The EH Model: Analytical Exploration of Energy-Harvesting Architectures

Joshua San Miguel, Karthik Ganesan, Mario Badr, and 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, Toronto ON M5S, Canada. E-mail: {joshua.sanmiguel, karthik.ganesan, mario.badr}@mail.utoronto.ca, enright@ece.utoronto.ca.*

Manuscript received 28 Sept. 2017; revised 2 Nov. 2017; accepted 19 Nov. 2017. Date of publication 26 Nov. 2017; date of current version 19 Mar. 2018.
(Corresponding author: Joshua San Miguel.)
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/LCA.2017.2777834

Easily estimating the energy spent towards making forward progress is no simple task; doing so—whether as an architect or a programmer—requires a solid understanding of the interplay between backup/restore overheads, re-executions and actual useful progress. This has led researchers to propose a wide range of processor designs ranging from simple in-order cores [1] to complex out-of-order cores [9] with a diverse mix of memory technologies (e.g., flash [10] and FRAM [4]). Despite these proposals, our community lacks consensus on the best practices for building efficient energy-harvesting systems and is in need of a general tool to better understand and explore this space.

We present the *EH model*, an analytical model that explores the complex trade-offs that arise in energy-harvesting systems. Our work targets architects; with the application's behaviour known a priori (e.g., average task length, frequency of non-volatile writes), our model can estimate the impact that architectural parameters will have on forward progress. Our work also provides insights to programmers seeking to maximize the performance of their programs on given energy-harvesting architectures. We validate our model and show that EH captures the trends of real hardware accurately.

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. Fig. 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 p, an estimate of forward progress represented as the fraction of the energy supply E expended on useful work (i.e., not spent on backups, restores and dead execution). To begin, we characterize the fundamental factors of energy E:

$$E - (e_P + n_B \cdot e_B + e_D + e_R) + e_C = 0 \tag{1}$$

- 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.
- e_R is energy spent on restoring backed-up state.
- e_C is energy gained from charging during the active period.

An application will expend some amount of energy per cycle while executing on an architecture (ϵ). During an active period, a certain number of cycles will be used to make forward progress (τ_P); the total energy spent on forward progress is computed via Equation 2. Note that the execution energy cost ϵ includes not only the processor but also any sensors and peripherals that are active.

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.

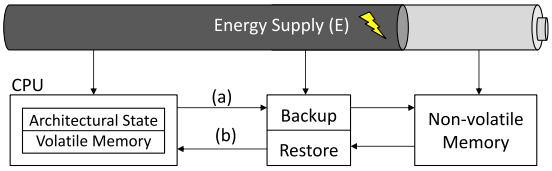

Fig. 1. An abstract energy-harvesting device.

Techniques that reduce ϵ (e.g., duty cycling sensors, dynamic voltage scaling [8]) are always beneficial for forward progress,

$$e_P = \epsilon \cdot \tau_P \tag{2}$$

During active periods, computation must be backed up to non-volatile memory (Fig. 1a) to make forward progress. Some approaches may back up multiple times within an active period [1], [6], [9]; others only once [10]. We can characterize how often they occur with the number of cycles between backups (τ_B). The total number of backups is then:

$$n_B = \frac{\tau_P}{\tau_B} \tag{3}$$

The energy spent on each backup (e_B) is dictated by the cost of saving the current state to non-volatile memory (Ω_B) scaled by the number of bytes written per backup. A system may choose to back up a fixed number of bytes each time (A_B), such as architectural state (e.g., program counter, registers) [9]. A system may also incur a variable backup cost that is proportional to any changes in application state (α_B) since the last backup (e.g., dirty data in a volatile cache must be saved, the amount of which depends on the application's write footprint). We calculate energy expended on backups as,

$$e_B = \Omega_B \cdot (A_B + \alpha_B \cdot \tau_B) \tag{4}$$

Ideally, we would back up our data just before our energy supply (E) is depleted. If this is not the case, then the application will have expended energy on computations that were never stored in non-volatile memory. Thus, some number of cycles before the next backup (τ_D) will contribute to dead energy (e_D). We consider the average case of dead cycles:[2]

$$0 \leq \tau_D \leq \tau_B, \qquad \tau_D = \tau_B/2, \text{ on average} \tag{5}$$

We calculate the amount of dead energy expended in a similar way to the energy spent on forward progress (Equation (2)). In this case, we replace τ_P with τ_D since application state after dead computation is not saved for the next restore:

$$e_D = \epsilon \cdot \tau_D \tag{6}$$

Before an application can resume execution, it must expend energy (e_R) to restore its last saved state (Fig. 1b). Mirroring the backup overhead (Equation (4)), this incurs the cost of accessing non-volatile memory (Ω_R) scaled by the amount of architectural (A_R) and application (α_R) state that must be restored. A_R represents a fixed number of bytes that need to be restored at the start of each active period (e.g., register file). α_R represents the variable cost of reverting or cleaning up any uncommitted state left over from the dead execution (τ_D) of the previous active period (e.g., flushing dead instructions in non-volatile processors [9]). From this, we compute the restore energy as,

$$e_R = \Omega_R \cdot (A_R + \alpha_R \cdot \tau_D) \tag{7}$$

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 (e_C, Equation (8)) as the device charging rate (ϵ_C) 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]

$$e_C = \epsilon_C \cdot \left(\tau_P + \frac{n_B \cdot (A_B + \alpha_B \cdot \tau_B)}{\sigma} + \tau_D + \frac{(A_R + \alpha_R \cdot \tau_D)}{\sigma}\right) \tag{8}$$

Putting It All Together. Our model outputs the percentage of energy spent on forward progress p as $\epsilon \cdot \tau_P / E$. Solving for p in Equation (1) yields:

$$p = \frac{1 - \frac{(\epsilon - \epsilon_C)\cdot\tau_B/2}{E} - \frac{(\Omega_R - \epsilon_C/\sigma)\cdot(A_R + \alpha_R\cdot\tau_B/2)}{E}}{(1 + \frac{(\Omega_B - \epsilon_C/\sigma)\cdot(A_B+\alpha_B\cdot\tau_B)}{(\epsilon-\epsilon_C)\cdot\tau_B}) \cdot (1 - \frac{\epsilon_C}{\epsilon})} \tag{9}$$

In the first term of the denominator, forward progress is scaled down by the ratio of backup overhead ($(\Omega_B - \epsilon_C/\sigma) \cdot (A_B + \alpha_B \cdot \tau_B)$) to how much useful work is committed on each backup ($(\epsilon - \epsilon_C) \cdot \tau_B$). 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 p goes to ∞ when ϵ_C approaches ϵ. 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 (ϵ) to 1 percent of the active period's energy supply (E) for illustrative purposes, focusing on general trends as opposed to the exact values.

TABLE 1
EH Model Parameters and Outputs

Parameter	*Units*	*Description*
$E \in \mathbb{R}_{>0}$	joules	energy supply per active period
$\epsilon \in \mathbb{R}_{>0}$	joules/cycle	execution energy per cycle
$\epsilon_C \in \mathbb{R}_{\geq 0}$	joules/cycle	charging energy per cycle
$\sigma \in \mathbb{R}_{>0}$	bytes/cycle	non-volatile memory bandwidth
$\Omega_B \in \mathbb{R}_{\geq 0}$	joules/byte	backup cost
$\Omega_R \in \mathbb{R}_{\geq 0}$	joules/byte	restore cost
$A_B \in \mathbb{R}_{\geq 0}$	bytes	architectural state per backup
$A_R \in \mathbb{R}_{\geq 0}$	bytes	architectural state per restore
$\alpha_B \in \mathbb{R}_{\geq 0}$	bytes/cycle	application state per backup
$\alpha_R \in \mathbb{R}_{\geq 0}$	bytes/cycle	application state per restore
$\tau_B \in \mathbb{R}_{>0}$	cycles	time between backups
$\tau_P \in \mathbb{R}_{\geq 0}$	cycles	time spent on forward progress
$p = \epsilon \cdot \tau_P / E$	% of E	% energy spent on forward progress

2.1 Exploration: Optimal Time Between Backups

How many cycles apart should backups be (τ_B) to maximize forward progress? Fig. 2 shows how progress (normalized to ϵ) varies with the time between backups (τ_B) and backup cost (Ω_B). 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:

2. We discuss the impact of variability in Section 2.3.

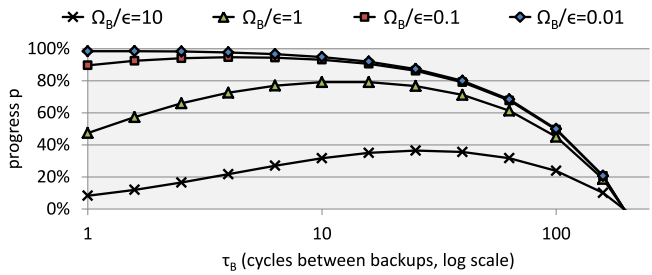

Fig. 2. Progress p with varying τ_B and backup cost Ω_B (normalized to ϵ). Assumes $E = 100$, $e_C = 0$, $A_B = \epsilon = 1$, $\alpha_B = 0.1$ and $\Omega_R = 0$.

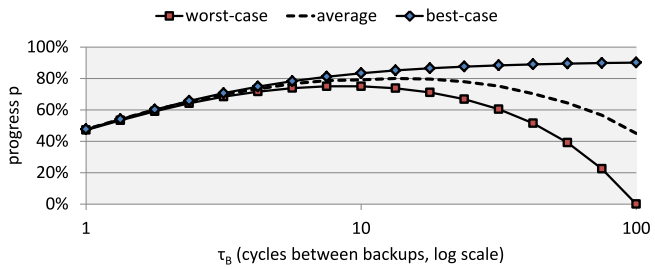

Fig. 3. Progress p over τ_B, varying from worst-case to best-case τ_D. Assumes $E = 100$, $e_C = 0$, $\Omega_B = A_B = \epsilon = 1$, $\alpha_B = 0.1$ and $\Omega_R = 0$.

$$\tau_{B,opt} = \frac{\Omega_B \cdot A_B}{\Omega_B \cdot \alpha_B + \epsilon} \cdot \left(\sqrt{2 \cdot \frac{E}{\epsilon} \cdot \frac{\Omega_B \cdot \alpha_B + \epsilon}{\Omega_B \cdot A_B} + 1} - 1 \right) \tag{10}$$

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 (α_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 (τ_B approaches $+\infty$), the performance improvement of reducing the restore cost ($\frac{\partial p}{\partial e_R}$) outweighs that of the backup cost ($\frac{\partial p}{\partial e_B}$). Since progress is inversely proportional to backup and restore overhead, both partial derivatives are negative (i.e., a lower $\frac{\partial p}{\partial e_R}$ means that reducing e_R yields better performance). Investigating further, we solve for the number of cycles between backups at the break-even point ($\frac{\partial p}{\partial e_B} = \frac{\partial p}{\partial e_R}$):

$$\tau_{B,be} = \frac{2}{3} \cdot \frac{E - e_B - e_R}{\epsilon} \tag{11}$$

From this, we have the following takeaways:

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

As expected, with very frequent backups (i.e., low τ_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 ($\tau_{B,be}$). 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 τ_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.

Fig. 3 shows how progress varies under the worst-case ($\tau_D = \tau_B$) 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 architect can sacrifice the average case and opt for a much lower τ_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 ($\tau_D = \tau_B$):

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

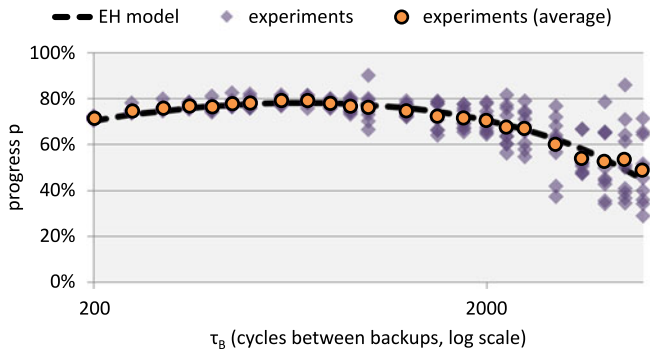

Fig. 4. Comparison of measured data to the trend predicted by EH.

Though similar to the average case in Equation (10), the key takeaway is that $\tau_{B,opt(wc)}$ and $\tau_{B,opt}$ are never 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 ϵ 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 interrupts 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 $\frac{E}{\epsilon}$ of 5000. We measure the forward progress of our application as we sweep τ_B from 200 to 5000 cycles (10 runs each), which are the minimum and maximum in our system, respectively. Fig. 4 shows that our experimental results match closely with the values predicted by our model. As we sweep τ_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 τ_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

[1] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini, "Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems," *IEEE Embedded Syst. Lett.*, vol. 7, no. 1, pp. 15–18, Mar. 2015.

[2] D. Bharadia, K. R. Joshi, M. Kotaru, and S. Katti, "BackFi: High throughput WiFi backscatter," in *Proc. ACM Conf. Special Interest Group Data Commun.*, 2015, pp. 283–296.

[3] A. P. Chandrakasan, D. C. Daly, J. Kwong, and Y. K. Ramadass, "Next generation micro-power systems," in *Proc. IEEE Symp. VLSI Circuits*, 2008, pp. 2–5.

[4] A. Colin and B. Lucia, "Chain: Tasks and channels for reliable intermittent programs," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Languages Appl.*, 2016, pp. 514–530.

[5] J. Hester, T. Scott, and J. Sorber, "Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors," in *Proc. 12th ACM Conf. Embedded Netw. Sensor Syst.*, 2014, pp. 330–331.

[6] M. Hicks, "Clank: Architectural support for intermittent computation," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 228–240.

[7] B. Lucia and B. Ransford, "A simpler, safer programming and execution model for intermittent systems," in *Proc. 36th ACM SIGPLAN Conf. Program. Language Design Implementation*, 2015, pp. 575–585.

[8] K. Ma, et al., "Spendthrift: Machine learning based resource and frequency scaling for ambient energy harvesting nonvolatile processors," in *Proc. 22nd Asia South Pacific Des. Autom. Conf.*, 2017, pp. 678–683.

[9] K. Ma, et al., "Architecture exploration for ambient energy harvesting nonvolatile processors," in *Proc. IEEE 21st Int. Symp. High Performance Comput. Archit.*, 2015, pp. 526–537.

[10] B. Ransford, J. Sorber, and K. Fu, "Mementos: System support for long-running computation on RFID-scale devices," in *Proc. 16th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2011, pp. 159–170.

[11] A. Wang, V. Iyer, V. Talla, J. R. Smith, and S. Gollakota, "FM backscatter: Enabling connected cities and smart fabrics," in *Proc. USENIX Symp. Netw. Syst. Des. Implementation*, 2017, pp. 243–258.