ETAP: Energy-aware Timing Analysis of Intermittent Programs

FERHAT ERATA, Yale University, USA
EREN YILDIZ, Ege University, Turkey
ARDA GOKNIL, SINTEF Digital, Norway
KASIM SINAN YILDIRIM, University of Trento, Italy
JAKUB SZEFER, Yale University, USA
RUZICA PISKAC, Yale University, USA
GOKCIN SEZGIN, UNIT Information Technologies R&D Ltd., Turkey

Energy harvesting battery-free embedded devices rely only on ambient energy harvesting that enables standalone and sustainable IoT applications. These devices execute programs when the harvested ambient energy in their energy reservoir is sufficient to operate and stop execution abruptly (and start charging) otherwise. These intermittent programs have varying timing behavior under different energy conditions, hardware configurations, and program structures. This article presents Energy-aware Timing Analysis of intermittent Programs (ETAP), a probabilistic symbolic execution approach that analyzes the timing and energy behavior of intermittent programs at compile time. ETAP symbolically executes the given program while taking time and energy cost models for ambient energy and dynamic energy consumption into account. We evaluate ETAP by comparing the compile-time analysis results of our benchmark codes and real-world application with the results of their executions on real hardware. Our evaluation shows that ETAP's prediction error rate is between 0.0076% and 10.8%, and it speeds up the timing analysis by at least two orders of magnitude compared to manual testing.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**; • **Computing methodologies** → **Model development and analysis**; **Symbolic and algebraic algorithms**; • **Computer systems organization** → **Embedded software;**

Additional Key Words and Phrases: Intermittent computing, energy harvesting, symbolic execution, timing analysis

This work was supported in part by the Italian Ministry for University and Research (MUR) under the program Dipartimenti di Eccellenza (2018-2022), NSF grants #1651945 and #1813797, the SINTEF projects G-IoT (Green Internet of Things) and Sustainable IoT funded by the Research Council of Norway, and XIVT (eXcellence In Variant Testing) project funded by the Scientific and Technological Research Council of Turkey under project #9180066.
Authors' addresses: F. Erata, J. Szefer, and R. Piskac, Yale University 51 Prospect Street, office AKW 212 New Haven, CT 06511, USA; emails: {ferhat.erata, jakub.szefer, ruzica.piskac}@yale.edu; E. Yildiz, Department of Computer Engineering, Ege University, 35100, Bornova, Izmir, Turkey; email: eren.yildiz@ege.edu.tr; A. Goknil, Sintef Digital, Forskningsveien 1, 0373 Oslo, Norway; email: arda.goknil@sintef.no; K. S. Yildirim, University of Trento, Department of Information Engineering and Computer Science, Via Sommarive, 9, Povo, 38123 TN, Trento, Italy; email: kasimsinan.yildirim@unitn.it; G. Sezgin, UNIT Information Technologies R&D Ltd., Izmir, Turkey; email: gokcin.sezgin@unitbilisim.com.

1539-9087/2023/01-ART23 $15.00
https://doi.org/10.1145/3563216

ACM Reference format:
Ferhat Erata, Eren Yildiz, Arda Goknil, Kasim Sinan Yildirim, Jakub Szefer, Ruzica Piskac, and Gokcin Sezgin. 2023. ETAP: Energy-aware Timing Analysis of Intermittent Programs. *ACM Trans. Embedd. Comput. Syst.* 22, 2, Article 23 (January 2023), 31 pages.
https://doi.org/10.1145/3563216

1 INTRODUCTION

Advancements in energy-harvesting circuits and ultra-low-power microelectronics enable modern battery-free devices that operate by using only harvested energy. Recent works have demonstrated several promising applications of these devices, such as battery-free sensing [29, 38, 41] and autonomous robotics [80]. These devices also pave the way for new stand-alone, sustainable applications and systems, such as body implants [34] and long-lived wearables [77], where continuous power is not available and changing batteries is difficult.

Battery-free devices can harvest energy from several sporadic and unreliable ambient sources, such as solar [58], radio-frequency [38], and even plants [43]. The harvested energy is accumulated in a tiny capacitor that can only store a marginal amount of energy. The capacitor powers the microcontroller, sensors, radio, and other peripherals. These components drain the capacitor frequently. When the capacitor drains out, the battery-free device experiences a power failure and switches to the harvesting of energy to charge itself. When the stored energy is above a threshold, the device reboots and becomes active again to compute, sense, and communicate. Successive charge-discharge cycles lead to frequent power failures, and in turn, *intermittent execution* of programs. Each power failure leads to the loss of the device's volatile state, i.e., registers, memory, and peripheral properties.

To cope with power failures and execute programs intermittently so the computation can progress and memory consistency is preserved, researchers developed two recovery approaches supported by runtime environments. The first one is to store the volatile program state into non-volatile memory with checkpoints placed in the program source [7, 9, 40, 42, 48, 51, 63, 78, 81]. Checkpointing approaches differ based on how checkpoints are triggered. In static checkpointing [4, 48, 63, 78], programmers can place checkpoints at arbitrary points in the code or set a timer for checkpointing. However, voltage tracker hardware triggers the CPU to get a checkpoint in the dynamic checkpointing [7, 8, 15, 42]. In both approaches, checkpointing size can be static or dynamically changing at runtime. Another one is the task-based programming model [18, 50, 80], in which programs are a collection of idempotent and atomic tasks. Several studies extended these recovery approaches by considering different aspects, e.g., I/O support [65], task scheduling [41, 52], adaptation [5, 54], virtualization [23], and timely execution [39, 44].

Considerable research has been devoted to compile-time analysis to find bugs and anomalies of intermittent programs [53, 70] and structure them (via effective task splitting and checkpoint placement) based on worst-case energy-consumption analysis [1, 19]. Despite these efforts, no attention has been paid to analyzing the timing behavior of intermittent programs affected by several factors, such as the energy availability of the deployment environment, the power consumption of the target hardware, capacitor size, program input space, and program structure (checkpoint placement). Without such an analysis, programmers will never know *at compile-time* if their intermittent programs execute as they are intended to do in a real-world deployment. Worse still, it is extremely costly and time-consuming to analyze the timing behavior of intermittent programs on real deployments, because programmers need to run the programs multiple times on the target hardware with various ambient energy profiles, hardware configurations, and program inputs and structuring.

As an example, consider a deployment environment with low ambient energy and frequent power failures. The computing progresses slowly due to long charging periods, and in turn, the program throughput may not meet the expectations (e.g., a batteryless long-range remote visual sensing system takes a picture every 5 minutes and transmits the relevant ones every 20 minutes). If the program execution time is not what is expected, then programmers might increase the capacitor size, change the target hardware, or remove some checkpoints. These changes may not always lead to what is intended (e.g., the bigger the capacitor size is, the longer the charging takes). Therefore, programmers might have to deploy their programs into the target platform several times and run them in the operating environment to check if they meet the desired throughput. Having estimations at a low cost *without need to test and re-test multiple times on real hardware*, they can rapidly restructure the program (e.g., add or remove checkpoints) or reconfigure the hardware (e.g., change the microcontroller, its frequency, or capacitor size) to improve the program execution time and, in turn, the throughput.

Goal and Challenges. Our goal in this article is to estimate statically, *at compile-time*, the timing behavior of intermittent programs considering different energy modes, hardware configurations, program input space, and program structure. The state-of-the-art approaches exploit stochastic models to represent the dynamic energy-harvesting environments, e.g., several models based on empirical data for RF [55, 56] and solar [26] energy-harvesting environments. Similarly, several techniques [1, 19] extract the energy-consumption characteristics of the target microcontroller instruction set to model its energy consumption. These two stochastic models, capacitor size, charging/discharging model, program input space, and program structure form a multidimensional space. The main challenge that we solve is to devise a program analysis solution that considers this multidimensional space to derive, *at compile-time*, the execution time probabilities of the given intermittent program.

Probabilistic symbolic execution [27] is an ideal solution to tackle this challenge, since it is a compile-time program analysis solution quantifying the likelihood of properties of program states concerning program non-determinism. However, existing techniques consider only typical program non-determinism, e.g., program input probability distribution. Therefore, we need a dedicated symbolic execution technique that integrates intermittent program non-determinism (charging/discharging model and environment energy profile) and other intermittent program characteristics (capacitor size and program structure).

Contributions. In this article, we propose, develop, and assess ***Energy-aware Timing Analysis of intermittent Programs (ETAP)***, a probabilistic symbolic execution approach for estimating the timing behavior of intermittent programs. We give an overview of ETAP's functionality in Figure 1. ETAP generates the execution time probability distributions of each function in the input program. Programmers may annotate programs with timing requirements for the execution time between any two lines of code or the time required to collect and process a given amount of data. ETAP also reports the execution time probability distributions for those requirements. It supports the static checkpoint programming model but can easily be extended to analyze dynamic checkpointing and task-based programs. ETAP extends Clang [16] to recognize ETAP-specific configurations and uses LLVM compiler [45]. It employs libraries of R environment [62] for symbolic computations and SMT [20] solver to detect infeasible paths. Our symbolic execution technique is designed not to be specific to any microcontroller architecture.

We compared our analysis to the program executions in a radio-frequency energy-harvesting testbed setup. The evaluation results show that ETAP's prediction error rate is between 0.0076% and 10.8%. To summarize, our contributions include:

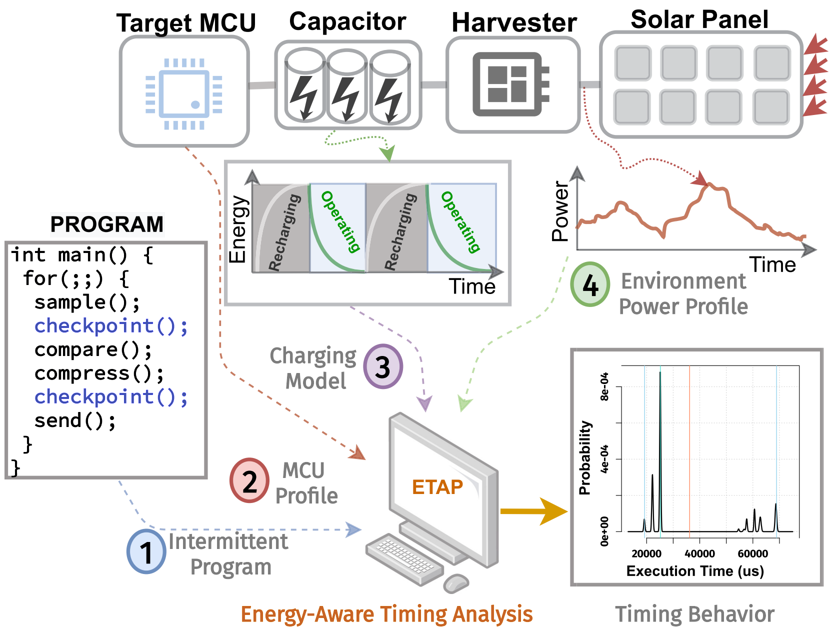


Fig. 1. The execution time of an intermittent program depends on energy-harvesting dynamics, capacitor size, the power consumption of the target microcontroller (MCU), and checkpoint overhead. Without hardware deployment, ETAP predicts timing behavior of an intermittent program.

(1) **Novel Analysis Approach.** We introduce a novel probabilistic symbolic execution approach that predicts the execution times of intermittent programs and their timing behavior considering intermittent execution dynamics.
(2) **New Analysis Tool.** We introduce the first compile-time analysis tool that enables programmers to analyze the timing behavior of their intermittent programs without target platform deployment.
(3) **Evaluation on Real Hardware.** ETAP correctly predicts the execution time of intermittent programs with a maximum prediction error less than 1.5% and speeds up the timing analysis by at least two orders of magnitude compared to manual testing.

We share ETAP as a publicly available tool[1] with the research community. We believe that ETAP is a significant attempt to provide the missing design-time analysis tool support for developing intermittent applications.

The rest of the article is structured as follows: Section 2 provides the background information regarding intermittent programming, timing behavior analysis, and probabilistic symbolic analysis. Section 3 introduces the challenges for timing analysis of intermittent programs. In Section 4, we present an overview of ETAP. Sections 5 to 8 describe the core technical solutions. Section 9 reports on the results of the empirical validation conducted with four benchmarks and a sense and send application on real hardware. In Section 10, we discuss the related work. Section 11 presents some insights into the evaluation and limitations of the approach. We conclude the article in Section 12.

2 MOTIVATION AND BACKGROUND

A typical battery-free device such as WISP [67], WISPCam [57], Engage [21], and Camaroptera [58] includes (i) an energy harvester converting incoming ambient energy into electric current;

[1] https://github.com/ferhaterata/etap.

(ii) energy storage, typically a capacitor, to store the harvested energy to power electronics; and (iii) an ultra-low-power microcontroller orchestrating sensing, computation, and communication. The microcontrollers in these platforms, e.g., MSP430FR series [74], comprise a combination of volatile (e.g., SRAM) and non-volatile (e.g., FRAM [75]) memory to store data that will persist upon power failures.

2.1 Programming Intermittent Systems

Battery-free platforms operate *intermittently* due to frequent power failures. Several programming models (supported by runtimes) have been proposed to mitigate the effects of unpredictable power failures and enable *computation progress* while preserving *memory consistency* (e.g., References [18, 44, 80]). Generally speaking, these models *backup* the volatile state of the processor into the non-volatile memory so the computation can be recovered from where it left upon reboot. Moreover, *memory consistency* should also be ensured so the backed-up state in the non-volatile memory will not be different from the volatile one, or vice versa.

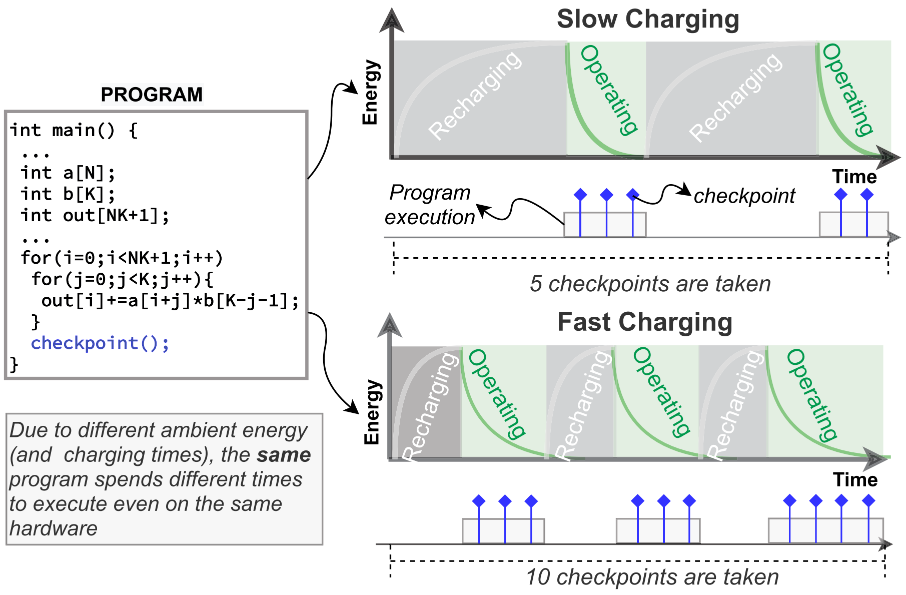
Programming models for intermittent computing have two classes. *Checkpointing* systems [44] snapshot the volatile state, i.e., the values of registers, stack, and global variables, in persistent memory at specific points—either defined by the programmer at compile-time or decided at runtime. Thanks to checkpoints, the state of the computation can be recovered after a power failure using the snapshot of the system state. *Task-based* systems [18, 80] employ a static task-based model. The programmer decomposes a program into a collection of tasks at compile-time and implements a task-based control flow (connecting task outputs with task inputs). The runtime keeps track of the active task, restarts it upon recovery from intermittent power failures, guarantees its atomic completion, and then switches to the next task in the control flow.

2.2 The Need for Timing Behavior Analysis

Using existing intermittent programming models, programmers mainly focus on the progress, memory consistency, and functional correctness of their programs. Without analyzing the timing behavior affected by several factors, programmers cannot be sure if their intermittent programs meet throughput expectations in a deployment environment.

Figure 2 depicts how the execution time of a checkpointed code changes under different environmental conditions. In high-energy environments, the capacitor charges faster, the program progresses quickly, the checkpoints are triggered more frequently, and the execution needs a shorter time. The same program takes much longer in low-energy environments. The device is mostly off to charge its capacitor and becomes active only for a short time to perform computation.

Some runtime environments, e.g., References [39, 44, 80], support the expression of timeliness in program source and check if sensing and computation are handled timely *at runtime*. They can catch the sensor readings that lost their timeliness due to power failures and long charging times, throw away these values, or omit their computation. Even though catching data and computation expiration are beneficial, programmers cannot reason about the timing behavior of their intermittent program at compile time due to many factors such as power failure frequency and capacitor charging time, or checkpoint placement. For instance, sensor readings and computations might expire continuously in specific energy-harvesting conditions due to a wrong checkpoint placement or insufficient capacitor size. The runtime environments can catch these expirations to prevent unnecessary processing and save precious harvested energy. However, the program does not produce any meaningful results due to frequent data expirations. With timing behavior analysis, programmers will know, in advance, how their programs will execute on the target environment (e.g., if there will be frequent data expirations) and perform the necessary actions to let their programs meet their expectations.



PROGRAM

```
int main() {
 ...
 int a[N];
 int b[K];
 int out[NK+1];
 ...
 for(i=0;i<NK+1;i++)
  for(j=0;j<K;j++){
   out[i]+=a[i+j]*b[K-j-1];
  }
  checkpoint();
}
```

*Due to different ambient energy (and charging times), the **same** program spends different times to execute even on the same hardware*

Fig. 2. Dynamics of Intermittent Execution. A sample application composed of matrix multiplication in a nested loop. The runtime behavior of the intermittent program (e.g., execution time) depends on the environmental factors (e.g., ambient energy), checkpoint placement and overhead, and the target platform (e.g., power consumption and capacitor size). Due to different ambient energy (and charging times), the same program may have different execution times on the same hardware.

2.3 Factors Affecting Timing Behavior

Energy Harvesting Environment. The availability of ambient energy sources is unpredictable. The harvested energy depends on several factors, such as the energy source type (e.g., solar or radiofrequency), distance to the energy source, and the efficiency of the energy-harvesting circuit. A probabilistic model of the energy-harvesting environment can be derived based on observations and profiling [26, 55, 56]. When incoming power is strong enough, the capacitor charges rapidly, and the device becomes available quickly after a power failure. At low input power, the charging is slower and takes more time.

Energy Storage. The interaction between the capacitor and the processor in battery-free devices plays a crucial role in the program execution time. When the capacitor is fully charged, the device turns on and starts program execution. When the stored energy in the capacitor drops below a threshold, the device dies. One of the factors that affect the device on time is the capacitor size. If the capacitor size is large, then the device has more energy to spend until the power failure, but charging the capacitor takes more time. Prior works, e.g., Reference [55], proposed models that capture the charging behavior of capacitors.

Target Platform. The power requirements of the target platform affect the end-to-end delay of the program execution. A program might take a long time to finish on platforms with high power requirements, since the capacitor discharges faster. Hence, the program might drain the capacitor more frequently (since instructions consume more energy in a shorter time). Therefore, the device is interrupted by frequent power failures, and it is unavailable and charging its capacitor for long periods. We can model the target platform by using the instruction-level energy-consumption profiles, as suggested in References [1, 19]. Another factor that affects the energy consumption of the target platform is the condition of the peripherals. Since peripherals have different energy consumption and timing behavior, they can not be standardized. Therefore, programmers should model peripherals and give the models as input to ETAP. For instance, programmers should measure the sensor energy consumption and sensing time in a sensing application for ETAP inputs. They should also annotate the sensing function in the intermittent program to be analyzed.

Intermittent Runtime. The programming model and the underlying runtime affect the execution time of intermittent programs. For instance, the checkpointing overhead is architecture-dependent, since the number of registers and the volatile memory size change from target to target. Moreover, checkpoint placement is also crucial: The more frequent the checkpoints are, the more energy consumed, but less computation is lost upon a power failure.

Program Inputs. Intermittent program inputs are mostly the sensor readings that depend on the environmental phenomena. Different inputs lead to different execution flow, and in turn, energy consumption. The energy consumption affects the frequency of power failures and the charging time.

3 CHALLENGES FOR TIMING ANALYSIS OF INTERMITTENT PROGRAMS

The main challenge is to devise a technique that facilitates the compile-time timing analysis of intermittent programs, considering the factors mentioned in the previous section. Since these factors can be represented using stochastic models [1, 19, 26, 55, 56], *probabilistic symbolic execution*—a static analysis technique calculating the probability of executing parts of a program [27]—becomes an excellent fit for timing behavior analysis. Unfortunately, existing probabilistic symbolic execution engines [14, 25, 27] do not support low-level code analysis; they are built on top of Symbolic PathFinder, which is a symbolic execution engine for Java bytecode. Furthermore, they consider only typical program non-determinism, e.g., program input probability distribution. Most importantly, ETAP needs a symbolic memory model with a non-volatile memory abstraction for backups, program restarts, and energy awareness to steer the path exploration algorithm by visiting all potential divergent power-failure paths in an intermittent program. Therefore, it is not feasible to integrate or reuse the existing probabilistic symbolic execution techniques for analyzing intermittent programs. Thus, we introduce a hand-rolled *probabilistic* symbolic execution engine supporting *intermittent program semantics*.

3.1 Probabilistic Symbolic Analysis

Symbolic execution analyzes a program to discover which program inputs execute which program parts. It searches the execution tree of a program using symbolic values for program inputs. From the execution tree, it generates program paths with a path condition, i.e., a conjunction of constraints on program inputs (path constraints). When symbolic execution reaches a node of the execution tree, it evaluates the path condition describing the path from the root to that node. If the condition is satisfiable, then it continues in that branch of the tree. If not, then the branch is known to be unreachable. The output of satisfiability checking is either false or true. It does not provide how frequently a path or a basic block executes. However, probabilistic symbolic execution estimates finer approximations of the probability of path conditions being true [27]. For a random input following a *discrete uniform* distribution, we can count the number of solutions to a *path constraint* and divide it by the product of the input domain to get the probability of the path condition. One approach for counting the number of a set of path constraints is to use *model counters*. Model counting is known as the problem of determining the number of assignments satisfying a given formula [32, 68, 76]. Its procedures are employed to compute the path probabilities in probabilistic symbolic execution [24, 27, 59].

ETAP follows a different approach to support sensor inputs as random variables following complex distributions. It models path execution environments with instances of an abstract memory model in which LLVM instructions are probabilistically interpreted. We employ two methods in evaluating probabilities of arithmetic instructions: the *linear location-scale transformation of a random variable* and the *sum of two random variables* (a.k.a. *convolution*). Linear transformations of named distributions have mostly analytical solutions; however, we need to employ computational approaches for non-linear transformations. It is the same for convolution operations. For instance, *the sum of two independent normally distributed (Gaussian) random variables* also follows a *normal distribution* (its *mean* is the sum of the two means, and its *variance* being the sum of the two variances). However, *the sum of two random variables*, which follow *Uniform* and *Normal* distributions, cannot be expressed with a *closed-form expression*. Therefore, in these cases, ETAP approximates these probabilities using numerical methods provided by libraries of the R framework.

3.2 Probabilistic Symbolic Storage for Checkpoints/Backups and Restarts

A symbolic memory model (a.k.a. *symbolic environment* or *symbolic store*) describes the approach used by a symbolic execution engine to handle memory operations such as loads and stores performed by the analyzed program. We could perform these operations concerning a specific memory object or a memory address [6, 12] in symbolic execution engines for low-level code. Since ETAP employs LLVM, its implementation tracks objects stored in memory, including their memory location and type and size information. While simulating *restarts* after power failures, it moves those objects to a separate "*non-volatile region*" to provide an infrastructure for different *checkpointing* or *backup* strategies (e.g., static checkpointing [4, 48, 63, 78] or hardware-triggered (dynamic) checkpointing [7, 8, 42]) in its symbolic memory model.

LLVM arithmetic and logical operators (instructions) on random variables are overloaded to produce *transformed* probability distributions. Once the symbolic execution reaches a branch point, ETAP integrates the probability function of the transformed distribution into the execution branches over the region of interest. This function tells us the probability of taking that branch. After stepping over a branch point, we update the random variables in the *symbolic store* of the path with the information gained.

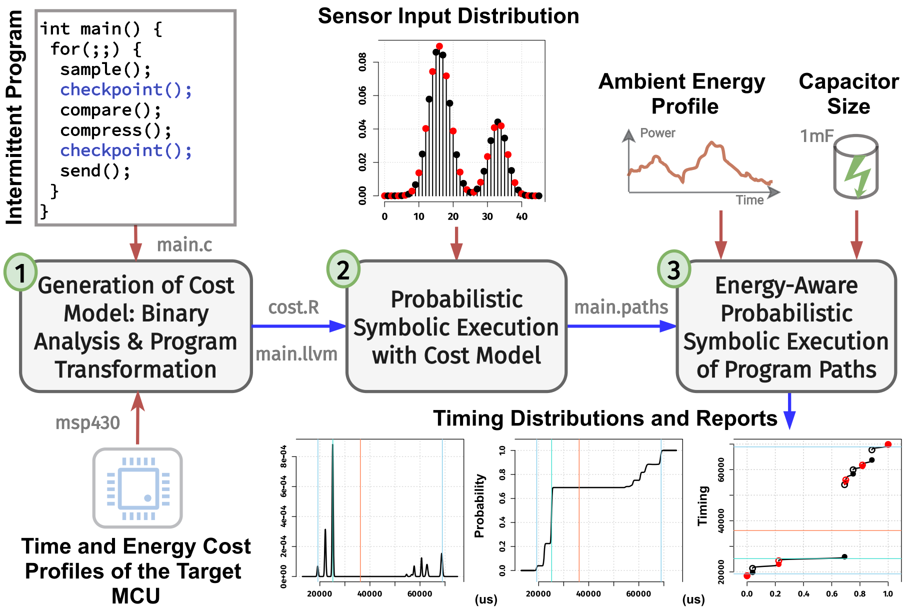
3.3 Energy-awareness

Another challenge for probabilistic timing analysis of intermittent programs is to model power failures over program executions: divergent execution paths induced by power failures and their emergent behavior. Our key insight is to drive *probabilistic symbolic execution* with a probability distribution representing the remaining energy budget at the capacitor that gets updated after executing each basic block. Since each basic block's energy-consumption behavior depends on executed instructions and called function, ETAP must also consider the probability distribution of energy consumption of each basic block as well as function calls.

All these aspects will be elaborated on in the subsequent sections.

4 ETAP: SYSTEMS OVERVIEW

The process in Figure 3 presents an overview of ETAP. Its steps are fully automated. Sections 6–8 provide details of each step. In Step 1, ETAP takes an intermittent program and time and energy cost profiles of the target architecture (*main.c* and *msp430* in Figure 3). It automatically generates a cost model with the program having split program blocks in LLVM IR [45] (*cost.R* and *main.ll*). The cost model is an R specification having LLVM program blocks to which time and energy costs are assigned as probabilistic cost expressions.

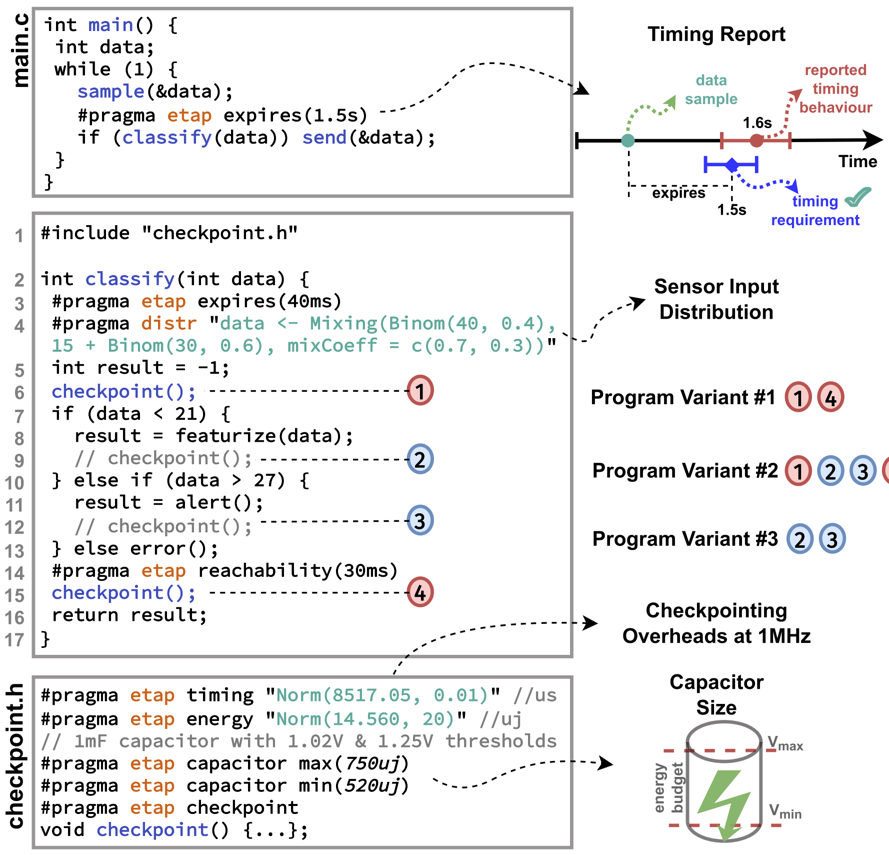

Intermittent Program
main.c
Sensor Input Distribution
Ambient Energy Profile
Capacitor Size
Power
Time
1mF
1 Generation of Cost Model: Binary Analysis & Program Transformation
cost.R
main.llvm
2 Probabilistic Symbolic Execution with Cost Model
main.paths
3 Energy-Aware Probabilistic Symbolic Execution of Program Paths
msp430
Time and Energy Cost Profiles of the Target MCU
Timing Distributions and Reports
Probability
Timing
(us)

```
int main() {
 for(;;) {
  sample();
  checkpoint();
  compare();
  compress();
  checkpoint();
  send();
 }
}
```

Fig. 3. ETAP consists of three stages after inputs are specified (see Section 5): (i) Generation of cost models using binary analysis and program transformation (see Section 6), (ii) Probabilistic Path Exploration that generates probabilities of execution traces based on sensor input distributions and program semantics (see Section 7), and (iii) Energy-aware symbolic execution based on intermittent execution of program semantics, microcontroller's energy consumption, and capacitor's energy capacity (see Section 8). Under given configurations, ETAP generates reports about timing distributions.

We designed our symbolic execution technique not to be specific to any microcontroller architecture and instruction set. Therefore, our symbolic execution runs not on microcontroller instructions but the program blocks in LLVM IR. This design choice contributes to the scalability of ETAP, since generating paths on program block-level leads to fewer divergent *power failure paths* to be analyzed. ETAP needs the time and energy cost profiles of the instruction types of each new target architecture platform organized under addressing modes. In Section 9, we derived the cost profiles for the MSP430FR5994 platform [74] through empirical data collection. It is a one-time effort for each new platform. ETAP models power failures during the transition from one basic block to another instead of after each instruction execution. However, compilers may generate relatively big basic blocks and, thus, symbolic execution over program blocks observe less power failures, which may lead to coarse approximations of energy costs in explored paths. Therefore, to increase the precision of the symbolic analysis, in Step 1, ETAP automatically splits blocks having outlier energy costs (maximum outliers) among all the program block energy costs. We employed the IQR method [47] to detect outlier blocks in a given program based on energy costs.

In Step 2, ETAP symbolically executes the program blocks in LLVM IR (*main.ll*) with the cost model (*cost.R*) to generate program paths and path execution probabilities (*main.paths*) based on the sensor input distribution. In Step 3, ETAP takes the ambient energy profile, capacitor size, program paths, and path execution probabilities as input. Each path is symbolically executed with stochastic ambient energy to estimate execution time probability distributions of program paths and functions for intermittent execution.



main.c
```
int main() {
 int data;
 while (1) {
   sample(&data);
   #pragma etap expires(1.5s)
   if (classify(data)) send(&data);
 }
}
```

```
1  #include "checkpoint.h"

2  int classify(int data) {
3   #pragma etap expires(40ms)
4   #pragma distr "data <- Mixing(Binom(40, 0.4),
   15 + Binom(30, 0.6), mixCoeff = c(0.7, 0.3))"
5   int result = -1;
6   checkpoint();
7   if (data < 21) {
8     result = featurize(data);
9     // checkpoint();
10  } else if (data > 27) {
11    result = alert();
12    // checkpoint();
13  } else error();
14  #pragma etap reachability(30ms)
15  checkpoint();
16  return result;
17 }
```

checkpoint.h
```
#pragma etap timing "Norm(8517.05, 0.01)" //us
#pragma etap energy "Norm(14.560, 20)" //uj
// 1mF capacitor with 1.02V & 1.25V thresholds
#pragma etap capacitor max(750uj)
#pragma etap capacitor min(520uj)
#pragma etap checkpoint
void checkpoint() {...};
```

Fig. 4. Example Intermittent Program with ETAP Pragmas. In the `classify` function, `featurize` and `alert` function calls are expected to take relatively more time and consume more energy. Therefore, to save the progress of computation, the programmer may apply alternative checkpoint configurations such as adding checkpoints before and after every major computation conservatively (Program `Variant` #2 has `checkpoints` 1, 2, 3, and 4); or placing checkpoints at `enter` and `exit` of the function (`Variant` #1 has only `checkpoint` 1 and 4 —checkpoints colored with red in the figure); or saving the progress of `featurize` or `alert` computations separately (`Variant` #3 has checkpoints 2 and 3 —checkpoints colored with blue in the figure). The question that ETAP can answer is which checkpoint placement strategy would meet the given timing requirement best under a set of given capacitor alternatives.

5 SPECIFICATION OF ETAP INPUTS

Engineers specify ETAP inputs (e.g., timing requirements and probability distributions of sensor inputs) via the "*#pragma*" directive [28]. Figure 4 presents an example intermittent program (*main.c*) and a header file (*checkpoint.h*) for the checkpoint operation. The runtime environment provides the header file [44], which programmers extend with pragmas for the time and energy cost distributions of the checkpoint operation (τ_{cp}, E_{cp}), ambient energy-harvesting time profile ($\tau_{harvest}$), and capacitor size (E_{min}, E_{max}).

The main function of the example program samples, classifies, and sends the input data in a loop (*main.c*); each loop iteration should execute in less than 1.5 second ("*#pragma etap expires(1.5s)*" in *main.c*). Function *classify* interprets input data, alerts users, or reports an error (Lines 2–17). It should finish in less than 40 milliseconds (Line 3). It has two checkpoints (Lines 6 and 15); the

ALGORITHM 1: DFS for Probabilistic Path Exploration

```
1 Blocks(path) ← Blocks(path) ⊕ Name(curr)
2 Timing(path) ← Timing(path) ∗ Get(Costs, curr)
3 Eval(Instructions(curr), env)
4 foreach succ in Successors(curr) do
5     prob ← prob × BranchProbability(curr, succ, env)
6     if prob > 0 ∧ ¬IsMaxLoopReached then
7         Paths ← DFS(Paths, Costs, (b, succ, prob), CloneEnv(env), path)
8 return Paths ⊕ (Blocks(path), prob, Timing(path))
```

second checkpoint should be reached from the first checkpoint in less than 30 milliseconds ("*#pragma etap reachability(30ms)*" in Line 14). The "reachability" pragma is to get the probability distribution of reaching at the specified checkpoint. The "expires" pragma is to get the probability distribution of timing at the specified point. The input distribution is discretized and modeled ("*#pragma distr*" in Line 4). It is the mixture of two signals following the binomial distribution (*Binom(40, 0.4)* and *Binom(30, 0.6)*).

6 GENERATION OF COST MODELS

ETAP generates a cost model with the intermittent program that has split program blocks in LLVM IR (Step 1 in Figure 3). It employs Clang [16] to compile the input program into the LLVM IR code. Each checkpoint call needs to be the first instruction in its basic block, because the symbolic execution returns to the beginning of the basic block for a power failure. Therefore, ETAP splits checkpoint calls, which are not the first instruction, from their basic blocks.

The LLVM IR code is compiled into the assembly code for the target hardware architecture. ETAP performs a binary analysis to map the hardware instructions in the assembly code to the basic blocks. It calculates the time and energy cost distributions of each basic block. To do so, it convolves the cost distributions of all instructions in that block based on their types and addressing modes (see Table 3). Some basic blocks may need much more energy than other basic blocks, which may lead to coarse approximations of energy costs of the program paths. ETAP performs a semantic- and cost-preserving program transformation to obtain more precise energy costs of the program paths. Considering all the block energy costs for each function in the program, it detects blocks having relatively high energy costs (i.e., takes them as outliers) and automatically splits them. We used the IQR method [47] to find and split these blocks. We defined the energy cost threshold value as $Q_3 + 1.5(IQR)$ (Q_3 is the third quartile, and IQR is the interquartile range).

Figure 5 presents the **control flow graph (CFG)** of *classify* in Figure 4 (with four checkpoints). Time and energy costs are modeled as normal distributions and encoded in an R specification (*cost.R* in Figure 3). Each node represents a basic block. Blocks *a* and *b*, *c* and *d*, and *g* and *h* were initially single blocks ETAP splits into two, as the checkpoint calls were not the first instruction. No blocks are split for the threshold.

7 PROBABILISTIC PATH EXPLORATION

ETAP symbolically executes the basic blocks with the cost models in the R environment with depth-first search (Step 2 in Figure 3). It generates program paths and their execution probabilities for *program execution without power failures*. The program paths are later re-executed symbolically with stochastic ambient energy to estimate its execution time probability distribution for intermittent execution (see Section 8).

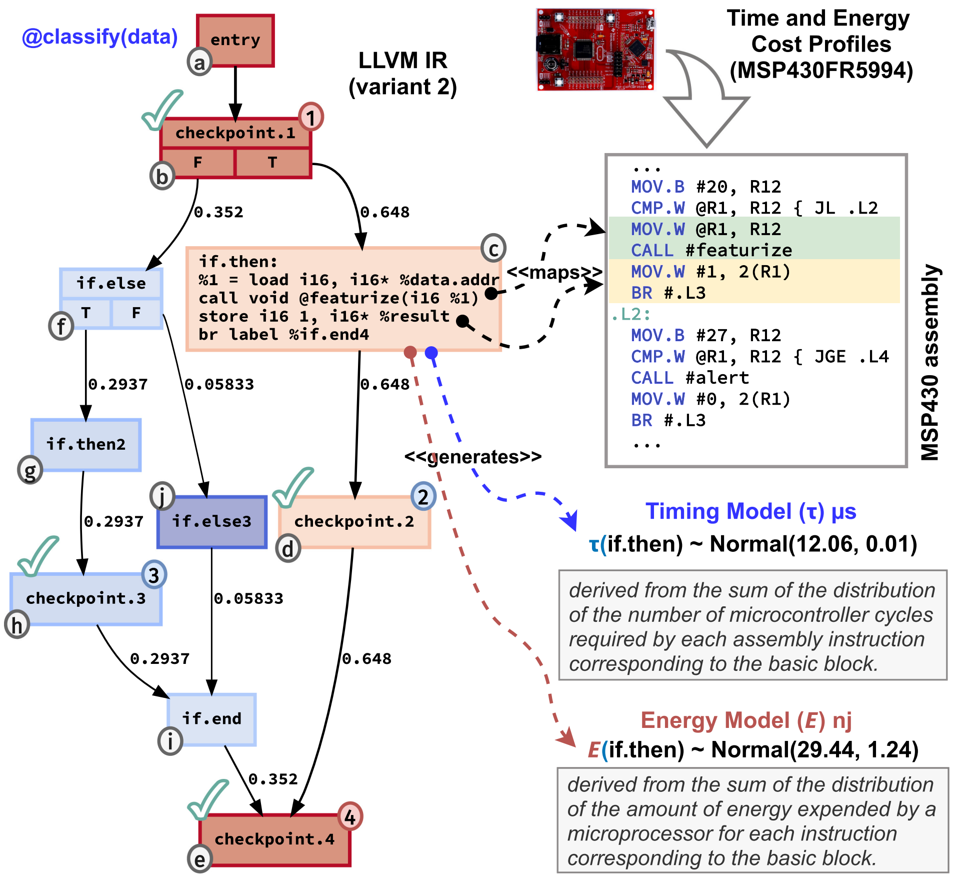

@classify(data)
entry
LLVM IR (variant 2)
Time and Energy Cost Profiles (MSP430FR5994)
checkpoint.1
F T
0.352
0.648
if.then:
%1 = load i16, i16* %data.addr
call void @featurize(i16 %1)
store i16 1, i16* %result
br label %if.end4
<<maps>>
...
MOV.B #20, R12
CMP.W @R1, R12 { JL .L2
MOV.W @R1, R12
CALL #featurize
MOV.W #1, 2(R1)
BR #.L3
.L2:
MOV.B #27, R12
CMP.W @R1, R12 { JGE .L4
CALL #alert
MOV.W #0, 2(R1)
BR #.L3
...
MSP430 assembly
if.else
T F
0.2937
0.05833
0.648
<<generates>>
if.then2
if.else3
checkpoint.2
Timing Model (τ) μs
τ(if.then) ~ Normal(12.06, 0.01)
derived from the sum of the distribution of the number of microcontroller cycles required by each assembly instruction corresponding to the basic block.
checkpoint.3
0.2937
0.05833
0.2937
0.648
if.end
0.352
Energy Model (E) nj
E(if.then) ~ Normal(29.44, 1.24)
derived from the sum of the distribution of the amount of energy expended by a microprocessor for each instruction corresponding to the basic block.
checkpoint.4
a b c d e f g h i j
1 2 3 4

Fig. 5. Control Flow Graph of Function `classify` with Program *Variant* #2. The figure shows the timing (τ) and energy (E) cost models of basic block *if.then*, which are automatically derived from the mapping between the basic block in LLVM IR and the compiled target MSP430 assembly. Basic block *if.then* calls `featurize()`; basic block *if.then2* calls function `alert()`; and basic block *if.else3* calls function `error()`. ETAP generates a basic block for each `call` to `checkpoint()` function.

ETAP calculates the path execution probability and the execution time probability distribution for each program path in the function to be analyzed. Algorithm 1 gives the depth-first search for probabilistic path exploration.

ETAP adds the visited (current) basic block of the CFG to the path (Line 1). It convolves the execution time probability distribution of the path and the execution time cost distribution of the visited block (Line 2). The block instructions in the path environment are evaluated to decide which successor block to take in which branching conditions (Line 3). ETAP models path execution environments with instances of a symbolic memory model in which LLVM instructions are probabilistically interpreted. For instance, in the LLVM instruction "*%inc = add nsw i16 %i.0, 1*," if "*%i*" is a random variable (r.v.) that follows a discrete probability distribution, the 16-bit integer register *%inc* becomes an r.v., $\%inc \sim \ \%i.0 + 1$, through linear location-scale transformation [10]. If the instruction was among two registers, "*%inc = add nsw i16 %i.0, %i.1*" and in that "*%i.0*" and "*%i.1*" are independent r.v.s, then the expression would be interpreted as the sum (integral) of two r.v.s (convolution) [10]. The path execution probability is a conditional probability and recalculated for each new successor block. The new path probability is the multiplication of the current path probability and the branch probability for the successor basic block (Line 5). Figure 5 shows the probabilities in the example CFG. The paths are recursively explored until the path probability is equal to zero or the maximum number of iterations for a loop is reached (Lines 6–7).

LLVM operators on distributions are overloaded to produce transformed distributions. When symbolic execution has reached a branch point, ETAP integrates the probability function of the transformed distribution that the execution branches on over the region of interest. This tells us the probability of taking that branch. After stepping over a branch point, we perform state forking, similar to mainstream symbolic execution engines [6, 13], and update the random variables in the symbolic environment of the path with the information gained.

Linear transformations of named distributions have mostly analytical solutions; however, we need to employ computational approaches if the transformation is non-linear (e.g., a logarithmic transformation). This is also the case for convolution operations. For instance, the sum of two independent normally distributed (Gaussian) random variables also follows a normal distribution, with its mean being the sum of the two means and its variance being the sum of the two variances; however, the sum of two random variables, which follow Uniform and Normal distributions, respectively, cannot be expressed with a closed-form expression, and therefore, in these cases, ETAP approximates these probabilities using numerical methods.

For function *classify* in Figure 5, ETAP explores three paths i.e., π_1: "$a \rightarrow b(cp) \rightarrow c \rightarrow d(cp) \rightarrow e(cp)$," π_2: "$a \rightarrow b(cp) \rightarrow f \rightarrow g \rightarrow h(cp) \rightarrow i \rightarrow e(cp)$," and π_3: "$a \rightarrow b(cp) \rightarrow f \rightarrow j \rightarrow i \rightarrow e(cp)$," where cp indicates the blocks with the checkpoint operation. It calculates path execution probabilities (or weights) associated with each path ($\omega_1 = 0.648$, $\omega_2 = 0.058$, and $\omega_3 = 0.294$). The nodes labeled with small letters represent the basic blocks (e.g., a, b, c). Each basic block is heat-map-colored based on its execution probability. Each path has an execution time probability distribution (τ_1, τ_2, and τ_3) that represents the path's timing behavior while it is continuously powered. It is the convolution of the execution time cost distribution of the basic blocks along the path. Since the timing behavior is *compositional* when there is no power failure, ETAP considers checkpoint (τ_{cp}) and other function calls in the program path (τ_{alert}, $\tau_{featurize}$, and τ_{error}) while convolving the cost distributions as follows:

$$\begin{aligned}
\tau_1 &\sim a(\tau) * b(\tau) * \tau_{cp} * c(\tau) * \tau_{featurize} * d(\tau) * \tau_{cp} * e(\tau) * \tau_{cp} \\
\tau_2 &\sim a(\tau) * b(\tau) * \tau_{cp} * f(\tau) * g(\tau) * \tau_{alert} * h(\tau) * \tau_{cp} * i(\tau) * e(\tau) * \tau_{cp} \\
\tau_3 &\sim a(\tau) * b(\tau) * \tau_{cp} * f(\tau) * j(\tau) * \tau_{error} * i(\tau) * e(\tau) * \tau_{cp}.
\end{aligned}$$

The execution time probability distribution of function `classify` for program execution without power failures is a convex combination of all the timing distributions of its execution paths: $\tau_{classify} \sim \omega_1\tau_1 + \omega_2\tau_2 + \omega_3\tau_3$. It is a univariate mixture distribution and represents the timing behavior of the function while the energy is plentiful in the environment.

8 ENERGY-AWARE ANALYSIS OF PROGRAM PATHS

ETAP symbolically executes each program path with stochastic ambient energy to estimate its execution time probability distribution for intermittent execution (Step 3 in Figure 3). Figure 6 presents an example path without power failures (π_2: "$a(cp) \rightarrow b \rightarrow c \rightarrow d(cp) \rightarrow e \rightarrow f(cp)$" where cp indicates the blocks with the checkpoint operation) and illustrates its energy-aware symbolic analysis. We relabel the blocks in path π_2 to increase the readability in Figure 6. We also initiate the symbolic analysis from block a to simplify the presentation.

The analysis starts with a non-deterministic initial capacitor energy budget, i.e., a uniform random variable on the interval of the capacitor's maximum and minimum thresholds, $E_{init} \sim \text{Unif}(E_{min}, E_{max})$. ETAP divides the path into regions between two sequential checkpoints and

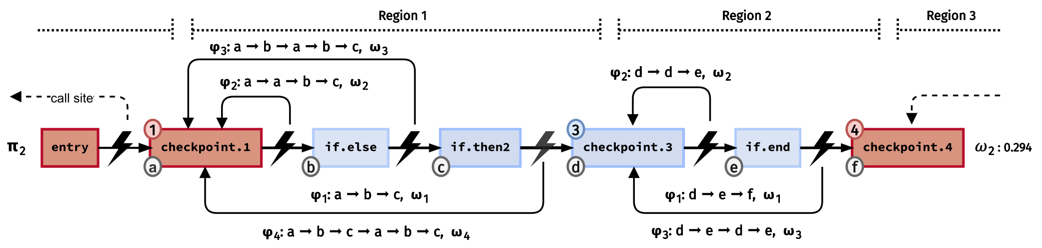


Fig. 6. An Example Path derived for *continuously-powered* execution and its analysis for *intermittent execution.*

between the beginning/end of the path and the first/last checkpoint in the path ("$a \longrightarrow b \longrightarrow c$," "$d \longrightarrow e$," and "$f$"). The path is analyzed region-by-region. The power failure probabilities are calculated for each block in the first region based on the initial capacitor energy distribution (E_{init}). If a power failure is probable, then ETAP creates a path that returns to the checkpoint block. In addition to the initial program path in the first region in Figure 6 (φ_1: "$a \longrightarrow b \longrightarrow c$"), three new paths are derived for power failures (e.g., φ_3: "$a \longrightarrow \dot{b} \longrightarrow a \longrightarrow b \longrightarrow c$," where the power failure occurs in block b). Each new path has only one power failure, because ETAP reports *non-terminating* ones [19] that need more energy than the capacitor size.

For each path, ETAP subtracts the energy cost distributions of program blocks from the capacitor energy through convolution; it sums (convolves) the time cost distributions of the blocks. The result is the time and energy cost distributions of the paths in the first region (τ_1, τ_2, τ_3, τ_4, E_1, E_2, E_3, and E_4). To reduce the number of convolution operations in the next region, ETAP derives univariate mixture distributions (e.g., τ_{mix_1}) from the cost distributions of the paths with power failure (e.g., in region 1, $\tau_{mix_1} \sim (\omega_2\tau_{\varphi_2} * \omega_3\tau_{\varphi_3} * \omega_4\tau_{\varphi_4}) * \tau_{harvest}$, where ω_2, ω_3, and ω_4 are failure probabilities). These distributions are input time costs for the paths in the next region derived from the power failure paths in the current region (e.g., $\varphi_2 \sim \tau_{mix_1} * d_\tau * (d_\tau * e_\tau)$ for φ_2 in the second region in Figure 6).

The current capacitor energy distribution is calculated for each path in the first region. For the capacitor energy of the path without power failure, ETAP subtracts the energy cost distributions of the blocks from the initial capacitor energy distribution through convolution, i.e., $E_{current} \sim E_{init} * -(a_\epsilon * b_\epsilon * c_\epsilon)$. In the power failure paths, the capacitor is charged up to the maximum threshold energy level during power failure (sleep mode). Therefore, while calculating their capacitor energy, ETAP considers the maximum energy the capacitor can store, e.g., $E_{current} \sim E_{max} * -(a_\tau * b_\tau * a_\tau * b_\tau * c_\tau)$ for path φ_3: "$a \longrightarrow b \longrightarrow a \longrightarrow b \longrightarrow c$." It uses the final program paths, capacitor energy levels, and cost distributions of the *current region* as the initial parameters of the *next region*. The same mixture and convolution operations are repeated in each new region. In the last region, the final energy and time probability distributions are obtained for the path under analysis (see *Region 3* in Figure 6).

As we described above, ETAP obtains the execution time probability distributions of each path of the function under analysis. These distributions are also used to generate reports on how likely timing requirements are met (see the 2nd column in Table 1). ETAP symbolically executes each path of the function in Figure 5 for intermittent execution. The mixture distribution of the execution time distributions of these paths is the execution time probability distribution of the function.

Figure 7 shows the reports of the execution time probability distributions of function *classify* for intermittent execution under three different capacitor types (1 mF, 2 mF, and 5 mF) and three different checkpoint placements (variant 1, variant 2, and variant 3 in Figure 4). The first functions

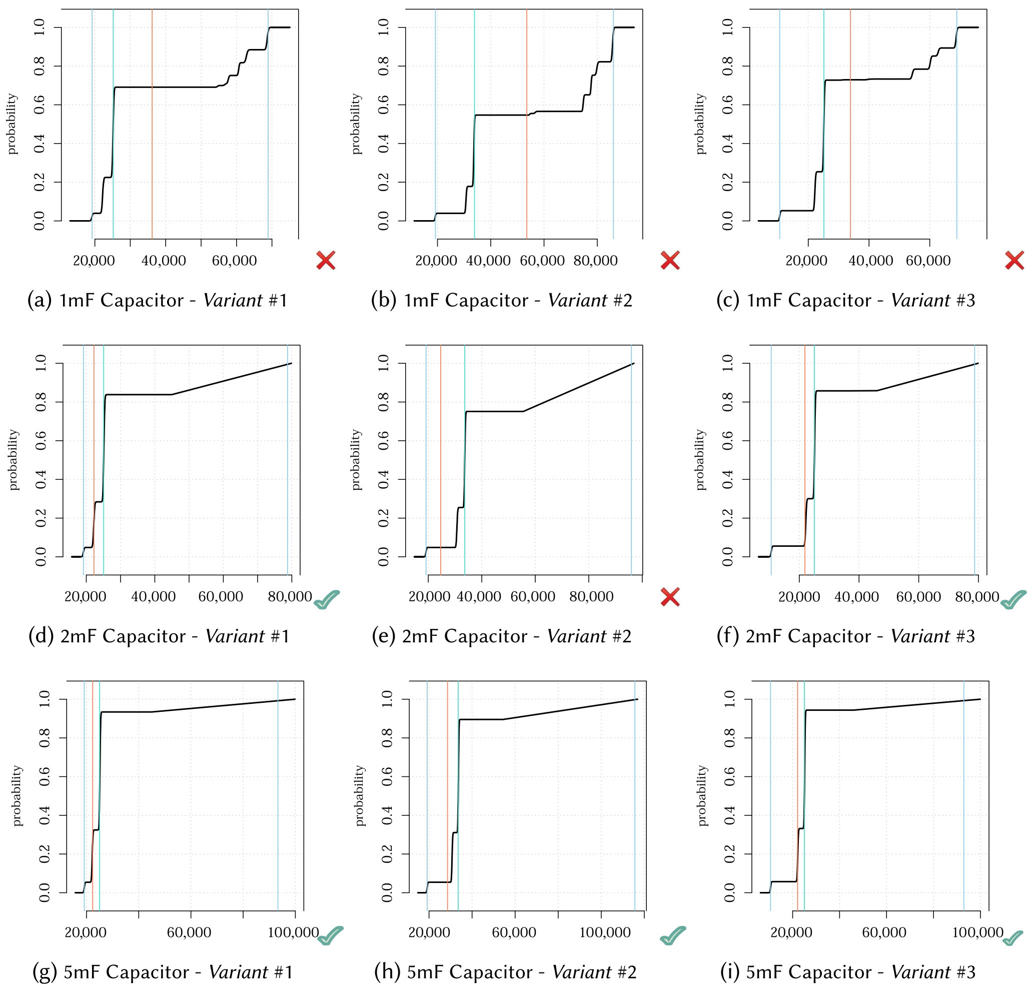


Fig. 7. Reports of the Execution Time Probability Distributions (Cumulative Distribution Function) of function *classify* after symbolically executing with 1 MHz clock-rate under three different capacitor types and three different checkpoint placements with the given $\tau_{\textbf{harvest}}$ profiles. The values on x-axis are in microseconds. The blue vertical lines represent the *first quartile*, the *median*, and the *third quartile*, respectively; the red vertical line is the *mean* of the probability distributions.

in Figure 7 are the **probability density function (PDF)**, and the second ones are the **cumulative distribution function (CDF)**. As shown in Table 1, ETAP also generates a report for each configuration.

When the capacitor size increases, the probability of failure paths decreases for all variants, which is expected. Configurations (d), (f), (g), (h), and (i) in Figure 7 satisfy the timing requirement (see Line 3 in Figure 4) with 0.8 probability ($P_r(\tau_{classify} \leq 40ms) > 0.8$). If we increase the capacitor size from 2 mF to 5 mF, then the probability of meeting the requirement becomes 0.9 for all variants. This increase our confidence in meeting the requirement, especially for configurations (g), (h), and (i).

Program *variant* #1 and #3 have more stable timing behavior compared to *variant* #2 (less divergent timing behavior in the probability distributions), and *variant* #3 performs slightly better

Table 1. Timing Report of "*#pragma Etap Expires(40ms)*" for the Given Configuration Space in Figure 7

Configuration	Timing Probability $P_r(\tau_{classify} \leq 40\text{ms})$	Meets Req.?	Timing Confidence Intervals (ms) *95% CI*	*90% CI*	*80% CI*
Figure 7(a)	0.691	✗	[19.2, 68.9]	[21.8, 68.7]	[22.1, 68.3]
Figure 7(b)	0.546	✗	[19.2, 86.1]	[39.4, 85.9]	[30.7, 85.7]
Figure 7(c)	0.732	✗	[10.6, 68.9]	[11.0, 68.7]	[22.0, 68.2]
Figure 7(d)	0.838	✓	[19.2, 78.7]	[21.6, 78.4]	[22.0, 70.9]
Figure 7(e)	0.751	✗	[19.2, 95.9]	[30.2, 95.7]	[30.6, 90.4]
Figure 7(f)	0.857	✓	[10.6, 78.7]	[10.9, 78.3]	[22.0, 70.0]
Figure 7(g)	0.934	✓	[19.1, 93.3]	[19.5, 90.4]	[22.0, 25.5]
Figure 7(h)	0.895	✓	[19.1, 116]	[19.5, 110]	[30.5, 104]
Figure 7(i)	0.943	✓	[10.6, 92.9]	[10.9, 84.3]	[21.9, 25.4]

Table 2. Charging Times (*ms*) of Capacitors in 95% *Confidence Interval*

40 cm distance	1 mF	2 mF	5 mF
$\tau_{harvest}$	[9.48 ms, 11.95 ms]	[18.99 ms, 25.64 ms]	[41.47 ms, 57.53 ms]

than *variant* #1 for all confidence intervals. Therefore, we can conclude that it would be better to use a 5 mF capacitor powering the program *variant* #3 for function *classify* (see Figure 7(f)). 5 mF capacitor has a higher energy level, and the probability of meeting the timing requirement would be higher due to less power-failure probability. However, bigger capacitors would require more time to store enough energy to power up the application, which would decrease the sampling rate of the sensing application. In this analysis, the setup harvests energy from the emitted **radio frequency (RF)** signals at a distance of 40 cm. The charging time distributions of the capacitors we used are given in 95% confidence intervals in Table 2.

We can conclude that it would be better to *minimize* the capacitor size while meeting the *timing requirement*. Also, a 2 mF capacitor would be the optimal selection for this setup under the given timing requirement and energy profile.

9 EVALUATION

We now evaluate our prototype on four benchmarks and one real-world application to demonstrate that: (i) ETAP correctly predicts the execution time of intermittent programs and (ii) it significantly reduces the analysis time and efforts.

Section 9.1 presents our testbed setup (checkpoint implementation, energy-harvesting tools, and host platform) for our evaluation. In Section 9.2, we describe the target platform and energy environment profiling used in the evaluation. Section 9.3 gives the results of our evaluation on the benchmark programs. In Section 9.4, we give the evaluation results for a DNN-based person detection application.

9.1 Testbed Setup

Target Platform. We used TI's MSP430FR5994 LaunchPad [74] development board as a target platform. The operating frequency of the **microcontroller (MCU)** was set to 1 MHz for our testbed experiments, however, we also investigated 4 MHz, 8 MHz, and 16 MHz clock-rates. The MCU supports 20-bit registers and 20-bit addresses to access an address space of 1 MB. To infer instruction-level energy consumption and obtain timing models, we used TI's EnergyTrace software [72] sampling the energy consumption of programs at runtime by using the specialized debugger circuitry on the development board. The precision of EnergyTrace was limited due to its low sampling rate.

Checkpoint Implementation. This MCU has 256 KB of FRAM (non-volatile memory) to store data that persists when there is no power. It also includes 4 KB of SRAM (volatile memory) to store program variables with automatic scope. We implemented (i) a checkpoint routine (*checkpoint()*) that copies the volatile computation state (20-bit general-purpose registers, program counter, and 4 KB SRAM) into FRAM and (ii) a recovery routine (*recovery()*) that restores the computation state after a power failure by using the latest successful checkpoint data. The energy and timing costs of these routines are constant. We configured ETAP with these costs by using pragmas, as shown in Figure 4.

Energy-harvesting Tools. We used the Powercast TX91501- 3W power transmitter [60] emitting **radio frequency (RF)** signals at 915 MHz center frequency. The transmitter was connected to a P2110-EVB receiver [61] co-supplied with a 6.1 dBi patch antenna. The receiver accumulated harvested energy from the emitted RF signals into 50 mF, 10 mF, and 5 mF supercapacitors to power the MSP430FR5994 launchpad board. We used Arduino Uno [3] with 10-bit **ADC (analog to digital converter)** to measure the instantaneous harvested power by the P2110-EVB receiver.

Host Platform. The evaluation was performed on 11th Gen Intel Core™i7-1185G7 @ 3.00 GHz × 8 cores with 32 GB RAM on Ubuntu 20.04.4 LTS operating system. ETAP's implementation is sequential, but, in the future, we aim to discharge symbolic computation queries concurrently. ETAP used one core and a maximum of 16 GB of memory. The memory consumption was relatively high, mainly due to the high rate of symbolic computations over random variables introduced by our benchmarks.

9.2 Profiling Target Platform and Energy Environment

The target platform and energy environment profiling is a one-time effort and eliminates the need for continuous target deployment and on-the-fly analysis efforts.

9.2.1 Profiling Target Platform. ETAP runs on the *platform-independent* LLVM instruction set [46]. There is no one-to-one correspondence between the LLVM instruction set and the target architecture instruction set. Therefore, we employed a block-based mapping strategy as mentioned in Section 6. In the cost model generation step of ETAP (Step 1 in Figure 3), the target hardware instructions were automatically mapped into the LLVM basic blocks. The time and energy cost distributions of each basic block were calculated through the hardware instruction cost distributions.

We used the Saleae logic analyzer [66] and EnergyTrace tool to collect the energy and timing costs of the MSP430 instructions from the MSP430FR5994 Launchpad at 1 MHz clock frequency. Table 3 presents a simplified version of the cost model we used to predict basic block costs. MSP430 is a 16-bit RISC instruction-set architecture with no data-cache [1]. It has 27 native and 24 emulated instructions. Our analysis shows that the energy and timing costs of the instructions depend on the formats and addressing modes. The MSP430 architecture has seven modes to address its operands: register (00), indexed (01), absolute (10), immediate (11), symbolic, register indirect, and register indirect auto increment. The timing and energy behavior of each instruction highly depends on these addressing modes, which we can group as double operand instructions (Format I), single operand instructions (Format II), and special instructions (Format III) [73]. We sampled the addressing modes and their combinations grouped by their formats and then statistically inferred the distribution of sample means employing bootstrapping method [47]. Apart from those

Table 3. Stochastic Cost Models of the MSP430 Instruction Set Architecture Based on Formats and Addressing Modes

Addressing Modes	Example Instruction	Timing Models (μs)	Energy Model (nj)
Format I	Double Operand		
00 0	`mov r5, r9`	$N(1.02, 0.01)$	$N(4.52, 0.62)$
00 1	`add r5, 3(r9)`	$N(3.02, 0.01)$	$N(7.08, 0.62)$
01 0	`mov 2(r5),r7`	$N(3.02, 0.01)$	$N(6.97, 0.62)$
01 1	`add 3(r4), 6(r9)`	$N(5.02, 0.01)$	$N(10.1, 0.62)$
10 0	`and @r4, r5`	$N(2.02, 0.01)$	$N(5.80, 0.62)$
10 1	`xor @r5, 8(r6)`	$N(4.02, 0.01)$	$N(8.33, 0.62)$
11 0	`mov #20, r9`	$N(2.02, 0.01)$	$N(5.55, 0.62)$
11 1	`mov @r9+, 2(r4)`	$N(4.02, 0.01)$	$N(8.34, 0.62)$
Format II	Single Operand		
00	`push r5`	$N(3.01, 0.01)$	$N(8.34, 0.62)$
01	`call 2(r7)`	$N(4.02, 0.01)$	$N(10.1, 0.62)$
10	`push @r9`	$N(3.52, 0.01)$	$N(8.33, 0.62)$
11	`call #81h`	$N(4.02, 0.01)$	$N(10.1, 0.62)$
Format III	Special Instruction or Intrinsic		
Jumps	`jmp r5, r9`	`Constant`(2)	$N(5.8, 0.62)$
multiplication	`call #_mspabi_mpyi`	$N(15.94, 0.27)$	$N(16.38, 0.23)$
division	`call #_mspabi_divu`	$N(16.39, 0.23)$	$N(16.68, 0.17)$
memcopy	`call #memcpy`	$N(13.06, .01) * cN(9.06, .01)$	$N(35.04, 1.38) * cN(24.21, 1.24)$

We sampled the addressing modes and their combinations grouped by their formats and then statistically inferred the distribution of sample means employing bootstrapping method [47]. Since we observed significant differences among measurements while sampling `call #memcopy` and `call #memset` intrinsic functions due to the data-dependent power consumption of these functions, we derived regression models for those that depend on the number of data words copied or set.

instructions, we modeled the subroutines, `call #mspabi_mpyi` and `call #mspabi_divu`, generated by the compiler, since MSP430 does not have a hardware multiplier and divider. Besides, we derived regression models for `call #memset` and `call #memcopy` intrinsic functions. For *call #memcopy*, the stochastic models for timing and energy are $\#\text{memcpy}_\tau \sim N(13.06, .01) * c \cdot N(9.06, .01)$ and $\#\text{memcpy}_\epsilon \sim N(35.04, 1.38) * c \cdot N(24.21, 1.24)$, where c is the explanatory variable for the number of data words copied.

9.2.2 Energy Profiles of the Environment. ETAP requires the ambient energy profile and the capacitor size to infer power failure rates and the off- and on-times of the device. Therefore, we collected the ADC samples of the instantaneously received power from different setups for 20 seconds. We created different settings by placing the RF power transmitter and receiver in line of sight at different distances. The measurements were repeated 10 times for each setup. Using the energy profile and energy equation of capacitors ($E = 1/2 \cdot C \cdot V^2$), one can infer a bootstrapping distribution [47] to model the average waiting time required to charge a capacitor with the given voltage threshold. ETAP uses the maximum and minimum energy thresholds to calculate charging and execution time. These energy thresholds are calculated by putting the minimum and maximum voltage thresholds in the energy equation formula, which assumes a constant capacitance value. However, the supercapacitor capacitance is not constant and depends on the current voltage value. In our experimental setting, the voltage range provided by the P2110-B energy-harvester board [60] is

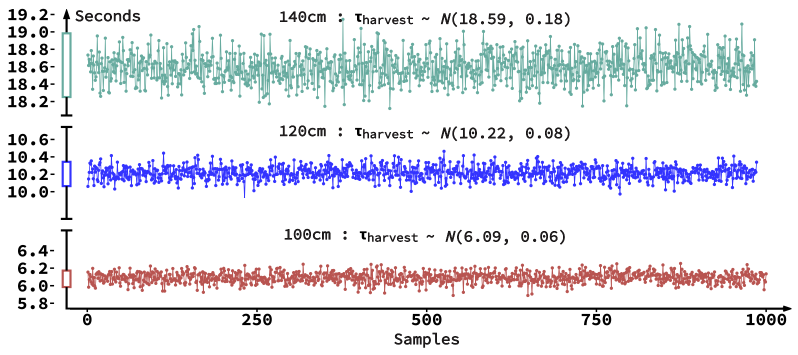


Fig. 8. RF energy profiles: charging-time samples for a 50 *mF* supercapacitor at different distances.

Table 4. Benchmarks

Benchmark	# of Instructions		# of Blocks		# of Power	Source
	Static	Dynamic	Static	Dynamic	Failure Paths	
BitCount	566	103,702	66	7,439	922	[33]
CRC	92	1,568	17	408	363	[33]
Dijkstra	282	24,585	36	2,884	592	[33]
FIR Filter	390	144,400	13	1,880	175	[2]

Instructions and block counts are derived while MSP430 is *continuously powered.*

from 1.02 V to 1.25 V, which are pretty close values. For simplicity, we used the capacitor energy equation formula, since it still provides a good approximation for the energy levels of the supercapacitor. Alternatively, programmers can also measure the real energy thresholds more precisely by using tools like EKHO [37].

We run our benchmarks using energy-harvesting hardware. Since the instantaneous power obtained from energy-harvesting devices varies, depending on the distance between the receiver and the transmitter, we chose three different distances in our testbed (100 *cm*, 120 *cm*, and 140 *cm*). Figure 8 presents the energy profiles sampled at these distances. The MCU starts to operate *intermittently* at 100 *cm* to the transmitter. The harvested power significantly decreases after 120 *cm*, and the MCU dies quite frequently.

9.3 Evaluation Results for the Benchmarks

We used four benchmarks having different computation demands and memory access frequency affecting the execution time (see Table 4). In particular, our benchmarks implement network algorithms and data filtering in real IoT and Edge computing applications.

9.3.1 ETAP Prediction Accuracy. We compared ETAP's predictions to the actual measurements obtained from the testbed (see Section 9.1 for the evaluation setup). The difference between the ETAP estimation and the actual measurements varies between 0.007% and 1.3% (see Table 5). Despite the limitations of our measurement devices, and in turn, coarse-grained ambient energy profiles, ETAP predicted the power failure probabilities with high accuracy. Since the device starts to operate continuously at a distance of less than 100 cm, the prediction error significantly decreases to at most 0.34% and at least 0.022%. Moreover, it accurately modeled the charging and discharging times by using the probabilistic distribution of the ambient energy. Note that the prediction

Table 5. Prediction for Intermittent Execution Times Compared with Actual Measurements

Benchmark	Distance (*cm*)	Actual Measurement (*ms*) *Mean*	*Std. Dev.*	ETAP Prediction (*ms*) *Mean*	*Std. Dev.*	Error Rate (%)
BitCount	*<100*	6,150.038	0.013	6,145.405	0.003	0.075
	100	33,550.353	19.919	33,693.902	99.793	0.4279
	120	99,815.832	1,008.207	99,618.170	303.237	0.0197
	140	206,575.550	219.747	206,546.729	267.430	0.0143
CRC	*<100*	3,689.654	0.001	3,690.537	0.001	0.022
	100	23,337.998	23.339	23,655.506	29.708	1.3605
	120	54,626.126	771.290	54,048.093	154.173	1.0582
	140	95,888.740	1,369.499	96,297.124	421.776	0.4259
Dijkstra	*<100*	3,212.243	0.013	3,201.285	0.006	0.341
	100	27,136.569	459.938	27,469.100	56.679	1.2254
	120	40,455.846	636.112	40,506.445	74.279	0.1251
	140	73,032.293	88.785	73,026.767	156.892	0.0076
FIR Filter	*<100*	7,282.946	0.019	7,277.968	0.004	0.068
	100	51,716.969	33.936	51,664.758	173.920	0.1010
	120	97,558.188	98.225	97,702.685	136.667	0.1481
	140	163,037.556	138.595	163,634.595	481.34	0.3662

accuracy of ETAP depends on the accuracy of the hardware cost models and the ambient energy profile.

The prediction errors in Table 5 (e.g., ETAP's predictions for CRC and Dijkstra are less accurate at closer distances to the energy harvester) are due to our imperfect probabilistic models. Moreover, ETAP does not consider the fact that the battery-free device is simultaneously charging while discharging. Therefore, as the distance gets longer, the amount of charge loaded on the capacitor during the discharge period decreases, and the predictions become more accurate.

Summary of the ETAP Accuracy Results for the Benchmarks. ETAP predicted the execution time of RF-powered intermittent programs **almost perfectly** for the benchmarks. We observed a reasonable maximum prediction error during our evaluation, which was less than 1.5%.

9.3.2 ETAP Analysis Time. To assess how significantly ETAP reduces the analysis time and efforts, we measured the time required to deploy an RF-powered application onto our testbed (see Section 9.1), run the application 50 times, and collect a sufficient amount of data to reason about the timing behavior through statistical sampling. The 4th column in Table 6 presents the time it takes for the experimental measurements at each distance and excludes the manual analysis efforts such as data extraction and preparation for data analysis.

As shown in Table 6, the manual experiment time increases as the number of power failures during intermittent operation increases (as the distance from the transmitter increases) while ETAP's analysis time does not change with regard to distances of the harvesting kit, and it mainly depends on the number of dynamic program blocks and instruction counts (see Table 4). Moreover, ETAP checks the likelihood of a power failure after each block, which increases its analysis time. ETAP apparently requires less manual effort and is significantly faster than manual testing. ETAP

Table 6. Time Study for Intermittent Execution

Benchmark	ETAP Analysis Time (s)		Harvesting Profiling Time (s)	Experiment Time (s)			Capacitor Size
	Path Exploration	*Intermittent Execution*		*100 cm*	*120 cm*	*140 cm*	
Bitcount	24.165	181.192	200	1,698	5,042	10,412	50 mF
CRC	1.283	10.636	200	1,249	2,797	3,895	50 mF
Dijkstra	10.590	73.035	200	1,391	2,029	3,842	50 mF
Fir Filter	18.709	127.897	200	2,591	4,897	8,203	50 mF
Bitcount	24.347	256.727	200	2,865	3,852	4,113	10 mF
CRC	1.296	10.259	200	1,593	2,108	2,460	10 mF
Dijkstra	10.075	69.147	200	1,278	1,954	2,030	10 mF
Fir Filter	18.592	45.084	200	346	488	520	10 mF
Bitcount	24.436	71.504	200	3,109	4,602	5,216	5 mF
CRC	1.314	10.156	200	1,857	2,878	3,086	5 mF
Dijkstra	10.318	108.201	200	1,600	2,252	2,307	5 mF
Fir Filter	18.923	26.479	200	406	608	689	5 mF

ETAP's automated analysis compared to manual experiment times. ETAP's analysis has two stages: First, it performs probabilistic *path exploration* (cf. Section 7) and then runs energy-aware *intermittent analysis* (cf. Section 8). We report here the analysis times of these two stages. *Path exploration* symbolically simulates *continuously powered* execution and enumerates paths accordingly. Therefore, the same benchmark takes a similar path exploration analysis time with different capacitors; however, ETAP's *intermittent analysis* time varies, depending on the capacitor size due to its impact on the number of divergent power-failure paths. Environment models on different distances do not impact the ETAP analysis times, since ETAP models the waiting time of the capacitor to charge up.

performed more precise path-based symbolic analysis at least 100 times faster than experimental measurements under low-energy harvesting conditions.

Summary of the ETAP Analysis Time Results for the Benchmarks. ETAP speeds up the timing analysis of intermittent programs for the benchmarks by **at least two orders of magnitude** and eliminates the burden of the manual data analysis effort.

9.4 ETAP Evaluation Using an Image Sensing Application

We evaluated ETAP using an image-sensing application to demonstrate the performance of ETAP's path-based probabilistic analysis. We used MSP430FR5994 MCU at 1 MHz, Powercast TX91501-3W power transmitter, and P2110-EVB receiver co-supplied with a 6.1 dBi patch antenna in the experimental setup. Furthermore, we set the distance between receiver and transmitter as 100 cm. We utilized 5 mF, 10 mF, and 50 mF supercapacitors as energy storage. Since ETAP requires peripheral functions' energy and time costs as inputs, we measured them using TI's EnergyTrace software [72]. We simulated peripheral functions rather than using peripherals. Since ETAP gets these functions' costs as input, the simulation of peripherals does not have an impact on ETAP analysis performance.

Application Implementation. Similar to the image sensing pipeline presented in Desai et al. [22], our application consists of four main steps (see Algorithm 2): (i) capturing the image, (ii) image differencing, (iii) DNN inference, and (iv) sending the image. The application control graph has three paths based on the image differencing and DNN inference step results.

ALGORITHM 2: Image Sensing Application

```
1  $idx \leftarrow 0$                                      ▷ Buffer index selects current or previous image
2  snapBuffer[$2$]                                          ▷ Store current and previous images
3  while 1 do
4      snapBuffer[$idx$] ← CaptureImage()
5      difference ← GetDifferencePercentage(snapBuffer)
6      if difference > %30 then
7          inference ← DNN(snapBuffer[$idx$])
8          if inference > %90 then
9              SendImage(snapBuffer[$idx$])
10     $idx \leftarrow idx \oplus 0x01$                     ▷ change buffer idx to swap current and previous buffer
```

Table 7. Harvesting Times (*ms*) of Capacitors in 95% *Confidence Interval* at the Environment Where the `Sense-and-Send` Application Runs

100 cm distance	50 mF	10 mF	5 mF
$\tau_{harvest}$	[1,490,980 μs, 152,323 μs]	[290,331 μs, 293,208 μs]	[147,747 μs, 149,212 μs]

- *Path 1* is the path in which the first if condition (*line 6*) is false. The program takes the image (*line 4*), compares it with the previous image (*line 5*), and changes buffer `idx` to save the next image (*line 10*).
- *Path 2* is the path in which the first if condition (*line 6*) is true and the second if condition (*line 8*) is false. The program takes an image (*line 4*), compares it with the previous image (*line 5*), runs DNN to detect a person (*line 7*), and changes buffer `idx` to save the next image (*line 10*).
- *Path 3* is the path in which both if conditions are true. The program takes an image (*line 4*), compares it with the previous image (*line 5*), runs DNN to detect a person (*line 7*), sends the image, which contains a person (*line 9*), and changes buffer `idx` to save the next image (*line 10*).

If there is enough difference between current and previous images, then the DNN algorithm runs to detect a person. If the person is detected, then the image is sent. Since peripherals functions were simulated, we defined uniformly distributed random values for image differencing, and DNN inference steps resulted in the code. We mainly inserted checkpoints at the beginning and end of each function and the inner loops at DNN.

ETAP Configuration. We compared ETAP's predictions to the actual measurements on the image sensing application. We analyzed the three data-dependent paths individually and their overall execution separately in four experiments. Furthermore, we also similarly analyzed them with ETAP using the same binaries. The timing and energy consumption of functions `CaptureImage()` and `SendImage()` in Algorithm 2 were modeled as $CaptureImage_{\tau} \sim N(13.9346\ \mu s, 0.0005)$ and $CaptureImage_{\epsilon} \sim 24,447.524$ nJ; and $SendImage_{\tau} \sim N(53.6934\ \mu s, 0.0019)$ and $SendImage_{\epsilon} \sim 817,614.653$ nJ. Harvesting times, $\tau_{harvest}$ for all capacitors, are given in Table 7.

We symbolically executed function `GetDifferencePercentage()` and `DNN()` in Algorithm 2. To align the sensor input behavior of the physical experiments with the ETAP's offline probabilistic analysis, the return values of functions `GetDifferencePercentage()` and `DNN()` were overridden by random values following Unif(1, 100) and Unif(60, 100) distributions using ETAP's

Table 8. Evaluation of the *Sense and Send Application*

Execution Paths	Cap. Size	ETAP Analysis Time (s)		Actual Measurement (s)		ETAP Prediction (s)		Error Rate (%)
		Path Exploration	*Intermittent Execution*	*Mean*	*Std. Dev.*	*Mean*	*Std. Dev.*	
Path 1	50 mF	7.826	53.568	0.352	0.532	0.314	0.482	10.80
	10 mF	8.225	51.033	0.325	0.141	0.349	0.167	7.38
	5 mF	8.373	52.151	0.348	0.114	0.371	0.122	6.61
Path 2	50 mF	86.001	544.292	1.509	0.767	1.531	0.228	1.46
	10 mF	91.108	535.289	1.273	0.519	1.312	0.408	3.06
	5 mF	89.447	569.962	1.710	0.891	1.882	0.615	10.06
Path 3	50 mF	91.843	562.086	1.320	0.700	1.353	0.763	2.50
	10 mF	94.764	573.437	1.730	0.489	1.862	0.314	7.63
	5 mF	92.398	571.485	1.965	0.101	2.094	0.201	6.56
Mixture	50 mF	84.657	1,149.556	1.167	0.897	1.135	0.397	2.81
	10 mF	80.744	1,129.452	1.214	0.687	1.12	0.319	8.39
	5 mF	82.670	1,142.744	1.327	0.707	1.448	0.394	9.12

configuration capability (see Section 5). Thus, we obtained similar path execution probabilities in the ETAP analysis and the physical experiments for the overall application execution (see *mixture* in Table 8).

Evaluation Results for the Image Sensing Application. The error rate of the ETAP's estimations ranges between 1.46% and 10.8% (see Table 8). The main reason for the differences in ETAP's prediction error rates between benchmarks and path analysis of image sensing application is that function `DNN()` of the application has more than one million loop iterations. After such numbers of iterations, ETAP starts encountering precision errors while generating a mixture probability distribution over all explored power failure paths and their execution probabilities. Therefore, ETAP analyzed one complete run of the application having an initial capacitor energy level following a *uniform* energy distribution—representing a maximum *uncertainty*. However, the actual experiments were conducted 50 times (runs), with the first run starting from a random initial capacitor level. The runs, except the first run, in the actual experiments were not *independent* of each other for the remaining energy at the capacitor, which led to an increase in the error rates of the ETAP's predictions for the image sensing application.

The high *standard deviation* compared to the benchmark experiments is due to the fact that the charging time of the capacitor dominates the total execution time after a power failure happens. For example, the normal execution time for path 1 is 0.143 second, while the average discharging time for 50 mF, 10 mF, and 5 mF capacitors are approximately 1.49 second, 296 ms, and 100 ms, respectively. Therefore, power failure does not occur at path 1 for 50 mF and 10 mF capacitors. However, it is still possible to have a power failure depending on the initial energy level of the capacitor. This scenario reveals the highest error rate that may occur in ETAP analysis. If there is no more than one power interruption (i.e., in path 1), then the error rate will increase as the capacitor size increases, since the charging time increases.

Due to the high number of loop iterations in function `DNN()`, paths 2 and 3 take more analysis time than path 1 takes and much more than the benchmarks take (see Table 6). When more than one power failure occurs at each execution (i.e., in path 2 and path 3), the error rate increases as capacitor size decreases. Since the capacitor size decreases, the number of power failure paths non-linearly increases. And, the uncertainty in the resulting distribution increases.

While evaluating mixture path analysis at the ETAP's end, we used uniformly distributed results for `DNN()` and `GetDifferencePercentage()` due to peripheral functions simulated. Therefore, the execution can branch into any of these paths. We run the application code 50 times in the real experiments and took a *sampling* distribution of the execution time to compare the ETAP result. As shown in the mixture path row of Table 8, the ETAP result reflects the weighted average of those three paths based on path probabilities. The mixture results are also close the weighted average of the three paths in experimental measurements. When the number of samples increases, the experimental results also become closer to the weighted mean of the three paths.

Summary of the ETAP Evaluation Results for the Application. Considering the evaluation results in Table 8, ETAP has successfully predicted the execution times of different application paths (with different sizes) executed on different hardware settings (different capacitor sizes), with prediction error rates ranging between 1.46% and 10.8%. The relatively high error rates for the image sensing application are due to precision errors that ETAP encountered while generating a mixture probability distribution over power failure paths and their execution probabilities in very large number of loop iterations.

10 RELATED WORK

We present prior work that relates to our approach in the context of intermittent computing, timing and energy analysis of intermittent programs, and probabilistic program analysis.

Timing Analysis of Intermittent Programs. Some runtimes provide programming constructs to assign timestamps to sensed data and check if the data expire. InK [80] is a reactive task-based runtime that employs preemptive and power failure-immune scheduling for timing constraints of task threads. Mayfly [39] makes the passing of time explicit, binding data to the time it was collected and keeping track of data and time through power failures. TICS [44] provides programming abstractions for handling the passing of time through intermittent failures and making decisions about when data can be used or thrown away. Different from these runtimes, ETAP predicts the timing behavior of intermittent programs before deployment and introduces zero overhead.

Energy Analysis of Intermittent Programs. CleanCut [19] detects non-terminating tasks in a task-based intermittent program. To do so, it samples the energy consumption of program blocks with a special debugging hardware and over-approximates path-based energy consumption. Similar to CleanCut, EPIC [1] traverses control-flow graph to compute best- and worst-case estimates of dynamic energy consumption for a given intermittent program. ETAP similarly analyzes the energy costs of paths but considers precise program semantics and knows execution probabilities of each path by means of probabilistic symbolic execution. In fact, profiling is a one-time effort for each target and has intrinsic support for detection of forward-progress violations. ETAP's main goal is to perform energy-aware timing analysis to reason about timing behavior of intermittent programs.

Other Analysis Tools for Intermittent Programs. IBIS [70] performs a static taint analysis to detect bugs caused by non-idempotent I/O operations in intermittent systems. ScEpTIC [53] has similar objectives as IBIS, i.e., detecting intermittent bugs. Ocelot [71] enforces intermittent programs written in Rust language to maintain temporal consistency. Those tools are complementary and can be used with ETAP to detect such intermittence bugs. But none of these tools perform timing analysis of intermittent programs. Moreover, ETAP is a novel probabilistic program analysis approach.

Probabilistic Program Analysis. Various probabilistic symbolic execution techniques have been proposed in the literature (e.g., References [11, 14, 24, 25, 27, 49, 59]). Geldenhuys et al. [27] propose probabilistic symbolic execution as an extension of Symbolic PathFinder [59]. Luckow et al. [49] extend probabilistic symbolic execution to compute a scheduler resolving program non-determinism to maximize the property satisfaction probability. Chen et al. [14] employ probabilistic symbolic execution to generate a performance distribution that captures the input probability distribution over the execution times of the program. Filieri et al. [25] extract failure and success program paths to be used for probabilistic reliability assessment against constraints representing subsets of all input variables range over finite discrete domains. All these techniques consider only typical program non-determinism, e.g., program input probability distribution. They analyze the control flow of continuously powered program execution, and they do not consider the power-failure-induced control flow. To analyze the power-failure-induced control flow, we need symbolic execution at the IR- (e.g., LLVM) or at the binary level, which is not supported by the current probabilistic symbolic execution engines. Therefore, it is not feasible to exploit, integrate, or reuse the existing techniques for the analysis of intermittent programs. Our goal is to overcome this problem and develop a dedicated probabilistic symbolic execution technique that analyzes the control flow graph of the intermittent programs. That is why we introduce power-failure-induced edges, and we use the charging/discharging model, environment energy profile, and other intermittent program characteristics (capacitor size and program structure).

11 DISCUSSION

Comparison to Simple Stochastic Simulation. Stochastic simulations may not cover all execution paths, since they randomly generate simulation inputs and simulate the program based on these input values. To execute all possible paths, we would need to perform numerous simulation runs. Even then, there might still be some paths that have not been examined, and it is not always possible to give a good estimate how many simulations we need to obtain a result accurate enough. Contrarily, ETAP employs probabilistic symbolic execution to get the absolute path frequency by using probability distributions, rather than randomly generating inputs and executing the program paths. This is a more accurate approach compared to simulation, since we are guaranteed to cover all execution paths needed to analyze the timing behavior of the intermittent program.

Scalability of Probabilistic Symbolic Execution. We do not expect serious scalability issues for ETAP analyzing intermittent programs running on batteryless devices, since these programs are relatively small by nature. This is also why we chose our benchmarks among the most common benchmarks that are widely accepted by the intermittent computing community (e.g., References [18, 19]). We already presented the analysis times for the four applications in Table 6's ETAP Analysis Time column. The Bitcount benchmark took about a minute. We also believe that a thorough scalability analysis will be beneficial to understand the limits of ETAP in terms of the maximum program size that we can analyze as of now.

The Prediction Accuracy Achieved by ETAP in the Unpredictable Nature of Energy Harvesting. The environment models that have more inherent variability will influence the accuracy and precision of the prediction negatively, such as energy modeling with mobile transmitter or receiver or both. But the focus of ETAP is not on generating the environment model, but on performing the analysis given the user-provided model. The user can provide a more sophisticated probabilistic model for incoming profiles, bimodal, mixture, and so on. However, the RF profile is not stable; it is actually chaotic. But it can be modeled with a Gaussian distribution easily at certain distances (by applying bootstrapping method on energy traces) as long as the testbed and the RF source are not mobile. We model the environmental energy profile in terms of the average waiting time required to fully charge a capacitor with specific capacitance values on the harvester kit. For instance, the average waiting time required to charge a 50 mF capacitor varies from 8 seconds to 9.5 seconds at 100 cm away from the RF source. And the more the testbed is positioned away from the source, the waiting time exponentially increases.

Dealing with Loops. If the program has a bounded-loop, then simply unrolling the loop would be sufficient; however, there might be conditions data-dependent upon the program's input that result in branch points in the computation tree. In these cases, the path exploration algorithm (Algorithm 1) traverses the loop until the probability of branching approaches zero. Our computation tree might have infinite depth, as some loops may be unbounded. Therefore, ETAP terminates analysis after exploring a user-provided limit (Line 6 in Algorithm 1). In addition, all the benchmarks used in the article include loops.

Timing Requirements. Timing requirements are optional inputs of ETAP. Without timing requirements, ETAP reports the timing distribution of each function in the LLVM module. Programmers only provide the function to be analyzed, and ETAP generates distributions for the timing and energy consumption of that function. If a more fine-grained analysis is needed, then timing requirements can be input to get a quantification report about the success rate of the requirements.

Ambient Energy Profile. While designing ETAP, we assumed that programmers follow a what-if analysis and evaluate checkpoint placement and timing behavior of functions under different ambient energy profiles. Even though ambient energy is unpredictable in intermittent systems, programmers should have an opinion about the energy level of the ambient to decide on intermittent constraints of their applications, such as capacitor size or checkpoint frequency. Similarly, they should have some information on the ambient energy behavior. If the ambient energy source is more erratic, then they should consider the worst ambient energy case (e.g., the farthest position of the RF receiver if the receiver is mobile) to analyze the timing behavior of the application.

Energy Cost of Peripherals. The energy-consumption characteristics of peripherals are quite different from each other. Moreover, some peripherals' energy consumption can change according to their hardware state. For instance, a transmitter has different energy consumption in receive or transmit mode. Thus, it is difficult to propose a general energy model that captures all the peripherals. ETAP needs to know which instructions change the peripheral state and energy-consumption characteristics of each hardware state. Therefore, programmers should measure the energy consumption of the peripherals by using tools (such as Ekho [37] or EDB [17]) to generate energy cost models. They should provide these cost models and the instructions that change the peripheral state as ETAP as inputs. These efforts are out of our current scope, and we left peripherals support as future work.

Energy Cost Model. Our evaluation shows that our energy cost model is already sufficient to perform an accurate timing analysis of intermittent programs. We observed less than 2% estimation error with our approach considering the benchmarked applications. Therefore, we did not see any reason to devise or incorporate a better model. It is possible to integrate better models (i.e., more accurate probability distributions that can provide a fine-grained representation of the pipelining and cache effects) into ETAP to increase the analysis accuracy. Considering the focus of our article (i.e., probabilistic symbolic execution approach and intermittent computing), proposing a better energy cost model is orthogonal to our contributions.

Requirement of Hardware Support and Cost of Energy Profiling. ETAP can use probabilistic or analytical models derived from either the real measurements conducted on the hardware or directly provided (for example, by a hardware vendor). There are many open-source energy-harvesting and power-consumption traces available, e.g., Reference [64]. Therefore, ETAP users can analyze their programs without the need for any hardware measurements, as long as they are provided via some of the mentioned means.

Using Energy Approximation. One may argue that compile-time analysis using energy cost approximations (e.g., Reference [4]) can be employed to predict the execution time of intermittent programs. Approximating the energy consumption of a code block can only give the time it takes to execute it continuously. The code block can be interrupted at any point during its execution, and there are extra recovery operations due to power failures. This situation leads to power-failure-induced control flow, which is highly dynamic. Therefore, it is impossible to infer the execution time by using simple energy-consumption approximations. And, we need a custom technique to infer the execution time concerning the dynamic power-failure-induced control flow.

Dynamic Checkpoint Size. For simplicity, ETAP assumes that all volatile data are backed up upon *checkpoint* calls [4, 48, 63, 78]. Thanks to this fixed checkpoint size, ETAP can use a fixed probability distribution to model timing and energy costs of checkpoints. However, there are studies that focus on reducing the contents to back up (e.g., References [44, 69, 82]), since storing all volatile data in the cache (or on-chip memory) would be energy-consuming. ETAP can be extended to model the behavior of dynamically changing size of data to back up, thanks to its *symbolic memory model* (cf. Section 2.4.2). Since ETAP's *symbolic store* tracks memory operations and simulates *backups* and *restarts*, checkpoints can be converted to parametric operations, similar to those regression models that we developed for timing and energy consumption of `memset` and `memcopy` operations (see Section 9.2.1).

Other Checkpointing Strategies. ETAP can be extended to support dynamic checkpoints, which are triggered by the voltage monitoring hardware when the energy level drops below a threshold [7, 8, 42]. ETAP's probabilistic symbolic analysis tracks the probability distribution of the remaining energy after executing each basic block for each program path and quantifies where power failures happen (cf. Section 2.4.3). ETAP can be similarly adapted to simulate dynamic checkpoints instead of power failures.

Drift between Distributions. There might be a drift between the distributions at measurement versus deployment time. It may not always be possible to know the exact energy-harvesting and sensor input distributions in the field. Therefore, we design ETAP to give programmers insight into how their intermittent programs behave under different ambient energy and sensor input conditions. Programmers are expected to use ETAP with different sensor input and energy-harvesting distributions and estimate the timing behavior of their programs.

MSP430 and Other Microcontrollers. As MSP430 is a de facto standard for intermittent computing, we target the MSP430 instruction set and its implementation, the MSP430FR5994 board, to assess ETAP. We empirically show that we can model the energy and timing behavior of the instruction set through sampling distributions. Even though instruction-level models to approximate timing and energy consumption of basic blocks result in accurate analysis on an MSP430 setup, it would be difficult to develop such models for a sophisticated processor having a memory hierarchy and a complex pipeline. However, ETAP supports assigning complex probability distributions to instructions and basic blocks. For instance, to model cache-hit and cache-miss behavior at instruction level, we need to know the timing behavior of a load instruction in the case of cache-miss and a cache-hit separately. ETAP can use a mixed probability distribution that can be, for instance, a weighted average of 95% of hit-distribution and 5% of miss-distribution. Another method in the literature [19] is to sample energy consumption of each basic block by executing it on the target microcontroller many times to generate accurate sampling distributions for each basic block. However, this method would require a special testing setup for the target microcontroller. In the future, we will explore more sophisticated chips.

12 CONCLUSION

We presented a novel static analysis approach, ETAP, which estimates the timing behavior of intermittent programs, affected by several factors such as ambient energy, the power consumption of the target hardware, capacitor size, program input space, and program structure. Considering the effects of power failures, ETAP symbolically executes the given program to generate the execution time probability distributions of each function in the program. To do so, it requires probabilistic energy and timing cost models of the target platform, capacitor size, and program input space. Our evaluation showed that ETAP exhibits prediction error rate ranging between 0.0076% and 10.8% for a set of benchmark codes and real-world application.

When implementing intermittent programs, programmers must consider several new challenges unfamiliar to most application developers that target continuously powered IoT systems. For instance, without compile-time analysis methods, programmers will never know at compile-time if their intermittent programs execute as they intend to do in a real-world deployment (e.g., meeting throughput requirements). Worse still, analyzing the timing behavior of intermittent programs on real deployments is costly and time-consuming, because programmers need to run the programs multiple times on the target hardware. ETAP is the first step to achieve a broad vision [30] addressing those unique software-engineering challenges for programming the batteryless edge.

The execution time and throughput of intermittent programs depend on multiple hardware and software design factors such as the capacitor size, the energy consumption of the target hardware, the efficiency of the energy-harvester unit, and the program structure (e.g., checkpoint placement and the size and number of tasks in task-based models). The changes in the hardware and software configuration may not always lead to what is intended (e.g., the bigger the capacitor size is, the longer the charging takes). Using the output of ETAP, programmers can follow a what-if analysis, e.g., reconfiguring the hardware and restructuring their program to ensure the desired timing behavior of their program. This what-if analysis is currently manual and not guided. As future work, we plan to employ metaheuristic algorithms [31, 79] to guide this analysis [35, 36].

REFERENCES

[1] Saad Ahmed, Muhammad Nawaz, Abu Bakar, Naveed Anwar Bhatti, Muhammad Hamad Alizai, Junaid Haroon Siddiqui, and Luca Mottola. 2020. Demystifying energy consumption dynamics in transiently powered computers. *ACM Trans. Embed. Comput. Syst.* 19, 6 (2020), 1–25.

[2] Mario Almeida, Stefanos Laskaridis, Ilias Leontiadis, Stylianos I. Venieris, and Nicholas D. Lane. 2019. EmBench: Quantifying performance variations of deep neural networks across modern commodity devices. In *the 3rd International Workshop on Deep Learning for Mobile Systems and Applications (EMDL'19)*. 1–6.

[3] Arduino. 2021. Arduino Uno rev-3. Retrieved from https://store.arduino.cc/usa/arduino-uno-rev3.

[4] Sara S. Baghsorkhi and Christos Margiolas. 2018. Automating efficient variable-grained resiliency for low-power IoT systems. In *International Symposium on Code Generation and Optimization (CGO'18)*. 38–49.

[5] Abu Bakar, Alexander G. Ross, Kasim Sinan Yildirim, and Josiah Hester. 2021. REHASH: A flexible, developer focused, heuristic adaptation platform for intermittently powered computing. *Proc. ACM Interact., Mob., Wear. Ubiq. Technol.* 5, 3 (2021), 1–42.

[6] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Comput. Surv.* 51, 3 (2018), 1–39.

[7] Domenico Balsamo, Alex S. Weddell, Anup Das, Alberto Rodriguez Arreola, Davide Brunelli, Bashir M. Al-Hashimi, Geoff V. Merrett, and Luca Benini. 2016. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 35, 12 (2016), 1968–1980.

[8] Domenico Balsamo, Alex S. Weddell, Geoff V. Merrett, Bashir M. Al-Hashimi, Davide Brunelli, and Luca Benini. 2014. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embed. Syst. Lett.* 7, 1 (2014), 15–18.

[9] Naveed Anwar Bhatti and Luca Mottola. 2017. HarvOS: Efficient code instrumentation for transiently-powered embedded sensing. In *the 16th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'17)*. 209–220.

[10] Joseph K. Blitzstein and Jessica Hwang. 2019. *Introduction to Probability*. CRC Press.

[11] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Păsăreanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. 866–877.

[12] Luca Borzacchiello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. 2019. Memory models in symbolic execution: Key ideas and new thoughts. *Softw. Test., Verific. Reliab.* 29, 8 (2019), e1722.

[13] Cristian Cadar and Martin Nowack. 2021. KLEE symbolic execution engine in 2019. *Int. J. Softw. Tools Technol. Transf.* 23, 6 (2021), 1–4.

[14] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating performance distributions via probabilistic symbolic execution. In *the 38th International Conference on Software Engineering (ICSE'16)*. 49–60.

[15] Jongouk Choi, Larry Kittinger, Qingrui Liu, and Changhee Jung. 2022. Compiler-directed high-performance intermittent computation with power failure immunity. In *the IEEE 28th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 40–54.

[16] Clang. 2021. Clang: a C language family frontend for LLVM. Retrieved from https://clang.llvm.org/.

[17] Alexei Colin, Graham Harvey, Alanson P. Sample, and Brandon Lucia. 2017. An energy-aware debugger for intermittently powered systems. *IEEE Micro* 37, 3 (2017), 116–125.

[18] Alexei Colin and Brandon Lucia. 2016. Chain: Tasks and channels for reliable intermittent programs. In *the 31st ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'16)*. 514–530.

[19] Alexei Colin and Brandon Lucia. 2018. Termination checking and task decomposition for task-based intermittent programs. In *the 27th International Conference on Compiler Construction (CC'18)*. 116–127.

[20] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[21] Jasper de Winkel, Vito Kortbeek, Josiah Hester, and Przemysław Pawełczak. 2020. Battery-free Game Boy. *Proc. ACM Interact., Mob., Wear. Ubiq. Technol.* 4, 3 (2020), 1–34.

[22] Harsh Desai, Matteo Nardello, Davide Brunelli, and Brandon Lucia. 2022. Camaroptera: A long-range image sensor with local inference for remote sensing applications. *ACM Trans. Embed. Comput. Syst.* 21, 3 (2022).

[23] Çağlar Durmaz, Kasım Sinan Yıldırım, and Geylani Kardas. 2022. Virtualizing intermittent computing. *IEEE Internet Things J.* (2022).

[24] Matthew B. Dwyer, Antonio Filieri, Jaco Geldenhuys, Mitchell Gerrard, Corina S. Păsăreanu, and Willem Visser. 2015. Probabilistic program analysis. In *the International Summer School on Generative and Transformational Techniques in Software Engineering*. Springer, 1–25.

[25] Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability analysis in Symbolic PathFinder. In *the 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 622–631.

[26] Kai Geissdoerfer, Raja Jurdak, Brano Kusy, and Marco Zimmerling. 2019. Getting more out of energy-harvesting systems: Energy management under time-varying utility with PreAct. In *the 18th International Conference on Information Processing in Sensor Networks (IPSN'19)*. 109–120.

[27] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *the 21st International Symposium on Software Testing and Analysis (ISSTA'12)*. 166–176.

[28] GNU. 2021. The Pragma Directive. Retrieved from https://gcc.gnu.org/onlinedocs/cpp/Pragmas.html.

[29] Graham Gobieski, Brandon Lucia, and Nathan Beckmann. 2019. Intelligence beyond the edge: Inference on intermittent embedded systems. In *the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. 199–213.

[30] Arda Goknil and Kasım Sinan Yildirim. 2022. Towards sustainable IoT applications: Unique challenges for programming the batteryless edge. *IEEE Softw.* 39, 5 (2022), 92–100.

[31] David E. Goldberg and John Henry Holland. 1988. Genetic algorithms and machine learning. *Mach. Learn.* 3 (1988).

[32] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. 2009. Model counting. In *Handbook of Satisfiability*. IOS Press, 633–654.

[33] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *the 4th Annual IEEE International Workshop on Workload Characterization (WWC'01)*. IEEE, 3–14.

[34] Philipp Gutruf, Vaishnavi Krishnamurthi, Abraham Vázquez-Guardado, Zhaoqian Xie, Anthony Banks, Chun-Ju Su, Yeshou Xu, Chad R. Haney, Emily A. Waters, Irawati Kandela et al. 2018. Fully implantable optoelectronic systems for battery-free, multimodal operation in neuroscience research. *Nat. Electron.* 1, 12 (2018), 652–660.

[35] Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Inf. Softw. Technol.* 43, 14 (2001), 833–839.

[36] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. 2010. Search based software engineering: Techniques, taxonomy, tutorial. In *Empirical Software Engineering and Verification - International Summer Schools, LASER 2008-2010, Revised Tutorial Lectures (Lecture Notes in Computer Science)*, Vol. 7007. Springer, 1–59.

[37] Josiah Hester, Timothy Scott, and Jacob Sorber. 2014. Ekho: Realistic and repeatable experimentation for tiny energy-harvesting sensors. In *the 12th ACM Conference on Embedded Network Sensor Systems*. 330–331.

[38] Josiah Hester and Jacob Sorber. 2017. Flicker: Rapid prototyping for the batteryless internet-of-things. In *the 15th ACM Conference on Embedded Network Sensor Systems (SenSys'17)*. 19:1–19:13.

[39] Josiah Hester, Kevin Storer, and Jacob Sorber. 2017. Timely execution on intermittently powered batteryless sensors. In *the 15th ACM Conference on Embedded Networked Sensor Systems (SenSys'17)*. 1–13.

[40] Matthew Hicks. 2017. Clank: Architectural support for intermittent computation. *ACM SIGARCH Comput. Archit. News* 45, 2 (2017), 228–240.

[41] Bashima Islam and Shahriar Nirjon. 2020. Zygarde: Time-sensitive on-device deep inference and adaptation on intermittently-powered systems. *Proc. ACM Interact., Mob., Wear. Ubiq. Technol.* 4, 3 (2020), 1–29.

[42] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. 2014. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *the 27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*. IEEE, 330–335.

[43] Christos Konstantopoulos, Eftichios Koutroulis, Nikolaos Mitianoudis, and Aggelos Bletsas. 2015. Converting a plant to a battery and wireless sensor with scatter radio and ultra-low cost. *IEEE Trans. Instrum. Meas.* 65, 2 (2015), 388–398.

[44] Vito Kortbeek, Kasim Sinan Yildirim, Abu Bakar, Jacob Sorber, Josiah Hester, and Przemysław Pawełczak. 2020. Time-sensitive intermittent computing meets legacy software. In *the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 85–99.

[45] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, 75–86.

[46] LLVM. 2021. LLVM Language Reference Manual. Retrieved from https://llvm.org/docs/LangRef.html.

[47] Robin H. Lock, Patti Frazer Lock, Kari Lock Morgan, Eric F. Lock, and Dennis F. Lock. 2020. *Statistics: Unlocking the Power of Data*. John Wiley & Sons.

[48] Brandon Lucia and Benjamin Ransford. 2015. A simpler, safer programming and execution model for intermittent systems. In *the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. 575–585.

[49] Kasper Luckow, Corina S. Păsăreanu, Matthew B. Dwyer, Antonio Filieri, and Willem Visser. 2014. Exact and approximate probabilistic symbolic execution for nondeterministic programs. In *the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. 575–586.

[50] Kiwan Maeng, Alexei Colin, and Brandon Lucia. 2017. Alpaca: Intermittent execution without checkpoints. In *the 32nd Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'17)*. 1–30.

[51] Kiwan Maeng and Brandon Lucia. 2018. Adaptive dynamic checkpointing for safe efficient intermittent computing. In *the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 129–144.

[52] Kiwan Maeng and Brandon Lucia. 2020. Adaptive low-overhead scheduling for periodic and reactive intermittent execution. In *the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20)*. 1005–1021.

[53] Andrea Maioli, Luca Mottola, Muhammad Hamad Alizai, and Junaid Haroon Siddiqui. 2019. On intermittence bugs in the battery-less internet of things (WIP paper). In *the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*. 203–207.

[54] Amjad Yousef Majid, Carlo Delle Donne, Kiwan Maeng, Alexei Colin, Kasim Sinan Yildirim, Brandon Lucia, and Przemysław Pawełczak. 2020. Dynamic task-based intermittent execution for energy-harvesting devices. *ACM Trans. Sensor Netw. (TOSN)* 16, 1 (2020), 1–24.

[55] Deepak Mishra, Swades De, and Kaushik R. Chowdhury. 2015. Charging time characterization for wireless RF energy transfer. *IEEE Trans. Circ. Syst. II: Express Briefs* 62, 4 (2015), 362–366.

[56] M. Yousof Naderi, Kaushik R. Chowdhury, and Stefano Basagni. 2015. Wireless sensor networks with RF energy harvesting: Energy models and analysis. In *the IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1494–1499.

[57] Saman Naderiparizi, Aaron N. Parks, Zerina Kapetanovic, Benjamin Ransford, and Joshua R. Smith. 2015. WISPCam: A battery-free RFID camera. In *the IEEE International Conference on RFID (RFID'15)*. IEEE, 166–173.

[58] Matteo Nardello, Harsh Desai, Davide Brunelli, and Brandon Lucia. 2019. Camaroptera: A batteryless long-range remote visual sensing system. In *the 7th International Workshop on Energy Harvesting and Energy-Neutral Sensing Systems (ENSsys'19)*. 8–14.

[59] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic execution of Java bytecode. In *the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*. 179–180.

[60] Powercast Corp. 2021. Powercast Hardware. Retrieved from http://www.powercastco.com.
[61] Powercast Corp. 2021. Powercast Hardware. Retrieved from https://www.powercastco.com/wp-content/uploads/2.016/11/p2110-evb1.pdf.
[62] R Core Team. 2017. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. Retrieved from https://www.R-project.org/.
[63] Benjamin Ransford, Jacob Sorber, and Kevin Fu. 2011. Mementos: System support for long-running computation on RFID-scale devices. In *the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 159–170.
[64] RAWDAD. 2022. The Columbia/EnHANTs dataset. Retrieved from https://crawdad.org/columbia/enhants/20110407/.
[65] Emily Ruppel and Brandon Lucia. 2019. Transactional concurrency control for intermittent, energy-harvesting computing systems. In *the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. 1085–1100.
[66] Saleae. 2021. Saleae Logic Pro 16 Analyzer. Retrieved from https://support.saleae.com/datasheets-and-specifications/datasheets.
[67] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. 2008. Design of an RFID-based battery-free programmable sensing platform. *IEEE Trans. Instrum. Meas.* 57, 11 (2008), 2608–2615.
[68] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. 2004. Combining component caching and clause learning for effective model counting. In *the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*.
[69] Weinning Song, Xiaojun Cai, Mengying Zhao, Zhaoyan Shen, and Zhiping Jia. 2021. A lightweight online backup manager for energy harvesting powered nonvolatile processor systems. *J. Syst. Archit.* 113 (2021), 101900.
[70] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O dependent idempotence bugs in intermittent systems. In *the 34th Annual ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'19)*. 1–31.
[71] Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2021. Automatically enforcing fresh and consistent inputs in intermittent systems. In *the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 851–866.
[72] Texas Instruments. 2021. EnergyTrace Technology. Retrieved from https://www.ti.com/tool/energytrace.
[73] Texas Instruments. 2021. MSP Datasheet. Retrieved from https://www.ti.com/sc/docs/products/micro/msp430/userguid/as_5.pdf.
[74] Texas Instruments. 2021. MSP430FR5994 Mixed-Signal Microcontroller. Retrieved from https://www.ti.com/product/MSP430FR5994.
[75] Texas Instruments, Inc. 2021. FRAM FAQs. Retrieved from http://www.ti.com/lit/ml/slat151/slat151.pdf.
[76] Marc Thurley. 2006. sharpSAT–counting models with advanced component caching and implicit BCP. In *the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT'06)*. 424–429.
[77] Hoang Truong, Shuo Zhang, Ufuk Muncuk, Phuc Nguyen, Nam Bui, Anh Nguyen, Qin Lv, Kaushik Chowdhury, Thang Dinh, and Tam Vu. 2018. CapBand: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys'18)*. 54–67.
[78] Joel Van Der Woude and Matthew Hicks. 2016. Intermittent computation without hardware support or programmer intervention. In *the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 17–32.
[79] Peter J. M. Van Laarhoven and Emile H. L. Aarts. 1987. Simulated annealing. In *Simulated Annealing: Theory and Applications*. Springer, 7–15.
[80] Kasım Sinan Yıldırım, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemyslaw Pawelczak, and Josiah Hester. 2018. InK: Reactive kernel for tiny batteryless sensors. In *the 16th ACM Conference on Embedded Networked Sensor Systems (SenSys'18)*. 41–53.
[81] Eren Yıldız, Lijun Chen, and Kasim Sinan Yıldırım. 2022. Immortal threads: Multithreaded event-driven intermittent computing on ultra-low-power microcontrollers. In *the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, 339–355. Retrieved from https://www.usenix.org/conference/osdi22/presentation/yildiz.
[82] Mengying Zhao, Chenchen Fu, Zewei Li, Qingan Li, Mimi Xie, Yongpan Liu, Jingtong Hu, Zhiping Jia, and Chun Jason Xue. 2017. Stack-size sensitive on-chip memory backup for self-powered nonvolatile processors. *IEEE Trans. Comput.-aid. Des. Integ. Circ. Syst.* 36, 11 (2017), 1804–1816.

Received 22 January 2022; revised 20 July 2022; accepted 4 September 2022