The Case for Approximate Intermittent Computing

Fulvio Bambusi*, Francesco Cerizzi*, Yamin Lee*, and Luca Mottola*+†
*Politecnico di Milano (Italy), †RI.SE Sweden, +Uppsala University (Sweden)

ABSTRACT

We present the concept of *approximate intermittent computing* and concretely demonstrate its application. Intermittent computations stem from the erratic energy patterns caused by energy harvesting: computations unpredictably terminate whenever energy is insufficient and the application state is lost. Existing solutions maintain equivalence to continuous executions by creating persistent state on non-volatile memory, enabling *stateful computations to cross power failures*. The performance penalty is massive: system throughput reduces while energy consumption increases. In contrast, approximate intermittent computations trade the accuracy of the results for sparing *the entire overhead* to maintain equivalence to a continuous execution. This is possible as we use approximation to *limit the extent of stateful computations to the single power cycle*, enabling the system to *completely* shift the energy budget for managing persistent state to useful computations towards an *immediate* approximate result. To this end, we effectively reverse the regular formulation of approximate computing problems. First, we apply approximate intermittent computing to human activity recognition. We design an anytime variation of support vector machines able to improve the accuracy of the classification as energy is available. We build a hw/sw prototype using kinetic energy and show a 7x improvement in system throughput compared to state-of-the-art system support for intermittent computing, while retaining 83% accuracy in a setting where the best attainable accuracy is 88%. Next, we apply approximate intermittent computing in a sharply different scenario, that is, embedded image processing, using loop perforation. Using a different hw/sw prototype we build and diverse energy traces, we show a 5x improvement in system throughput compared to state-of-the-art system support for intermittent computing, while providing an equivalent output in 84% of the cases.

1 INTRODUCTION

Ambient energy harvesting enables battery-less embedded sensing [1, 30, 35, 36, 38, 65, 69]. However, energy from the environment is generally erratic, causing frequent and unanticipated power failures. For example, harvesting energy from RF transmissions to compute a simple CRC may lead to 16 power failures over a 6 sec period [13]. Executions thus become *intermittent*, as they consist of intervals of active computation interleaved by possibly long periods of recharging energy buffers [13].

Prior art. Due to resource constraints, power failures normally cause a device to lose the application state. To ensure forward progress across power failures, a variety of techniques exists that make use of *persistent state* stored on non-volatile memory (NVM), as we elaborate in Sec. 2. Persistent state is loaded back from NVM when energy is back, so executions resume closer to the point of power outage rather than performing a complete reboot.

Most existing solutions [3, 8, 9, 14, 19, 42, 44, 45, 51, 62, 68, 71] aim to make intermittent executions equivalent to their continuous counterparts, as we articulate in Sec. 2. Given the same inputs, for example, certain sensor readings, the results of intermittent executions must be exactly the same as those of a continuous one. To achieve this, existing solutions employ persistent state to allow *stateful computations* to cross power failures. The price for this is enormous, in both *energy* consumption and latency until a result is available, and thus system *throughput*. The energy overhead may reach up to 350% the cost of the application processing, mainly due to the use of energy-hungry NVM technology [68]. Depending on energy patterns and the time for energy buffers to recharge, a 10 ms processing in a continuous execution may take minutes in an intermittent one [36], reducing throughput.

Nonetheless, using persistent state to allow stateful computations to cross power failures has further implications. In mixed-volatile platforms [39], slices of main memory are mapped to NVM, for example, with FRAM technology. As a result, intermittence anomalies [50, 61] appear due to re-execution of non-idempotent code, requiring additional time and energy to be corrected. When using FRAM, wait cycles may be necessary to synchronize read/write operations with the microcontroller, further increasing energy overhead [39]. Finally, if the state of computations is to be retained across power failures, the state of peripherals must also be accounted for [6, 12, 16, 46]. This increases the size of the persistent state, as it must include information related to peripheral states that may not be reflected in the systems' main memory, adding to the energy overhead.

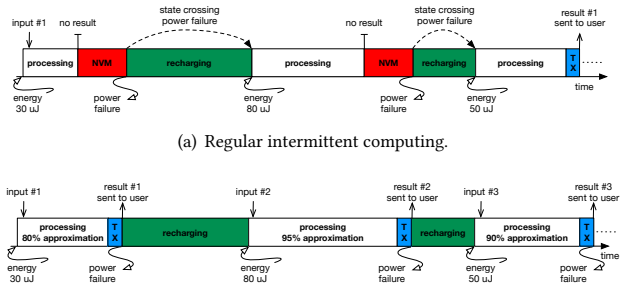
Approximate intermittent computing. Our work starts from the observation that a number of embedded sensing applications, as in smart health [66], ambient intelligence [22], and environment monitoring [57], expose two specific characteristics:

1) Latency to obtain a result is key; a fitness tracker must process input signals as rapidly as possible, because the outcome might require immediate reactions. The longer it takes for a sample to be processed, the lower is the value of the analysis.
2) Newer inputs are more important than older ones. A fitness tracker should process the most recent samples first, as they are representative of the current situation. Any further processing may merely represent a "best-effort" task.

Most importantly, *approximate results* are often tolerated. This stems from the nature of data processing in these applications, including computer vision, machine learning, signal processing, and pattern identification. These algorithms offer probabilistic guarantees in the first place and are robust to data errors, for example, due to sensor inaccuracies [53].

These observations prompt us to establish a concept of *approximate intermittent computing*. In regular intermittent computing, shown in Fig. 1(a), every input is processed until the precise outcome is eventually computed. Because processing of an input rarely concludes before the first power failure, persistent state is employed

978-1-6654-9624-7/22/$31.00 ©2022 IEEE
DOI 10.1109/IPSN54338.2022.00044



(a) Regular intermittent computing.

(b) Approximate intermittent computing.

Figure 1: Approximate intermittent computing trades the accuracy of the final result for the time and energy required to allow stateful computations to cross power failures. *Such a shift allows applications to completely employ the available energy for useful application processing.*

to make stateful computations cross power failures. This means that the latency to return a result to the user, for example, by transmitting a packet, extends over multiple power cycles and crucially includes the potentially large periods to recharge energy buffers. While the system goes back and forth between main memory and NVM to dump and re-load application state for crossing power failures, many newer inputs are missed.

In contrast, as shown in Fig. 1(b), approximate intermittent computing dictates that whenever a precise result cannot be obtained within a single power cycle, the system shall work towards an *immediate approximate outcome*. Approximation is the knob the system exploits to reduce the energy costs so that a result can be returned to the user *before* the first power failure. *As a result, approximation allows us to shrink the extent of stateful computations to the single power cycle.* Based on available energy, the system dynamically selects a level of approximation to process the current sample that ensures precisely this. This means that, instead of seeking the most efficient way to make stateful computations cross power failures, as most existing solutions do, we effectively *remove the problem* in the first place, as stateful computations are bound to conclude before the first power failure.

Our approach is fundamentally different, however, than the regular application of approximate computing techniques. Indeed, application requirements normally dictate rigid lower bounds on the accuracy of results [32, 53]. Approximate algorithms aim at ensuring this minimum accuracy by saving the greatest amount of resources, for example, energy. In our case, the problem is effectively reversed. Applications may not impose minimum accuracy requirements and be satisfied with whatever is attainable, whereas *a strict upper bound on energy consumption* exists due to the finite energy buffers. Approximate algorithms must accordingly be designed to attain the greatest accuracy within a finite energy budget.

The ability to shrink stateful computations to the single power cycle entails that *no persistent state needs to be carried over to the next power cycle*, and available energy is *exclusively employed for useful computations*, rather than for NVM operations; therefore:

1) the system is *ready to process new inputs as soon as it restarts* after a power failure, as previous iterations necessarily concluded;
2) *intermittence anomalies are not an issue*, because re-executions that combine volatile and non-volatile data do not occur;
3) *handling peripheral states across power failures is unnecessary*, as computations always resume from the same point in the code;
4) as there is no persistent state to maintain, devices *need not be equipped with NVM* to run intermittent programs.

The key trade-off we explore is the reduction in accuracy against the energy savings obtained by sparing the need of persistent state and the improvements in the latency to return the result to the user. Latency is here, nonetheless, inversely proportional to system throughput. The very notion of accuracy, however, along with the way it is measured and the definition of acceptable bounds, is inherently application-specific. This is *not only germane to our work*, but a general characteristic of approximate computing [53].

Concrete cases. We first consider the case of human activity recognition [4] using on-body acceleration and angular velocity sensors. This application shows the characteristics discussed earlier, In particular, an accurate report of activity state, received much later than when the corresponding sensor data is originally gathered, is often useless. Reducing the latency from the acquisition of sensor data to producing the output is key. Further, human activity recognition is an ideal candidate for energy harvesting [33, 63], as batteries are detrimental to user experience and increase a device's footprint. Machine learning techniques, such as support vector machines [15], are often employed to perform activity recognition [4].

We develop an *anytime* variation of support vector machines, described in Sec. 3, where accuracy of the classification is improved incrementally by processing one feature of the input samples at a time. This is the knob that determines the level of approximation versus the energy cost of processing; the more features are processed, the more accurate is the classification, but also the higher is the energy cost. Limiting the extent of stateful computations using anytime support vector machines requires to tie the number of processed features to the expected accuracy; we study this aspect analytically depending on the statistical nature of input data and the number of output classes.

Running classification tasks on resource-constrained embedded devices also requires additional efforts to fit a complex processing pipeline within a limited processing and memory budget [29, 40]. In Sec. 4, we report on the prototype we build, including the customized hardware, the off-line data processing, and two alternative software implementations. As we spare the need of persistent state, our prototype is the first device we are aware of to run real-world intermittent programs with no use of NVM. The two implementations showcase the spectrum of possibilities offered by approximate intermittent computing. One implementation employs available energy greedily, producing the most accurate result within the available energy budget. The other implementation allows developers to set a lower bound in accuracy and postpones processing until the current energy budget ensures the required accuracy.

Our evaluation of approximate intermittent computing applied to human activity recognition, reported in Sec. 5, is based on a combination of emulation and real-world experiments involving a total of 15 volunteers across a total of 24 days, producing ≈842 hours of experiment data. By comparing the performance in accuracy and system throughput, we show that approximate intermittent

computing provides a 7x improvement in system throughput compared to state-of-the-art system support for regular intermittent computing, while retaining an 83% accuracy in a setting where the best attainable accuracy is 88%. Moreover, with approximate intermittent computing, returning the result to the user always occur within the same power cycle *by design*; with regular intermittent computing instead, the time when the classification is returned is entirely a function of energy patterns, extends across tens of power cycles, and includes the times to recharge the energy buffer.

Following the same path as in the case of human activity recognition, Sec. 6 demonstrates the applicability of approximate intermittent computing to a sharply different application, namely, embedded image processing. We employ loop perforation techniques [53] as the knob to trade a loss of accuracy for a reduction in energy cost, allowing the system to return a result to the user before the first power failure, and again sparing the need of persistent state. We build another hardware/software prototype, which we feed with diverse energy traces to explore the impact of various energy patterns on the accuracy of the output and energy consumption, compared to the same state-of-the-art system support for regular intermittent computing. In this setting, approximate intermittent computing achieves a 5x improvement in system throughput, while providing an equivalent output in 84% of the cases.

We release as open-source [11] the software artifacts and hardware schematics concurring to the results we present next, enabling others to continue exploring approximate intermittent computing.

2 RELATED WORK

Our combines intermittent computations with a novel use of approximate computing techniques.

Intermittent computing. Most existing works [36] focus on how to make programs maintain the same semantics as a continuous execution, which requires stateful computations to cross power failures. Common to these works is the use of some form of persistent state on NVM. Two flavors exist.

Some solutions employ a form checkpointing [3, 8, 9, 14, 45, 62, 68]. This consists in replicating the application state on NVM at specific points in the code, where it is retrieved back once the system resumes with sufficient energy. Systems such as Hibernus [8, 9] operate based on interrupts fired from a hardware device that prompts the application to take a checkpoint, for example, whenever the energy level falls below a threshold. Differently, systems exist that place function calls in application code to proactively checkpoint [14, 45, 62, 68]. The specific placement is a function of program structure and energy provisioning patterns.

Other approaches offer abstractions that programmers use to define and manage persistent state [19, 42, 44, 51, 71] and time profiles [37]. These approaches particularly target mixed-volatile platforms, while taking care of intermittence anomalies due to repeated executions of non-idempotent code [50, 61]. For example, Alpaca [44] defines tasks as individual execution units that run with transactional semantics against power failures and subsequent reboots, and channels to exchange data across tasks.

Approximate intermittent computing represents a different design standpoint. Rather than retaining equivalence to continuous executions at the cost of making stateful computations cross power failures, we trade accuracy in the final results for better energy efficiency, making it possible to shrink the extent of stateful computations to the single power cycle. Unlike a few existing works that explored similar directions [28, 43], we push this design to the point of taking away the need of persistent state, thus allowing systems to burn energy exclusively for useful application processing, while not requiring custom hardware. Approximate intermittent computing is applicable where inaccurate results are tolerable, but is unfit for applications requiring precise computations, as discussed in Sec. 6.

Additional issues arising in the execution of intermittent programs require testing the executions [18, 20, 49] and profiling their energy consumption [2, 27]. These techniques and tools are orthogonal to our work and may be applied to approximate intermittent programs as well. We do rely on an existing energy estimation tool for intermittent programs in our prototypes of Sec. 4 and Sec. 6.

Approximate computing. The need to reduce resource consumption in data-intensive applications originally motivates the development of approximate algorithms, namely, algorithms sacrificing the accuracy in the final result to gain in key performance metrics, such as processing times, memory occupation, or energy consumption. A vast body of work exists on the subject [32, 53].

From an algorithmic standpoint, our work is not different than most existing literature in approximate computing. As the very notion of accuracy, the way it is measured, and the definition of acceptable bounds are application-specific, the specific data processing techniques we employ are also necessarily tied to a specific class of applications. This is, in fact, one of the major limitations of approximate computing in general [32]. Nonetheless, these techniques are merely the specific instantiation of approximate intermittent computing we study here for the applications we consider. Other instantiations are also possible that target different classes of applications, as we argue in Sec. 6.

What is fundamentally different, however, is the formulation of the approximate computing problem, as we hinted earlier. Instead of dealing with rigid lower bounds on the accuracy of results dictated by application requirements [32, 53], we are to work with a fixed energy envelope, determined by the size of energy buffers. In our case, algorithms must improve the accuracy obtained within the finite energy budget, whatever that may be, rather than improving resource usage, such as energy, within the constraint of a minimum accuracy bound. The traditional design of approximate algorithms is therefore often inapplicable [32, 53], as the effort spent in computing is upper bound unlike in mainstream scenarios.

Output degradation in intermittent computing. Closest to our work are systems that adapt the application execution based on available energy, possibly degrading the quality of the output tin situations of energy scarcity. One example is CatNap [47], a programming model and run-time system that allows programmers to identify a subset of the code as time-critical. When available energy is lower than expected, CatNap defers the execution of the non time-critical code to ensure timely execution of the time-critical one. If the schedule becomes unfeasible in situations of extreme energy scarcity, CatNap further degrades the quality of the output by either running different programmer-provided code for the same functionality, or the same code but less frequently.

Works also exist that use multi-resolution inference and multi-exit strategies to handle a graceful degradation of the machine

learning performance to meet temporal deadlines or depending on available energy. For example, Sonic [29] provides specialized support for intermittent executions, Zygarde [40] aims at taking power failures into account to reduce the inference latency, whereas ePerceptive [55] builds upon Sonic by adding a dynamic multi-exit strategy. These works do represent initial steps in a similar direction as approximate intermittent computing, yet they do not seek to constrain the stateful processing steps within the same power cycle, and therefore still require the use of persistent state.

3 CASE IN POINT

We make a first concrete case for approximate intermittent computing using human activity recognition as a target application. Our choice is not casual. While human activity recognition is representative of the two characteristics discussed in the Introduction, it is also a nice fit for energy harvesting. Kinetic energy abounds on human bodies [33, 63], whereas batteries are detrimental to user experience, as they increase a device's footprint.

We specifically consider the study of Anguita et al. [4] in human activity recognition. The application consists in classifying acceleration and angular velocity readings as representative of six possible human activities, including walking, walking upstairs, walking downstairs, standing, sitting, and laying. Using a regular support vector machine, Anguita et al. obtain an absolute accuracy of 93.9%.

Albeit the design of anytime support machines we describe next is originally motivated by human activity recognition, their applicability extends beyond that. The underlying reasoning and mathematical properties *are not*, indeed, tied to the specific application.

3.1 Preliminaries

We provide necessary background to later understand our design.

Support vector machines. A geometrical analogy helps understand support vector machines, shown in Fig. 2. Consider the case of only two possible classes A and B, for simplicity. Support vector machines use training data to identify a hyperplane that provides the best distinction between the two classes. The hyperplane is the one that is maximally distant from the closest vector representing the input sample. These vectors are called support vectors. The separating hyperplane is commonly identified by solving a quadratic programming problem [54] using training data. The computational complexity of training is $O(n^3)$, with n being the cardinality of the training set. The output of the training phase is a tuple of coefficients with the same dimensionality as the input samples.

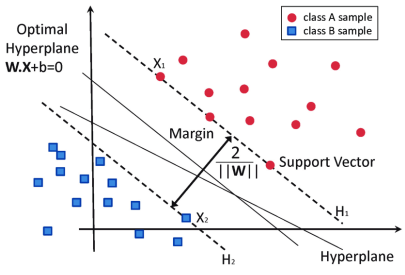


Figure 2: Geometrical representation of separating hyperplane and support vectors in two dimensions.

Support vector machines rely on the assumption that the data set is linearly separable, that is, at least one hyperplane W exists such that all the input samples in A are on one side of W, and all input samples in B are on the other. This is not necessarily the case. If the data set is not linearly separable, the data is projected in high-dimensionality spaces where it becomes so, using kernel functions [54]. These retain the applicability of support vector machines in cases where the original data is not linearly separable, but incur in additional memory consumption and processing.

Support vector machines are usable also when the classification task extends to multiple classes. Two methods are available. One-versus-rest (OvR) methods classify input samples as belonging to the class whose hyperplane is the farthest away. Instead, one-versus-one (OvO) methods use one classifier for each pair of classes C_i, C_j. The class that matches most frequently among all possible pairs is returned as output. As OvO methods require one hyperplane for each pair of classes and a corresponding comparison when classifying input samples, they incur in greater processing and memory cost in both training and classification. OvR methods are therefore usually favored on embedded devices.

Approximation for support vector machines. Approximate variations of support vector machines exist. For example, Decoste et al. [23] design interval-valued support vector machines. Whether an input sample $\boldsymbol{x}$ belongs to class C_1 or C_2 may be computed by determining the sign of

$$\sum_{p \in C_1} d_{xp}^2 - \sum_{q \in C_2} d_{xq}^2, \tag{1}$$

where d_{xp}^2 is the squared distance between the input sample $\boldsymbol{x}$ and the support vector $\boldsymbol{p}$. Instead of computing Eq. 1 directly, Decoste et al. [23] present an algorithm to compute corresponding bounds that are continuously narrowed until the sign of Eq. 1 is determined.

Wagstaff et al. [70] train two different models, one that is extremely accurate but incurs in high processing cost during classification, the other that is less precise but extremely simple to use. Input samples are first classified using the simple model, and the probability that the classification is correct is accordingly computed. In cases this falls below a threshold, the accurate model is used for re-running the classification.

Both approaches are, however, largely inapplicable for approximate intermittent computing. Interval-valued support vector machines incur, in the worst case, in greater processing cost than regular support vector machines [23]. As this is proportional to energy consumption on the devices we target, this characteristic defeats the very purpose of using approximations. The approach of Wagstaff et al. [70] offers little flexibility, as there are only two possibilities for tuning the classification accuracy: one is either happy with the simple model, or runs the accurate one as well.

3.2 Anytime Support Vector Machines

Like Anguita et al. [4], we use the OvR method. Say $\boldsymbol{w_1}, \boldsymbol{w_2} \ldots \boldsymbol{w_c}$ are the vectors representing the hyperplanes of the c classes. An input sample $\boldsymbol{x}$ may be classified by identifying the class corresponding to the hyperplane that yields the largest inner product with $\boldsymbol{x}$. Our claim is that we can achieve an approximate classification by using

fewer features than the n available, that is

$$\boldsymbol{w_i x} = \sum_{j=1}^{n} w_{ij} x_j \approx \sum_{j=1}^{p} w_{ij} x_j, \tag{2}$$

where $1 < i < c$ and $p < n$. Intuitively, we use only a subset of the available features to compute the classification. The closer is p to n, the more accurate is the classification, and viceversa.

The classification may then be computed incrementally, by caching approximate results and adding more features as energy is available. This is particularly beneficial whenever the additional features are not immediately available from sensor data, but require additional processing on the latter before they can be used for classification. By limiting the classification step to a subset of the features, we save the processing overhead of not just the classification step itself, but also of the computation required to compute the additional features from raw sensor data.

To employ this form of approximation in intermittent computing, the key question is how to tie the number of features we do process to the expected accuracy in the final classification. This essentially represents the trade-off between energy cost and accuracy. In the general case, we are interested in computing the probability that a classification using $p < n$ features is coherent with the one obtained using n features, that is, it is the same as the most accurate classification we can achieve. We want to study this probability as a function of p. We call $class_{mi}$ the classification of the i-th sample using m features. Therefore, we are to calculate, depending on p

$$P(class_{pi} = class_{ni}). \tag{3}$$

How to compute this probability depends on whether *i)* the features are independent or correlated, and *ii)* the classification is binary or extends to multiple classes. Without loss of generality and to ease the presentation, we illustrate next the case of independent features with two or more classes; the analytical details and derivations for the cases considering correlated features are available elsewhere [10] (Chapter 5, Section 5.1-5.4).

Let us consider two classes and independent features. When using all n available features, the classification of an input sample $\boldsymbol{x_i}$ is determined by the sign of

$$S_i = \boldsymbol{w x_i} = \sum_{j=1}^{n} c_j x_{ij}, \tag{4}$$

where $\boldsymbol{w} = [c_1, c_2, \ldots, c_n]$ is the vector representing the hyperplane used to classify the input samples, and $\boldsymbol{x_i} = [x_{i1}, x_{i2}, \ldots, x_{in}]$ is the i-th input sample we process.

An approximate classification for the i-th input sample may be obtained by computing the sign of

$$S_{ip} = \boldsymbol{w x_i} = \sum_{j=1}^{p} c_j x_{ij}, \quad p < n. \tag{5}$$

Note that the sign of Eq. 4 is coherent with that of Eq. 5 as long as

$$S_{ip} \geq -R_{ip} = -\sum_{j=p+1}^{n} c_j x_{ij}, \tag{6}$$

where R_{ip} represents the contribution to S_i of the features we are *not* considering. Intuitively, Eq. 6 states that such contribution is not sufficient to flip the sign of S_{ip} compared to S_i, and hence the two classifications are coherent.

Eq. 6 also suggests what are the features that should be processed first. In fact, for the same input sample, R_{ip} is as small as the features $p+1, \ldots, n$ correspond to smaller coefficients c_j. This entails that the most efficient order to process features should be based on magnitude of the corresponding coefficients in the separating hyperplane. Features with larger coefficients bear a stronger contribution in determining the final classification, and are therefore those we should process first. We confirm this observation experimentally in Sec. 5.

At run-time, the features corresponding to the larger coefficient may change; for example, because the distribution of the underlying data changes. One possibility to cater for this situation may be to periodically collect run-time data and re-run the training phase. This process may run on a back-end machine, hence not requiring resources from the sensing devices. Should the updated hyperplane information be sharply different than the original ones, we may issue an update for the sensing devices. Many solutions exist to perform this efficiently and securely [7].

In the case of independent and normally distributed coefficients $c_1, c_2, \ldots, c_n$, we eventually derive that

$$P(class_{pi} = class_{ni}) = 2 \int_{k=0}^{\infty} f_{S_{ip}}(k)(1 - F_{R_{ip}}(k))dk, \tag{7}$$

where $f_{S_{ip}}$ and $F_{R_{ip}}$ may be determined numerically [10] (Chapter 9, Appendix 1), making Eq. 7 cheap to compute.

The case of multiple classes follows as a natural extension. Say $C_1, C_2, \ldots, C_c$ are the possible classes, and $\boldsymbol{w_1}, \boldsymbol{w_2} \ldots \boldsymbol{w_c}$ are the vectors representing the corresponding hyperplanes. For a generic class h, Eq. 4 may be extended to the case of multiple classes as

$$S_{hi} = \boldsymbol{w_h x_i} = \sum_{j=1}^{n} c_{hj} x_{ij}. \tag{8}$$

The input sample i is classified as belonging to class C_h when using all n available features such that

$$class_{ni} = argmax_h(S_{hi}), \quad 1 \leq h \leq c. \tag{9}$$

It is possible to repeat the same reasoning of Eq. 6 for each individual class C_h compared to all others. Therefore, in the case of independent and normally distributed coefficients, the probability that the classification using $p < n$ features is coherent with the one obtained using n features is given by Eq. 7 for a generic class C_h, multiplied by the probability that h is precisely the one solving Eq. 9. In this case as well, we eventually derive an expression that may computed numerically [10] (Chapter 5, Section 5.4.3; Chapter 9, Section 9.2). It is also possible to derive similar expressions in the case of correlated coefficients, by taking into account the corresponding covariance matrix [10] (Chapter 5, Section 5.4.4; Chapter 5, Section 5.4.5; Chapter 9, Section 9.2.1). In this case too, the value of the expressions may be computed numerically.

4 PROTOTYPE

We describe the hardware we build, the data processing for training and classification, and how we implement two alternative classification pipelines on resource-constrained devices.

4.1 Hardware

Figure 3: ReVibe modelQ kinetic energy harvester.

We manufacture a custom board hosting an MSP430-FR5659 MCU to perform human activity recognition using kinetic energy. The MCU is equipped with 64Kb of volatile RAM together with 512Kb of FRAM as NVM. We use the FRAM exclusively when running regular intermittent computing techniques. Acceleration and angular velocity readings are obtained through an Analog Devices ADXL362 accelerometer and an STM Electronics L3GD20H gyroscope respectively, both connected through SPI. The board features BLE connectivity with a Nordic nRF51822 chip.

For energy harvesting, we use a ReVibe modelQ [25] kinetic transducer, shown in Fig. 3, which we order with a customized resonance frequency based on the spectral profile of raw accelerometer data we gather in a short pre-deployment trial. We choose the modelQ over alternatives, for example, the modelD [24] of the same manufacturer, because of the smaller form factor and the higher power output at the target frequencies.

Similar to existing deployments of intermittent computing [1, 30, 35], the harvester is attached to a BQ25505 combined booster and buck converter that charges a 1470μF capacitor we use as energy buffer. We determine its size through a mixed analytical and experimental approach [67], striking a balance between charging times and available energy. A too large capacitor may take long to charge to a sufficient level, yielding large periods of no system operation. A too small capacitor may not suffice to supply enough energy for worst-case processing scenarios. Whenever required, the capacitor's current charge is read through an Analog Devices ultralow-power LTC1417 analog-to-digital converter.

We create 12 identical hardware prototypes. The device is admittedly large, but still wearable with no major issues by the volunteers involved in the trial of Sec. 5. The computing board including MCU, BLE transceiver, charging circuitry, and application sensors measures 5cm x 6cm. This part of the device is easy to miniaturize [48, 67]. The key for practical usage is thus the kinetic energy harvester, shown in Fig. 3, which is roughly half the size of a AA battery. It can therefore replace traditional energy sources by reducing the overall device footprint and yet providing zero-maintenance operation [1]. Even smaller thermoelectric energy harvesters also exist [48, 67] that offer comparable energy performance.

4.2 Data and Training

We use the original dataset of Anguita et al. [5] for training. They sample acceleration and angular velocity readings at 50 Hz from a group of 30 volunteers within an age interval of 19-48 years, each following a predetermined protocol of activities while carrying a smartphone. The experiments are video-recorded to facilitate the data labeling. A 3rd order Butterworth filter with a cutoff frequency of 20Hz is used to remove the noise, as 99% of the signal energy is found below that [4]. A second low-pass filter accounts for gravity. Learning is then accomplished in the regular way, using the SVM Python library from the `scipy` package.

Out of the raw data, Anguita et al. [4] compute a total of 561 features for training and classification. Not all of these features are, however, generating linearly separable samples, therefore demanding the use of kernel functions. Because of the increased overhead due to these, we limit ourselves to the 140 signal features that generate linearly separable samples. Together with the specific implementation techniques we employ, this also ensures that in a continuous execution, all 140 features may be used for classification before the new sensor readings are gathered. This represents the most accurate classification we can possibly provide. The features we compute range from simple window operators such as average and standard deviation, to sophisticated ones such as fast Fourier transforms and spectral density distributions.

Based on Sec. 3.2, we numerically compute the probability that a classification using only $p < 140$ features is coherent with the classification obtained with all 140 available features. For each feature, we use energy estimation tools for intermittent computing [2] to profile the energy necessary to add that specific feature to the existing classification. Such an energy cost is fixed for a feature, but varies across features mainly because of the processing to extract the feature from the raw sensor readings.

The entire data processing and energy profiling run on a standard desktop machine in less than an hour.

4.3 Software

We implement the classification pipeline using C/C++. The classification process starts as soon as a new window of sensor samples is gathered. We create two implementations:

Greedy. The Greedy implementation continues to add features to the existing classification, progressively refining the accuracy, until either just the right amount of energy is left to send out a BLE packet with the 1-byte output, or all available features are used. In the latter case, a BLE packet is generated thereafter and the node switches to the lowest-power mode available that allows the system to wake up again in one minute for the next iteration of sensor sampling. No issue arises if the system dies because of energy depletion at this stage; the result is already returned to the user.

Smart. Based on the information provided by the offline phases, the Smart implementation first determines whether the available energy is sufficient to achieve a classification accuracy above a user-defined threshold A and finds in a look-up table the corresponding number p' of features to be used. If energy is insufficient, it skips this round of classification and switches to the lowest-power mode, similar to Greedy, waiting for the next sensor samples. Otherwise, it immediately uses all p' samples and then switches to Greedy mode.

Note that the operation of the Smart implementation ensures that the user-defined threshold is met for all input samples that are actually processed. It also ensures that any energy left or obtained while running is employed to further refine the accuracy of the classification before returning it to the user.

Both implementations employ fixed-point arithmetics due to the lack of hardware support for floating-point operations on the MCU we target and space-efficient data structures to store *i)* the models output by the training phase described by hyperplane information,

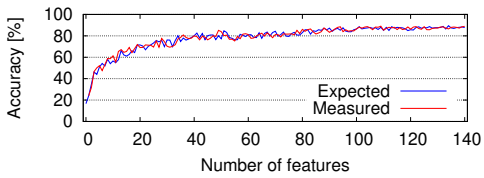


Figure 4: Emulation experiments: expected and measured accuracy as a function of the number of features used for classification. *The expected accuracy computed according to Sec. 3.2 is constantly close to the measured accuracy. The analysis of Sec. 3.2 can, indeed, be used to forecast the accuracy as a function of the number of features used for classification. Using all available features, the accuracy is around 88%, in line with the results of Anguita et al. [4].*

ii) the mapping between the p processed features to the expected classification accuracy, and *iii)* information on the energy cost to add the i-th feature to the existing classification. The information for *ii)* is essentially the result of numerically computing the right-hand side of Eq. 7 for a given p; the actual computation is performed off-line and only the result is stored in the MCU memory. The information for both *ii)* and *iii)* are compacted in a single lookup table that ensures $O(1)$ access overhead given a minimum classification accuracy A to achieve, as used in the Smart implementation, or the next feature i' to process, as used in Greedy. These information occupy ≈18Kb of the 64Kb of available memory, leaving ample room for additional functionality if necessary.

5 EVALUATION

We consider four key metrics. The *accuracy* of classification represents the matching between recognized human activities and ground truth, whenever available. This metric tops at the accuracy provided by an execution running without interruptions, which always uses all available features. We call this baseline Uninterrupted. Whenever ground truth is not available, we measure the *coherence* of the classification returned by approximate intermittent computing against Uninterrupted or regular intermittent computing, as discussed in Sec. 3. The system *throughput* measures the number of returned classifications throughout an experiment duration, whereas the *latency* indicates when those classifications are emitted compared to when the sensor samples are acquired.

For approximate intermittent computing, we study the performance of the Smart implementation, using a 80% or 60% lower bound in accuracy, and of Greedy, as described in Sec. 4. In addition to Uninterrupted, we use Chinchilla as a baseline [45]. Chinchilla over-provisions code with checkpoints to ensure forward progress with scarce energy, then dynamically disables checkpoints to adapt to situations of energy abundance. Because of this, Chinchilla efficiently matches the varying energy levels in our target scenarios. We also clock the MCU at 8MHz to avoid wait states when writing or reading checkpoints on FRAM. The performance we measure for this baseline thus represents a best case.

We study the relevant trade-offs from multiple angles, using different tools and settings:

(1) in Sec. 5.1, we use emulation experiments to compare the expected and measured accuracy of classification as a function of the number of features used for classification, providing quantitative support to the analysis in Sec. 3.2;
(2) in Sec. 5.2, we use emulation experiments to compare the latency, accuracy, and throughput of the implementations in Sec. 4 against either Uninterrupted or Chinchilla;
(3) in Sec. 5.3, we involve six volunteers for about 56 hours each to compare the latency, accuracy, and throughput of the implementations in Sec. 4 against Uninterrupted, using two identical devices on the same person's wrist;
(4) in Sec. 5.4, we use the same setup as the previous case with another six volunteers for about 58 hours each to run a comparison against an implementation using Chinchilla.

The volunteers we involve include senior members of our lab and their spouses or parents. The diversity of activities they are involved in, ranging from coding or studying to driving or exercising, caters for a range of different settings. The experiments in Sec. 5.1 and Sec. 5.2 are enabled by labeled sensor data and energy traces we collect with three of these volunteers using a battery-powered version of the prototype of Sec. 4 for about 56 hours each. Note that these data is different from the dataset used for training. The emulation experiments use an extension of the MSPSim emulator [26, 27] that provides support for using FRAM as NVM and accounts for the corresponding energy consumption. Experiments using the real prototype output the classification over BLE to a smartphone the user carries. The Uninterrupted executions are obtained with same battery-powered version of our prototype mentioned earlier.

5.1 Expected Accuracy

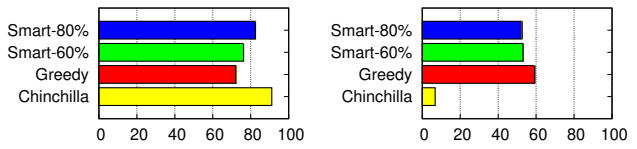
The analysis of Sec. 3.2 provides a conceptual and quantitative basis for the application of approximate intermittent computing. Here we check that this analysis can indeed be relied upon, using the labeled data we collect, including ground truth.

Fig. 4 shows the results of our emulation experiments comparing the expected and measured accuracy of classification, as a function of the number of processed features. The expected accuracy, computed based on the analysis of Sec. 3.2, is constantly close to the measured accuracy. The delta between the curves also appears largely independent of the number of features processed, providing evidence of the general applicability of our analysis.

Fig. 4 also offers a more general opportunity to gain a qualitative insight into the behavior of approximate intermittent computing. As expected, the blue curve starts at 16,6% because with no features, determining the correct classification equates to a random event with uniform distribution over the six possible classes. As the number of features we process increases, both curves rapidly grow. The first few features, which in our case come from processing the FFT of the input signal, significantly contribute to improving the accuracy of classification. The curves eventually flatten out as the contribution of the latest features we process only marginally improves the obtained accuracy. Both expected and measured accuracy top at around 88%. This is in line with the results of Anguita et al. [4], who obtain a 93.9% accuracy using many more features compared to our system and the same training data.

5.2 Comparing with Ground Truth

Using labeled data, we replay the execution of approximate intermittent computing, as well as of Uninterrupted and Chinchilla, using the same sensor data and energy traces.



(a) Classification accuracy compared to ground truth [%].

(b) System throughput normalized to continuous execution [%].

Figure 5: Emulation experiments: classification accuracy and system throughput normalized to UNINTERRUPTED. *The GREEDY implementation captures energy fluctuations most efficiently, improving system throughput at the expenses of slightly lower accuracy. The SMART implementations ensure a lower bound in accuracy for every processed sample, at the cost of a slight reduction in throughput. Using Chinchilla provides the best possible classification accuracy, but must invest significant energy in handling persistent state, severely impacting the throughput.*

Activity	*Walking*	*Walking up*	*Walking down*	*Standing*	*Sitting*	*Laying*
Walking	332	10	7	0	0	0
Walking up	28	110	18	0	0	0
Walking down	18	12	88	0	0	0
Standing	0	0	0	185	32	0
Sitting	0	0	0	17	236	3
Laying	0	0	0	0	0	218
Accuracy	87.83%	86.61%	77.87%	91.58%	88.05%	98,64%

(a) Confusion matrix for Chinchilla implementation.

Activity	*Walking*	*Walking up*	*Walking down*	*Standing*	*Sitting*	*Laying*
Walking	2,158	145	96	0	0	0
Walking up	421	732	202	0	0	0
Walking down	128	104	602	0	0	0
Standing	0	0	0	1,115	412	0
Sitting	0	0	0	415	1,612	403
Laying	0	0	0	0	0	1,417
Accuracy	79.71%	73.35%	66.95%	79.64%	77 .85%	77,85%

(b) Confusion matrix for GREEDY implementation.

Figure 6: Confusion matrixes of the classification results comparing the output of the Chinchilla and GREEDY implementation with ground truth. Rows represent the actual class and columns represent the predicted class.

Fig. 5 illustrates the results in accuracy and throughput. Fig. 5(a) shows how Chinchilla provides, for the single input sample, the best possible accuracy as it always uses all available features. Fig. 5(b) illustrates, however, that doing so comes at a tremendous cost in terms of throughput, because Chinchilla stretches the processing of every single samples across multiple power cycles, missing the opportunity to process new samples. Approximate intermittent computing, in contrast, trades a limited loss of accuracy for much greater throughput. One of the fundamental benefits is apparent here: the energy budget is entirely spent for useful application processing, rather than for managing persistent state on the energy-hungry NVM. The loss of accuracy is around 11% worst-case, yet the throughput improvements reach up to 7x that of Chinchilla.

Fig. 6 provides a closer look at how different accuracy results are obtained as a function of actual and predicted activities. Fig. 6(a) is obtained using Chinchilla and shows that two clusters of activities exist as in the original dataset [5]. The act of walking, possibly upstairs or downstairs, is distinctly recognized compared to activities that do not entail as much body movement, such as standing, sitting, or laying. Using Chinchilla, indeed, we systematically employ all available features as done by Anguita et al. [4], at the cost of processing fewer samples because of the overhead for employing persistent state and waiting for energy buffers to recharge.

Although the two clusters of activities still exist when considering the confusion matrix for the GREEDY implementation, illustrated in Fig. 6(b), the per-class accuracy results are different. As we are using kinetic energy harvesting, one may expect that activities that entail significant movement also carry a higher energy content, and are thus more accurately recognized because more features are used for classification. On the other hand, it is also the case that the activities with a lower energy content are the ones that are simpler to recognize in the original dataset [5], hence fewer features are usually sufficient to obtain an accurate classification. We argue that these two aspects largely compensate each other.

Comparing either configuration of SMART with GREEDY, Fig. 5(a) shows a slightly higher accuracy for the former. This is due to those samples that, in situations of energy scarcity, SMART discards as the number of features the system can process is too limited to match the required lower bound. GREEDY proceeds anyways by computing the classification with fewer features than usual, likely obtaining a less accurate output. The effect of this is also apparent in Fig. 5(b). The samples that SMART decides to drop do not produce an output, causing a reduction of the throughput. The same observations apply also between the two configurations of SMART, as a higher lower bound for accuracy improves the latter at the expense of additional dropped samples and therefore lower throughput.

Note how Fig. 5(b) also generally shows the impact of using ambient energy, in that energy harvesting causes a device not to run as often as a battery-powered one that can afford to execute uninterrupted. This is essentially the price to pay for a battery-less system. Still, using approximate intermittent computing, more than half of the classifications that UNINTERRUPTED would produce are indeed returned by the user when using kinetic energy. Let apart experiences using solar radiation as the energy source, these results generally represent a significant improvement compared to existing deployments of intermittent computing [1, 35, 67].

We also investigate the impact of sensor noise on the accuracy results. The dataset used for training includes filtered data as described in Sec. 4.2. In contrast, to limit processing overhead, we use the raw sensor data at run-time. We verify that this is the most efficient choice: re-running the emulation experiments with the addition of the same filtering steps used to build the training dataset yields a marginal accuracy improvement, in the range of a few percentage points. However, system throughput decreases by 22% on average for *all* systems we consider, due to the energy consumed for applying the filtering step to *every* sensor sample.

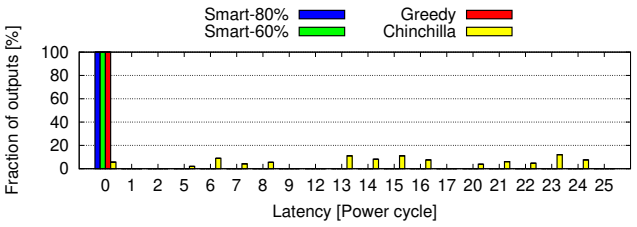


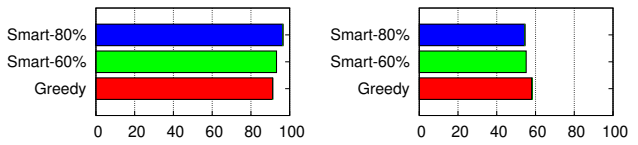
Figure 7: Emulation experiments: distribution of the latency to return the classification, measured in number of power cycles between when sensor data is acquired until the classification is emitted. *Approximate intermittent computing always returns the result to the user within the same power cycle. Because intermittent computing uses persistent state to cross periods of energy unavailability until all available features are processed, the time when the classification is returned is entirely a function of energy patterns.*

The improvements in throughput stem from the reduced latency to return the result to the user. Fig. 7 illustrates this metric, measured in the number of power cycles, that is, the number of times the device wakes up with new energy since the sensor data is acquired and until the classification is emitted. Chinchilla is at the mercy of the energy source; computations stop and resume, using persistent state, until sufficient energy is eventually available to process all features. The latency therefore covers also the periods for recharging the energy buffer. A non-negligible fraction of the outputs are even returned tens of power cycles later than when the sensor data is acquired. In contrast, using approximate intermittent computing, the classification is returned within the same power cycle *by design*: the number of features we process is tuned for returning the output before the first power failure. Therefore, even though we generally do not aim at purely real-time operation, approximate intermittent computing reduces the application latency to the minimum time feasible on an energy-harvesting device.

5.3 Comparing with Uninterrupted Executions

We use two identical prototypes on the same person's wrist. One of them is running either of the approximate intermittent computing implementations of Sec. 4, the other one is battery-powered and executes uninterrupted. With six volunteers, we run two instances of every approximate intermittent computing implementation and six uninterrupted executions we compare with. We align the power cycle information with the uninterrupted execution based on the interval the BLE packets are received at the user smartphone. Should this interval be lower than .2sec, which equals the duration of sensor sampling, we consider the two classifications to be aligned.

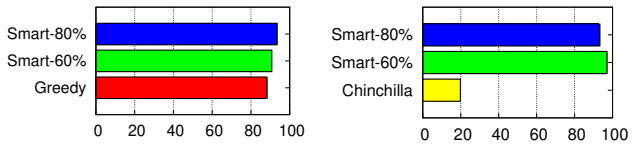
Fig. 8(a) shows the results we obtain in coherence of the classification. This time we cannot reason on absolute accuracy as in Sec. 5.2, because ground truth is not available. In at least 91.2% of the cases, however, the classification of human activities returned by approximate intermittent computing is the same as in an uninterrupted execution. However, approximate intermittent computing runs in a completely self-sustained manner, using kinetic energy. The coherence is higher for SMART because of the reasons explained earlier: the lower bound on expected accuracy makes SMART discard samples in situations where the (too) little available energy would yield a less accurate classification.



(a) Classification coherence compared to continous execution [%].

(b) System throughput normalized to continuous execution [%].

Figure 8: Real-world experiments: coherence of the classification of approximate intermittent computing against UNINTERRUPTED and system throughput normalized to UNINTERRUPTED. *Approximate intermittent computing returns the same classification of an uninterrupted execution in the majority of the cases. More than half of the samples processed by an uninterrupted execution are processed by approximate intermittent computing too.*



(a) Classification coherence compared to Chinchilla [%].

(b) System throughput normalized to GREEDY [%].

Figure 9: Real-world experiments: coherence of the classification of approximate intermittent computing against Chinchilla, and system throughput normalized to that of GREEDY. *The performance in coherence mirrors Fig. 8(a), as Chinchilla processes all available samples like UNINTERRUPTED. The throughput enabled by Chinchilla is lower than approximate intermittent computing, as it can process fewer samples.*

Unsurprisingly, Fig. 8(b) shows again that relying on ambient energy severely impacts system throughput, These results are in line with the emulation results of Fig. 5(b), giving us confidence on the correctness of the setup. Compared to Fig. 8(a), the relative trends between the different implementations are reversed; GREEDY shows higher throughput as it opportunistically consumes energy whenever available, returning more results at the expense of lower accuracy. We do not report on latency here, as all implementations we test here return the classification within the same power cycle.

5.4 Comparing with Chinchilla

We use the same setup as in Sec. 5.3, but replace the uninterrupted executions with an implementation that uses Chinchilla. Because the two devices on a person's wrist are exposed to the same movements, we verify that their energy patterns are almost identical. To align power cycle information with Chinchilla, we check what checkpoints are taken since the sensors samples are acquired and until the result is transmitted. This allows us to compute how many power cycles in the past the samples originate from.

Fig. 9(a) shows the coherence of the classification obtained by the implementations of approximate intermittent computing compared to Chinchilla. The results here mirror those of Fig. 8(a). The implementation using Chinchilla exploits all available features anyways, exactly like UNINTERRUPTED. Therefore, the accuracy it achieves is the same as UNINTERRUPTED and thus the coherence with the classification of approximate intermittent computing is also similar. What is different, however, is the number of sensor samples that

Chinchilla manages to process. As this implementation must cross periods of energy unavailability using persistent state, processing of a single sample extends across multiple power cycles, preventing the the acquisition of newer samples.

One consequence of this is a reduction in system throughput, as shown in Fig. 9(b). This time the data is normalized to the best performing implementation among the ones we test, which is Greedy. Chinchilla can only provide a fraction of the results that approximate intermittent computing can dispense. As we observed before, improved throughput is enabled by reduced latency to return the result to the user. By design, approximate intermittent computing returns the classification before the first power failure. The time it takes for Chinchilla to return the result is a function of energy patterns, as seen in Fig. 7. This latency stretches across even tens of power cycles and includes the periods for recharging energy buffers. While Chinchilla writes and reads from NVM to cross these periods, approximate intermittent computing can capture newer samples, as it already concluded processing the previous ones.

6 GENERALITY

The applicability of approximate intermittent computing extends beyond the case of human activity recognition or the use of specific machine learning techniques. Here we summarize the design, implementation, and evaluation of a sharply different application. Further details, including an extensive evaluation, are also available [17]. We conclude by discussing the limitations of our approach.

6.1 Application

Embedded image processing enables applications such as smart parking, preventive maintenance, and pervasive surveillance [58]. To accomplish the application tasks, image processing techniques such as corner detection are used to extract key features from the picture. In a smart parking application, for example, corner information may be used to determine whether a spot is occupied. The results of processing pictures at run-time are compared to the results obtained for a set of reference pictures, for example, showing the empty parking spot, to determine the final result.

Applications relying on embedded image processing often exhibit the two characteristics discussed in the Introduction. Occupancy information must reach the end user rapidly; for example, to update information on the available spots. The information is as valid as it is most current, as stale information is essentially of no use. Most importantly, corner detection only offers probabilistic guarantees, as most image processing techniques do. The processing is already robust to data errors, for example, due to distortions in the pictures, and is therefore amenable to further approximation.

Embedded image processing using energy-harvesting devices enables zero-maintenance deployments, yet is extremely challenging. The energy cost of image capture is significant and even the simplest camera sensor easily generates 25Kb of data for a single image capture [56]. Capturing pictures and processing this amount of data for every frame may be prohibitive on energy-harvesting devices, requiring either very large energy buffers that incur in high leakage currents, require long times to recharge, and increase footprints, or the frequent use of persistent state to stretch the processing across multiple execution rounds [21, 56].

	Anytime SVM	**Loop perforation**
Approximation knob	Number of features	Loop iterations
Energy estimation	Single feature	Single loop iteration
Output parameter	Activity classification	Number/position of corners

Figure 10: Relation between key concepts of approximate intermittent computing in the applications we consider.

6.2 Data Processing

Similar to many image processing techniques, corner detection is implemented by applying forms of iterative processing. We apply *loop perforation* [32] to create the knob that approximate intermittent computing requires to trade accuracy for energy consumption. Loop perforation skips a certain fraction of the loop iterations to save resources. Although application-specific policies to determine *what* iterations to skip exist [32], the choice is most often random. By properly tuning the fraction of loop iterations not executed as a function of available energy, loop perforation allows us to conclude processing before the first power failure, and hence to spare the need of persistent state and energy-hungry operations on NVM.

The key concepts emerging from the application of approximate intermittent computing to human activity recognition return here, as summarized in Fig. 10. The parameter that determines the level of approximation, and consequently the energy saving, is the number of loop iterations not executed, similar to the features that the anytime support vector machines do not consider. The extent of data processing as a function of available energy is obtained by estimating, using the same tools as before [2], the energy consumed by the single loop iteration; likewise, this information is the energy to process the single feature in human activity recognition. The output is represented by the number and position of detected corners, similar to the human activity classification obtained earlier. Based on this information, we quantify the accuracy achieved; as there is no "ground truth" here, accuracy is defined relative to an execution that does not skip any loop iteration.

6.3 Evaluation

We describe first our prototype implementation, along with the metrics and baselines we consider, and summarize the results next.

Prototype. We create a hardware/software prototype based on a TI Launchpad equipped with the same MSP430-FR5659 MCU used earlier. We use the on-board FRAM to store the test pictures we use for evaluation and the output of image processing. We manually retrieve the latter from FRAM at the end of every experiment. The energy cost for these operations is factored out. Generally, a federated energy architecture [21] would allow one to use separate capacitors for image capture and processing, shortening the times for recharging. Although the energy cost for image capture is fixed, approximate intermittent computing allows one to improve the energy efficiency in image processing. To power the latter functionality, we use the same capacitor and charging circuit as in Sec. 4.

We supply energy using a Renesas digital power supply driven by an RL78/I1A controller, based on five energy traces obtained from diverse sources and in different settings. Fig. 11 shows an excerpt, plotting the instantaneous voltage reading over time. The RF trace is from Mementos [62], recorded using a WISP device [64]. The other four traces are from EPIC [2] and are recorded using a

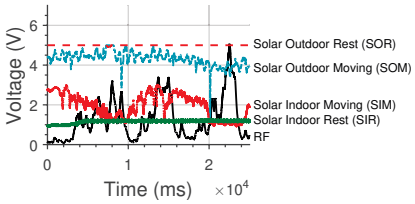
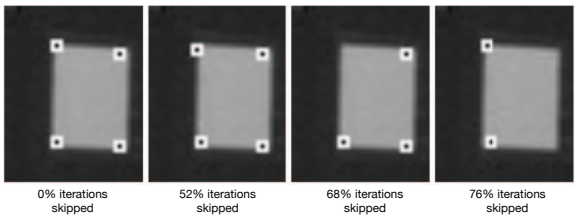
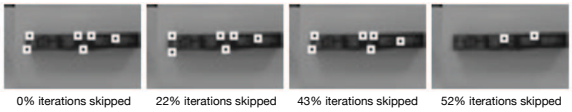


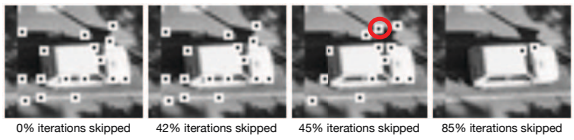
Figure 11: Energy traces used to power an embedded image processing pipeline. *SOM is most stable and has highest energy content. RF is most variable and with least energy content.*



(a) Rectangle.



(b) Pen.



(c) Car.

Figure 12: Representative examples of the output of corner detection, depending on the fraction of loop iterations not executed. *For complex pictures, up to 42% of the loop iterations may be skipped without severely impacting the quality of the result.*

mono-crystalline solar panel attached to an Arduino Uno in settings including outdoor mobile (SOM), indoor mobile (SIM), outdoor static (SOR), and indoor static (SIR). Similar to Ekho [34], this setup allows us to replicate the exact V-I curve the device would experience if attached to the actual energy harvester while considering the equivalent resistance offered by the device, and yet retain repeatability across experimental settings.

Whenever the device wakes up with new energy, it randomly loads one of the test pictures and performs corner detection. If energy is left at the end of the processing, the MCU switches to the lowest power mode that allows a 30sec timer to eventually trigger another round of image processing. The approximate intermittent computing implementation works the same as Greedy, skipping a number of loop iterations that allows the system to output the result just before energy is exhausted. The energy traces are the basis for comparing the approximate intermittent computing implementation with Chinchilla, as in Sec. 5. In addition, we also consider an implementation that incurs no power failures, hence disregarding the energy traces. Both the Chinchilla implementation and the one

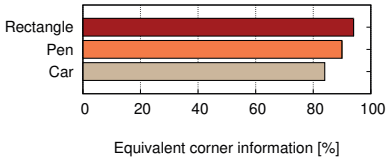


Figure 13: Quantifying the accuracy of approximate intermittent computing, compared to an execution that skips no loop iterations. *Based on the number and position of detected corners, approximate intermittent computing returns an equivalent output in at least 84% of the cases.*

experiencing no power failures skip no loop iterations; in particular, the latter processes one picture every 30sec with no interruptions, returning the result of the complete corner detection functionality at the end of processing.

The metrics we consider next are the same as in Sec. 5, but measured according to the specific features of the application at hand. **Results.** Fig. 12 graphically shows representative examples of the outputs we obtain, as a function of the loop iterations skipped. Note that the latter quantity here is precisely proportional to the share of saved energy. The picture in Fig. 12(a) is a simple test. Approximate intermittent computing may skip up to more than half of the loop iterations, and yet return corner information that are equal in number, and very similar in positioning, compared to an execution that skips no iteration. With more complex pictures, as in Fig. 12(b) and Fig. 12(c), this observation applies up to situations where more than 42% of the loop iterations are not executed. Beyond that, the overall number of detected corners reduces and spurious detections also appear, as indicated by the red circle in Fig. 12(c).

Fig. 13 quantifies the average accuracy of approximate intermittent computing across all energy traces by showing the fraction of pictures whose corner information are equivalent to those obtained by skipping no loop iterations. We define equivalence as the same number of corners appearing in the output, and each corner's position in the approximate intermittent execution to be closer to the position of the same corner when skipping no loop iteration, than to any other corner. The latter condition ensures that a corner may not be confused with a different one. Based on this, for example, the leftmost three pictures in Fig. 12(b) represent an equivalent output. Depending on the picture, approximate intermittent computing returns an equivalent output as an execution that skips no loop iterations in *at least* 84% of the cases.

As seen in Sec. 5, approximate intermittent computing trades a loss in accuracy for the ability to produce the result in the same power cycle. This reduces latency, enabling higher throughput. Fig. 14 shows the performance in the latter, normalized to that of an execution that experiences no power failures. The approximate intermittent computing implementation constantly outperforms the one using Chinchilla. Generally, traces that are richer in energy correspond to larger improvements for approximate intermittent computing. This is because it uses energy more efficiently than Chinchilla, where a significant fraction of that is spent handling persistent state and thus subtracted from application processing.

Interestingly, the better use of energy in approximate intermittent computing is visible in the time dynamics as well. This aspect is apparent as one observes that the performance of approximate

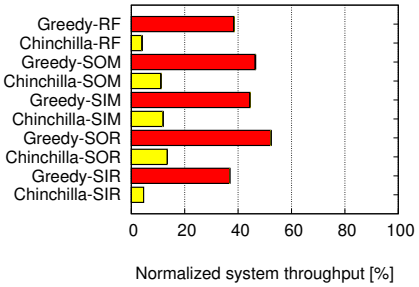


Figure 14: System throughput of corner detection normalized to that of an execution with no power failures. *Because of the higher energy efficiency, traces with higher energy content amplify the gains of approximate intermittent computing.*

intermittent computing in Fig. 14 is very similar for the RF and SIR traces: these two are very different in time, yet provide roughly the same total amount of energy to the system. We make better use of the energy available, no matter when it becomes available, by tuning the target accuracy, based on current conditions. In contrast, Chinchilla suffers from the rapid dynamics in the RF trace.

The trends in latency to produce the output of corner detection are similar to the ones discussed in Sec. 5.2 and Sec. 5.4. Approximate intermittent computing systematically produces a result within the same power cycle, regardless of the energy trace, whereas Chinchilla stretches the processing over many power cycles, as a function of the energy patterns.

6.4 Boundaries and Limitations

Applications where the data pipelines are amenable to approximation are, in principle, candidates for approximate intermittent computing. The aforementioned examples of smart health [66], ambient intelligence [22], and environment monitoring [57] extend the range of applications we concretely demonstrate here.

Existing literature offers a multitude of techniques that may be employed in approximate intermittent computing [53]. One example is neural networks. Zhang et al. [72] develop techniques to characterize the impact of neurons on the obtained accuracy. Based on this information, they determine how to approximate the computation and memory accesses of certain less critical neurons to improve energy efficiency. Although their work on memory accesses is not immediately applicable to our target platforms due to discrepancies in the memory layout, their work on the computing part does enjoy immediate applicability. The energy consumed to process the single neuron is measurable in ways similar to how we measure the energy for single features in Sec. 4.

It is also possible to foresee different implementations of the same functionality co-exist on the same device, featuring varying trade-offs between accuracy and energy consumption. At run-time, depending on available energy and expected consumption of the different implementations, one is chosen for execution. Such a technique, usually called "multi-accuracy programs" [53], may be used in approximate intermittent computing as long as sufficient program memory is available and is applicable in domains such as classification [60], signal processing [31], and image filtering [52].

The efficiency of the resulting system, however, is a function of how we can accurately estimate the energy consumption as a function of the level of approximation. This is required to determine how far can the system go applying approximation. Energy estimation tools for low-power embedded computing exist aplenty [41], along with versions that are specific to intermittent computing [2, 18, 20, 27]. These tools are necessary companions to approximate intermittent computing. Most of them work off-line, as the run-time overhead of energy estimation may be prohibitive. Whenever these tools cannot provide precise estimates, for example, because executions are highly dependent on run-time information, approximate intermittent computing may be difficult to apply.

By its own nature, approximate intermittent computing only offers probabilistic guarantees on correctness. Many of the inaccurate results returned in our application prototypes, nonetheless, may be corrected through some form of post-processing, as they are often represented by single outliers in long sequences of accurate outputs. This would further improve the accuracy of the system as a whole. Despite this possibility, whenever applications require exact results or depend on the absolute precision of data, approximate intermittent computing is simply not applicable. This is the case, for example, when sensor devices are used to drive closed-loop control systems, where accuracy is key to achieve stable behaviors [59].

7 CONCLUSION

We presented the concept of approximate intermittent computing and demonstrated its application in two diverse application cases. Regular intermittent computing retains the equivalence to continuous executions by using persistent state on NVM to cross periods of energy unavailability. In contrast, we showed how a moderate loss in the accuracy of the output provides huge gains in terms of energy consumption in that, if properly tuned, the system can finish processing prior to the first power failures. This makes it possible for the system to spare the need to maintain persistent state on the energy-hungry NVM, and thus allows the energy budget to be spent *entirely* for useful application processing. As a result, outputs are returned to the end user within the same power cycle compared to when the input sensor data is gathered. System throughput increases consequently, as the system is ready to process new inputs as soon as it restarts after a power failure. We showed, for example, that in human activity recognition approximate intermittent computing provides a $7x$ improvement in system throughput compared to regular intermittent computing. It also retains an average 83% accuracy compared to ground truth, in a setting where the best attainable accuracy is 88%. Using imagine processing techniques for corner detection, we achieve a $5x$ improvement in system throughput compared to regular intermittent computing, while retaining equivalence in the number and position of detected corners in at least 84% of the cases, compared to a continuous execution.

Acknowledgments. We thank the shepherd and reviewers for the feedback received on the initial submission. This work was supported partly by the Google Faculty Award programme and by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] M. Afanasov, N. A. Bhatti, D. Campagna, G. Caslini, F. M. Centonze, K. Dolui, A. Maioli, E. Barone, M. H. Alizai, J. H. Siddiqui, and L. Mottola. 2020. Battery-Less Zero-Maintenance Embedded Sensing at the Mithræum of Circus Maximus. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*.

[2] S. Ahmed, A. Bakar, N. Anwar Bhatti, M. Alizai, J. Siddiqui, and L. Mottola. 2019. The betrayal of constant power× time: Finding the missing joules of transiently-powered computers. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*.

[3] S. Ahmed, M. H. Bhatti, N. A. Alizai, J. H. Siddiqui, and L. Mottola. 2019. Efficient Intermittent Computing with Differential Checkpointing. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*.

[4] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. Reyes-Ortiz. 2012. Human activity recognition on smartphones using a multiclass hardware-friendly support vector machine. In *International workshop on ambient assisted living*. Springer, 216–223.

[5] D. Anguita, A. Ghio, L. Oneto, X. Parra, and J. Reyes-Ortiz. 2013. A public domain dataset for human activity recognition using smartphones. In *Esann*, Vol. 3. 3.

[6] A. R. Arreola, D. Balsamo, G. V. Merrett, and A. S. Weddell. 2018. RESTOP: Retaining External Peripheral State in Intermittently-Powered Sensor Systems. *Sensors* (2018).

[7] N. Asokan, T. Nyman, N. Rattanavipanon, A. Sadeghi, and G. Tsudik. 2018. ASSURED: Architecture for secure software update of realistic embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018).

[8] D. Balsamo, A. S. Weddell, A. Das, A. R. Arreola, D. Brunelli, B. M. Al-Hashimi, G. V. Merrett, and L. Benini. 2016. Hibernus++: A Self-Calibrating and Adaptive System for Transiently-Powered Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2016).

[9] D. Balsamo, A. S. Weddell, G. V. Merrett, B. M. Al-Hashimi, D. Brunelli, and L. Benini. 2015. Hibernus: Sustaining Computation During Intermittent Supply for Energy-Harvesting Systems. *IEEE Embedded Systems Letters* (2015).

[10] F. Bambusi. 2020. A Case for Approximate Intermittent Computing. bit.ly/3K0TZxo. Technical report.

[11] Fulvio Bambusi, Francesco Cerizzi, Yamin Lee, and Luca Mottola. 2020. Approximate Intermittent Computing: Schematics and Source Code. aic.neslab.it.

[12] G. Berthou, T. Delizy, K. Marquet, T. Risset, and G. Salagnac. 2018. Sytare: a Lightweight Kernel for NVRAM-Based Transiently-Powered Systems. *IEEE Trans. Comput.* (2018).

[13] N. A. Bhatti, M. H. Alizai, A. A. Syed, and L. Mottola. 2016. Energy Harvesting and Wireless Transfer in Sensor Network Applications: Concepts and Experiences. *ACM Transactions on Sensor Networks* (2016).

[14] N. A. Bhatti and L. Mottola. 2017. HarvOS: Efficient Code Instrumentation for Transiently-powered Embedded Sensing. In *Proceedings of the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

[15] B. Boser, I. Guyon, and V. Vapnik. 1992. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*. 144–152.

[16] A. Branco, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. Intermittent Asynchronous Peripheral Operations. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems (SENSYS)*.

[17] F. Cerizzi. 2020. Approximate energy-aware framework to support intermittent computing. bit.ly/3hrJk2q. Technical report.

[18] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. 2016. An Energy-interference-free Hardware-Software Debugger for Intermittent Energy-harvesting Systems. *SIGOPS Operating Systems Review* (2016).

[19] A. Colin and B. Lucia. 2016. Chain: Tasks and Channels for Reliable Intermittent Programs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[20] A. Colin and B. Lucia. 2018. Termination Checking and Task Decomposition for Task-based Intermittent Programs. In *Proceedings of the 27th International Conference on Compiler Construction (CC 2018)*.

[21] A. Colin, E. Ruppel, and B. Lucia. 2018. A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[22] D. Cook, J. Augusto, and V. Jakkula. 2009. Ambient intelligence: Technologies, applications, and opportunities. *Pervasive and Mobile Computing* 5, 4 (2009), 277–298.

[23] D. DeCoste. 2002. Anytime interval-valued outputs for kernel machines: Fast support vector machine classification via distance geometry. (2002).

[24] ReVibe Energy. [n.d.]. modelD Piezoelectric Energy Harvester. Retrieved July 8th, 2020 from https://revibeenergy.com/modeld/

[25] ReVibe Energy. [n.d.]. modelQ Piezoelectric Energy Harvester. https://revibeenergy.com/modelq/

[26] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. Marrón. 2009. COOJA/MSPSim: interoperability testing for wireless sensor networks. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*. 1–7.

[27] M. Furlong, J. Hester, K. Storer, and J. Sorber. 2016. Realistic Simulation for Tiny Batteryless Sensors. In *Proceedings of the 4th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'16)*.

[28] K. Ganesan, J. San Miguel, and N. Jerger. 2019. The What's Next Intermittent Computing Architecture. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 211–223.

[29] G. Gobieski, B. Lucia, and N. Beckmann. 2019. Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 199–213.

[30] A. Gomez, L. Sigrist, T. Schalch, L. Benini, and L. Thiele. 2017. Efficient, Long-Term Logging of Rich Data Sensors Using Transient Sensor Nodes. *ACM Transactions on Embeddded Computing Systems* (2017).

[31] B. Grigorian, N. Farahpour, and G. Reinman. 2015. BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing. In *International Symposium on High Performance Computer Architecture (HPCA)*. IEEE.

[32] J. Han and M. Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*. IEEE, 1–6.

[33] A. Haroun, I. Yamada, and S. Warisawa. 2016. Investigation of kinetic energy harvesting from human body motion activities using free/impact based micro electromagnetic generator. *Diabetes Cholest Metabol* 1, 104 (2016), 13–16.

[34] J. Hester, T. Scott, and J. Sorber. 2014. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys '14)*.

[35] J. Hester and J. Sorber. 2017. Flicker: Rapid Prototyping for the Batteryless Internet-of-Things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[36] J. Hester and J. Sorber. 2017. The Future of Sensing is Batteryless, Intermittent, and Awesome. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems (SENSYS)*.

[37] J. Hester, K. Storer, and J. Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. 1–13.

[38] N. Ikeda, R. Shigeta, J. Shiomi, and Y. Kawahara. 2020. Soil-Monitoring Sensor Powered by Temperature Difference between Air and Shallow Underground Soil. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)* (2020).

[39] Texas Instruments. 2017 (last access: September 18th, 2020). MSP430-FR5969 datasheet. https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf.

[40] B. Islam and S. Nirjon. 2020. Zygarde: Time-Sensitive On-Device Deep Inference and Adaptation on Intermittently-Powered Systems. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 4, 3 (2020), 1–29.

[41] V. Konstantakos, A. Chatzigeorgiou, S. Nikolaidis, and T. Laopoulos. 2008. Energy consumption estimation in embedded systems. *IEEE Transactions on instrumentation and measurement* 57, 4 (2008), 797–804.

[42] B. Lucia and B. 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)*.

[43] K. Ma, X. Li, J. Li, Y. Liu, Y. Xie, J. Sampson, M. Kandemir, and V. Narayanan. 2017. Incidental computing on IoT nonvolatile processors. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE.

[44] K. Maeng, A. Colin, and B. Lucia. 2017. Alpaca: Intermittent Execution Without Checkpoints. *Proceedings of the ACM Programming Languages* (2017).

[45] K. Maeng and B. Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[46] K. Maeng and B. Lucia. 2019. Supporting Peripherals in Intermittent Systems with Just-in-Time Checkpoints. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) (PLDI)*.

[47] K. Maeng and B. Lucia. 2020. Adaptive Low-Overhead Scheduling for Periodic and Reactive Intermittent Execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[48] M. Magno, D. Brunelli, L. Sigrist, R. Andri, L. Cavigelli, A. Gomez, and L. Benini. 2016. InfiniTime: Multi-sensor wearable bracelet with human body harvesting. *Sustainable Computing: Informatics and Systems* 11 (2016).

[49] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2019. On Intermittence Bugs in the Battery-Less Internet of Things (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*.

[50] A. Maioli, L. Mottola, M. H. Alizai, and J. H. Siddiqui. 2021. Discovering the Hidden Anomalies of Intermittent Computing. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks (EWSN 2021)*.

[51] A. Y. Majid, C. Delle Donne, K. Maeng, A. Colin, K. S. Yildirim, B. Lucia, and P. Pawelczak. 2020. Dynamic Task-Based Intermittent Execution for Energy-Harvesting Devices. *ACM Transactions on Sensor Networks* (2020).

[52] L. McAfee and K. Olukotun. 2015. EMEURO: A framework for generating multi-purpose accelerators via deep learning. In *International Symposium on Code Generation and Optimization (CGO)*.

[53] S. Mittal. 2016. A survey of techniques for approximate computing. *ACM Computing Surveys (CSUR)* 48, 4 (2016), 1–33.

[54] M. Mohri, A. Rostamizadeh, and A. Talwalkar. 2018. *Foundations of machine learning*. MIT press.

[55] A. Montanari, M. Sharma, D. Jenkus, M. Alloulah, L. Qendro, and F. Kawsar. 2020. ePerceptive: Energy Reactive Embedded Intelligence for Batteryless Sensors. In *Proceedings of the 18th Conference on Embedded Networked Sensor Systems*.

[56] S. Naderiparizi, A. Parks, Z. Kapetanovic, B. Ransford, and J. Smith. 2015. WISPCam: A battery-free RFID camera. In *2015 IEEE International Conference on RFID (RFID)*. IEEE, 166–173.

[57] M. Othman and K. Shazali. 2012. Wireless sensor network applications: A study in environment monitoring system. *Procedia Engineering* 41 (2012), 1204–1210.

[58] S. Pedre, T. Krajník, E. Todorovich, and P. Borensztejn. 2016. Accelerating embedded image processing for real time: a case study. *Journal of Real-Time Image Processing* 11, 2 (2016), 349–374.

[59] N. Ploplys, P. Kawka, and A. Alleyne. 2004. Closed-loop control over wireless networks. *IEEE Control Systems Magazine* 24, 3 (2004).

[60] J. Ross Quinlan. 1987. Simplifying decision trees. *International journal of man-machine studies* 27, 3 (1987), 221–234.

[61] B. Ransford and B. Lucia. 2014. Nonvolatile Memory is a Broken Time Machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*.

[62] B. Ransford, J. Sorber, and K. Fu. 2011. Mementos: System Support for Long-running Computation on RFID-scale Devices. *ACM SIGARCH Computer Architecture News* (2011).

[63] E. Romero, M. Neuman, and R. Warrington. 2009. Kinetic energy harvester for body motion. *Proc. PowerMEMS* (2009), 237–240.

[64] A. Sample, D. Yeager, P. Powledge, A. Mamishev, and J. Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement* 57, 11 (2008), 2608–2615.

[65] E. Sazonov, H. Li, D. Curry, and P. Pillay. 2009. Self-Powered Sensors for Monitoring of Highway Bridges. *IEEE Sensors Journal* (2009).

[66] A. Solanas, C. Patsakis, M. Conti, I. Vlachos, V. Ramos, F. Falcone, O. Postolache, P. Pérez-Martínez, R. Di Pietro, D. Perrea, et al. 2014. Smart health: A context-aware health paradigm within smart cities. *IEEE Communications Magazine* 52, 8 (2014), 74–81.

[67] M. Thielen, L. Sigrist, M. Magno, C. Hierold, and L. Benini. 2017. Human body heat for powering wearable devices: From thermal energy to application. *Energy conversion and management* (2017).

[68] J. Van Der Woude and M. Hicks. 2016. Intermittent Computation Without Hardware Support or Programmer Intervention. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI)*.

[69] K. Vijayaraghavan and R. Rajamani. 2010. Novel Batteryless Wireless Sensor for Traffic-Flow Measurement. *IEEE Transactions on Vehicular Technology* (2010).

[70] K. Wagstaff, M. Kocurek, D. Mazzoni, and B. Tang. 2010. Progressive refinement for support vector machines. *Data Mining and Knowledge Discovery* 20, 1 (2010), 53–69.

[71] K. S. Yildirim, A. Y. Majid, D. Patoukas, K. Schaper, P. Pawelczak, and J. Hester. 2018. InK: Reactive Kernel for Tiny Batteryless Sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems (SENSYS)*.

[72] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu. 2015. ApproxANN: An approximate computing framework for artificial neural network. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.