Published in the Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)

Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems

Graham Gobieski
Carnegie Mellon University
gobieski@cmu.edu

Brandon Lucia
Carnegie Mellon University
blucia@cmu.edu

Nathan Beckmann
Carnegie Mellon University
beckmann@cs.cmu.edu

Abstract

Energy-harvesting technology provides a promising platform for future IoT applications. However, since communication is very expensive in these devices, applications will require inference "beyond the edge" to avoid wasting precious energy on pointless communication. We show that application performance is highly sensitive to inference accuracy. Unfortunately, accurate inference requires large amounts of computation and memory, and energy-harvesting systems are severely resource-constrained. Moreover, energy-harvesting systems operate *intermittently*, suffering frequent power failures that corrupt results and impede forward progress.

This paper overcomes these challenges to present the first full-scale demonstration of DNN inference on an energy-harvesting system. We design and implement SONIC, an intermittence-aware software system with specialized support for DNN inference. SONIC introduces *loop continuation*, a new technique that dramatically reduces the cost of guaranteeing correct intermittent execution for loop-heavy code like DNN inference. To build a complete system, we further present GENESIS, a tool that automatically compresses networks to optimally balance inference accuracy and energy, and TAILS, which exploits SIMD hardware available in some microcontrollers to improve energy efficiency. Both SONIC & TAILS guarantee correct intermittent execution without any hand-tuning or performance loss across different power systems. Across three neural networks on a commercially available microcontroller, SONIC & TAILS reduce inference energy by 6.9× and 12.2×, respectively, over the state-of-the-art.

1 Introduction

The maturation of energy-harvesting technology and the recent emergence of viable intermittent computing models creates the opportunity to build sophisticated battery-less systems with most of the computing, sensing, and communicating capabilities of existing battery-powered systems. Many future IoT applications require frequent decision making, e.g., when to trigger a battery-draining camera, and these decisions must be taken locally, as it is often impractically expensive to communicate with other devices. Future IoT applications will require *local* inference on raw sensor data, and their performance will be determined by inference accuracy. Using energy numbers from recent state-of-the-art systems, we show that such local inference can improve end-to-end application performance by 480× or more.

Recently, deep neural networks (DNNs) [46, 72, 75] have made large strides in inference accuracy. DNNs enable sophisticated inference using limited, noisy inputs, relying on rich models learned from many examples. Unfortunately, while DNNs are much more accurate than traditional alternatives [32, 58], they are also more computationally demanding.

Typical neural networks use tens of millions of weights and require billions of compute operations [46, 72, 75]. These networks target high-powered, throughput-optimized processors like GPUs or Google's TPU, which executes up to 9 trillion operations per second while drawing around 40 watts of power [45]. Even a small DNN (e.g., LeNet [48]) has over a million weights and millions of operations. The most efficient DNN accelerators optimize for performance as well as energy efficiency and consume hundreds of mW [11, 13, 26, 34].

Challenges: In stark contrast to these high-performance systems, energy-harvesting devices use simple microcontrollers (MCUs) built for extreme low-power operation. These MCUs systems run at low frequency (1–16 MHz) and have very small memories (tens or hundreds of kilobytes). Their simple architectures limit them to executing a few million operations per second, while consuming only 1–3mW—a power envelope two orders of magnitude lower than recent DNN accelerators.

DNN inference on these devices is unexplored, and several challenges must be overcome to enable emerging IoT applications on energy-harvesting systems built from commodity components. Most importantly, energy-harvesting systems operate *intermittently* as power becomes available, complicating the development of efficient, correct software. The operating period depends on the properties of the power system, but is short—typically around 100,000 instructions. As a result, *existing DNN inference implementations do not tolerate intermittent operation*.

Recent work proposed software systems that guarantee correct execution on intermittent power for arbitrary programs [16, 39, 40, 51, 54, 80]. These systems add significant runtime overheads to ensure correctness, slowing down DNN inference by on average 10× in our experiments. What these systems have missed is the opportunity to *exploit the structure of the computation to lower the cost of guaranteeing correctness.* This missed opportunity is especially costly for highly structured and loop-heavy computations like DNN inference.

Our approach and contributions: This paper presents the *first demonstration of intermittent DNN inference* on real-world neural networks running on a widely available energy-harvesting system. We make the following contributions:

- We first analyze where energy is spent in an energy-harvesting system and show that inference accuracy largely determines IoT application performance (Sec. 3). This motivates using DNNs despite their added cost over simpler but less accurate inference techniques.
- Building on this analysis, we present GENESIS, a tool that automatically compresses networks to maximize IoT application performance (Sec. 5). GENESIS uses known compression techniques [9, 14, 35, 60]; our contribution is that GENESIS optimally balances inference energy vs. accuracy.
- We design and implement SONIC, a software system for DNN inference with specialized support for intermittent execution (Sec. 6). To ensure correctness at low overhead, SONIC introduces *loop continuation*, which exploits the regular structure of DNN inference to selectively violate task-based abstractions from prior work [54], allowing direct modification of non-volatile memory. Loop continuation is safe because SONIC ensures loop iterations are idempotent through *loop-ordered buffering* (for convolutional layers) and *sparse undo-logging* (for sparse fully-connected layers). These techniques let SONIC resume from where it left off after a power failure, eliminating task transitions and wasted work that plague prior task-based systems.
- Finally, we build TAILS to show how to incorporate hardware acceleration into SONIC (Sec. 7). TAILS uses hardware available in some microcontrollers to accelerate matrix multiplication and convolution. TAILS automatically calibrates its parallelism to ensure correctness with intermittent power.

We evaluate SONIC & TAILS on a TI MSP430 microcontroller [2] using an RF-energy harvester [4, 5] (Secs. 8 & 9). On three real-world DNNs [42, 48, 69], SONIC improves inference efficiency by 6.9× on average over Alpaca [54], a state-of-the-art intermittent system. TAILS exploits DMA and SIMD to further improve efficiency by 12.2× on average.

We conclude with future research directions for parallel intermittent architectures that avoid limitations of current energy-harvesting MCUs and provide new features to support intermittence efficiently (Sec. 10).

2 Background

Energy-harvesting devices operate using energy extracted from their environment. Harvested energy is not continuously available, so an energy-harvesting device operates *intermittently* as energy allows. Prior work showed that intermittent execution leaves memory inconsistent, compromises progress, and suffers from non-termination conditions. Moreover, the typical energy-harvesting device is severely resource-constrained, adding resource management complexity to programming. To motivate the contributions of SONIC & TAILS, we summarize the challenges of intermittent execution on a resource-constrained device and describe the inefficiencies of prior intermittent execution models.

2.1 Intermittent execution on energy-harvesting systems

An energy-harvesting device operates intermittently when harvestable power in the environment is below the device's operating power. To operate despite weak or periodically unavailable power, a device slowly accumulates energy in a hardware buffer (e.g., a capacitor) and operates when the buffer is full. The device drains the buffer as it operates, then it turns off and waits for the buffer to fill again.

Software executes in the *intermittent execution model* on an energy-harvesting device [10, 44, 51, 56, 57, 67]. In intermittent execution, software progresses in bursts, resetting at frequent power failures. Existing devices [2, 78] mix volatile state (e.g., registers and SRAM) and non-volatile memory (e.g., FRAM). A power failure clears volatile state while non-volatile memory persists. Repeated power failures impede progress [67], and may leave memory inconsistent due to partially or repeatedly applied non-volatile memory updates [51]. These progress and consistency issues lead to incorrect behavior that deviates from any continuously-powered execution [15].

Prior work addressed progress and memory consistency using software checkpoints [40, 51, 80], non-volatile processors (NVPs) [52, 53], and programming models based around atomic tasks [16, 39, 54]. A task-based system restarts after power loss with consistent memory at the most recent task or checkpoint. We focus on task-based models because prior work showed that they are more efficient than checkpointing models [16, 54, 55] and because they do not rely on specialized hardware to backup architectural state after each instruction that makes NVPs more complex and less performant.

Task-based intermittent execution models: Task-based intermittent execution models avoid frequent checkpoints by restarting from a task's start after power failure, at which point all register and stack state must be re-initialized. To ensure memory consistency, tasks ensure that the effect of a partial task execution is not visible to a subsequent re-execution. Specifically, data that are read then written (i.e., a WAR dependence) may expose the result of an interrupted task. Task-based systems avoid "the WAR problem" with redo-logging [54] and static data duplication [16].

Task-based systems guarantee correct execution, but at a significant run-time cost. Redo-logging and static duplication both increase memory and compute in proportion to the

amount of data written. Transitioning from one task to the next takes time, so short tasks that transition frequently suffer poor performance. Long tasks better amortize transition costs, but re-execute more work after a power failure. Worse, a task that is too long faces *non-termination* if the energy it requires exceeds the energy that the device can buffer.

A key challenge that we address with SONIC & TAILS *is ensuring correct execution of DNN inference while avoiding the overheads of prior task-based systems.* We achieve this through SONIC's *loop continuation*, which safely "breaks the rules" of existing task-based systems by allowing WAR dependencies for loop index variables (Sec. 6). This is safe because SONIC ensures that each loop iteration is idempotent. Loop continuation yields large gains because it effectively eliminates redo-logging, task transitions, and wasted work.

Resource constraints: Intermittent systems are severely resource-constrained. In this paper we study an intermittent system built using a TI MSP430 microcontroller (MCU), which is the most commonly used processor in existing intermittent systems [17, 36–38, 70]. Such an MCU's frequency is typically 1–16MHz, leaving a substantial performance gap compared to, e.g., a full-fledged, 2GHz Xeon-based system. An intermittent system's MCU usually also houses all the memory available to the system, including embedded SRAM, which is volatile, and embedded FRAM, which is non-volatile. Embedded memories are small and capacity varies by device. A typical MSP430 low-power MCU includes 1–4KB of SRAM and 32–256KB of FRAM. While continuously powered embedded systems may interface with larger memories via a serial bus (i^2c or SPI), most intermittent systems do not due to their high access energy and latency. The typical operating power of an intermittent device is around 1mW.

2.2 Efficient DNN inference

Deep neural networks (DNN) are becoming the standard for inference applications ranging from understanding speech to image recognition [46, 72, 75]. The architecture community has responded with accelerators that improve the performance of inference and training and reduce power consumption. Some architectures focus on dense computations [11–13], others on sparse computations [26, 34, 47, 81], and still others on CNN acceleration [6, 7, 24, 64, 68, 73]. Industry has followed this trend, embracing custom silicon for DNNs [45].

Other recent work focused on algorithmic techniques for reducing the cost of DNN inference. Near-zero weights can often be "pruned" without losing much accuracy [35, 60]. Inference also does not need full-precision floating-point and reducing weight precision [23, 34] reduces storage and computation costs. Additional reductions in storage and computation comes from factoring DNN computations [9, 14, 43, 62, 74, 76, 77].

Despite these efforts, power consumption remains orders-of-magnitude too high for energy-harvesting systems. DNN inference consumes hundreds of milliwatts even on the most efficient accelerators [3, 13, 34]. Recent power-efficient DNN work from the circuits community [28, 66] reduces power somewhat, but compromises on programmability.

More importantly, across all of these prior efforts, *intermittent operation remains unaddressed*. It is the key problem addressed in this work.

3 Motivation for intermittent inference

Many attractive IoT applications will be impractical without intelligence "beyond the edge." Communication is too expensive on these devices for solutions like cloud offloading to be practical. Instead, energy-harvesting devices must decide *locally* how to spend their energy, e.g., when to communicate sensor readings or when to activate an expensive sensor, such as a high-resolution camera.

This section makes the case for inference on energy-harvesting, intermittently operating devices. We show how communication dominates energy, even with state-of-the-art low-power networking, making cloud offloading impractical. We analyze where energy is spent and show that, to a first order, *inference accuracy determines system performance*, motivating the use of DNNs in these applications. Using this analysis we will later compare different DNN configurations and find one that maximizes application performance (Sec. 5).

3.1 The need for inference beyond the edge

Many applications today offload most computation to the cloud by sending input data to the cloud and waiting for a response. Unfortunately, communication is not free. In fact, on energy-harvesting devices, communication costs orders-of-magnitude more energy than local computation and sensing. These high costs mean that *it is inefficient and impractical for energy-harvesting devices to offload inference to the edge or cloud*, even on today's most efficient network architectures.

For example, the recent OpenChirp network architecture lets sensors send data over long distances with extremely low power consumption. To send an eight-byte packet, a terrestrial sensor draws 120mA for around 800ms [25]. Using the recent Capybara energy-harvesting power system [17], such a sensor would require a *900mF* capacitor bank to send a single eight-byte packet. This large capacitor array imposes an effective duty cycle on the device, because the device must idle while charging before it can transmit. A Capybara sensor node with its 2cm $\times$ 2cm solar array in direct sunlight (an optimistic setup) would take around 120 seconds to charge a 900mF capacitor bank [17]. Hence, sending a single 28×28 image with 1B per pixel (e.g., one MNIST image [49]) to the cloud for inference would take *over an hour*.

In contrast, our full-system SONIC prototype performs inference locally in just 10 seconds operating on weak, harvested RF energy—an improvement of more than 360$\times$.

Parameter	Description
IMpJ	Our figure of merit, the number of "interesting" messages sent per Joule of harvested energy.
p	Base rate (probability) of "interesting" events.
t_p	True positive rate in inference.
t_n	True negative rate in inference.
E_{sense}	Energy cost of sensing (e.g., taking a photo).
E_{comm}	Energy cost of communicating one sensor reading.
E_{infer}	Energy cost of a inference on one sensor reading.

Table 1. Description of each parameter in our energy model.

SONIC & TAILS thus open the door to entirely new classes of inference-driven applications on energy-harvesting devices.

3.2 Why accuracy matters

We now consider an example application to show how inference accuracy determines end-to-end application performance. This analysis motivates the use of state-of-the-art inference techniques, namely DNNs, over less accurate but cheaper techniques like support-vector machines.

To reach these conclusions, we employ a high-level analytical model, where energy in the system is divided between sensing, communication, and inference. (Sensing includes all associated local processing, e.g., to set up the sensor and post-process readings.) We use local inference to filter sensor readings so that only the "interesting" sensor readings are communicated. Our figure of merit is the number of interesting sensor readings that can be sent in a fixed amount of harvested energy (which is also a good proxy for execution time). We denote this as IMpJ, or interesting messages per Joule. Though this metric does not capture the interesting readings that are *not* communicated due to inference error (i.e., false negatives), our analysis demonstrates the need for high accuracy, and hence false negatives are uncommon.

This simple model captures many interesting applications of inference beyond the edge: e.g., wildlife monitoring, disaster recovery, wearables, military, etc. For concreteness, we consider a wildlife-monitoring application where sensors with small cameras are deployed across a wide area with OpenChirp connectivity. These sensors monitor a local population of, say, hedgehogs and send pictures over radio when they are detected. The goal is to capture as many images of hedgehogs as possible, and images without have no value.

Baseline without inference: Our baseline system does not support local inference, so it must communicate every image. Communication is expensive, so this baseline system does not perform well. Suppose sensing costs E_{sense} energy, communicating one sensor reading costs E_{comm} energy, and interesting events occur at a base rate of p (see Table 1). Then the baseline system spends $E_{\text{sense}} + E_{\text{comm}}$ energy per event, only p of which are worth communicating, and its IMpJ is:

$$\text{Baseline} = \frac{p}{E_{\text{sense}} + E_{\text{comm}}} \tag{1}$$

Ideal: Although impossible to build, an ideal system would communicate only the interesting sensor readings, i.e., a fraction p of all events. Hence, its IMpJ is:

$$\text{Ideal} = \frac{p}{E_{\text{sense}} + p\, E_{\text{comm}}} \tag{2}$$

Local inference: Finally, we consider a realistic system with local, imperfect inference. In addition to sensing energy E_{sense}, each sensor reading requires E_{infer} energy to decide whether it is worth communicating. Suppose inference has a true positive rate of t_p and a true negative rate of t_n. Since communication is very expensive, performance suffers from incorrectly communicated, uninteresting sensor readings at a rate of: $(1-p)\,(1-t_n)$. Its IMpJ is:

$$\text{Inference} = \frac{p\, t_p}{(E_{\text{sense}} + E_{\text{infer}}) + \big(p\, t_p + (1-p)\,(1-t_n)\big)\; E_{\text{comm}}} \tag{3}$$

Case study: Wildlife monitoring: We now apply this model to the earlier wildlife monitoring example. Hedgehogs are reclusive creatures, so "interesting" photos are rare, say $p = 0.05$. Low-power cameras allow images to be taken at low energy, e.g., $E_{\text{sense}} \approx 10\text{mJ}$ [61]. As we saw above, communicating an image is expensive, taking $E_{\text{comm}} \approx 23{,}000\text{mJ}$ over OpenChirp [25]. Finally, we consider two systems with local inference: a naïve baseline implemented using prior task-based intermittence support (specifically Tile-8 in Sec. 6.2) and SONIC & TAILS, our proposed technique. Their inference energies are gathered from our prototype (Sec. 8), taking $E_{\text{infer,naïve}} \approx 198\text{mJ}$ and $E_{\text{infer,TAILS}} \approx 26\text{mJ}$, respectively.

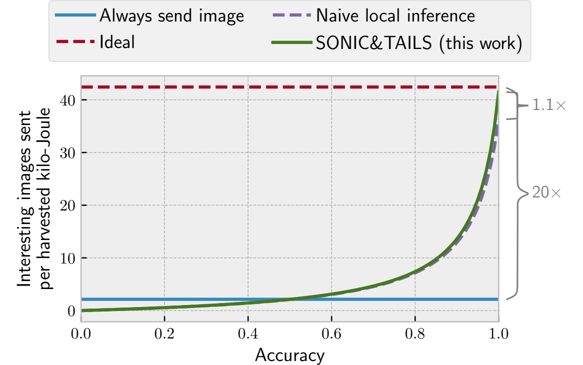


Figure 1. Inference accuracy determines end-to-end system performance in an example wildlife monitoring application. Interesting events are rare and communication is expensive; local inference ensures that energy is only spent on interesting events.

Fig. 1 shows each system's IMpJ after plugging these numbers into the model. For simplicity, the figure assumes that true positive and negative rates are equal, termed "accuracy". Since communication dominates the energy budget, local inference enables large end-to-end benefits on the order of $1/p = 20\times$. However, for these gains to be realized in practice, inference must be accurate, and the benefits quickly

deteriorate as inference accuracy declines. Qualitatively similar results are obtained when p varies, though the magnitude of benefit changes (increasing with smaller p).

This system is dominated by the energy of sending results. Inference is relatively inexpensive, so naïve local inference and SONIC & TAILS perform similarly (though SONIC & TAILS outperforms Naïve by up to 14%). To see the benefits of efficient inference, we must first address the system's communication bottleneck.

Sending only inference results: Depending on the application, even larger end-to-end improvements are possible by sending only the *result* of inference rather than the full sensor reading. For instance, in this wildlife monitoring example, the energy-harvesting device could send a single packet when hedgehogs were detected, rather than the full image. The effect is to significantly decrease E_{comm} for the systems with local inference, mitigating the system's bottleneck. In our wildlife monitoring example, E_{comm} decreases by 98×.

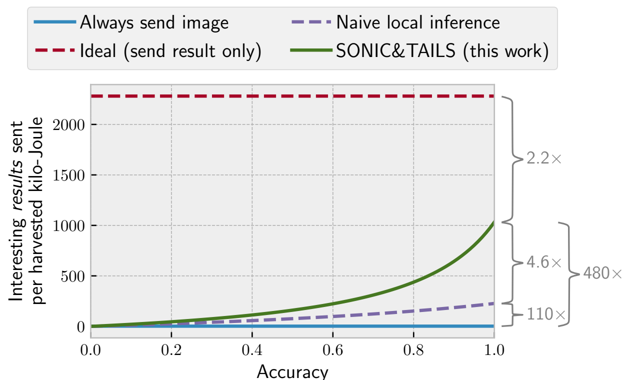


Figure 2. Local inference (i.e. Naive and SONIC & TAILS) lets energy-harvesting devices communicate only *results* of inference, enabling dramatic increases in end-to-end system performance.

Fig. 2 shows end-to-end performance when only sending inference results. Local inference allows dramatic reductions in communication energy: SONIC & TAILS can detect and communicate 480× more events than the baseline system without local inference. These reductions also mean that inference is a non-negligible energy cost, and SONIC & TAILS *outperform naïve local inference by* 4.6×. Finally, the gap between Ideal and SONIC & TAILS is 2.2×. This gap is difficult to close further on current hardware; we discuss ways to address it in Sec. 10.

4 System overview

This paper describes the first system for performing DNN inference efficiently on intermittently-operating, energy-harvesting devices. Fig. 3 shows the new system components in this work and how they produce an efficient, intermittence-safe executable starting from a high-level DNN model description. There are three main components to the system: GENESIS, SONIC, and TAILS.

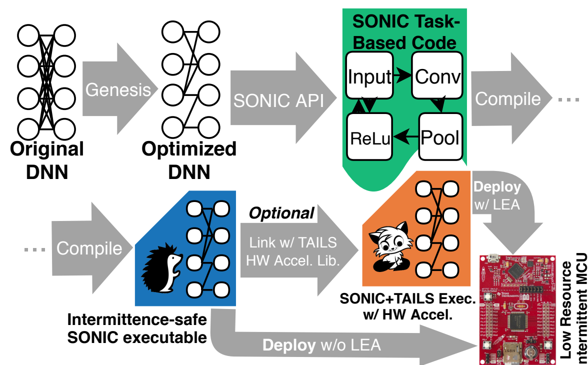


Figure 3. Overview of implementing a DNN application using SONIC & TAILS. GENESIS first compresses the network to optimize interesting messages sent per Joule (IMpJ). SONIC & TAILS then ensure correct intermittent execution at high performance [29].

GENESIS (generating energy-aware networks for efficiensy on intermittent systems) is a tool that automatically optimizes a DNN, starting from a programmer's high-level description of the network. GENESIS attempts to compress each layer of the network using well-known separation and pruning techniques. GENESIS's goal is to *find a network that optimizes IMpJ* while meeting resource constraints. As Fig. 3 shows, GENESIS's input is a network description and its output is an optimally compressed network. Sec. 5 describes GENESIS.

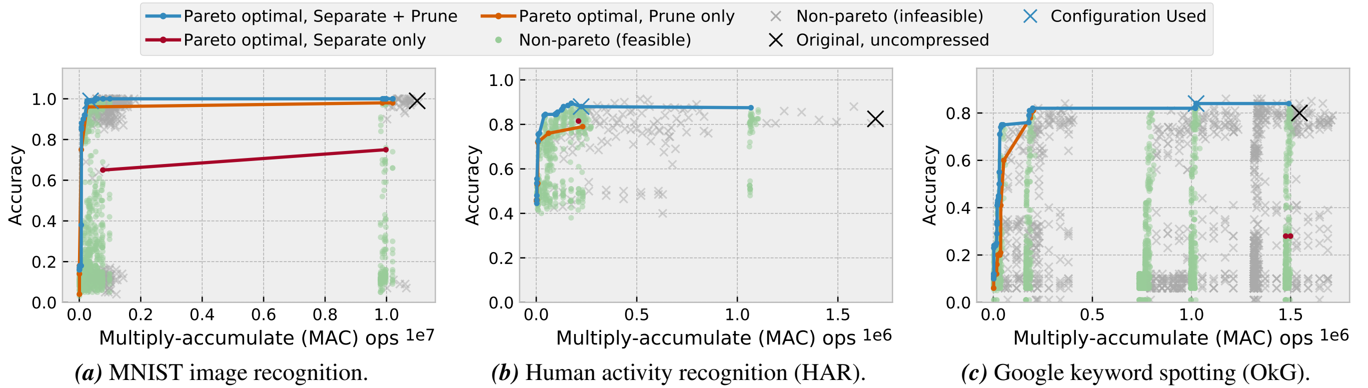
SONIC (software-only neural intermittent computing) is an intermittence-safe, task-based API and runtime system that includes specialized support for DNN inference that *safely "breaks the rules" of existing task-based systems to improve performance*. SONIC is compatible with existing task-based frameworks [16, 54], allowing seamless integration into larger applications. Sec. 6 describes SONIC in detail.

TAILS (tile-accelerated intermittent LEA support) is an alternative to the SONIC runtime library that leverages hardware vector acceleration, specifically targeting the TI Low Energy Accelerator (LEA) [1]. To use TAILS, the programmer need only link their compiled binary to the TAILS-enabled runtime system. This runtime includes all of SONIC's optimizations and a suite of hardware-accelerated vector operations, such as convolutions. Sec. 7 describes TAILS in detail.

Starting with a high-level network description, a programmer can use GENESIS, SONIC, and TAILS to build an efficient, intermittent DNN-enabled application that meets resource constraints, is robust to intermittent operation, and leverages widely available hardware acceleration. Our code and datasets can be found at: https://github.com/CMUAbstract/SONIC.

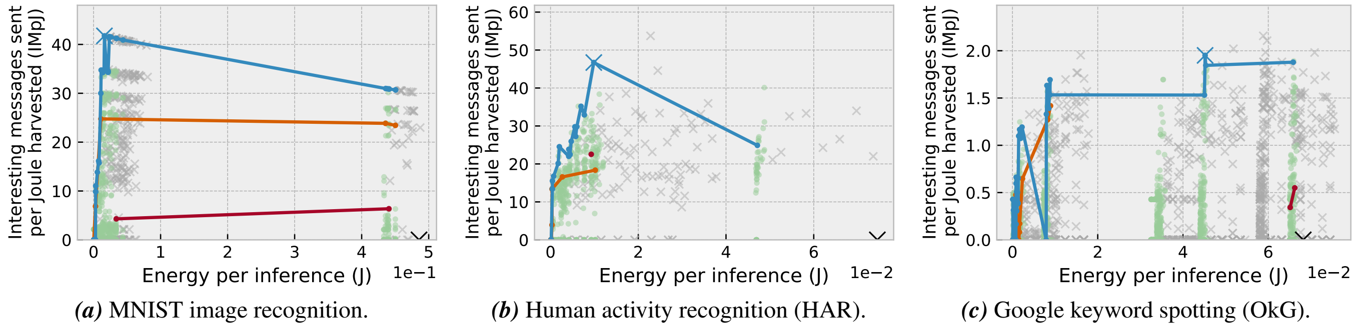
5 Optimal DNN compression with GENESIS

The first challenge to overcome in SONIC & TAILS is fitting neural networks into the resource constraints of energy-harvesting systems. In particular, the limited memory capacity of current microcontrollers imposes a hard constraint on



(a) MNIST image recognition. ***(b)*** Human activity recognition (HAR). ***(c)*** Google keyword spotting (OkG).

Figure 4. GENESIS explores the inference accuracy-cost tradeoff for different neural network configurations.



(a) MNIST image recognition. ***(b)*** Human activity recognition (HAR). ***(c)*** Google keyword spotting (OkG).

Figure 5. GENESIS uses our end-to-end application performance model (Eq. 3) to select the best feasible network configuration.

networks. We have developed a tool called GENESIS that automatically explores different configurations of a baseline neural network, applying separation and pruning techniques (Sec. 2) to reduce the network's resource requirements. GENESIS improves upon these known techniques by optimally balancing inference energy and true positive/negative rates to maximize IMpJ, building on the the model in Sec. 3.

5.1 Neural networks considered in this paper

This paper considers three networks, summarized in Table 2. To represent image-based applications (e.g., wildlife monitoring and disaster recovery), we consider MNIST [49]. We consider MNIST instead of ImageNet because ImageNet's large images do not fit in a resource-constrained device's memory. To represent wearable applications, we consider human activity recognition (HAR). HAR classifies activities using accelerometer data [42]. To represent audio applications, we consider Google keyword spotting (OkG) [69], which classifies words in audio snippets.

We also evaluated binary neural networks and several SVM models and found that they perform poorly on current energy-harvesting MCUs. A 99%-accurate binary network for MNIST required 4.4MB of weights [18], exceeding the device's scant memory, and compressing this to 360KB lost nearly 10% accuracy [8]. Likewise, no SVM model that fit on the device was competitive with the DNN models [50]: measured by IMpJ, SVM under-performed by 2× on MNIST and by 8× on HAR, and we could not find an SVM model for OkG that performed anywhere close to the DNN.

5.2 Fitting networks on energy-harvesting systems

GENESIS evaluates many compressed configurations of a network and builds a Pareto frontier. Compression has trade-offs in four dimensions, difficult to capture with a pareto curve; these include true negative rate, true positive rate, memory size (i.e., parameters), and compute/energy (i.e., operations). Fully-connected layers typically dominate memory, whereas convolutional layers dominate compute. GENESIS compresses both.

GENESIS compresses each layer using two known techniques: separation and pruning. Separation (or rank decomposition) splits an $m \times n$ fully-connected layer into two $m \times k$ and $k \times n$ matrix multiplications, or an $m \times n \times k$ convolutional filter into three $m \times 1 \times 1$, $1 \times n \times 1$, and $1 \times 1 \times k$, filters [9, 14]. GENESIS separates layers using the Tucker tensor decomposition, using the high-order orthogonal iteration algorithm [21, 22, 79]. Pruning involves removing parameters below a given threshold, since they have small impact on results [35, 60].

GENESIS sweeps parameters for both separation and pruning across each layer of the network, re-training the network after compression to improve accuracy. GENESIS relies on the Ray Tune black box optimizer with the Median Stopping Rule to explore the configuration space [30, 59]. Fig. 4 shows the results for the networks in Table 2. Each marker on the figure

Network	Layer	Uncompressed Size	Compression Technique	Compressed Size	Compression	Accuracy
Image classification (MNIST)	Conv	$20 \times 1 \times 5 \times 5$	HOOI	3×1D Conv	11.4×	99.00%
	Conv	$100 \times 20 \times 5 \times 5$	Pruning	1253	39.9×	
	FC	200×1600	Pruning, SVD	5456	109×	
	FC	500×200	Pruning, SVD	1892	—	
	FC	10×500	—	—	—	
Human activity recognition (HAR)	Conv	$98 \times 3 \times 1 \times 12$	HOOI	3×1D Conv	2.25×	88.0%
	FC	192×2450	Pruning, SVD	10804	58.1×	
	FC	256×192	Pruning, SVD	—	—	
	FC	6×256	—	—	—	
Google keyword spotting (OkG)	Conv	$186 \times 1 \times 98 \times 8$	HOOI, Pruning	3×1D Conv	7.3x	84.0%
	FC	96×1674	Pruning, SVD	16362	11.8×	
	FC	128×96	Pruning, SVD	2070	—	
	FC	32×128	SVD	4096	2×	
	FC	128×32	SVD	4096	—	
	FC	128×12	—	—	—	

Table 2. Neural networks used in this paper.

represents one compressed configuration, shown by inference accuracy on the y-axis and inference energy on the x-axis. Feasible configurations (i.e., ones that fit in our device's small memory; see Sec. 8) are shown as green circles and infeasible configurations are grey ×s. Note that the original configuration (large ×) is infeasible for all three networks, meaning that they cannot be naïvely ported to the device because their parameters would not fit in memory.

Fig. 4 also shows the Pareto frontier for each compression technique. Generally, pruning is more effective than separation, but the techniques are complementary.

5.3 Choosing a neural network configuration

GENESIS estimates a configuration's IMpJ using the model from Sec. 3, specifically Eq. 3. The user specifies E_{sense} and E_{comm} for their application as well as per-compute-operation energy cost. From these parameters, GENESIS estimates E_{infer} for each configuration, and uses the inference accuracy from the prior training step to estimate application performance. The user can specify which class in the training set is "interesting," letting GENESIS compute true positive t_p and negative t_n rates for the specific application.

Fig. 5 shows the results by mapping each point in Fig. 4 through the model. For these results, we use E_{sense} from Sec. 3, per-operation energy from our SONIC & TAILS prototype in Sec. 8, and estimate E_{comm} from input size assuming OpenChirp networking [25].

GENESIS chooses the feasible configuration that maximizes estimated end-to-end performance (i.e., IMpJ). Fig. 5 shows that this choice is non-trivial. True positive, true negative, and inference energy affect end-to-end application performance in ways that are difficult to predict. Simply choosing the most accurate configuration, as the twisty blue curve suggests in Fig. 5, is insufficient since it may waste too much energy or underperform other configurations on true positive or true negative rates.

6 Efficient intermittent inference with SONIC

SONIC is the first software system optimized for inference on resource-constrained, intermittently operating devices. SONIC supports operations common to most DNN computations, exposing them to the programmer through a simple API. SONIC's functionality is implemented as a group of *tasks* supported by the SONIC runtime system, which is a modified version of the Alpaca runtime system [54]. These tasks implement DNN functionality, and the SONIC runtime system guarantees correct intermittent operation.

Specializing intermittence support for DNN inference yields large benefits. Prior task-based intermittent execution models [16, 54] can degrade performance by up to 19× and by 10× on average (Sec. 9). SONIC dramatically reduces these overheads to just 25%–75% over a standard baseline of DNN inference that does not tolerate intermittent operation.

SONIC achieves these gains by eliminating the three major sources of overhead in prior task-based systems: redo-logging, task transitions, and wasted work (Sec. 2). Our key technique is *loop continuation*, which selectively violates the task abstraction for loop index variables. Loop continuation lets SONIC directly modify loop indices without frequent and expensive saving and restoring. By writing loop indices directly to non-volatile memory, SONIC checkpoints its progress after each loop iteration, eliminating expensive task transitions and wasting work upon power failure.

Loop continuation is safe because SONIC ensures that each loop iteration is idempotent. SONIC ensures idempotence in convolutional and fully-connected layers through *loop-ordered buffering* and *sparse undo-logging*. These two techniques ensure idempotence without statically privatizing or dynamically checkpointing data, avoiding the overheads imposed by prior task-based systems.

6.1 The SONIC API

The SONIC API lets the programmer describe a DNN's structure through common linear algebra primitives. Just as a programmer chains tasks together in a task-based intermittent programming model [16, 39, 54], the programmer chains SONIC's tasks together to represent the control and data flow of a DNN inference pipeline. SONIC's API exposes functionality that the programmer invokes like any other task in their program (specifically, a *modular task group* [16, 54]). Though SONIC "breaks the rules" of a typical task-based intermittent system, the programmer does not need to reason about these differences when they are writing a program using the SONIC API. The program-level behavioral guarantee that SONIC provides is the same as the one underlying other task-based intermittent execution models: a SONIC task will execute atomically despite power interruptions by ensuring that repeated, interrupted attempts to execute are idempotent.

6.2 The SONIC runtime implementation

DNN inference is dominated by loops within each layer of the neural network. SONIC optimizes DNN inference by ensuring that these loops execute correctly on intermittent power while adding much less overhead than prior task-based systems.

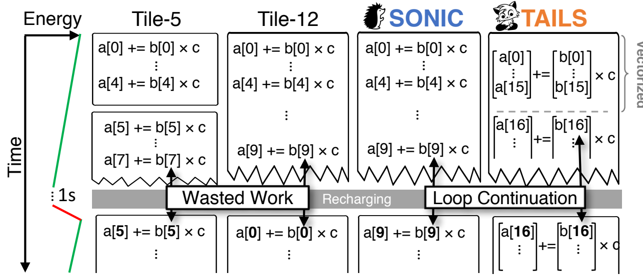


Figure 6. Executing a loop using two fixed task-tilings and with SONIC's loop continuation mechanism. Loop continuation avoids the re-execution and non-termination costs of task-tiling. TAILS uses SIMD to perform more work in a fixed energy budget (Sec. 7).

Loops in task-based systems: A typical task-based intermittent system sees two kinds of loops: *short loops* and *long loops*. All iterations of a *short loop* fit in a single task and will complete without consuming more energy than the device can buffer. A short loop maintains control state in volatile memory and these variables clear on power failure. When power resumes, the task restarts and completes. Data manipulated by a short loop are usually non-volatile (i.e., "task-shared" [54]) and if read and updated, they must be backed up (either statically or dynamically) to ensure they remain consistent. The problem with short loops is that they always restart from the beginning, wastefully repeating loop work that was already done. In contrast, a *long loop* with many iterations does not fit in a single task; a long loop demands more energy than the device can buffer and may never terminate. A programmer must split loop iterations across tasks, requiring a task transition on each iteration and requiring control state and data to be non-volatile and backed up. The problem with long loops is that may not terminate and, when split across tasks, impose hefty privatization and task transition overheads.

Task-tiling is a simple way to split a loop's iterations into tasks. A task-tiled loop executes a fixed number of iterations per task. Task-tiling amortizes task transitioning overhead, but risks executing more iterations in a single task than the device's energy buffer can support, causing non-termination. Figure 6 shows the intermittent execution (energy trace on left) of a loop computing a dot product using two fixed tile sizes of five (Tile-5) and twelve (Tile-12). Tile-5 wastes work when four iterations complete before a failure. Tile-12 prevents forward progress because the device buffers insufficient energy to complete twelve iterations.

6.2.1 Loop continuation

SONIC's *loop continuation* is an intermittence-safe optimization that avoids wasted work, unnecessary data privatization, and task transition overheads in tasks containing long-running loop nests. Loop continuation works by directly modifying loop control variables and memory manipulated in a loop nest, rather than splitting a long-running loop across tasks. Loop continuation permits loops of arbitrary iteration count within a single task, with neither non-termination nor excessive state management overhead. Loop continuation stores a loop's control variables and data manipulated directly in non-volatile memory *without backing either up*. When a loop continuation task restarts, its (volatile) local variables are reinitialized at the task's start. The loop control variables, however, retain their state and the loop continues from the last attempted iteration.

Fig. 7 and Listing 1 show how loop continuation works by storing the loop control state, **i**, for Task_Convolve in non-volatile memory. SONIC ensures that the loop's control variables **i** is correct by updating it at the end of the iteration and *not resetting it upon re-execution* (the loop in Listing 1 starts from i). A power failure during or after the update to the control variable may require the body of the loop nest to repeat a single iteration, but it never skips an iteration.

Figure 6 shows SONIC executing using loop continuation. Despite the power interruption, execution resumes on the ninth loop iteration, rather than restarting the entire loop nest or every fifth iteration like Tile-5 does.

6.2.2 Idempotence tricks

Normally, restarting from the middle of a loop nest could leave manipulated data partially updated and possibly inconsistent. However, loop continuation is safe because SONIC's runtime system ensures each loop iteration is idempotent using either *loop-ordered buffering* or *sparse undo-logging*. SONIC never requires an operation in an iteration to read a value produced by another operation in the same iteration. Thus, an iteration that repeatedly re-executes due to power interruption will always see correct values.

Loop-ordered buffering: Loop-ordered buffering is a double-buffering mechanism used in convolutional layers (and dense fully-connected layers) that ensures each loop iteration is idempotent without expensive redo-logging (cf., [54]). Since the MSP430 devices do not possess sophisticated caching mechanisms, *rather than optimizing for reuse and data locality,* SONIC *optimizes the number of items needed to commit.* By re-ordering the loops in DNN inference and double-buffering partial activations as needed, SONIC is able to *completely eliminate* commits within a loop iteration.

Evaluating a sparse or dense convolution requires SONIC to apply a filter to a layer's entire input activation matrix. SONIC orders loop iterations to apply each element of the filter to each element of the input activation (i.e., multiplying them) before moving on to the next element of the filter. For idempotence, SONIC writes the partially accumulated value to an intermediate output buffer, rather than applying updates to the input matrix in-place. After applying a single filter element to each entry in the input and storing the partial result in the intermediate buffer, SONIC swaps the input buffer with the intermediate buffer and moves on to the next filter value.

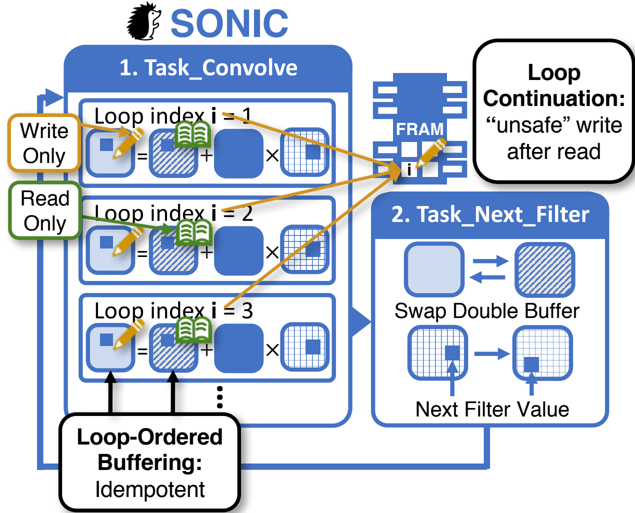


Figure 7. SONIC uses *loop continuation* and *loop-ordered buffering* to reduce overheads of correct intermittent execution. *Loop continuation* maximizes the amount of computation done per task by allowing computation to pick up where it left off before power failure.

```
def Task_Convolve():
    for i in i...len(src):  # note: i is NOT reset to 0 here
        f = src[i] * filter[pos]
        dest[i] = (inter[i] + f) if pos > 0 else f

    if pos < len(filter):
        transition Task_Next_Filter
    else:
        pos, i = 0, 0 # reset for next invocation
        transition caller # return from SONIC

def Task_Next_Filter():
    atomic { # swap buffers; reset i; next filter element
        dest, inter = inter, dest
        i = 0
        pos++
    }
    transition Task_Convolve
```

Listing 1. Pseudocode corresponding to Fig. 7. src, dest, inter, i, and pos are non-volatile. Task_Convolve implements loop continuation. Task_Next_Filter atomically swaps buffers and updates variables to move to the next element of the convolutional filter.

Since SONIC never reads and then writes to the same memory locations within an iteration, it avoids the WAR problem described in Sec. 2 and loop iterations are thus idempotent. Fig. 7 shows how under loop-ordered buffering, SONIC never reads and writes to the same matrix buffer while computing a partial result in Task_Convolve (dest is distinct from inter in Listing 1). After finishing this task, SONIC transitions to Task_Next_Filter, which swaps the buffer pointers and gets the next value to apply from the filter.

Sparse undo-logging: While loop-ordered buffering is sufficient to ensure each loop iteration is idempotent, it is sometimes unnecessarily wasteful. The problem arises because loop-ordered buffering swaps between buffers after every task, so it must copy data between buffers in case it is read in the future—even if the data has not been modified. This copying is wasteful on sparse fully-connected layers, where most filter weights are pruned and thus few activations are modified in a single iteration. With loop-ordered buffering, SONIC ends up spending most of its time and energy copying unmodified activations between buffers.

To eliminate this inefficiency, SONIC introduces *sparse undo-logging* which ensures idempotence through undo-logging instead of double buffering. To ensure atomicity, sparse undo-logging tracks its progress through the loop via two index variables, the *read* and *write* indices. When applying a filter, SONIC first copies the original, unmodified activation into a canonical memory location, and then increments the read index. SONIC then computes the modified activation and writes it back to the original activation buffer (there is no separate output buffer). Then it increments the write index and proceeds to the next iteration. This two-phase approach guarantees correct execution, since sparse undo-logging resumes computing the output value from the buffered original value if power fails in the middle of an update.

Sparse undo-logging ensures that the work per task grows with the number of modifications made, not the size of the output buffer (unlike loop-ordered buffering). However, sparse undo-logging doubles the number of memory writes per modified element, so it is inefficient on dense layers where most data are modified. In those cases, loop-ordered buffering is significantly more efficient. We therefore only use sparse undo-logging in sparse fully-connected layers. Finally, unlike prior task-based systems such as Alpaca, sparse undo-logging ensures idempotence with *constant* space overhead and *no* task transition between iterations.

Related work: Prior work in persistent memory [27] uses techniques similar to our sparse undo-logging. This work is in the high-performance domain, and therefore focuses on cache locality and scheduling cache flushes and barriers. In contrast, our prototype has no caches, and we exploit this fact in loop-ordered buffering to re-arrange loops in a way that would destroy cache performance on conventional systems. Moreover, SONIC is more selective than [27], only using undo-logging in sparse fully-connected layers where it outperforms double buffering.

7 Hardware acceleration with TAILS

TAILS improves on SONIC by incorporating widely available hardware acceleration to perform inference even more efficiently. A programmer may optionally link their SONIC application to the TAILS runtime system, enabling the application to use direct-memory access (DMA) hardware to optimize block data movement and to execute operation in parallel using a simple vector accelerator like the TI Low-Energy Accelerator (LEA) [1]. LEA supports finite-impulse-response discrete-time convolution (FIR DTC), which directly implements the convolutions needed in DNN inference.

TAILS's runtime system enables the effective use of LEA in an intermittent system by *adaptively binding hardware parameters at run time to maximize operational throughput without exceeding the device's energy buffer*. Our TAILS prototype adaptively determines the DMA block size and LEA vector width based on the number of operations that successfully complete using the device's fixed energy buffer. After calibrating these parameters, TAILS uses them to configure available hardware units and execute inference thereafter.

7.1 Automatic one-time calibration

Before its first execution, a TAILS application runs a short, recursive calibration routine to determine DMA block size and LEA vector size. The routine determines the maximum vector size that it is possible to DMA into LEA's operating buffer, process using FIR DTC, and DMA back to non-volatile memory without exceeding the device's energy buffer and impeding progress. If a tile size does not complete before power fails, the calibration task re-executes, halving the tile size. Calibration ends when a FIR DTC completes and TAILS uses that tile size for subsequent computations.

7.2 Accelerating inference with LEA

Once TAILS determines its tile size, the application runs, using DMA and LEA to compute dense and sparse convolutions and dense matrix multiplications. LEA has limitations: it only supports dense operations and can only read from the device's small 4KB SRAM (not the 256KB FRAM). TAILS uses DMA to move inputs into SRAM, invokes LEA, and DMAs the results back to FRAM. Dense layers are natively supported: fully-connected layers use LEA's vector MAC operation, and convolutions use LEA's one-dimensional FIR DTC operation. To support two- and three-dimensional convolutions, TAILS iteratively applies one-dimensional convolutions and accumulates those convolutions' results. TAILS uses loop-ordered buffering to ensure that updates to the partially accumulated values are idempotent (Sec. 6.2.2).

Sparse operations require more effort. TAILS uses LEA for sparse convolutions by first making filters dense (padding with zeros). Making the filters dense is inexpensive because each filter is reused many times, amortizing its creation cost. However, this does mean that LEA performs unnecessary work, which sometimes hurts performance. For this reason, we use LEA's dot-product operation instead of FIR-DTC for $1 \times p \times 1$ factored convolutional layers.

Finally, sparse fully-connected layers are inefficient on LEA because filters do not get reuse. We found that TAILS spent most of its time on padding filters, and, despite significant effort, we were unable to accelerate sparse fully-connected layers with LEA. For this reason, TAILS performs sparse fully-connected layers in software exactly like SONIC.

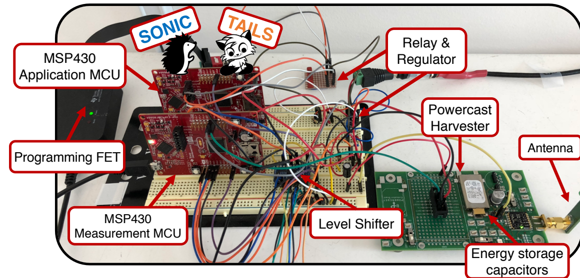


Figure 8. Diagram of the measurement setup.

8 Methodology

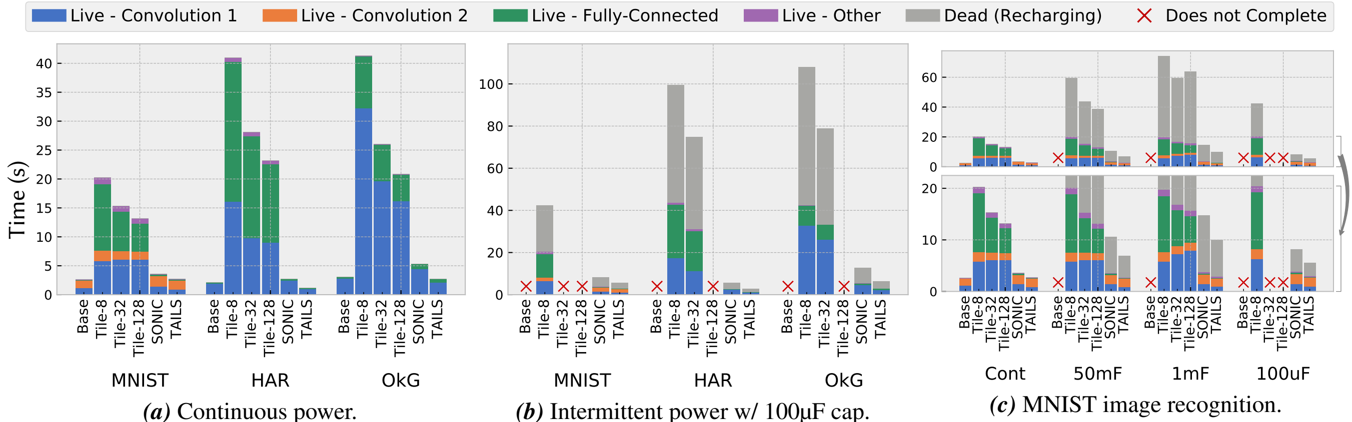
We implement SONIC and TAILS on the TI-MSP430FR5994 [2] at 16MHz in the setup in Fig. 8. The board is connected to a Powercast P2210B [4] harvester 1m away from a 3W Powercaster transmitter [5]. We ran all configurations on continuous power and on intermittent power with three different capacitor sizes: 1mF, 50mF, and 100µF.

Running code on the device: We compile with MSPGCC 6.4 and use TI's MSPDriverlib for DMA and TI's DSPLib for LEA. We use GCC instead of Alpaca's LLVM backend because LLVM lacks support for 20-bit addressing and produces slower code for MSP430 than GCC.

Measurement: We use a second MSP430FR5994 to measure intermittent executions. GPIO pins on the measurement MCU connect through a level-shifter to the intermittent device, allowing it to count reboots and signal when to start and stop timing.

We automate measurement with a combination of software and hardware that compiles a configuration binary, flashes the binary to the device, and communicates with the measurement MCU to collect results. The system actuates a relay to switch between continuous power for reprogramming and intermittent power for testing.

Measuring energy: By counting the number of charge cycles between GPIO pulses, we can determine the amount of energy consumed in different code regions. For a more fine-grained approach, we built a suite of microbenchmarks to count how many times a particular operation (e.g., a load from FRAM) can run in single charge cycle. We then profile how many times each operation is invoked during inference and scale by per-operation energy to get a detailed energy breakdown.



(a) Continuous power. *(b)* Intermittent power w/ 100µF cap. *(c)* MNIST image recognition.

Figure 9. Inference time on three neural networks. The naïve baseline is fast, but does not tolerate intermittent execution. Tiled implementations can ensure correct execution, but only at high cost (up to 19× slowdown) and sometimes do not complete. SONIC ensures correct execution and is nearly as fast as the naïve baseline, and TAILS is even faster. (9a) All three networks on continuous power, where SONIC & TAILS add dramatically lower overheads than prior task-based systems. (9b) All three networks on intermittent power (100µF capacitor), where the baseline and most tiled implementations do not complete. (9c) The MNIST network across all four power systems. SONIC & TAILS always completes and has consistently good performance; HAR and OkG show similar results.

Baselines for comparison: We compare SONIC & TAILS to four DNN inference implementations. The first implementation is a standard, baseline implementation that does not tolerate intermittent operation (it does not terminate). The other three implementations are based on Alpaca [54] and split up loops by tiling iterations, as in Fig. 6.

9 Evaluation

We now evaluate our prototype to demonstrate that: *(i)* SONIC & TAILS guarantee correct intermittent execution; *(ii)* SONIC & TAILS greatly reduce inference energy and time over the state-of-the-art; and *(iii)* SONIC & TAILS perform well across a variety of networks without any hand-tuning.

9.1 SONIC & TAILS significantly accelerate intermittent DNN inference over the state-of-the-art

Fig. 9 shows the inference time for the three networks we consider (Table 2). For each network, we evaluated six implementations running on four different power systems. We break inference time into: dead time spent recharging; live time spent on each convolution layer (which dominates); live time spent on the fully-connected layers; and everything else.

First, notice that SONIC & TAILS guarantees correct execution for every network on every power system. This is not true of the naïve baseline, which does not run correctly on intermittent power, or of most tilings for prior task-based intermittent systems. The only other implementation that reliably executes correctly is Tile-8, since its tiling is small enough to always complete within a single charge cycle. The other tilings fail on some configurations: Tile-32 fails on MNIST with a 100µF capacitor, and Tile-128 fails on all networks at 100µF.

SONIC & TAILS guarantee correct execution at much lower overheads than Tile-8. Averaging across networks, Tile-8 is gmean 13.4× slower than the naïve baseline on continuous power, whereas SONIC is 1.45× slower and TAILS is actually 1.2× *faster* than the baseline. That is to say, SONIC improves performance on average by 6.9× over tiled Alpaca [54], and TAILS improves it by 12.2×. Moreover, execution time is consistent across capacitor sizes for SONIC & TAILS.

Larger tile sizes amortize overheads somewhat, but since they do not complete on all networks or capacitor sizes, they are an unattractive implementation choice. SONIC & TAILS guarantee correct intermittent execution across all capacitor sizes, while also being faster than the largest tilings: even compared to Tile-128, SONIC is on average 5.2× faster on continuous power and TAILS is 9.2× faster.

Both DMA and LEA improve TAILS's efficiency. We tested configurations where DMA and LEA are emulated by software and found that LEA consistently improved performance by 1.4×, while DMA improved it by 14% on average.

Ultimately, these results indicate that inference is viable on commodity energy-harvesting devices, and SONIC & TAILS significantly reduce overheads over the state-of-the-art.

9.2 Loop continuation nearly eliminates overheads due to intermittence

Fig. 10 shows that the overheads of SONIC & TAILS come mainly from control required to support intermittence. The darker-hatched regions of the bars represent the proportion of time spent computing a layer's kernel (i.e., the main loop), while the lighter regions represent control overheads (i.e., task transitions and setup/teardown). Most of the difference in performance between the baseline and SONIC is attributable to the lighter, control regions. This suggests that SONIC imposes small overhead over the naïve baseline, which accumulates

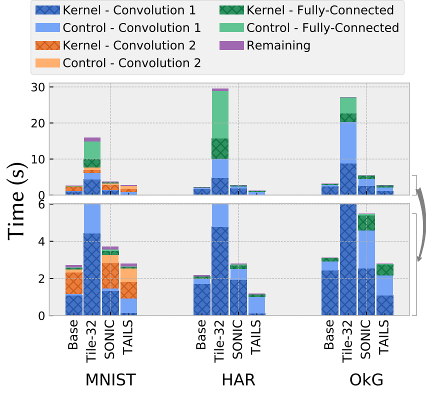


Figure 10. Proportions of time spent computing the kernel of a layer. SONIC & TAILS add small overheads over a naïve baseline, unlike prior task-based systems (Tile-32).

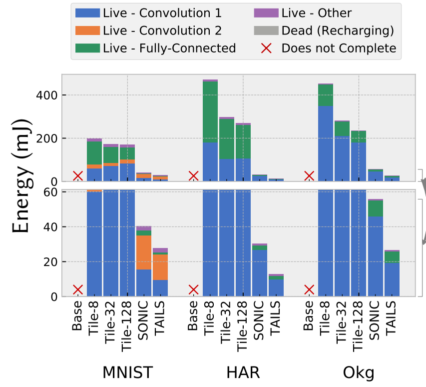


Figure 11. Energy of three neural networks with a 1mF capacitor. SONIC & TAILS require substantially less energy than the state-of-the-art.

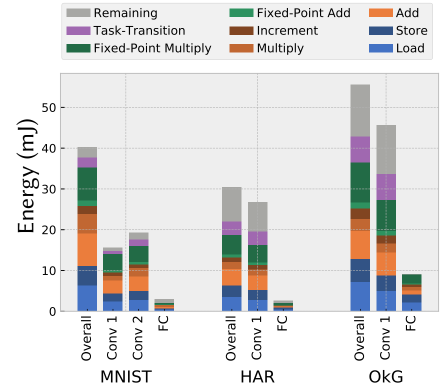


Figure 12. Energy profile of SONIC broken down by operation and layer. Multiplication, control, and memory accesses represent significant overheads.

values in registers and avoids memory writes (but does not tolerate intermittence).

TAILS's overhead also comes from control; TAILS significantly accelerates kernels. TAILS's control overhead is large due to LEA's fixed-point representation, which forces TAILS to bit-shift activations before invoking FIR-DTC. Moreover, LEA does not have a left-shift operation (it does have a right-shift), so these shifts must be done in software. These shifts account for most of the control time in Fig. 10.

Fig. 10 also shows the time breakdown for Tile-32. Unlike SONIC & TAILS, Tile-32 spends significantly more time in both control and the kernel. This is because Alpaca uses redo-logging on all written values to ensure idempotence, so every write requires dynamic buffering (kernel time) and committing when the task completes (control time). SONIC & TAILS effectively eliminate redo-logging, avoiding these overheads.

9.3 SONIC & TAILS use much less energy than tiling

Energy-harvesting systems spend a majority of their time powered off recharging, so execution time is largely determined by energy efficiency. Fig. 11 shows that SONIC & TAILS achieve high performance because they require less energy than other schemes. Inference energy is in direct proportion to the dead time spent recharging in Fig. 9. Since dead time dominates inference time, SONIC & TAILS get similar improvements in inference energy as they do in terms of inference time.

9.4 Where does SONIC's energy go?

Fig. 12 further characterizes SONIC by showing the proportion of energy spent on different operations. The blue regions represent memory operations, the orange regions are control instructions, the green regions are arithmetic instructions within the kernels, the purple regions are the task-transition overhead, and the grey regions are the remaining, unaccounted-for energy. The control instructions account for 26% of SONIC's energy, and a further 14% of system energy comes from FRAM writes to loop indices. Ideally, these overheads would be amortized across many kernel operations, but doing this requires a more efficient architecture.

10 Future intermittent architecture research

Our experience in building SONIC & TAILS demonstrates there is a large opportunity to accelerate intermittent inference via a parallel architecture with built-in support for intermittent operation. However, typical microcontrollers for energy-harvesting systems are poorly suited to efficient inference, and we have identified several opportunities to significantly improve them with better hardware support. Current microcontrollers are sequential, single-cycle processors, and so spend very little of their energy on "useful work" [41]. For example, by deducting the energy of `nop` instructions from Fig. 12, we estimate that SONIC spends 40% of its energy on instruction fetch and decode. This cost is a waste in highly structured computations like DNN inference, where overheads easily amortize over many operations.

LEA should bridge this efficiency gap, but unfortunately LEA has many limitations. Invoking LEA is expensive. Each LEA invocation should therefore do as much work as possible, but LEA's parallelism is limited by its small (4KB) SRAM buffer. This small buffer also forces frequent DMA between SRAM and FRAM, which cannot be overlapped with LEA execution and does not support strided accesses or scatter-gather. LEA has surprising gaps in its support: it does not support vector left-shift or scalar multiply, forcing TAILS to fall back to software. In software, integer multiplication is a memory-mapped peripheral that takes four instructions and nine cycles. All told, these limitations cause SONIC & TAILS to spend much more energy than necessary. There is ample room to improve inference efficiency via a better architecture.

Thus far, architectures for intermittent computing have focused on how hardware can efficiently guarantee correctness [40, 52, 53]. While there is certainly scope for architectural support, correctness requires a full-stack approach.

Handling correctness in the architecture alone is insufficient because it ignores system-level effects, such as I/O (e.g., sensors and radios), data timeliness, and interrupts that must be part of an end-to-end correctness guarantee [17, 39]. Moreover, an architecture-only approach is energy-inefficient because it must conservatively back up architectural state in non-volatile memory after each instruction. Software can instead identify precisely what state is needed for correctness (e.g., loop indices in SONIC). We therefore see more opportunity in targeted architectural support (e.g., caches with just-in-time checkpointing to avoid frequent, expensive writes to non-volatile memory for index variables), than in conservative models that ask nothing of software [52, 53].

Furthermore, to enable compute-heavy applications like inference and signal processing, future intermittent architectures must aggressively optimize for energy efficiency. The key is to eliminate or amortize wasted energy (e.g., in fetch, decode, register file, and FRAM)—we estimate that a new architecture would save 14% of system energy just by eliminating frequent FRAM writes to loop indices alone!

Intermittent architectures must navigate several fundamental design challenges to optimize energy efficiency. Highly specialized architectures (e.g., ASICs) are the most efficient, but sacrifice programmability. Such specialization is premature in intermittent computing systems because the dominant applications in this domain are yet to be determined; programmability remains essential. Programmable architectures can achieve ASIC-like efficiency on highly parallel codes by amortizing energy spent across many in-flight operations. Unfortunately, this requires high power and large amounts of state [19], both of which are non-starters in energy-harvesting systems. Hence, a balance of modest specialization and SIMD parallelism is needed to maximize energy-efficiency [20, 33]. We are currently exploring an intermittent parallel architecture inspired by streaming dataflow models [31, 63, 65, 71], striking an appealing balance between programmability, parallelism, and specialization to maximize efficiency without compromising the architecture's ability to provide correctness guarantees.

11 Conclusion

This paper has argued that intelligence "beyond the edge" will enable new classes of IoT applications, and presented the first demonstration of efficient DNN inference on commodity energy-harvesting systems. We presented a high-level analysis of why inference accuracy matters, and used this analysis to automatically compress networks to maximize end-to-end application performance. SONIC & TAILS then specialize intermittence support to guarantee correct execution, regardless of power system, while reducing overheads by up to 6.9× and 12.2×, respectively, over the state-of-the-art.

12 Acknowledgements

We thank the anonymous ASPLOS reviewers for their useful feedback. This work was generously funded by National Science Foundation Awards CCF-1815882 and CCF-1751029.

References

[1] [n. d.]. Low Energy Accelerator FAQ. http://www.ti.com/lit/an/slaa720/slaa720.pdf

[2] [n. d.]. MSP430fr5994 SLA. http://www.ti.com/lit/ds/symlink/msp430fr5994.pdf

[3] [n. d.]. Nividia Jetson TX2. https://developer.nvidia.com/embedded/develop/hardware

[4] [n. d.]. Powercast P2110B. http://www.powercastco.com/wp-content/uploads/2016/12/P2110B-Datasheet-Rev-3.pdf

[5] [n. d.]. Powercaster Transmitter. http://www.powercastco.com/wp-content/uploads/2016/11/User-Manual-TX-915-01-Rev-A-4.pdf

[6] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 1–13.

[7] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.

[8] Angus Galloway. 2018. Tensorflow XNOR-BNN. https://github.com/AngusG/tensorflow-xnor-bnn.

[9] Sourav Bhattacharya and Nicholas D Lane. 2016. Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM, 176–189.

[10] Michael Buettner, Ben Greenstein, and David Wetherall. 2011. Dewdrop: An Energy-Aware Task Scheduler for Computational RFID. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[11] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proc. of the 19th intl. conf. on Architectural Support for Programming Languages and Operating Systems*.

[12] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. 2014. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 609–622.

[13] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*.

[14] François Chollet. [n. d.]. Xception: Deep learning with depthwise separable convolutions. ([n. d.]).

[15] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. *SIGOPS Oper. Syst. Rev.* 50, 2 (March 2016), 577–589. https://doi.org/10.1145/2954680.2872409

[16] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*.

[17] Alexei Colin, Emily Ruppel, and Brandon Lucia. 2018. A Reconfigurable Energy Storage Architecture for Energy-harvesting Devices. In *ASPLOS*.

[18] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[19] David E Culler et al. 1988. Resource requirements of dataflow programs. In *ACM SIGARCH Computer Architecture News*, Vol. 16. IEEE Computer Society Press, 141–150.

[20] William J Dally, James Balfour, David Black-Shaffer, James Chen, R Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. 2008. Efficient embedded computing. *Computer* 41, 7 (2008).

[21] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications* 21, 4 (2000), 1253–1278.

[22] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. *SIAM journal on Matrix Analysis and Applications* 21, 4 (2000), 1324–1342.

[23] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. 2017. Understanding and Optimizing Asynchronous Low-Precision Stochastic Gradient Descent. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*.

[24] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. 2017. CirCNN: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 395–408.

[25] Adwait Dongare, Craig Hesling, Khushboo Bhatia, Artur Balanuta, Ricardo Lopes Pereira, Bob Iannucci, and Anthony Rowe. 2017. OpenChirp: A low-power wide-area networking architecture. In *Pervasive Computing and Communications Workshops (PerCom Workshops), 2017 IEEE International Conference on*. IEEE, 569–574.

[26] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *Proc. of the 42nd annual Intl. Symp. on Computer Architecture (Proc. ISCA-42)*.

[27] Hussein Elnawawy, Mohammad Alshboul, James Tuck, and Yan Solihin. 2017. Efficient Checkpointing of Loop-Based Codes for Non-volatile Main Memory. In *Parallel Architectures and Compilation Techniques (PACT), 2017 26th International Conference on*. IEEE, 318–329.

[28] L. Fick, D. Blaauw, D. Sylvester, S. Skrzyniarz, M. Parikh, and D. Fick. 2017. Analog in-memory subthreshold deep neural network accelerator. In *2017 IEEE Custom Integrated Circuits Conference (CICC)*. 1–4. https://doi.org/10.1109/CICC.2017.7993629

[29] Graham Gobieski, Nathan Beckmann, and Brandon Lucia. 2018. Intermittent Deep Neural Network Inference. In *SysML*.

[30] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D Sculley. 2017. Google vizier: A service for black-box optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1487–1495.

[31] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. 2012. Dyser: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro* 32, 5 (2012), 38–51.

[32] Chirag Gupta, Arun Sai Suggala, Ankit Goyal, Harsha Vardhan Simhadri, Bhargavi Paranjape, Ashish Kumar, Saurabh Goyal, Raghavendra Udupa, Manik Varma, and Prateek Jain. 2017. ProtoNN: Compressed and Accurate kNN for Resource-scarce Devices. In *International Conference on Machine Learning*. 1331–1340.

[33] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, Vol. 38. ACM, 37–47.

[34] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pdream, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proc. of the 43rd annual Intl. Symp. on Computer Architecture (Proc. ISCA-43)*.

[35] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization, and Huffman Coding. In *Proc. of the 5th Intl. Conf. on Learning*

Representationas (Proc. ICLR'16).

[36] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. 2016. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys '16)*. ACM, New York, NY, USA, 216–229. https://doi.org/10.1145/2994551.2994554

[37] Josiah Hester, Lanny Sitanayah, and Jacob Sorber. 2015. Tragedy of the Coulombs: Federating Energy Storage for Tiny, Intermittently-Powered Sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys '15)*. ACM, New York, NY, USA, 5–16. https://doi.org/10.1145/2809695.2809707

[38] Josiah Hester and Jacob Sorber. [n. d.]. Flicker: Rapid Prototyping for the Batteryless Internet of Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*.

[39] Josiah Hester, Kevin Storer, and Jacob Sorber. [n. d.]. Timely Execution on Intermi!ently Powered Ba!eryless Sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SenSys '17)*.

[40] Matthew Hicks. 2017. Clank: Architectural Support for Intermittent Computation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 228–240. https://doi.org/10.1145/3079856.3080238

[41] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*. IEEE, 10–14.

[42] Andrey Ignatov. [n. d.]. HAR. https://github.com/aiff22/HAR

[43] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).

[44] H. Jayakumar, A. Raha, and V. Raghunathan. 2014. QuickRecall: A Low Overhead HW/SW Approach for Enabling Computations across Power Cycles in Transiently Powered Computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*.

[45] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760* (2017).

[46] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.

[47] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 461–475. https://doi.org/10.1145/3173162.3173176

[48] Yann Le Cun, LD Jackel, B Boser, JS Denker, HP Graf, I Guyon, D Henderson, RE Howard, and W Hubbard. 1989. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine* 27, 11 (1989), 41–46.

[49] Yann LeCun. 1998. The MNIST database of handwritten digits. *http://yann. lecun. com/exdb/mnist/* (1998).

[50] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[51] Brandon Lucia and Benjamin Ransford. 2015. A Simpler, Safer Programming and Execution Model for Intermittent Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2015)*. ACM, New York, NY, USA, 575–585. https://doi.org/10.1145/2737924.2737978

[52] Kaisheng Ma, Xueqing Li, Jinyang Li, Yongpan Liu, Yuan Xie, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. 2017. Incidental computing on IoT nonvolatile processors. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 204–218.

[53] Kaisheng Ma, Yang Zheng, Shuangchen Li, Karthik Swaminathan, Xueqing Li, Yongpan Liu, Jack Sampson, Yuan Xie, and Vijaykrishnan Narayanan. 2015. Architecture exploration for ambient energy harvesting nonvolatile processors. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 526–537.

[54] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent Execution without Checkpoints. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, Vancouver, BC, Canada.

[55] Kiwan Maeng and Brandon Lucia. 2018. Adaptive Dynamic Checkpointing for Safe Efficient Intermittent Computing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 129–144. http://dl.acm.org/citation.cfm?id=3291168.3291178

[56] J. San Miguel, K. Ganesan, M. Badr, and N. E. Jerger. 2018. The EH Model: Analytical Exploration of Energy-Harvesting Architectures. *IEEE Computer Architecture Letters* 17, 1 (Jan 2018), 76–79. https://doi.org/10.1109/LCA.2017.2777834

[57] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. 2013. Idetic: A High-level Synthesis Approach for Enabling Long Computations on Transiently-powered ASICs. In *IEEE Pervasive Computing and Communication Conference (PerCom)*. http://aceslab.org/sites/default/files/Idetic.pdf

[58] Thomas M. Mitchell. 1997. *Machine Learning* (1 ed.). McGraw-Hill, Inc., New York, NY, USA.

[59] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *arXiv preprint arXiv:1712.05889* (2017).

[60] Tarek M Nabhan and Albert Y Zomaya. 1994. Toward generating neural network structures for function approximation. *Neural Networks* 7, 1 (1994), 89–99.

[61] Saman Naderiparizi, Zerina Kapetanovic, and Joshua R. Smith. 2016. WISPCam: An RF-Powered Smart Camera for Machine Vision Applications. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'16)*. ACM, New York, NY, USA, 19–22. https://doi.org/10.1145/2996884.2996888

[62] Preetum Nakkiran, Raziel Alvarez, Rohit Prabhavalkar, and Carolina Parada. 2015. Compressing deep neural networks using a rank-constrained topology.. In *Sixteenth Annual Conference of the International Speech Communication Association*.

[63] Tony Nowatzki, Vinay Gangadhar, and Karthikeyan Sankaralingam. 2015. Exploring the potential of heterogeneous von neumann/dataflow execution models. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 298–310.

[64] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks. In *Proc. of the 44th annual Intl. Symp. on Computer Architecture (Proc. ISCA-44)*.

[65] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel patterns. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 389–402.

[66] M. Price, J. Glass, and A. P. Chandrakasan. 2018. A Low-Power Speech Recognizer and Voice Activity Detector Using Deep Neural Networks. *IEEE Journal of Solid-State Circuits* 53, 1 (Jan 2018), 66–75. https://doi.org/10.1109/JSSC.2017.2752838

[67] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System Support for Long-Running Computation on RFID-Scale Devices. In *ASPLOS*.

[68] Ao Ren, Zhe Li, Caiwen Ding, Qinru Qiu, Yanzhi Wang, Ji Li, Xuehai Qian, and Bo Yuan. 2017. Sc-dcnn: highly-scalable deep convolutional neural network using stochastic computing. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 405–418.

[69] Tara N Sainath and Carolina Parada. 2015. Convolutional neural networks for small-footprint keyword spotting. In *16th Annual Conference of the International Speech Communication Association*.

[70] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. 2008. Design of an RFID-Based Battery-Free Programmable Sensing Platform. *IEEE Transactions on Instrumentation and Measurement* 57, 11 (Nov. 2008), 2608–2615.

[71] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W Keckler, and Charles R Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. IEEE, 422–433.

[72] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[73] M. Song, K. Zhong, J. Zhang, Y. Hu, D. Liu, W. Zhang, J. Wang, and T. Li. 2018. In-Situ AI: Towards Autonomous and Incremental Deep Learning for IoT Systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 92–103. https://doi.org/10.1109/HPCA.2018.00018

[74] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning.. In *AAAI*, Vol. 4. 12.

[75] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 1–9.

[76] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. 2015. Going deeper with convolutions. Cvpr.

[77] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2818–2826.

[78] TI Inc. 2014. Overview for MSP430FRxx FRAM. http://ti.com/wolverine. Visited July 28, 2014.

[79] Ledyard R Tucker. 1966. Some mathematical notes on three-mode factor analysis. *Psychometrika* 31, 3 (1966), 279–311.

[80] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*. 17.

[81] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 1–12.