j'}$
20             if $\tau_j \in G_i$, $\tau_{j'} \in G_{i'}$ and $G_i \neq G_{i'}$:
21               update remain WCEC of both graphs
22               if $G_{j'}$ has more WCEC:
23                 drop $G_{j'}$
24                 schedule $\tau_j$ to re-execute
25               else:
26                 drop $G_j$
27             else:
28               drop $G_j$
```

6.3 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

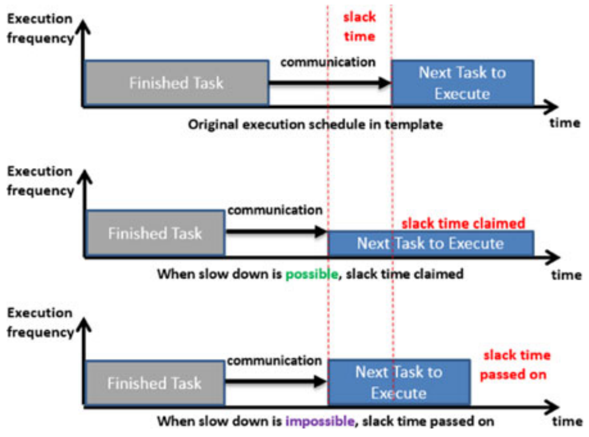


Fig. 8. Illustrative example of slack time reclamation.

task graph scheduling problem to design time. However, embedded systems can encounter 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 5.

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 task nodes, 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 Fig. 8. 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).

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

TABLE 4
Results of MILP Based Schedule Template Generation for a 4-Core Homogeneous Embedded Computing System

Schedule template ID	Energy budget	Objective value	Energy budget usage	Energy usage	Number of misses
0	0 J	9.000	0.0%	0 J	9
1	24 J	7.846	84.6%	20.3 J	7
2	48 J	5.920	92.0%	44.2 J	5
3	72 J	4.968	96.8%	69.7 J	4
4	96 J	4.726	72.6%	69.7 J	4
5	120 J	3.808	80.8%	97.0 J	3
6	144 J	2.904	90.4%	130.2 J	2
7	168 J	2.775	77.5%	130.2 J	2
8	192 J	1.923	92.3%	177.2 J	1
9	216 J	1.820	82.0%	177.2 J	1
10	240 J	0.965	96.5%	231.6 J	0

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$\sim$20). If true, both task nodes have the WCEC of their unfinished nodes updated and the task graph with the higher WCEC is dropped (step 21$\sim$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 soft and hard reliability-aware hybrid workload management framework. 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) [33]. We chose the Gurobi Optimizer [34] as our MILP solver to generate the optimal schedule templates. We generated task graph sets based on the *networking*, *telecom*, and *auto-industry* applications from the Embedded System Synthesis Benchmark Suite (E3S) [35] and the distribution of actual execution times of task nodes is obtained from [36]. We also used synthetic task graph sets from TGFF [41]. In the rest of this section, we first analyze characteristics of the generated schedule templates and then study system performance for our proposed hybrid workload management scheme compared to prior work.

7.2 Template Generation Analysis

In the first set of experiments, we check the quality and optimality of the schedule templates generated using our MILP approach on a four-core system. We scale task node execution time of four periodic task graphs from E3S 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, which sets a stringent timing requirement for a system with four cores. The resulting periodic task graphs with targeted utilization have periods ranging from 20 to 60 seconds and execution times at 1,000 MHz operating frequency ranging approximately from 16 to 48 seconds with maximum per-graph parallelism of 4. Besides, apart from the deadlines at task-graph termination nodes, we randomly select few task nodes in each task graph to assign explicit deadlines that result in even more stringent timing requirements (Note: utilization of the entire task graph stays the same as it is calculated based on maximum frequency; see Section 3.1.3). Based on the periods of the generated task graphs, we set the length of schedule window to be 1 minute, within which nine task graph instances arrive in the system for execution. We generated 11 schedule templates with energy budgets evenly distributed from 0 to E_{peak}, where E_{peak} is the assumed peak energy budget (240 Joules) available from our solar energy harvesting system.

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 96.5 percent 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. Note that there are three pairs of templates in Table 4 that are identical to each other with the same extent of energy usage and instance misses. Thus it is unnecessary to increase number of budget levels indefinitely (much beyond number of application task graph instances in a window) as the resulting smaller energy budget difference between levels will lead to identical and redundant schedule templates that increase storage overheads.

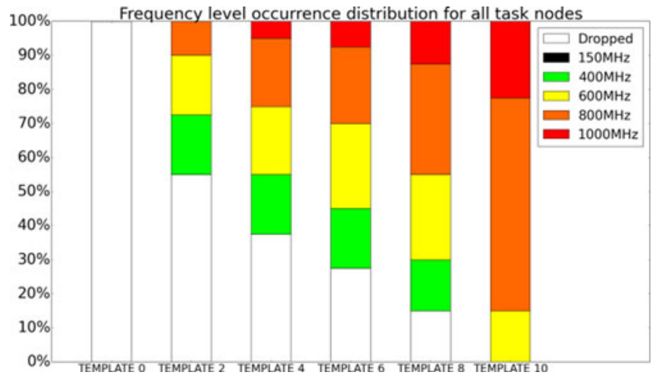


Fig. 9. Frequency level occurrence distribution for all task nodes.

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 Fig. 9. 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 150 MHz frequency is never used by any schedule; this is due to the fact that the frequency level of 150 MHz has lower efficiency and lower speed than the 400 MHz level (see Table 1). Therefore our MILP optimization approach rules out this sub-optimal frequency choice as it is always better to schedule at 400 MHz instead.

Table 5 shows a comparison between the MILP and ATG heuristics, in terms of execution time and memory footprint, for two problem instances of different sizes. While the MILP approach generates optimized schedule templates, we found that the approach is not scalable for larger problem sizes. ATG sacrifices some performance but provides significant speedup.

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

In this section, we compare 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) [37]. As we assume that our system only operates in daylight, system performance is evaluated 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 two recent works: 1) SDA [10], 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 2) LP+SA [38], 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

TABLE 5
Computation Resource Requirement of MILP and ATG

Method	Complexity		Memory footprint	Execution time
	Number of nodes	Number of edges		
ATG	36	44	42 MB	0.1 hour
MILP			257 MB	6.5 hour
ATG	150	193	61 MB	1 hour
MILP			7,693 MB	492 hour

near-optimal solution. To compare HyWM with these approaches, we adapt the techniques to our environment and problem formulation. As SDA is 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 input to LP+SA.

The results of our comparison study on task graph sets extracted from E3S are shown in Fig. 10. For the platform with four cores, it can be observed that SDA has the highest miss rate. This is because SDA cannot arrange specific execution schedules for task nodes along critical paths of task graphs and thus all nodes in a task graph are assigned the same frequencies, resulting in a less efficient schedule. LP+SA outperforms SDA as it can generate task dependency-aware offline schedules after comprehensive design space exploration unlike in 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 both of these efforts. HyWM-LP reduces absolute miss rate by 5.6 and 9.0 percent over LP+SA and SDA, respectively. In terms of relative performance improvement, HyWM-LP accomplishes an improvement of 12.9 and 20.1 percent over LP+SA and SDA, respectively. HyWM-ATG ends up with higher miss rates than HyWM-LP, however it still outperforms the other two techniques from prior work. HyWM-ATG can however serve as an alternative approach when scalability is an

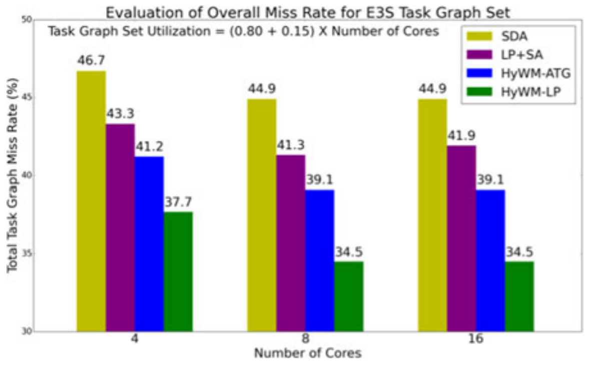


Fig. 10. Comparison between HyWM framework and prior work ([10], [38]) in terms of overall system task graph miss rate.

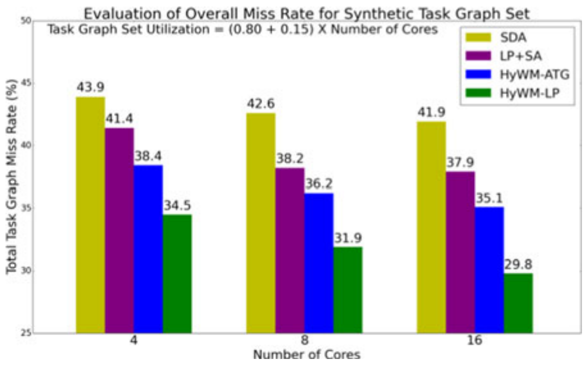


Fig. 11. Comparison of overall system task graph miss rate on synthetic task graph set with higher degree of parallelism.

issue, e.g., for larger problem sets. Fig. 10 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 (up to 23.2 percent reduction relatively) 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 the E3S task graph set, which has maximum per-graph parallelism of 4, 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. Fig. 11 shows another group of results based on a synthetic task graph set generated using TGFF [41], with the same targeted utilization as E3S but maximum per-graph parallelism increased to 8. We can observe in Fig. 11 that while performance differences among techniques are similar to the results shown in Fig. 10, all techniques continue to get miss rate reduction on a 16-core system, as there is additional parallelism to exploit in the synthetic task graphs set (in contrast, miss rate improvements for E3S saturate for the 16-core system as shown in Fig. 10).

7.4 Evaluation of System Performance with Soft 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 10^{-5} soft errors per second per core at maximum frequency [20]. As there is no prior work on soft error handling for systems with energy harvesting, we conduct multiple tests with run-time management features enabled progressively on a four-core system to show each feature's effectiveness, with results shown in Fig. 12. Each of the configurations shown in the figure are described as follows: ±*none*: This base case uses HyWM-LP with soft error injection and no run-time adjustment technique enabled, and has a miss rate of 45.4 percent. ±*slack reclamation*: System miss rate drops to 34.4 percent when the slack reclamation capability in run-time heuristic is activated. ±*drop*: 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 31.9 percent. ±*compare before drop*: When the heuristic adds support for dropping other task graphs with high WCET to allow re-execution of the faulty task node, the system sees a drop in miss rate to 30.2 percent. ±*backup energy*: when the fully-enabled heuristic is utilized that adds further support for utilizing backup energy to speed up faulty node re-execution, we end up with the lowest miss rate of 25.6 percent.

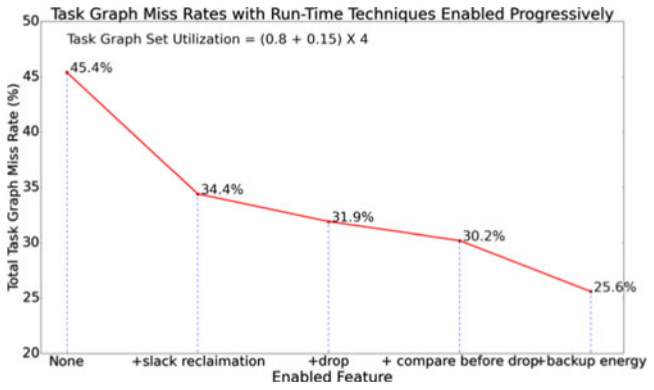


Fig. 12. Miss rate comparison with run-time techniques enabled progressively.

The results in Fig. 12 highlight the significance of slack reclamation and soft error handling in our run-time framework with a relative 43.6 percent miss rate reduction for the best configuration compared to the baseline case.

7.5 Evaluation of System Hard Reliability and MTTF

In this section, we explore the impact of aging on multicore embedded systems with energy harvesting. In the model, we set the critical currently density $J_0 = 1.5 \times 10^6\ A/cm^2$, the activation energy $E_a = 0.4\ 8\ eV$, and assume a slope parameter in the Weibull distribution $\beta = 2$ [22] (Section 3.1.5). We simulated execution of systems over a long period of time with solar harvesting profiles randomly selected from a preset pool. At the beginning of each schedule window, the aging progress is estimated based on average core frequencies, supply voltages, and core temperatures of previous schedule windows. All experiments in this section target eight-core systems executing the same workload as in experiments of previous sections.

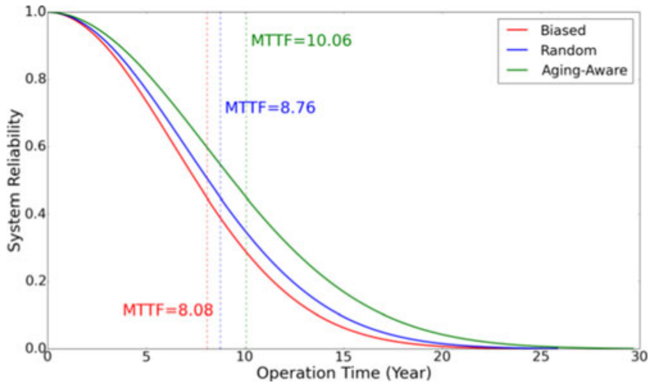


Fig. 13. Comparison of reliability and MTTF for different workload allocation schemes.

TABLE 6
System MTTF and Performance Comparison with Different Failure Thresholds

Failure Threshold*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
MTTF (years)	10.06	15.58	20.22	24.66	29.27	34.44	40.94	51.35
Processing Capability Before System Failure (%)	100	92.3	80.1	68.8	52.4	34.9	18.4	7.0
Average Processing Capability during System Lifetime (%)	100	96.5	92.5	87.7	81.7	75.0	67.3	60.3

Failure threshold: number of cores that must fail before a chip is considered unusable.

The first set of experiments is designed to evaluate the benefit of our aging-aware workload allocation scheme, which is compared with *Biased*, an allocation scheme that always allocates workload partitions with low to high workload intensities on to cores with low to high core id respectively, and *Random*, the original partition-to-core allocation randomly generated during schedule template generation. All experiments have failure thresholds set to 0 and the results on hard reliability and MTTF of system are shown in Fig. 13. We can observe that our *Aging-Aware* scheme (which is used in our HyWM framework) results in better hard reliability over time as it can reallocate workload partitions to balance aging progress among cores, ending up with 14.8 and 24.5 percent MTTF improvements compared to *Biased* and *Random* without diminishing system performance.

The last set of experiments performs sensitivity analysis for our aging-aware workload allocation scheme, focusing on system MTTF and performance analysis when different failure thresholds are considered. The results of this experiment are shown in Table 6. As we can see, increasing failure threshold allows the system to operate for longer periods of time (higher MTTF), however, this comes at the cost of a decrease in peak processing capability before failure and average system processing capability over

8 Conclusions

In this paper, we proposed a hybrid design-time and run-time framework for reliable resource allocation in multicore 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. With the increasing prevalence of energy-constrained computing, energy scavenging, execution time variability, and the rise in soft errors and hard failures 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 multicore embedded computing platforms. Our future work will explore expanding the versatility of energy harvesting-based platforms by considering mixed-criticality workloads, combination with utility power, and support for overnight operation.

Acknowledgments

This research is supported by grants from SRC, the US National Science Foundation (NSF) (CCF-1252500, CCF-1302693), and AFOSR (FA9550-13-1-0110).

References

[1] ARM Cortex-A9 Processor. (2008). Retrieved Nov 11, 2014 [Online]. Available: http://www.arm.com/products/processors/cortex-a/cortex-a9.php

[2] NVIDIA. The benefits of multiple CPU cores in mobile devices. (2010). Retrieved Feb. 21, 2012 [Online]. Available: http://www.nvidia.com/content/PDF/tegra_white_papers/Benefits-of-Multi-core-CPUs-in-Mobile-Devices_Ver1.2.pdf

[3] N. Kapadia and S. Pasricha, "VERVE: A framework for variation-aware energy efficient synthesis of NoC-based MPSoCs with voltage islands," in *Proc. Int. Symp. Quality Electron. Des.*, 2013, pp. 603–610.

[4] C. Li, W. Zhang, C. B. Cho, and T. Li, "SolarCore: Solar energy driven multicore architecture power management," in *Proc. IEEE 17th Int. Symp. High-Perform. Comput. Archit.*, 2011, pp. 205–216.

[5] X. Lin, Y. Wang, D. Zhu, N. Chang, and M. Pedram, "Online fault detection and tolerance for photovoltaic energy harvesting systems," in *Proc. Int. Conf. Comput.-Aided Des.*, 2012, pp. 1–6.

[6] Y. Zhang, Y. Ge, and Q. Qiu, "Improving charging efficiency with workload scheduling in energy harvesting embedded systems," in *Proc. 50th Annu. Des. Autom. Conf.*, 2013, pp. 1–8.

[7] C. Moser, D. Brunelli, L. Thiele, and L. Benini, "Lazy scheduling for energy harvesting sensor nodes," in *Proc. FIP TC 10 Working Conf. Distrib. Parallel Embedded Syst.*, 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] J. Lu and Q. Qiu, "Scheduling and mapping of periodic tasks on multicore embedded systems with energy harvesting," in *Proc. Int. Green Comput. Conf.*, 2011, pp. 1–6.

[10] Y. Xiang and S. Pasricha, "Harvesting-aware energy management for multicore platforms with hybrid energy storage," in *Proc. 23rd ACM Int. Conf. Great Lakes Symp. VLSI*, 2013, pp. 25–30.

[11] Y. Xiang and S. Pasricha, "Thermal-aware semi-dynamic power management for multicore systems with energy harvesting," in *Proc. Int. Symp. Quality Electron. Des.*, 2013, pp. 619–626.

[12] Y. Xiang and S. Pasricha, "Run-time management for multicore embedded systems with energy harvesting," in *IEEE Trans., Very Large Scale Integration (VLSI) Syst.*, vol. PP, no. 99, pp. 1–1, 2014.

[13] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *Proc. 11th Int. Symp. High-Perform. Comput. Archit.*, 2005, pp. 243–247.

[14] D. Zhu, R. Melhem and D. Mosse, "The effects of energy management on reliability in real-time embedded systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2004, pp. 35–40.

[15] D. Zhu and H. Aydin, "Energy management for real-time embedded systems with reliability requirements," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2006, pp. 528–534.

[16] B. Zhao, H. Aydin, and D. Zhu, "Generalized reliability-oriented energy management for real-time embedded applications," in *Proc. 48th Des. Autom. Conf.*, 2011, pp. 381–386.

[17] 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, p. 21, Apr. 2013.

[18] Y. Zou and S. Pasricha, "Reliability-aware and energy-efficient synthesis of NoC based MPSoCs," in *Proc. Int. Symp. Quality Electron. Des.*, 2013, pp. 643–650

[19] Y. Zou, Y. Xiang, and S. Pasricha, "Analysis of on-chip interconnection network interface reliability in multicore systems," in *Proc. IEEE 29th Int. Conf. Comput. Des.*, 2011, pp. 427–428.

[20] 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, Jun. 2012.

[21] A. K. Coskun, R. Strong, D. M. Tullsen, and T. S. Rosing, "Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 169–180, Jun. 2009.

[22] L. Huang, F. Yuan, and Q. Xu, "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms," in *Proc. Conf. Des. Autom. Test Eur.*, 2009, pp. 51–56.

[23] M. Basoglu, M. Orshansky, and M. Erez, "NBTI-aware DVFS: A new approach to saving energy and increasing processor lifetime," in *Proc. 16th ACM/IEEE Int. Symp. Low Power Electron. Des.*, 2010, pp. 253–258.

[24] J. Luo and N. K. Jha, "Power-conscious joint scheduling of periodic task graphs and aperiodic tasks in distributed real-time embedded systems," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, 2000, pp. 357–364.

[25] R. Sakellariou and H. Zhao, "A hybrid heuristic for DAG scheduling on heterogeneous systems," in *Proc. 18th Int. Parallel Distrib. Process. Symp.*, 2004, pp. 111.

[26] A. K. Coskun, T. T. Rosing, K. A. Whisnant, and K. C. Gross, "Static and dynamic temperature-aware scheduling for multiprocessor SoCs," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 16, no. 9, pp. 1127–1140, Sep. 2008.

[27] M. Veerachary, T. Senjyu, and K. Uezato, "Maximum power point tracking of coupled inductor interleaved boost converter supplied PV system," *IEE Electric Power Appl. IET*, vol. 150, no. 1, pp. 71–80, Jan. 2003.

[28] ReliaSoft. The Limitations of Using the MTTF as a Reliability Specification. (2003). Retrieved November 11, 2014 from [Online]. Available: http://www.weibull.com/hotwire/issue32/hottopics32.htm

[29] Intel. Intel XScale Core. (2004). Retrieved May 28, 2013 from [Online]. Available: http://download.intel.com/design/intelxscale/27347302.pdf

[30] Y. K. Kwok and I. Ahmad, "Benchmarking the task graph scheduling algorithms," in *Proc. 1st Merged Int. Parallel Process. Symp. Parallel Distrib. Process.*, 1998, pp. 531–537.

[31] V. Suhendra, C. Raghavan, and T. Mitra, "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2006, pp. 401–410.

[32] B. Xia and Z. Tan, "Tighter bounds of the First Fit algorithm for the bin-packing problem," *Discr. Appl. Math.*, vol. 158, no. 15, pp. 1668–1675, Aug. 2010.

[33] A. Makhorin. GLPK—GNU linear programming kit. (2012). Retrieved Mar. 20, 2015 from [Online]. Available: http://www.gnu.org/software/glpk/

[34] Gurobi. Gurobi Optimization. (2014). Gurobi optimizer reference manual, version 6.0. Retrieved March 20, 2015 from [Online]. Available: http://www.gurobi.com/documentation/6.0/refman.pdf

[35] R. Dick. Embedded System Synthesis Benchmarks Suites (E3S). (2008). Retrieved May 13, 2014 from http://ziyang.eecs.umich.edu/~dickrp/e3s/

[36] H. F. Sheikh and I. Ahmad, "Dynamic task graph scheduling on multicore processors for performance, energy, and temperature optimization," in *Proc. Int. Green Comput. Conf.*, 2013, pp. 1–6.

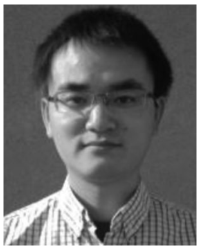
[37] NREL. National Renewable Energy Laboratory: Measurement and Instrumentation Data Center. (2002). Retrieved Februray 23, 2011 from [Online]. Available: http://www.nrel.gov/midc

[38] R. Watanabe, M. Kondo, M. Imai, H. Nakamura, and T. Nanya, "Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor SoC," in *Proc. Conf. Des. Autom. Test Eur.*, 2007, pp. 797–802.

[39] X. Wang, M. Tehranipoor, S. George, D. Tran, and L. Winemberg, "Design and analysis of a delay sensor applicable to process/environmental variations and aging measurements," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 20, no. 8, pp. 1405–1418, Aug. 2012.

[40] M. Hazewinkel, Ed., "Weibull distribution," in *Encyclopedia of Mathematics*. New York, NY, USA: Springer, 2001.

[41] R. P. Dick, D. L. Rhodes, and W. Wolf, "TGFF: Task graphs for free," in *Proc. 6th Int. Workshop. Hardware/Softw. Codes.*, 1998, pp. 97–101.



Yi Xiang (S'11) received the BS degree in microelectronics from the University of Electronic Science and Technology of China, Chengdu, China, in 2010. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering, Colorado State University, Fort Collins, CO. His current research interests include low-power computing, parallel embedded systems, heterogeneous computing, and CAD algorithms. He is a student member of the IEEE.

Sudeep Pasricha (M'02-SM'13) received the BE degree in electronics and communication engineering from Delhi Institute of Technology, India, in 2000, and the MS and PhD degrees in computer science from the University of California, Irvine, in 2005 and 2008, respectively. He is currently an associate professor of electrical and computer engineering at Colorado State University, Fort Collins. His research interests include energy efficiency and fault-tolerant design for network and memory architectures in multicore computing. He is or has been an organizing committee member and/or technical program committee member of various IEEE/ACM conferences such as DAC, DATE, CODES+ISSS, NOCS, ISQED, VLSID, and GLSVLSI. He received several awards for his research contributions, including the 2015 IEEE TCSC Award for Excellence for a mid-career researcher and the 2013 AFOSR Young Investigator Award, as well as Best Paper Awards at the ACM GLSVLSI 2015, IEEE AICCSA 2011, IEEE ISQED 2010, and ACM/IEEE ASPDAC 2006 conferences. He is a senior member of the IEEE.