Power-Efficient Embedded Processing with Resilience and Real-Time Constraints

Liang Wang[†], Augusto J. Vega[*], Alper Buyuktosunoglu[*], Pradip Bose[*], and Kevin Skadron[†]
[*]IBM T. J. Watson Research Center and [†]University of Virginia

***Abstract*—Low-power embedded processing typically relies on dynamic voltage-frequency scaling (DVFS) in order to optimize energy usage (and therefore, battery life). However, low voltage operation exacerbates the incidence of soft errors. Similarly, higher voltage operation (to meet real-time deadlines) is constrained by hard-failure rate limits. In this paper, we examine a class of embedded system applications relevant to mobile vehicles. We investigate the problem of assigning optimal voltage-frequency settings to individual segments within target workflows. The goal of this study is to understand the limits of achievable energy efficiency (performance per watt) under varying levels of system resilience constraints. To optimize for energy efficiency, we consider static optimization of voltage-frequency settings on a per-application-segment basis. We consider both linear and graph-structured workflows. In order to understand the loss in energy efficiency in the face of environmental uncertainties encountered by the mobile vehicle, we also study the effect of injecting random variations in the actual runtime of individual application segments. A dynamic re-optimization of the voltage-frequency settings is required to cope with such in-field uncertainties.**

***Keywords*—Reliability, energy-efficiency, linear optimization**

I. Introduction

A powerful control parameter for power management of embedded processor systems is dynamic voltage-frequency scaling (DVFS). However, soft error rates (SERs) are known to increase sharply as the supply voltage is scaled downward [10]. Hence, in order to preserve system resilience levels, it is important to apply voltage scaling carefully, keeping in mind the varying levels of vulnerability to SER within an application's execution profile. On the other hand, overclocking or turbo-boosting (with higher voltages applied if/as necessary) to meet real-time deadline, comes at the cost of higher power (or current) density and temperature (as well as higher gate-oxide field stress), which results in higher hard-failure rates.

In this paper, we consider a class of embedded systems that require high levels of power-performance efficiency while meeting mission-critical reliability specifications and real-time performance targets. Such systems require an energy optimization protocol that is cognizant of the variable resiliency needs and properties of the executed application. A representative example of such an embedded system of interest is a single unmanned aerial vehicle (UAV) or a swarm of such UAVs. These are typically engaged in remote sensing of ground images, for the purposes of reconnaissance, object recognition/tracking and tactical response.

We first describe PEARL, a novel software modeling framework that enables users to: (a) *statically* prepare application workflows for energy-optimized resilience; and (b) experiment with *run-time* deployment options in targeted embedded systems. PEARL stands for power efficient and resilient embedded processing with real-time constraints, and is built upon the earlier toolset described by Wang [13]. We then describe the use of PEARL to pursue *static* optimization of voltage-frequency settings across segments of a workflow. We handle both linear and directed acyclic graph (DAG)-structured workflows. Subsequently, we describe how *dynamic* (run-time) uncertainties are factored into the user-driven workflow optimization process. Our analysis shows up to 15% and 35% energy efficiency improvement over a simple baseline, for linear and graph workflows respectively. Moreover, our proposed dynamic iterative approach achieves up to 25% energy efficiency improvement over a conservative static approach, while maintaining the same level of confidence in meeting deadline constraints.

II. PEARL Facility and Methodology

A. Workflow Model

We use the term *workflow* to denote a collection of inter-dependent tasks as executed by a single mobile embedded system (e.g. UAV) or a swarm thereof. The basic building block of a workflow is an application or a task. For simplicity, we stipulate that any workflow-specific voltage-frequency management decision is applied only at the beginning of an application segment. Within an application, the voltage-frequency setting remains fixed.

A linear workflow (Figure 1a) is used to model a single embedded system, where a sequence of applications is ordered by inter-application dependencies. A graph workflow (Figure 1b) is used to model multiple embedded systems, across which inter-communications are required in order to solve the mission-assigned task cooperatively. Each embedded system runs its own thread of linear workflow, while applications within that linear workflow are also subject to dependencies from other embedded systems, known as inter-dependencies. Adding inter-dependencies, threads of linear workflows form a directed acyclic graph (DAG). For simplicity, we only consider dependencies across applications, and leave the overhead associated with inter-application communication as future work.

B. Individual Application Programs

We use applications from the PERFECT benchmark suite provided for research in the DARPA PERFECT program [1]. The PERFECT suite includes 12 applications taken from signal and image processing tasks: outer product (*oprod*), system solve (*syssol*), inner product (*iprod*), discrete wavelet transform (*dwt53*), histogram equalization (*histo*), 2D convolution (*2dconv*), path finding algorithms (*pfa1* and *pfa2*), back projection (*bp*), change detection (*cd*), lucas kanade (*lucas*), and debayer (*debayer*). To add diversity to our set of

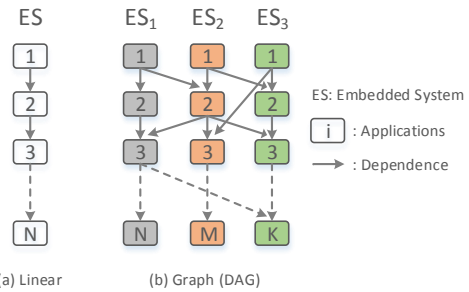

Fig. 1: Workflow illustration. A linear workflow (a) is a stitched sequence of applications, modeling a single embedded system (ES). A graph workflow (b) is a set of linear workflows with inter-dependencies, modeling multiple embedded systems with inter-system communications.

benchmarks, we also select 8 applications from SPEC2006 in the expectation that embedded systems will have an increasing burden of general purpose applications. Selected applications are *gcc*, *perlbench*, *bzip2*, *h264ref*, *mcf*, *sphinx*, *dealII*, and *lbm*, which constitute an application set with diverse computing domains and characteristics. We will use these 20 applications to synthesize workflows studied later in this paper.

C. Power and Performance Modeling

We use machine measurement from real hardware for power and performance characterizations for all the applications mentioned in the last section. The experimental system is a state-of-the-art multi-core server processor system with POWER ISA. For each application, we measure the power and performance (execution time) under 20 frequency levels with a step size of 100MHz. The frequency range covers operating levels such as turbo boosting, nominal operation, and low-power operations with low supply voltage. Note that, in our experimental setup, the appropriate supply voltage is determined by firmware for each frequency level. Therefore, the frequency and supply voltage always come as a pair for voltage-frequency scaling. For simplicity, we will refer to voltage-frequency settings by their corresponding frequency levels.

The length of each application plays a critical role in the voltage-frequency allocation problem. For example, if a single application dominates the workflow in terms of the execution time, it will have an overwhelming effect in determining the energy efficiency of the workflow. The solution, in this case, would be straightforward: just choose the most efficient operation level for the *dominating* application. Therefore, in order to remove such application-specific bias in our illustrative analysis, we scale the execution time of each application so that all of them run for the same length of time (10 seconds) at the nominal supply voltage. This assumption allows us to explore the maximum achievable improvement via full-workflow scheduling. Furthermore, the DVFS transition time (which is in the milliseconds range) is a very small fraction of each application's run time. We ignore the transition overhead associated with DVFS by only allowing DVFS transitions at application boundaries.

D. Resilience Modeling

System reliability, in the context of transient (soft) and permanent (hard) errors, is of critical importance for embedded systems carrying out mission-critical tasks. For soft error rate (SER) modeling, we estimate the failure rate (measured in standard units of failures in time or FITs) using an approach adopted from industrial practice [5], [9]. In such an evaluation methodology, machine-level derating (MD) and application-level derating (AD) are treated as decoupled factors. We model the system-level FIT as: $FIT_{SER} = SER_{latch} \times N_L \times AD \times MD$, where SER_{latch} refers to the raw per-latch SER; N_L is the number of latches; AD refers to the probability of application-level masking, given a corruption of the program-visible (register and memory) state; and MD is the probability of machine- (or microarchitecture) level masking in the context of a single latch bit upset.

To model SER_{latch}, we empirically fit it to the voltage sensitivity gradients published in [10] using the formula shown below:

$$\text{SER}_{\text{latch}} = e^{\alpha \cdot \text{Vdd} + \beta}$$

where Vdd is the supply voltage, and α,β are fitting constants.

To derive the AD factor for a given application, we use an Application Fault Injection (AFI) tool with a built-in programmable statistical fault injection routine. AFI makes use of the ptrace debugging facility that is available in POSIX-compatible operating systems to put together a framework in which any target application can be compiled and run, under user-controllable fault injection directives. Each injected error in program-visible architectural state leads to one of the four outcomes: 1) *Masked*, where there is no effect on the correctness of program execution or final output; 2) *SDC* or silent data corruption where a deviation of the program output, relative to the golden value is observed; 3) *Crash*, where the program terminates prematurely; and 4) *Hang*, where a hung state is observed such that there is no forward progress in the program execution. The overall AD factor is empirically derived as the "masked" rate as observed from AFI experiments. However, if the concern is mainly on SDC (which is often the case in numerical, compute-intensive codes), the AD factor may be computed as: $AD = 1 - rate_{SDC}$.

MD factors can be estimated using the same approach as used by the authors of Phaser [8], which factors in true data residency statistics for each functional unit. These statistics can be derived from a properly validated, cycle-accurate architectural simulator as explained in [8]. As has been observed in prior work on POWER machines [9], the AD factor tends to dominate over MD by a large factor, especially if the focus is only on SDC. Therefore, for simplicity of analysis, in this paper we only focus on AD, while effectively assuming that MD is invariant across the class of applications considered for a given (fixed) machine implementation.

In optimizing for power-performance efficiency (e.g. billions of instructions per second per watt or BIPS/W), the tendency is to push towards lower voltage-frequency operating points as much as possible, without violating latency (i.e. deadline) constraints. Therefore, our resilience modeling is mostly focused on soft errors, because, as discussed above, SER increases very sharply as the supply voltage is scaled downward, which constrains the useful range of DVFS. For operation at nominal or sub-nominal voltage-frequency regions, hard-fault incidence rates are well below design specification limits, so these effects need not be considered explicitly in resilience models within PEARL. Even when turbo-mode operation is required (very rarely) to meet deadline constraints in field operation, the hard-failure rate is actually within specification, since the processor is designed to tolerate the higher voltages and temperatures incurred in turbo-mode. Nonetheless, for completeness, we decided to consider power consumption as a proxy or gauge for hard-failure rates of the system. With increased voltage-frequency operation points, the current drawn is higher, so failures linked to current density (e.g. electromigration) are higher. Also, higher voltages translate to failure rates associated with mechanisms such as oxide breakdown and bias-temperature instability (BTI).

E. Workflow Synthesis

PEARL is a framework for workflow optimization. However, the target benchmark suites are composed of individual applications. Therefore, a workflow synthesizer, which has been embedded in the PEARL framework, is required to assemble workflows from applications according to user-specified parameters, such as application dependencies, execution time (duration), etc. The synthesizer is able to generate

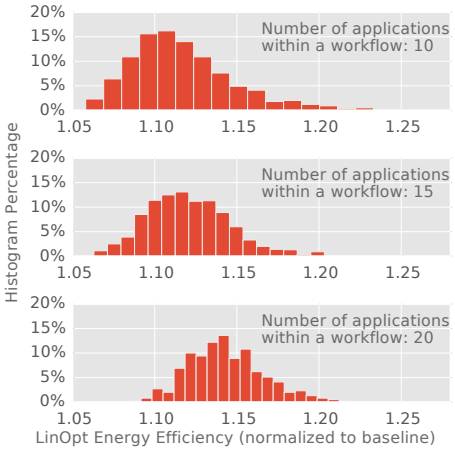

Fig. 2: Histogram of the maximum improvement in BIPS/W achieved by LinOpt over the baseline across 1,000 generated workflows with a given number of applications. Power and resilience constraints are set as the maximum of applications within a workflow running at the nominal frequency level. X-axis is the relative energy efficiency of LinOpt normalized to the baseline. Y-axis is histogram in percentage.

various workflows for both design space exploration and algorithm evaluation. In order to characterize the potential benefits of full-workflow scheduling, this paper synthesizes workflows using normalized applications, as described in subsection 2-C. Note, however, that the PEARL framework is capable of handling arbitrary workflows composed of applications with any variety of execution time.

III. Linear Workflow Optimization

For linear workflows, we use an approach based on linear programming (LinOpt), adapted from the one used by Wang et al. [13]. The algorithm optimizes overall energy efficiency defined in terms of billions of instructions per second per watt (BIPS/W), while obeying constraints of power, resilience, and deadline. To briefly show the benefit of optimization on a linear workflow, we study the distribution of the energy efficiency improvement obtained by LinOpt over a baseline heuristic. The baseline picks a single frequency level that is the lowest that meets all the constraints across all the applications. We use the embedded workflow synthesizer to generate workflows by selecting 10, 15, or 20 applications from the full suite of 20. The selection is done by using random sampling with replacement. For each generated workflow, we set the power and resilience constraints to be the maximum values of these metrics as observed across all the applications within the workflow at nominal frequency level. After that, we exhaustively search the space of feasible deadline constraints. We then report the best energy efficiency improvement over the baseline to demonstrate the maximum potential of the algorithm. As shown in Figure 2, LinOpt outperforms the baseline by around 15% for the majority of the 20-application workflows. In some less common cases, the improvement may go up to 25%. Comparing the three plots, the mass of the histogram is seen to shift to the right, suggesting longer workflows with more diverse behaviors will generally lead to better energy efficiency improvements from LinOpt.

IV. Graph Workflow Optimization

A. DAGopt

We propose an iterative optimization algorithm called DAGopt to statically optimize voltage-frequency settings of applications within a graph workflow. The basic idea of

Algorithm 1 Graph workflow optimization algorithm

function DAGopt(workflow, constraints)
  initialize $freq\_list$ by the lowest frequency level ($freq_{min}$)
  for all *app* in $workflow$ **do**
    update the execution time of *app* set by $freq_{min}$
  while not all applications processed **do**
    $C \leftarrow$ NextCriticalPath(workflow)
    $freq\_opt \leftarrow$ LinOpt(C, constraints, $freq\_list$)
    if LinOpt fails on path C **then**
      return Failure due to in-feasible constraints
    update $freq\_list$ with $freq\_opt$
    for all *app* in C **do**
      update the execution time of *app* according to $freq\_opt$
  return $freq\_list$
function NextCriticalPath(workflow)
  for all *app* that is not processed **do**
    $C_{to} \leftarrow$ *longest* path from source to *app*
    $C_{from} \leftarrow$ *longest* path from *app* to target
    $C \leftarrow C_{to} + C_{from}$
  return max C

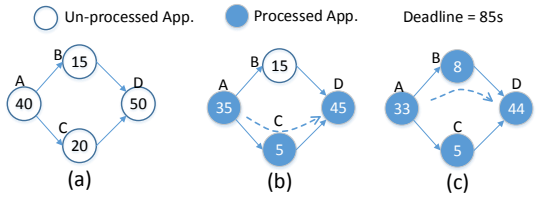

Fig. 3: DAGopt illustration on a synthetic workflow. Numbers in each node indicate the execution time of the corresponding application in seconds. Suppose that the deadline constraint is 85s. In (a), all applications are initialized with the lowest frequency levels. In (b), DAGopt identifies the critical path as $A \rightarrow C \rightarrow D$, then invokes LinOpt on the path. Upon a successful optimization, DAGopt sets optimized frequency levels as indicated by the new execution time, and marks all nodes within the path as *processed*. In (c), DAGopt identifies the *next_critical_path* as $A \rightarrow B \rightarrow D$ and invokes LinOpt on the path while only selecting higher frequency levels for A and D than their already chosen levels. DAGopt terminates after this step since all the applications within the workflow have been marked as processed.

DAGopt is to apply LinOpt iteratively on paths of a DAG until all applications within a graph workflow are optimized. For a given graph workflow, as shown in Algorithm 1, DAGopt initializes all applications with the lowest frequency level, and identifies the critical path as C: one that has the longest execution time. Then it applies LinOpt to C to choose the optimal frequency levels for each application within C. After a successful optimization of C, DAGopt updates the DAG with the execution time of all applications within C running at the optimized frequency level, and marks these applications as processed. In the next iteration, DAGopt identifies the *next_critical_path* as C^*, which is defined as the longest path that traverses at least one un-processed application. Then it applies LinOpt on C^* with the same deadline constraint of T, where frequency levels can only increase on any already-processed applications. As a result, any paths that share applications with C^* can only run faster after the current iteration. This restriction ensures that no paths processed in prior iterations would violate deadline constraints, while sacrificing energy efficiency due to the potential increase of frequency levels on already-processed applications. DAGopt keeps working on the *next_critical_path* until all applications have been marked as processed (therefore, no new *next_critical_path* can be found). The algorithm processes at least one application in each iteration so that it finishes in at most N-iterations, where N is the number of applications within a workflow. Figure 3 shows how the algorithm executes on an illustrative DAG.

B. DAGopt Evaluation

We first show a workflow illustrated in Figure 4, for which the DAGopt has a promising energy efficiency boost over the baseline. The workflow has two threads of linear workflows. Each linear workflow is composed of applications chosen from *2dconv*, *lbm*, and *bzip2*. We add inter-dependencies between the two threads, shown by the dashed arrows. Power and resilience constraints are set as the maximum power and maximum FIT estimates of all applications within a workflow running at nominal frequency. For demonstration purposes, we select 10 evenly-spaced sample points for deadline constraints from the range of critical path execution times, determined by running all applications within a workflow at the lowest and highest frequency levels. Results shown in Figure 5 suggest up to 35% energy efficiency improvement is achievable over the simple single-frequency selection heuristic. This is because, with the added inter-dependencies, application segments such as *bzip2* in the first thread can be slowed down considerably thanks to the dependence-imposed timing slack.

Finally, we carry out the energy efficiency distribution study that is similar to what we have shown in the previous section. We first randomly choose 14 applications to form two linear workflows, with seven applications per workflow. Then we randomly add a given number of inter-dependencies between the two linear workflows to finish a graph workflow. Using this approach, we generate 1,000 graph workflows, and compare DAGopt against the baseline which picks a single frequency for applications within the workflow. We use the same constraints that were just described. We choose the number of inter-dependencies to be 5, 10, and 15, and plot the histogram of maximum relative energy efficiency obtained by DAGopt over the baseline in Figure 6. It shows that with fewer inter-dependencies (e.g. 5), DAGopt is able to achieve an energy efficiency boost over the baseline by 10% to 20%. As the number of inter-dependencies increases to 15, the improvement in energy efficiency reduces to the range of 5% to 10%. This is because DAGopt becomes more limited in trade-off options among applications as the number of inter-dependencies increases.

V. Dynamic, Run-time Efficiency Optimization

A. Dynamic Iterative Approach

PEARL is also capable of emulating real systems by considering dynamic execution time variations. For each application within a workflow, we define its run-time variation factor (rvf) for a specific execution instance as the ratio of actual execution time to the pre-characterized value at the same frequency level. An rvf value greater than one indicates that the actual execution time of the corresponding application exceeds the expected pre-characterization value. In this case, there is a potential to miss the overall deadline for a workflow. On the other hand, an rvf value less than one indicates that the actual execution time is less than the pre-characterized value. The resulting slack could be potentially reclaimed to improve overall energy efficiency.

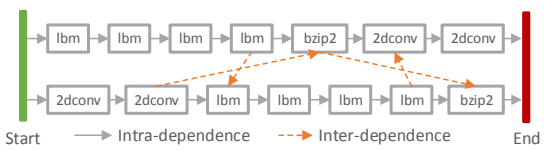

Fig. 4: An illustrative graph workflow, consisting of 2 threads with inter-dependencies indicated by dashed arrows.

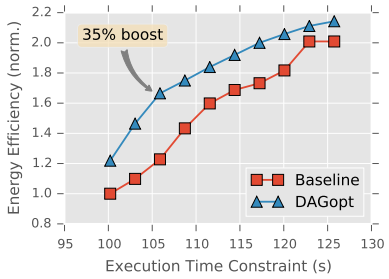

Fig. 5: DAGopt energy efficiency improvement over simple single-frequency selection heuristics, evaluated on an illustrative workflow (Figure 4). Power and resilience constraints are set as the maximum of applications within a workflow running at the nominal frequency level. Energy efficiency values, plotted on Y-axis, are normalized to the baseline under the execution time constraint of 100s (the leftmost data point). DAGopt outperforms the baseline in all cases, with up to 35% boost when the execution time constraint is 106s.

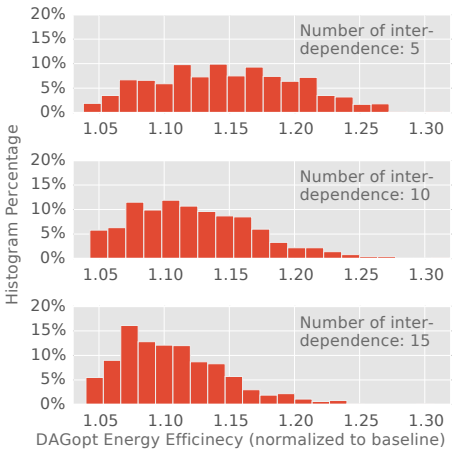

Fig. 6: Histogram of the maximum relative BIPS/W obtained by DAGopt over the baseline across 1,000 generated workflows. The number of inter-dependencies is varied from 5 to 15. Power and resilience constraints are set as the maximum of these metric values across applications within a workflow running at the nominal frequency level. The X-axis is the relative energy efficiency normalized to the baseline, Y-axis is the histogram distribution percentage. It shows that with fewer inter-dependencies, DAGopt is good to achieve an energy efficiency boost over the baseline by 10% to 20%, while as the number of inter-dependencies increases to 15, the improvement in energy efficiency reduces to the range of 5% to 10%.

The static optimization algorithm on a linear workflow tends to choose lower frequency settings to maximize energy efficiency. As a result, the total execution time of a given solution is usually close or equal to the deadline constraint. Therefore, such a static algrotihm's success becomes vulnerable to dynamic execution time variations, where the rvf value of at least one application in the workflow is greater than 1. With rvf following the Gaussian distribution of $\mathcal{N}(1, 0.1)$, frequency settings given by LinOpt miss deadline constraints in more than 50% of trials. Although it is possible to invoke LinOpt with a stricter deadline constraint than the original to tolerate execution time variations, it reduces the achieved energy efficiency significantly. For example, with the same rvf distribution, it reduces the average energy efficiency by 20% when statically invoking LinOpt with only 90% of the original deadline constraint.

In this section, we present an iterative dynamic algorithm (Figure 7a) to achieve a high level of confidence in meeting deadline constraints without significantly compromising the energy efficiency. The algorithm works as follows:

1) Before starting application A_i, we invoke LinOpt with an adjusted deadline by applying a scaling factor (t_f) to the original deadline constraint. Then, we launch A_i with the frequency level picked by LinOpt.

2) At the end of A_i, we observe that the actual elapsed execution time for A_i is t_i. We incorporate this slack (positive or negative) and update the deadline constraint for the residual workflow by deducting t_i from the residual deadline. If it is less than zero, the algorithm terminates and reports "in-feasible" as the deadline constraint is not met.
3) We go back to step 1) and iterate until the workflow is fully processed.

The algorithm is visualized step by step in Figure 7(b) using an illustrative workflow composed of *2dconv*, *dwt53*, and *histo*.

It is important to choose appropriate values for the deadline scaling factor t_f. When t_f equals 1, it invokes LinOpt with the original deadline constraint. In this case, the algorithm achieves maximized energy efficiency, while it is vulnerable to any applications that run slower than their pre-characterized latency. On the other hand, when t_f is less than 1, the algorithm invokes LinOpt with a more stringent deadline constraint. In this case, the algorithm creates slack that tolerates runtime delays while sacrificing energy efficiency. Note that the value of t_f can be larger than 1 in speculating that most applications finish earlier than their pre-characterized latency. However, we will not discuss this scenario in this paper.

We choose t_f based on the observation that delays in early phases of a workflow are more easily compensated for, while delays in later phases of a workflow are more dangerous because there is less opportunity to compensate with few remaining residual applications. Based on this observation, our algorithm exploits the most aggressive $\max(t_f)$ at the beginning for the first application to maximize potential energy efficiency gains, then gradually lowers t_f to the most conservative $\min(t_f)$ at the end for the final applications to stay within the deadline constraint. The pair of parameters $(\max(t_f), \min(t_f))$ enables the trade-off between energy efficiency and the level of confidence in meeting the deadline constraints. We choose a value of 1 for $\max(t_f)$, speculating that few applications finish earlier. We evaluate alternate $\min(t_f)$ values in the next subsection with regard to distributions of rvfs.

B. Evaluation

To study the impact of run-time variations, we use an illustrative workflow by stitching together four copies of the *lbm* application, two copies of *2dconv*, and one copy of *bzip2*, and replicate the 7-application sequence three times. The resulting workflow has 21 applications. For illustration purposes, we use Monte-Carlo simulation method to generate rvf sequences by following a Gaussian distribution for the workflow. For each rvf distribution, we sample 1,000 rvf sequence samples. We study three alternative distributions to cover various dynamic execution scenarios: a)*VarL* with $N(1, 0.1)$, which reflects the scenario that the system experiences significant, intermittent changes in operating environment; b)*VarS* with $N(1, 0.03)$, which reflects a scenario similar to *VarL* except that only moderate changes in operating environment are observed; c)*Delay* with $N(1.05, 0.02)$, which reflects the scenario in which steady deviations from the nominal operating environment lead to persistent delays for all applications (as indicated by the mean rvf that is larger than 1). We study the dynamic iterative approach with three values for parameter $\min(t_f)$: 0.9, 0.85, and 0.8. The three cases are denoted as DYN_0.9, DYN_0.85, and DYN_0.8, respectively. We compare the three dynamic approaches against the static LinOpt (denoted as ST_1). We include an alternate static case which targets 90% of the original deadline constraint (denoted as ST_0.9). Note that rvf distributions are chosen to illustrate the capabilities of PEARL. PEARL accepts distributions chosen by users.

As shown in Figure 8, static approaches are either too conservative, yielding lower energy efficiency (in the case of ST_0.9), or too vulnerable to dynamic execution time variations, yielding higher deadline miss ratio (in the case of ST_1). Dynamic approaches, on the other hand, achieve energy efficiency that is close to static LinOpt, which targets the original deadline constraint. Within dynamic approaches, the energy efficiency increases in a limited manner as $\min(t_f)$

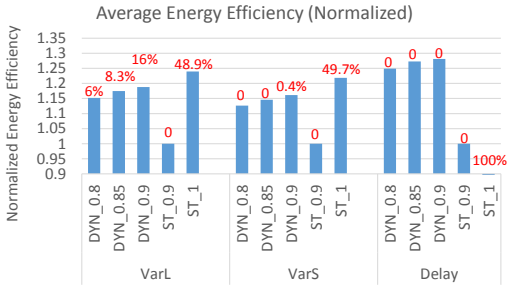

Fig. 8: Dynamic simulation results. For the described scenarios (*VarL*, *VarS*, and *Delay*), we compare the dynamic iterative approach with 3 values of $\min(t_f)$ (DYN_0.9, DYN_0.85, and DYN_0.8), and two two cases of static LinOpt with (ST_0.9) and without (ST_1) conservative provision on deadline constraints. The Y-axis is the average energy efficiency computed in BIPS/W normalized to ST_0.9. The percentage on top of a bar is its deadline miss ratio. Dynamic approaches generally achieve energy efficiency close to the aggressive static approach (ST_1), while maintaining miss ratios close to the conservative static approach (ST_0.9).

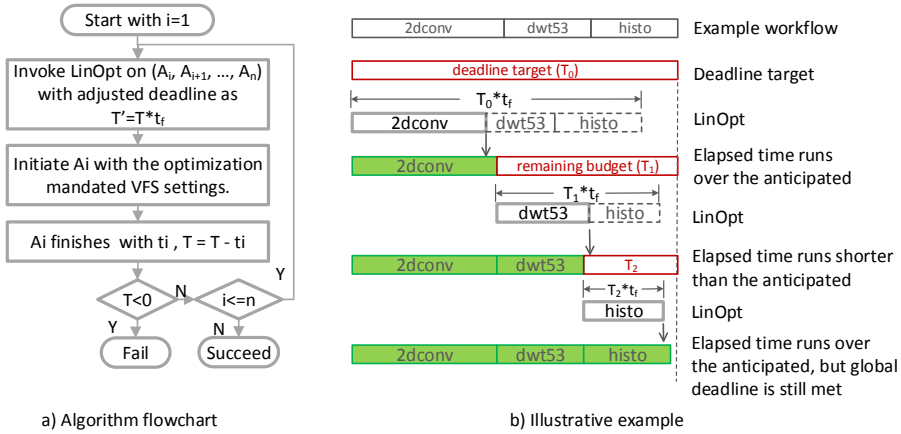

Fig. 7: An iterative algorithm adapting to dynamic execution time variations. In (a), the algorithm is presented as a flowchart, and in (b), a step-by-step example illustrates the dynamic optimization algorithm. The illustrative linear workflow is a stitched sequence of three applications: *2dconv*, *dwt53*, and *histo*.

increases. In the cases of *VarS* and *Delay* where distribution variations of rvf are small, dynamic approaches do not miss any deadlines, in any experimental trials, which is as good as the conservative static approach (ST_0.9). In the case of *VarL*, dynamic approaches with higher $\min(t_f)$ (e.g. DYN_0.9) suffer from a higher deadline miss ratio. The latter is more than double the miss ratio observed in dynamic approaches with lower $\min(t_f)$ (e.g. DYN_0.8), despite a modest energy efficiency improvement. Overall, the dynamic approach prefers a high $\min(t_f)$ when the variation of rvf is small, and a low $\min(t_f)$ is better when the variation of rvf is large.

VI. Related Work

Dynamic voltage and frequency scaling (DVFS) has been widely used for managing workload-driven power in a processor or system context. For example, Isci et al. [3] propose multi-core DVFS algorithms with the objective of maximizing chip throughput for a given power budget. In the real-time embedded systems domain, early work by Pillai et al. [6] and more recent work by Devadas et al. [2] and Qi et al. [7] are a few representative studies from a large body of work to meet real-time deadlines while minimizing power. Scheduling of tasks in the form of DAG has also been widely studied, for example, in Shang et al. [12] and Kanoun et al. [4]. However, none of them considered reliability as a system-level design constraint.

Combining reliability considerations with energy savings in a unified dynamic resource management framework is a topic area that has not been explored as widely as dynamic power management alone. Zhang et al. [15] address a reliability-aware power management problem, but in this case the reliability focus is only on checkpoint-restart based systems. Similar to our work, Zhao et al. [16] address the mutually opposing issues of energy efficiency and transient error probability (with DVS or DVFS control). However, Zhao et al. use a set voltage-dependence equation to model system resilience, without factoring in the application-level masking effects (as in our work). EPROF [14] exploits Integer Linear Programming (ILP) to optimize the trade-off among energy, performance, and reliability. However, the trade-off in EPROF targets heterogeneous multi-core systems composed of reliable cores consuming high power, and unreliable cores consuming low power. Performance, and reliability are modeled as intrinsic properties of hardware using proxies such as clock frequency and soft error rate. Our work relies on voltage-frequency scaling as the primary energy saving technique, and takes advantage of application-specific models for power and reliability. Finally, our work presents an approach to dynamic adaptation that accounts for run-time variations, which is not covered by EPROF. As in our static optimization work, the authors of [11] propose a similar linear-optimization based approach, while emphasizing the impact of application level correctness to the system resilience. However, in their work, the method of characterizing the application level resilience is SoC-specific, and does not apply to general-purpose processors, as studied in our work. In [13], Wang et al. present the idea to exploit ILP for linear workflow optimization. Our work extends this to more complex graph workflows, and provides a better dynamic approach that can achieve higher energy efficiency while still maintaining a high level of confidence in meeting deadline constraints.

VII. Conclusions and Future Work

Future ultra-efficient embedded systems with mission-critical resilience requirements in the deep-submicron design era will require a careful balance between static preparation and run-time adaptation of applications. In this paper, we first describe a software framework called PEARL for application preparation and runtime steering in the context of diverse performance, power, and resilience constraints. Then we present experimental analysis for a diverse set of application sequences to demonstrate the functionality and capabilities of the PEARL framework. Our analysis shows up to 15% and 35% energy efficiency improvement over a simple baseline, for linear and graph workflows respectively. Moreover, our proposed dynamic iterative approach achieves up to 25% improvement in energy efficiency over a conservative static approach, while maintaining the same level of confidence in meeting deadline constraints.

VIII. Acknowledgement

This work is sponsored by Defense Advanced Research Projects Agency, Microsystems Technology Office (MTO), under contract no. HR0011-13-C-0022. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government. This document is Approved for Public Release, Distribution Unlimited.

References

[1] "DARPA PERFECT Program," http://www.darpa.mil/Our_Work/MTO/Programs/Power_Efficiency_Revolution_for_Embedded_Computing_Technologies_(PERFECT).aspx.

[2] V. Devadas *et al.*, "Real-Time Dynamic Power Management Through Device Forbidden Regions," in *RTETAS*, 2008.

[3] C. Isci *et al.*, "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget," in *MICRO*, 2006.

[4] K. Kanoun *et al.*, "Onling Energy-Efficient Task-Graph Scheduling for Multicore Platforms," *TCADICS*, 2014.

[5] P. Kudva *et al.*, "Fault Injection Verification of IBM POWER6 Soft Error Resilience," in *Proc. of the Workshop on Architectural Support for Gigascale Integration*, 2007.

[6] P. Pillai *et al.*, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," in *SOSP*, 2001.

[7] X. Qi *et al.*, "Power Management for Real-Time Embedded Systems on Block-Partitioned Multicore Platforms," in *ICESS*, 2008.

[8] J. A. Rivers *et al.*, "Phaser: Phased Methodology for Modeling the System-Level Effects of Soft Errors," *IBM Jour. of Res. and Dev.*, vol. 52, no. 3, pp. 293–306, 2008.

[9] P. N. Sanda *et al.*, "Soft-Error Resilience of the IBM POWER6 Processor." *IBM Jour. Res. and Dev.*, vol. 52, no. 3, pp. 275–284, 2008.

[10] N. Seifert *et al.*, "Soft Error Susceptibilities of 22nm Tri-Gate Devices," *IEEE Trans. on Nuclear Science*, vol. 59, no. 6, Dec 2012.

[11] R. A. Shafik *et al.*, "Soft Error-Aware Voltage Scaling Technique for Power Minimization in Application-Specific Multiprocessor System-on-Chip," *JOLPE*, vol. 5, no. 2, pp. 145–156, 2009.

[12] M. Shang *et al.*, "An Efficient Parallel Scheduling Algorithm of Dependent Task Graphs," in *PDCAT*, 2003.

[13] L. Wang *et al.*, "Resilience and Real-Time Constrained Energy Optimization in Embedded Processor Systems,," in *SELSE*, 2014.

[14] Y. Yetim *et al.*, "EPROF: An Energy/Performance/Reliability Optimization Framework for Streaming Applications," in *ASPDAC*, 2012.

[15] Y. Zhang *et al.*, "A Unified Approach for Fault Tolerance and Dynamic Power Management in Fixed-Priority Real-Time Embedded Systems," *TCADICS*, vol. 25, no. 1, pp. 111–125, 2006.

[16] B. Zhao *et al.*, "Reliability-aware Dynamic Voltage Scaling for Energy-constrained Real-time Embedded Systems," in *ICCD*, 2008.