The EH Model: Analytical Exploration of Energy-Harvesting Architectures

Joshua San Miguel  Karthik Ganesan  Mario Badr  Natalie Enright Jerger

Abstract—Energy-harvesting devices—which operate solely on energy collected from their environment—have brought forth a new paradigm of intermittent computing. These devices succumb to frequent power outages that would cause conventional systems to be stuck in a perpetual loop of restarting computation and never making progress. Ensuring forward progress in an intermittent execution model is difficult and requires saving state in non-volatile memory. In this work, we propose the EH model to explore the trade-offs associated with backing up data to maximize forward progress. In particular, we focus on the relationship between energy and forward progress and how they are impacted by backups/Restores to derive insights for programmers and architects.

Index Terms—energy-harvesting, intermittent computing, analytical model

1 INTRODUCTION

The batteryless operation of compute devices introduces new and challenging trade-offs to programmers and computer architects. A key challenge is that ambient energy sources (e.g., photovoltaic, thermal, RF [11], WiFi [2]) do not provide a constant stream of power to run the device [3]. Also, they typically provide less average power than the device requires. To overcome this, a common approach is to first store energy in a capacitor, the energy supply, which then powers the device for a period of time [3]. However, as a result of this sporadic power supply, relying on ambient energy sources requires an execution model that must inherently support intermittent computation: computation may stop at any point in the application because the energy supply has depleted and cannot resume until sufficient energy has been harvested from the environment.

Intermittent computation requires that application state be backed up in non-volatile memory before energy is depleted and restored when energy is available again. There are many approaches to the backup process that differ in both when to save state and how much state to store [1], [6], [9], [10]: we generalize the concept of a backup to be any point at which no preceding instructions need to be re-executed. Ideally, an architecture would maximize energy usage and allow an application to make consistent forward progress [7]. We define forward progress as the energy spent on useful work (in contrast to energy spent on overheads such as saving or restoring state). For example, a common energy-harvesting application gathers sensor data from the environment and calculates statistics (e.g., mean, median) for logging purposes [4], [10]; in this case, forward progress is the energy spent on processing the sensor data to generate the needed statistics for logging. Another common application is activity recognition where data is processed from accelerometers to determine the user’s activity (i.e., walking, running, lying down) [4]. In such an application, forward progress is the energy spent processing the accelerometer data to determine the activity.

The authors are with the Edward S. Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto. (joshuaesanmiguel, karthik.ganesan, mario.badr)@mail.utoronto.ca, enright@ee.ece.utoronto.ca

2 THE EH MODEL

The goal of our model is to provide an accurate estimate of an application’s ability to make forward progress on a given energy-harvesting architecture. Figure 1 represents the abstract energy-harvesting device that we target, which can encompass a wide range of CPU designs. Within the context of intermittent computing, we divide application runtime into two phases: active and charging. During charging, no forward progress is made until the energy supply is filled to capacity. During the active period, the energy supply is expended on executing instructions and performing backups and restores. Forward progress is only made when the results of executed instructions are saved to non-volatile memory (intermediate results in volatile memory are lost during a power outage). Expending energy on instructions whose output is not saved before a power outage is wasteful—we call this dead energy. Our analytical model is designed to measure forward progress within an active period.¹

We list our model parameters in Table 1. The parameters can be characterized as distributions—in this work, we assume the average unless otherwise specified. The output of our model is , an estimate of forward progress represented as the fraction of the energy supply spent on useful work (i.e., not spent on backups, restores and dead execution). To begin, we characterize the fundamental factors of energy :

\[ E = (e_P + n_B \cdot e_B + e_D + e_R) + e_C = 0 \]  (1)

- \( e_P \) is energy spent on forward progress.
- \( e_B \) is energy spent on each of \( n_B \) backups.
- \( e_D \) is energy spent on dead execution.

¹ We do not consider the charging period—which can vary based on the ambient power source—a limitation of our model that we plan to address in future work.
We consider the average case of dead cycles: while executing on an architecture (which depends on the application’s write footprint). We calculate (e.g., dirty data in a volatile cache must be saved, the amount of 
period [1], [6], [9]; others only once [10]. We can characterize how progress. 
energy expended on backups as: 
the total energy spent on forward progress is computed 
that governs the rate at which an application moves forward. 
non-volatile memory (σ). From this, we compute the restore energy as: 
Often the energy budget is not fixed at E but rather increases over time since the device can continue charging during an active period. We model the charging energy (eC, Equation 8) as the device charging rate (eC) multiplied by the number of cycles in an active period (second term in Equation 8). The time spent per backup and restore scales inversely with the non-volatile memory bandwidth (σ), typically 1 byte per cycle on the MSP430 CPU commonly used in energy-harvesting systems [1], [4], [10]. 
Putting It All Together. Our model outputs the percentage of energy spent on forward progress p as e · τP/E. Solving for p in Equation 1 yields: 
In the first term of the denominator, forward progress is scaled down by the ratio of backup overhead (ΩB) to how much useful work is committed on each backup (e - eC · τB). This represents the backup cost-reward ratio that governs the rate at which an application moves forward. The second term represents additional progress made due to charging in the active period. Note that the charging rate is generally much lower than the consumption rate; progress goes to infinity when eC approaches e. In the numerator, progress is limited by one-time costs—dead energy (second term) and restore energy (third term)—which are incurred once per active period. Thus, even if the cost of backups were to be completely eliminated in the denominator, these one-time costs impose an upper bound on performance and must be minimized. 
There are many interesting implications that can be garnered from Equation 9. The remainder of this section explores a few of them in detail and discusses the insights they reveal. Unless otherwise stated, we ignore restore overhead and charging energy and set the execution energy (e) to 1% of the active period’s energy supply (E) for illustrative purposes, focusing on general trends as opposed to the exact values.

### 2.1 Exploration: Optimal Time Between Backups

How many cycles apart should backups be (τB) to maximize forward progress? Figure 2 shows how progress (normalized to e) varies with the time between backups (τB) and backup cost (ΩB). The first takeaway is that reducing backup cost is
always better for performance, as expected. As the backup cost approaches 0, forward progress favours more frequent backups since 1) this minimizes dead cycles, and 2) backups are cheap.

The second takeaway is that the optimal time between backups is not stagnant but rather varies depending on the backup cost. Solving for the roots of \( \frac{\partial p}{\partial \tau_B} \) (Equation 9), we obtain the optimal:

\[
\tau_{B,\text{opt}} = \frac{\Omega_B \cdot A_B}{\Omega_B \cdot \alpha_B + \epsilon} \left( \sqrt{\frac{2 \cdot E}{\epsilon} \cdot \frac{\Omega_B \cdot \alpha_B + \epsilon}{\Omega_B \cdot A_B} + 1} - 1 \right)
\]  

(10)

The optimal number of cycles between backups is dictated by the ratio \( \frac{\Omega_B \cdot A_B}{\Omega_B \cdot \alpha_B + \epsilon} \). The numerator represents the compulsory energy cost per backup while the denominator represents the energy cost proportional to how much work was done since the last backup. There is a trade-off between 1) backing up less frequently if the compulsory cost is high; and 2) backing up more frequently if the proportional cost is high. With Equation 10, programmers can estimate the optimal task length for their code (e.g., in systems like CHAIN [4], a task can be sized to match the optimal backup time), while system designers can configure the optimal period for checkpoints [10] and watchdog timers (e.g., in Clank [6], by choosing the appropriate duration between watchdog interrupts).

Note that in cases with only application state (\( \alpha_B \)) and no architectural state (\( A_B \)), there is no optimal time between backups. Removing the cost of backing up architectural state (\( A_B = 0 \)) in Equation 9 leaves us with a simple relationship: \( \lim_{\tau_B \to 0} p = 1 \). As a result, when considering only the backup cost of application state, it is always better to back up as frequently as possible since the overhead decreases proportionally with the time between backups (Equation 4).

### 2.2 Exploration: Cost of Backups and Restores

Given the number of cycles between backups, should we focus on minimizing backup or restore overhead? The decision arises in situations where 1) an architect must optimize for the average task length [4], or 2) a programmer must optimize for periodic backups imposed by the system (e.g., watchdog timers [6]). At first glance, one may assume that it is always better to optimize backup cost since restores are performed only once per active period. But if the time between backups is too great, there may be insufficient energy to backup resulting in no progress. In this section, we explore the interplay between the time between backups and the backup/restore overhead.

As the time between backups becomes significantly large (\( \tau_B \) approaches \( \infty \)), the performance improvement of reducing the restore cost (\( \frac{\partial p}{\partial \tau_R} \)) outweighs that of the backup cost (\( \frac{\partial p}{\partial \tau_B} \)). Since progress is inversely proportional to backup and restore overhead, both partial derivatives are negative (i.e., a lower \( \frac{\partial p}{\partial \tau_R} \) means that reducing \( \epsilon_R \) yields better performance). Investigating further, we solve for the number of cycles between backups at the break-even point (\( \tau_{B,\text{be}} = \frac{\epsilon_R}{\epsilon} \cdot \frac{\Omega_B - A_B}{E} \))

\[
\tau_{B,\text{be}} = \frac{2}{3} \cdot \frac{E - \epsilon_B - \epsilon_R}{\epsilon}
\]

(11)

From this, we have the following takeaways:

- If the time between backups is less than the break-even point (\( \tau_B < \tau_{B,\text{be}} \)), reduce the cost of backups.
- If the time between backups is greater than the break-even point (\( \tau_B > \tau_{B,\text{be}} \)), reduce the cost of restores.

As expected, with very frequent backups (i.e., low \( \tau_B \)), architects should focus on optimizing the backup mechanisms to improve performance. For example, a non-volatile processor designer can choose to discard the state of some structures (e.g., instruction fetch queue, branch predictor) if they expect to back up often [9]. Conversely, as the time between backups increases, the restore overhead starts to dominate; it becomes more likely that no backup is invoked at all within an active period. When this happens, all execution is dead. As a result, we actually start to see more restore invocations than backup invocations when the time between backups exceeds the break-even point (\( \tau_B > \tau_{B,\text{be}} \)). With this analysis, architects and programmers gain a better understanding of where to focus their efforts to optimize for the expected time between backups in their systems. For instance, in Clank [6], checkpoints occur due to idempotent violations and watchdog timers. Based on the observed frequency of these checkpoints, the break-even point can inform the Clank architect to optimize the restore or backup overhead.

### 2.3 Exploration: Variability of Dead Cycles

So far our analysis assumes the average \( \tau_D \) from Equation 5. However, due to non-determinism in both the architecture (e.g., fluctuations in energy source) and the application (e.g., input-dependent program behaviour) [5], the number of dead cycles can vary dramatically across active periods. In this section, we explore how this variability affects important design decisions.

Figure 3 shows how progress varies under the worst-case (\( \tau_D = \tau_{D,\text{max}} \)) and best-case dead cycles (\( \tau_D = 0 \)). The first takeaway is that variability diminishes as the time between backups approaches 0. This is expected since backing up more often decreases the likelihood of dead execution. Conversely, a long time between backups increases the risk of not backing up at all but opens up the possibility of the ideal scenario (i.e., a single backup invoked at the end of the backup period). This introduces an interesting trade-off. If aggressive performance gains are desired, an architect can design a system with a long time between backups that are scheduled in an intelligent or speculative way to consistently minimizes dead cycles. But if worst-case performance (i.e., tail latency) is important, an...
The architect can sacrifice the average case and opt for a much lower $\tau_B$. For example, Spendthrift [8] uses voltage and frequency scaling to maximize energy efficiency. Our work provides an upper bound on forward progress for Spendthrift and related work that try to minimize dead cycles.

Following from this, how many cycles apart should backups be invoked to maximize worst-case forward progress? At first glance, one may assume that Equation 10 is sufficient, however it was solved for the average case of dead cycles. Solving instead for the worst case ($\tau_B = \tau_{B,opt}$):

$$
\tau_{B,opt}(wc) = \frac{\Omega_B \cdot A_B}{\Omega_B \cdot \alpha_B + \epsilon} \cdot \left( \sqrt{\frac{E}{\epsilon} \cdot \frac{\Omega_B \cdot \alpha_B + \epsilon}{\Omega_B \cdot A_B} + 1} - 1 \right) \quad (12)
$$

Though similar to the average case in Equation 10, the key takeaway is that $\tau_{B,opt}(wc)$ and $\tau_{B,opt}$ are not equal. The optimal time between backups in the worst case is always less than that of the average case, an important consideration when designing for tail latency.

2.4 Summary

We present the EH model formulation and its implications for designing efficient energy-harvesting systems. We explore the optimal time between backups (Section 2.1) and how it can help 1) programmers to determine the granularity and size of tasks [10] and 2) architects to configure optimal watchdog timers [6]. We also provide architects with guidelines on when to optimize backup or restore overhead (Section 2.2). The model is capable of providing a lower and upper bound on performance when accounting for the non-determinism of intermittent execution (Section 2.3), useful for works that seek aggressive performance gains [8]. In the next section, we validate our model on real hardware.

3 Validation

To validate EH, we experimentally measure $\epsilon$ on an ARM Cortex-M3 50 MHz processor. We observe that the energy cost per cycle is fixed ($\epsilon = 1.18$ mJ) for all tested instructions (NOP, AND, OR, ADD, SUB, MUL, LW, SW). We configure backups with $\Omega_B = \epsilon$ in our experiments. We employ interrupters to mimic the periodic backups required, setting $\alpha_B = 0.1$ and $A_B = 50$ to account for the overhead of entering and exiting the interrupt routine. As in Sections 2.1 and 2.3, we keep the restore cost as zero but note that it is straightforward to add to our analysis.

For our experiments, we run an application that calculates the mean and variance of 64 data points, a common task in energy-harvesting systems. We emulate the average 100 ms active period commonly found in prior work [6], [10] by modelling the energy supply as a normal distribution with a mean $\mu$ of 5000. We measure the forward progress of our application as we sweep $\tau_B$ from 200 to 5000 cycles (10 runs each), which are the minimum and maximum in our system, respectively. Figure 4 shows that our experimental results match closely with the values predicted by our model. As we sweep $\tau_B$ to larger values, we see a greater variance in the dead cycles (Section 2.3), as they can now occur over a larger timespan of $\tau_B$.

4 Related Work

To the best of our knowledge, this is the first model that explores the forward progress made by energy-harvesting devices. To date, evaluations of different devices has been done in simulation [1], [6], [9]. Through simulation, Clank identified that re-execution cost (i.e., dead cycles) can outweigh the cost of checkpointing [6]. The EH model allows similar observations to be made without the implementation details of a backup/restore mechanism, revealing insights from simple formulas.

The cost of checkpointing has been explored. Chain advocates that checkpointing is too expensive in an energy-harvesting environment and proposes a new programming model to avoid it [4]. Ekho proposes an I–V curve to capture how current changes when checkpointing with a variable energy supply [5]. Our EH model is more architecture centric, focusing on the active period and requiring the energy supply to be fully charged before execution resumes. Future work could look at how to model architectures that resume execution before the energy supply has fully charged.

5 Conclusion

We propose the EH model, a step towards better understanding the unique implications and complex interactions that arise in energy-harvesting systems. Our analysis sheds new insights and trade-offs that we hope not only assist but also excite research in this field.

References


Figure 4. Comparison of measured data to the trend predicted by EH.