Termination Checking and Task Decomposition for Task-Based Intermittent Programs

Alexei Colin Brandon Lucia
Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

Emerging energy-harvesting computer systems extract energy from their environment to compute, sense, and communicate with no battery or tethered power supply. Building software for energy-harvesting devices is a challenge, because they operate only intermittently as energy is available. Programs frequently reboot due to power loss, which can corrupt program state and prevent forward progress. Task-based programming models allow intermittent execution of long-running applications, but require the programmer to decompose code into tasks that will eventually complete between two power failures. Task decomposition is challenging and no tools exist to aid in task decomposition.

We propose CleanCut, a tool that can check for and report non-terminating tasks in existing code, as well as automatically decompose code into efficient, terminating tasks. CleanCut is based on a statistical model for energy of paths through the program. We applied a prototype of CleanCut to four applications, including pattern-recognition, encryption, compression, and data filtering. Our experiments demonstrated the risk of non-termination in existing code and showed that CleanCut finds efficient task decompositions that execute 2.45x faster on average than manually placed boundaries.

CCS Concepts • **Computer systems organization → Embedded systems**;

Keywords intermittent computing; energy estimation

ACM Reference Format:
Alexei Colin Brandon Lucia. 2018. Termination Checking and Task Decomposition for Task-Based Intermittent Programs. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3178372.3179525

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CC'18, February 24–25, 2018, Vienna, Austria
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179525

1 Introduction

Recent advances in energy-harvesting technology, and the advent of extremely low-power computing devices has enabled computer systems powered entirely by energy extracted from their environment. Without a battery or tethered power, these devices are the key to emerging applications, like the internet of things (IoT) and implantable or ingestible medical devices [1, 24]. A typical energy-harvesting device collects energy into a small energy buffer (i.e., a capacitor), until a threshold level, at which the device begins to run. When operating, the device consumes energy very quickly, depleting the capacitor and powering down. These devices operate *intermittently* as energy is available in the energy buffer.

Recent work identified key progress [38] and memory consistency [30, 37] challenges faced by programmers targeting energy-harvesting systems. Other work proposed mechanisms to support non-trivial intermittent applications [3, 4, 9, 10, 19, 20, 30, 38, 47]. *Task-based intermittent programming models* [10, 30] ensure long-running applications execute correctly on intermittent devices. Such a model asks the programmer [30] or compiler [47] to decompose an application into tasks that execute atomically. Checkpointing volatile state and versioning non-volatile state makes tasks restartable, but does not ensure that a task re-execution after a power failure will have sufficient energy to complete. A task will run to completion only if it consumes less energy than the capacity of device's energy storage buffer. To complete tasks reliably, the device can depend only on stored energy and not on extra energy that might be harvestable from the environment during operation.

Task decomposition must be performed for existing task-based systems, but it is difficult to reason about how likely a task is to exhaust the buffered energy. An overly cautious programmer may place more task boundaries in code than necessary, wasting energy and imposing a time overhead. If, the programmer uses too few boundaries, the program may have a *non-terminating path* that requires more energy than the device can buffer. A non-terminating path consumes more energy than will ever be available, causing the task to repeatedly restart and fail forever. Code including such a non-terminating path represents a new type of software bug that is unique to intermittent applications. There is currently *no system support* to help find these bugs by assessing whether a task boundary placement includes non-terminating paths, nor for helping place task boundaries.

This work is the first to characterize power-failure-related non-terminating path bugs in intermittent programs. We develop CleanCut, the first system for *finding* non-terminating paths in intermittent programs and *eliminating* such paths by generating terminating task boundary assignments automatically. CleanCut's *checker* checks a task boundary assignment and reports non-terminating paths that need refinement. CleanCut's *placer* subdivides a program into tasks free of non-terminating paths. CleanCut minimizes overhead by approximately bisecting paths and preferring boundaries unlikely to be executed frequently.

Both the checker and placer use CleanCut's statistical model of the energy consumption of each program path. Following the insights in [26], we base our path model on a lower-level model of energy of *branch-free basic blocks*. CleanCut's top-level path energy model is compatible with both worst-case or average-case block energy models based on profiling and analytics [17, 26], performance counters [11], or simulation [6]. CleanCut ships with a simple average-case profiling-based block energy model.

We implemented CleanCut's analyses in LLVM and applied them to applications from prior work [10, 30, 31]. We show that CleanCut's checker identifies task boundary assignments with non-terminating path bugs, demonstrating its value as a debugging tool. We show that CleanCut's placer produces boundary placements that are free of non-terminating paths and have lower overhead than manually- or randomly-placed boundaries. To summarize our contributions:

- We develop the CleanCut task checker that finds non-terminating path bugs.
- We develop the CleanCut placer that places task boundaries, eliminating non-terminating paths and minimizing boundary overhead.
- We statistically model the energy cost of program paths with loops and I/O in terms of basic block energy.
- We evaluate CleanCut on real energy-harvesting hardware and demonstrated detected non-termination bugs and placements that outperform manual decompositions.

Section 2 reviews intermittent computing and Section 3 overviews CleanCut. Sections 4 and 5 describe CleanCut's Checker and Placer . The energy model is presented in Section 6. Section 7 provides implementation details, Section 8 evaluates CleanCut, and Section 9 discusses related work.

2 Background and Motivation

Embedded computers [43] and energy-harvesting devices (e.g., WISP5 [39]) are finding widespread adoption. Hardware advances have spawned research into general programming and execution models for software on *intermittently-powered* devices [3, 4, 7, 10, 20, 30, 38].

2.1 Energy-harvesting Devices

Energy-harvesting devices are embedded computing platforms composed of a microcontroller and peripherals such as sensors and radios. These devices extract their energy from the environment, e.g. radio waves, vibration, or a thermal gradient. Harvested energy sources are typically too weak (by orders of magnitude) to directly power a device, requiring devices to buffer energy in a capacitor. After buffering a threshold amount of energy the device turns on and begins executing software. Executing consumes energy more quickly than it accumulates, depleting the buffer and causing the device to power off. The active period of a device depends on the size of its energy buffer. A typical energy-harvesting device [39] may power cycle hundreds of times per second.

From the perspective of software, each power cycle is a reboot that impedes the progress of the computation. Volatile state of the device, including its register file, stack memory, and global variables, is erased, while non-volatile memory (e.g., FRAM [43]) retains its state across failures. Recent work [10, 30, 37, 47] observed that when volatile memory erases and non-volatile memory persists, reboots leave program state inconsistent. The issues with progress and consistency inspired research on compiler and system support for *intermittent programming models*.

2.2 Intermittent Programs and Execution Models

There are various intermittent programming and execution models each with different correctness and performance characteristics. The first efforts in this area focused on scheduling computations to complete under energy constraints [7, 41] and did not directly address intermittence. Later work enabled long-running computations on intermittently-powered devices, relying on checkpoints of volatile state [3, 4, 20, 32, 38] and versioning of non-volatile state [10, 30] with varying automation from the compiler [31, 47].

Task-based models [7, 10, 30, 31, 41] require programmers to manually decompose code into tasks by adding task boundaries to a program in a C-like language. The quality of a task boundary placement dictates whether a program terminates and determines the time and space overhead of the system. Task-based models maintain progress at the granularity of a task. Consequently, if any path through a task consumes more energy than the device can buffer, program execution will not advance past that task. Such *non-terminating path bugs* cause the program to partially execute a task repeatedly, each time failing to reach the task's terminal boundary.

To avoid these non-termination bugs, the programmer must reason about the energy that a task consumes along each of its control-flow paths. A programmer may attempt naively to avoid non-termination by inserting many boundaries (e.g., after every operation), but each boundary imposes an overhead to capture a checkpoint [20, 30, 38], commit a log [31], or store multi-versioned state [10]. Moreover, our data in Section 8 suggest that programmers might do a poor job of judging the energy cost of code regions.

Figure 1 shows how different static task boundary assignments lead to different behavior with three variants of an

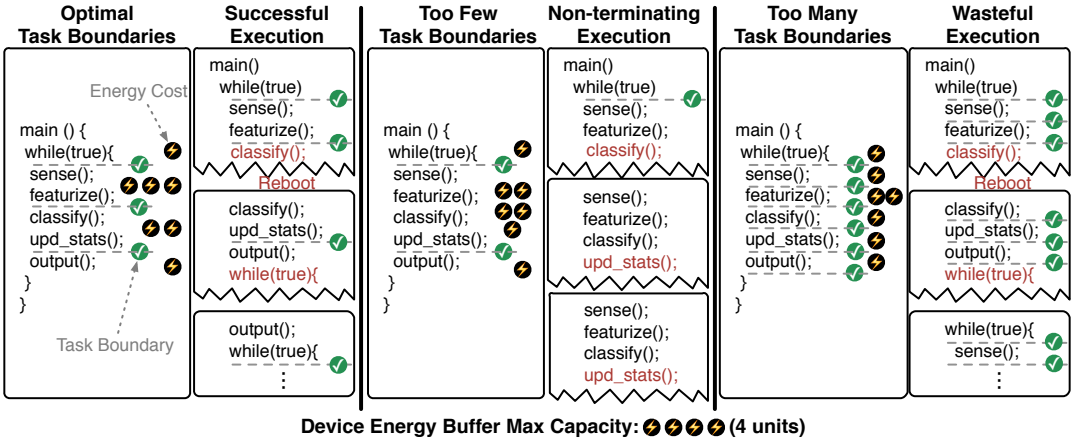

Figure 1. Different task decompositions cause different execution behavior.

activity recognition application from prior work [10, 30, 31]. The code featurizes and classifies data from a sensor, maintains statistics, and produces output. The energy cost of a task is illustrated in terms of abstract energy units, represented by the lightning bolt circles between the task's initial and terminal boundary. The figure assumes a device that can buffer at most four energy units. The left variant of the program is decomposed into tasks using three boundaries. The energy consumption of the resulting tasks does not exceed the device's energy capacity and the depicted execution makes progress with little boundary overhead, despite periodic reboots. The middle variant has a non-terminating path bug because it is decomposed with *too few* boundaries. This variant's most costly task consumes more energy than the device can buffer. Consequently, the application can never make progress, rebooting and re-executing the task indefinitely. The right variant is inefficient because it uses *too many* task boundaries because the energy cost of each task fits within the device capacity and boundaries execute more often than necessary, wasting time and energy.

Despite the importance of placing task boundaries, doing so remains a difficult, manual process for which there is no system support. The programmer must draw a correspondence between a code span and its energy cost, accounting for variation across inputs and the energy consumption of the full system, including peripherals. The connection between code and energy capacity of the device is opaque. The compiler provides no feedback about a task boundary placement. Instead, the programmer is left to guess whether tasks will terminate, or if boundary overhead will throttle throughput. To port to another platform, the programmer must decompose the code again. Adoption of task-based intermittence models is impeded by the lack of support to *check* that a decomposed program is free of non-terminating path bugs and to *place* task boundaries to avoid these bugs by construction. CleanCut fills both of these gaps.

3 CleanCut Overview

CleanCut is both a debugging tool and a program transformation analysis that helps a programmer place task boundaries in a program written for a task-based intermittent execution model. CleanCut has two modes of use, as a *checker* (Section 4) or as a *placer* (Section 5). Both analyze *paths* through a program, i.e. sequences of basic blocks allowed by the edges in the control-flow graph, and rely on a statistical model for the energy of a path (Section 6).

CleanCut's checker is a *debugging tool* that examines a task boundary placement and checks for non-terminating path bugs. A non-terminating path bug stems from a misuse of task boundaries that allows a path through a task to consume more energy than the maximum amount of energy that the target device can buffer. The program's source and the energy buffering capacity of the target device are inputs to the checker. If the checker finds a path that consumes more energy than the device can buffer (i.e., a non-terminating path bug), the checker reports the path to the programmer, along with the boundaries of the task containing the non-terminating path. The programmer can then adjust the task boundaries — by moving existing boundaries or adding new ones — to eliminate the bug. The checker is particularly useful to a programmer that prefers fine-grained manual control over boundaries to ensure that, for example, related I/O operations execute in the same task.

CleanCut's placer is a *program transformation* that adds task boundaries to a program to avoid non-terminating path bugs. The goal of the placer is to produce a task boundary assignment that is free of non-terminating paths and that minimizes the overhead of executing task boundaries. The placer works iteratively and each iteration evaluates the current task boundary assignment to identify non-terminating paths. The placer selects the non-terminating path of highest energy cost to subdivide, and inserts a new task boundary along the path to divide the path into two sub-paths of approximately equal energy cost. To minimize task boundary overhead, the placer avoids placing boundaries in loops with

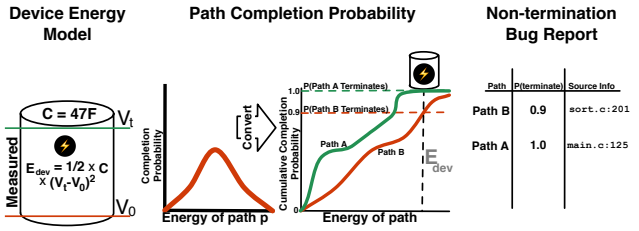

Figure 2. CleanCut Checker reports the termination probability of each path. CleanCut estimates the termination probability by evaluating the CDF of the path energy distribution at the energy capacity of the device.

a high iteration count and in functions that are called from many call sites. The placer is most useful to a programmer that has fewer platform-specific requirements in their application, and benefits more from a fully-automated workflow.

4 CleanCut Non-termination Checker

CleanCut's non-termination checker evaluates a task decomposition to report non-terminating paths to the programmer if any exist. CleanCut compares an estimate of the energy of each path to an estimate of the storage capacity of the energy buffer on the device and identifies non-terminating paths *statistically*, relying on the distributional representation of path energy consumption from the model. If there is a non-zero probability that a path energy exceeds the storage capacity, then CleanCut reports the paths along with the non-termination probability.

Modeling Device Energy Capacity. Energy available to execute a path is determined by the size of the capacitor installed on the device. To estimate the *effective* energy capacity, illustrated on the left in Figure 2, CleanCut measures energy consumed starting from the first application task until power failure, as described in Section 7.1. The estimate is the minimum observed sample. We assume that variations in capacity at runtime induced by temperature or degradation are negligible.

To identify non-terminating paths, CleanCut estimates the energy consumption for each path, as a distribution over energy values. The algorithm for estimating the probability density function (PDF) of each path energy is part of the energy model, described in Section 6. CleanCut then compares the path energy to the energy storage capacity of the device, as is shown in the center of Figure 2. The first step of the comparison is to transform a path's PDF into a cumulative density function (CDF) by integrating the PDF with respect to energy. The value of a path's PDF at a particular energy level represents the likelihood that the path will consume that much energy. The value of a path's CDF at a particular energy level represents the likelihood that the path will consume that amount of energy *or less*. The value of a path's CDF at the device's energy buffering capacity thus corresponds to the probability that the path is *not* a non-terminating path. We report potentially non-terminating paths to the programmer, each with its corresponding non-termination probability, as shown on the right in Figure 2.

Algorithm 1 CleanCut program decomposition algorithm.

```
1:  function DECOMPOSE(CFG G, device model D)            ▷ program G on device D
2:      B ← ∅                                            ▷ Initialize set of boundary locations
3:      do
4:          ▷ Evaluate the energy model and return max e s.t. Prob(energy = e) > 0
5:          P ← CALCPATHENERGIES(G, B)                   ▷ Stores energies into field Energy[]
6:          p ← arg max_{p∈P} Energy[p]                  ▷ Pick path of maximum energy
7:          if Energy[p] > Capacity[D] then              ▷ Is path predicted to exceed capacity?
8:              if |p| > 1 then                          ▷ Only splits at block granularity are supported
9:                  b ← SPLITPATH(p, D)                  ▷ Place a boundary
10:                 B ← B ∪ b                            ▷ Add the boundary to the decomposition
11:             else
12:                 return "NO PLACEMENT EXISTS"
13:     while Energy[p] > Capacity[D]
14:     return B
15: function SPLITPATH(path p, device model D)
16:     m ← 1 + arg max_k Σ_{i=0}^{k} Energy[p_i] < Energy[p]/2 for 0 ≤ k < |p|
17:     for i ← 0 to m do
18:         if IsLoop[p_i] ∧ Energy[p_i] > Capacity[D] then
19:             L ← BodyPaths[p_i]                       ▷ Block p_i is loop head, get loop body paths
20:             l ← arg max_{l∈L} Energy[l]
21:             return SPLITPATH(l, D)
22:     return arg max_{s∈[1,m]} (Σ_{k∈[1,s]} Energy[p_k] / Σ_{t∈[1,m]} Energy[p_t]) + DYNBOUNDARIES(p_s, p) / max_{t∈[1,m]} DYNBOUNDARIES(p_t, p)
23: function DYNBOUNDARIES(block b, path p)              ▷ Estimates dynamic transitions
24:     return Σ_{b∈InlinedInstancesOfBlock[p_s]} Π_{loop L | b∈L} LoopBound[L]
```

5 CleanCut Task Boundary Placer

The CleanCut task boundary placer inserts boundaries into a program to eliminate non-terminating paths while minimizing boundary overhead. The placer's core is the greedy algorithm listed in Algorithm 1. The main loop in DECOMPOSE repeatedly divides the path with the highest energy cost by placing a boundary along the path. Each iteration begins with estimating the energy for all paths through the program (Line 5) according to the energy model (Section 6) and storing the estimate as a distribution in the Energy[] field of each path object. For the division and comparison operations (but not addition), the distribution is reduced to a scalar value. The reduction operator is configurable to either the expectation or the maximum observed value; to model worst-case behavior we use the latter with the energy model from Section 6.

The algorithm then selects the highest energy path (Line 6) and, if its energy cost exceeds the device energy capacity (Line 7), the algorithm calls SPLITPATH to choose a location on the path for a boundary (Lines 8-10) using criteria explained in Section 5.1. The set of paths P is recomputed on the next iteration, because the new boundary affects not only the path being split but also all paths with a call to the function that contains the new boundary. The placer completes when the costliest path is within the energy capacity of the device (Line 13).

The algorithm must divide looping paths with a high energy cost, even if those looping paths are contained within an abstract loop block (Section 6.2.1). If the traversal over blocks in a path encounters an abstract loop block (Line 18), the algorithm *descends* into the abstract loop block if the energy cost of the loop exceeds capacity (Line 18) and inserts a boundary along the most costly path in the loop

body (Lines 19-20). A boundary placed along a path through a loop, invalidates the energy estimate for that loop until it is recomputed in the main loop (Line 5).

5.1 Minimizing Task Boundary Overhead

The location of a boundary determines its run-time energy and time overhead. Given a path p, SplitPath finds the location in p where a boundary will have the least impact. The algorithm identifies the *energy midpoint* of the path, i.e., the block at which energy accumulated from either end of the path is below half of the total path energy (Line 16). SplitPath places the boundary at one of the *candidate* split points between the start and the midpoint of the path. The algorithm could consider the split points between the midpoint and the end of the path but does not in order to save time.

SplitPath assigns each candidate split block a score and chooses the candidate with the highest score (Line 22). The *split score* measures the impact of a boundary using two components: the relative energy of the two segments after the split and the expected number of dynamic task boundaries. A static task boundary at block b leads to as many dynamic task boundaries as there are calls to b's parent function and iterations of (nested) loops that include b at runtime. Function DynBoundaries (Line 24) estimates the dynamic calls and loop iterations from the inlined version of the program CFG — where each call is recursively replaced with the callee's blocks — and from loop bounds. The placer algorithm assumes that the best candidate split point for a boundary is the one that leads to the fewest dynamic boundaries.

5.2 Placer Algorithm Analysis

Correctness. *If a placement exists that is free of non-terminating paths according to CleanCut's energy model, then the placer algorithm will find such a placement; otherwise it will report failure.* A *valid placement* is a placement for which CleanCut's model indicates that all path energy costs are below the device energy capacity. If the loop in Decompose terminates, then placement B is valid, because the negation of the loop condition (Line 13) implies that the maximum-energy path p is below capacity, which implies that all paths are below capacity. The loop in Decompose terminates if the least upper bound on the energy of a path in set of paths P^i (set P in iteration i) strictly decreases, i.e. $\max_{p \in P^i} \text{Energy}[p] > \max_{p \in P^{i+1}} \text{Energy}[p]$, because the right hand side of the inequality in the loop condition (Capacity[D]) is constant. The least upper bound on energy of P decreases in iteration i, if SplitPath is called on the maximum-energy path in iteration i and if SplitPath decreases the least upper bound.

SplitPath is called on all but the last iteration, because on the iteration in which SplitPath is not called, either the condition in Line 7 is false, which implies the loop condition is false, or the condition on Line 8 is false which violates the premise that a valid placement exists (i.e., some block exceeds the energy capacity). SplitPath decreases the least upper bound on energy of paths in P, because SplitPath inserts a boundary at the block at index $s \in [1, m]$ in maximum-energy path p (maximum selected in Line 22), which excludes at least block p_0 from the maximum-energy path in the next invocation of CalcPathEnergies (Line 5). That the maximum-energy path is shortened follows from the fact that (1) energy is strictly increasing in the number of blocks in the path, regardless of the type of the block, (2) boundaries are strictly appended to the set of boundaries B, and (3) adding a boundary to program CFG G with boundaries B cannot increase the length of any path in G.

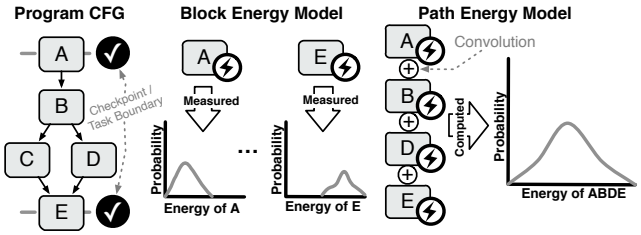

Figure 3. CleanCut models the energy of each path.

Complexity. Let $W(n, e)$ be the number of blocks traversed by the greedy placer algorithm for a program with n paths and the costliest path of energy e. At each iteration of the outer loop, the algorithm splits one path, which may create boundaries on every path in the worst case, doubling the number of paths for the next step, but cutting the maximum energy in half (since the split is done near the energy-midpoint). That is, $W(n, e) = n + W(2 * n, e/2)$ with $W(n, e) = n$ for $e < C$, where C is the device capacity. The recurrence is bounded by $O(n * 2^{\log e + 1})$.

6 CleanCut Energy Model

CleanCut's non-termination bug checker and task boundary placer both rely on a statistical model of the energy consumed by each control-flow path from one task boundary to another. The model computes a path's energy by combining the energy of its constituent basic blocks. We chose to model path energy based on basic block energy, as opposed to single instruction energy, to produce estimates closer to the observable average case rather than the theoretical worst-case, following the insights in [26]. In addition, since profiling is part of the programmer's workflow in CleanCut, we avoid relying on high-resolution measurement hardware to collect per-instruction estimates. With a block-based model, as with a *detailed* instruction-level model, energy estimates must be recomputed as code changes.

6.1 Block Energy Model

A basic block energy model compatible with CleanCut represents the consumed energy as a *probability distribution* that indicates how likely the block is to consume different amounts of energy. A distributional model captures the range

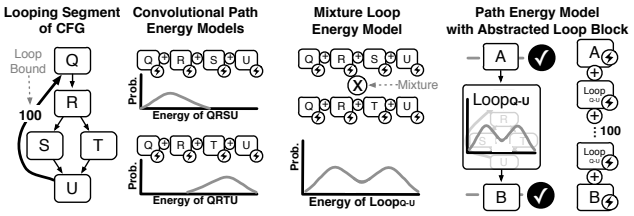

Figure 4. CleanCut's loop model mixes path energy models. The figure assumes uniform path likelihood in the loop, but CleanCut can weight paths in a loops mixture using a path profile.

of possible energy costs of a block, which is necessary to estimate the probability of a non-termination bug manifesting. Figure 3 on the left illustrates that CleanCut measures an energy distribution for each block in a program's control-flow graph, using the procedure described in Section 7.2.

We chose to make CleanCut's energy model *distributional* to avoid losing information about possible path energy consumptions. A distributional model captures more information than a scalar worst-case energy model about whether a non-termination bug will manifest. A distributional model lets CleanCut's non-termination bug checker report the likelihood that a path will not terminate to the programmer. Reporting non-terminating path bugs with their manifestation likelihood enables the programmer to prioritize potential non-termination issues.

CleanCut is designed to accept any distributional block energy model that can represent the distribution as a discrete histogram. Our prototype implementation of CleanCut uses a measurement-based block energy model, because it accounts for the total energy of the board, including sensors and radios, does not rely on any models of low-level circuit power behavior, and was effective in our evaluation (Section 8). The potential drawback of this measurement-based prototype is that it may not capture all of a block's energy behaviors, potentially underestimating the block's worst case energy as the maximum energy *observed* during measurement.

The potential drawbacks of the measurement-based model in our prototype are *not* inherent to CleanCut's path modeling approach, however, and CleanCut could instead use an analytical block model derived from device characteristics and application analysis [8, 23, 26, 45]. Using an analytical model has the advantage of being able to estimate *theoretical* worst-case energy, and can provide estimates that cover all program inputs. However, analytical models, too, have drawbacks: analytical models typically capture only processor power since other board components like sensors and radios require fundamentally different modeling methodologies. As better energy models arise, CleanCut can incorporate them.

6.2 Path Energy Model

CleanCut uses the block energy distributions to compute the energy distribution of each path in the program, as illustrated in Figure 3 on the right. CleanCut's path energy model accumulates the cost of blocks along a path from its *initial* task boundary to its *terminal* task boundary. A path is a non-branching sequence of basic blocks (Section 6.1), loop blocks (Section 6.2.1), or opaque blocks (Section 6.2.2). CleanCut's target programming model does not support recursion, which is uncommon in embedded software where predictability and static resource bounds are often required.

To compute path energy, CleanCut must aggregate the energy of the various types of blocks that comprise the path. If CleanCut represented block energy with a scalar, then it could calculate the energy of a path by simply adding the energy costs of the blocks that make up that path. However, CleanCut represents the energy of each block as a distribution, which precludes simple addition. To accumulate block costs, CleanCut *convolves* the energy distributions for the blocks along the path. Convolving the distributions for two random variables (i.e., two block energy distributions) produces a distribution for the random variable that is their sum. Any two arbitrary distributions can be convolved. CleanCut sequentially convolves blocks on a path yielding a distribution representing the energy cost of the path.

6.2.1 Loops

CleanCut handles loops by encapsulating their energy cost in an *abstract loop block*, as shown in Figure 4. CleanCut abstracts a loop's body by using a single distribution to represent the energy cost of all paths from the head of a loop to its back edge. A nested loop is recursively abstracted and incorporated into a path through the parent loop. Along a path with a loop, CleanCut convolves the loop body's energy distribution once per loop iteration along with the distributions of the other blocks on the path.

A loop body with many control-flow paths has a *modal* energy distribution, with a mode at the expected energy cost of each path. As illustrated in Figure 4, CleanCut computes this modal distribution by *mixing* the distributions for each of the paths through the loop body. To produce an energy model for a path containing a loop, CleanCut convolves the loop body's distribution with the path energy distribution a number of times equal to the estimated loop bound. By default, the loop body's mixture model *uniformly* combines the intra-loop distributions, treating each path through the loop as equally likely. CleanCut provides an implementation of Ball-Larus path profiling [2] that can determine the likelihood of each path by monitoring representative executions to use as weights in the mixture.

Loops present two main challenges to any analysis. First, the iteration count of an unbounded loop is statically unknowable. Second, an efficient analysis must not unroll the loop. To account for the iteration count of a loop, CleanCut requires the programmer to provide a *bound estimate*, as depicted on the arc (U, Q) in Figure 4. For unbounded loops, CleanCut gives the programmer a choice of either providing an annotation statically bounding its iteration count (similar

Algorithm 2 CleanCut path energy estimation algorithm.

```
1:  function CalcPathEnergies(CFG G, block b)
2:      if IsLeaf[b] ∨ IsBoundary[b] ∨ IsLoopSucc[b] then
3:          return {0}, ∅
4:      E ← ∅,    S ← ∅                          ▷ Path energies and successors
5:      if ¬ IsLoopHead[b] then                   ▷ Add energy of a block
6:          for s ∈ Successors[b] do
7:              E_s, S_s ← CalcPathEnergies(G, s)
8:              E ← E ∪ E_s,    S ← S ∪ S_s
9:      else                                      ▷ Add energy of a loop
10:         e_l ← 0,    S_l ← ∅                   ▷ Loop energy and successors
11:         for s ∈ Successors[b] do
12:             E_s, S_s ← CalcPathEnergies(G, s)
13:             E_l ← {e ∈ E_s : EndsAtBackedge[Path[e]]}
14:             e_l ← e_l ⊗ e for e ∈ E_l
15:             E ← E ∪ (E_s \ E_l)
16:             S_l ← S_l ∪ S_s
17:         e_l ← e_l × LoopIters[b]
18:         for s ∈ S_l do                        ▷ Add loop to paths after the loop
19:             E_s, S_s ← CalcPathEnergies(G, s)
20:             E ← E ∪ {e_l ⊕ e : e ∈ E_s}
21:             S ← S ∪ S_s
22:     if E ≠ ∅ then return {e ⊕ Energy[b] : e ∈ E}, S
23:     else return {0}, ∅
```

to k-bounded [14] profiles and often simple for embedded applications) or forcing a task boundary inside the loop, which effectively eliminates the loop from the task. For the former choice, to help the programmer determine the loop iteration count, CleanCut has a loop iteration count profiler that can measure the histogram of a loop's iteration counts. The accuracy of the profile for dynamically-bound loops is limited by the sensor inputs during profiling.

6.2.2 I/O Operations

CleanCut accounts for the energy cost of I/O operations. The energy of I/O that is contained within a basic block is accounted within the energy for the containing block. Composite multi-block I/O operations (e.g. polling a peripheral) are abstracted into *opaque blocks*. CleanCut measures the energy distributions for opaque blocks *in-place* during a dedicated instrumented run of the application.

6.3 Evaluating the Energy Model in the Compiler

CleanCut's compiler uses the recursive procedure shown in Algorithm 2 to calculate each path's energy probability density function (PDF). Before the algorithm runs, a preliminary pass splits any basic blocks with a call instruction and inlines the callee's blocks, recursively. The traversal starts at the entry block and recursively descends along each path until a task boundary or a program-terminating block (Line 2). A recursive call (Line 7) returns a list of energy distributions, E_s, for paths that start from the intermediate node and a list of entry blocks into successor tasks, S_s. Each frame adds the current block's energy to each sub-path that starts at a block's child by convolving ($\oplus$) the distributions (Lines 8, 22) and the current block's successors list, S, is extended with its children's successors, S_s (Line 8). To add the energy of a k-iteration loop to a path, the pass recursively computes the energy of each loop body path (Line 12), mixes them ($\otimes$)

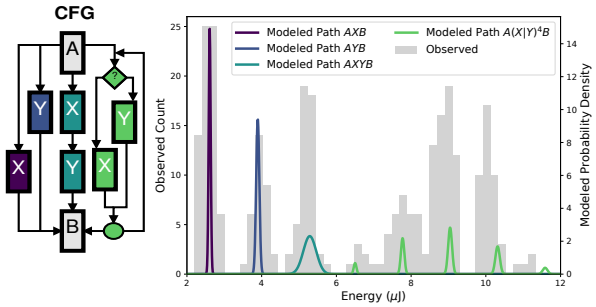

Figure 5. Modeled and observed distributions for energy of four paths through a benchmark application (left). The match between locations of the modes on the x-axis validates that CleanCut modeling abstractions correctly represent energy behavior.

(Line 14), and convolves the resulting block with itself ($\times$) k times (Line 17). The loop energy is then convolved with each path starting after the loop (Line 20). The set of loop body paths E_l excludes paths that descend into the loop body but reach a task boundary before a backedge (Line 15).

6.4 Energy Model Benchmarking

We applied the path energy computation to a microbenchmark to show that the distribution computed by recursive convolutions and mixtures matches the measured energy of the path. Figure 5 shows a CFG with four paths comprising simple sequences of blocks and a loop. Each path is composed of three or more blocks of four types, labeled A, B, X and Y, that differ in energy cost. Branches are decided uniformly randomly. The probability density function (PDF) curves in Figure 5 show each path's estimated energy distribution. [1] The bars in the plot show path energies measured on the WISP [39] during the 294 independent executions of the program over 5 minutes. There is no correspondence between the scales of the left and right y-axes beyond the relative heights of modes *within data for a single path.*

The key result is that the x-axis position of peaks in a path's modeled distribution corresponds to the path's peak in the observed energy values. The match for path $AXYB$ shows that the energy cost of a sequence of blocks, XY, is correctly modeled by the convolution of energy distributions for X and Y. The match for each of the 5 modes in the distribution for path $A(X|Y)^4B$ shows that the cost of a loop is correctly modeled by a mixture of energy distributions of the paths through its body. The data also show that CleanCut underestimated path energy variance and overestimated values in the upper range. CleanCut derives its variance estimate from the variance of energy of individual blocks and block energy variance is smoothed because block measurements are an average of replicas (Section 7.2). CleanCut overestimates block energy values in the upper range (10-12 μJ), because CleanCut's block model conservatively assumes that all blocks execute at the maximum voltage, consuming their worst-case energy.

[1] For a PDF f, $f(x)$ may exceed 1, but $\int f(x)\,dx \leq 1$.

7 CleanCut Implementation

The toolchain is organized as a tree of dependent analysis phases in GNU Make, with the checker and placer results near the root and requisite models and profiles at intermediate and leaf nodes. Independent phases run in parallel.

7.1 Energy Measurement

CleanCut programatically controls the Energy-interference-free Debugger (EDB) [9] connected to the capacitor on the target device to measure energy. For each measurement, CleanCut places two voltage watchpoints in the application code and EDB records the capacitor voltage at the watchpoints. Energy consumed between the watchpoints depends on the watchpoint voltage measurements, V_{from} and V_{to}, and device capacity, C, as $E = \frac{1}{2}C(V_{\text{from}}^2 - V_{\text{to}}^2)$. Using EDB, CleanCut directly measures full-system energy, including the energy consumed by peripherals, e.g. sensors and actuators.

Using our energy measurement setup, we measure the energy storage capacity on the device and block energy costs. Assuming V_{on} is the voltage when the initialization completes and the first application task begins and V_{off} is the MCU's brown-out threshold, CleanCut computes the *effective* capacity using $V_{\text{from}} = V_{\text{on}}$ and $V_{\text{to}} = V_{\text{off}}$. V_{on} is measured by running the application binary with an EDB watchpoint after power-on code. V_{off} is set from the MCU's specification (we validated that $V_{\text{off}} = 1.8 \pm 0.002$V for our MSP430FR5969 using an EDB watchpoint).

7.2 Block and Path Energy

To measure a block's energy cost, CleanCut extracts assembly generated by LLVM's backend for the target architecture, translates the instruction arguments to make the block runnable outside its context, replicates it, and inserts it into a harness binary for measurement. To make the block safe to execute repeatedly outside of its context, CleanCut replaces register references with a designated "scratch" register and memory references with random addresses in a designated range. CleanCut generates harness code with the application's clocking and peripheral configuration to reflect true energy consumption. After running the harness binary on the device for 20s and tracing watchpoints, CleanCut calculates the block energy from watchpoints as described in Section 7.1. CleanCut replicates the block being measured in the harness, to ensure that the measured energy is above EDB's watchpoint measurement resolution. The block's energy cost is the energy cost of the sequence of replicas, divided by the replication factor. After a code change, CleanCut only profiles blocks that changed.

To estimate the path energy distribution (PDF) described in Section 6.2, an LLVM pass first traverses the CFG according to Algorithm 2. The pass assembles an expression that symbolically represents the path energy distribution as a sequence of convolutions and mixtures of block distributions. To evaluate the resulting expression to a numerically-represented probability density function (PDF), CleanCut computes convolutions using NumPy [16] and mixtures as an element-wise linear combination of input PDFs.

7.3 Checker and Placer

The checker computes a cumulative distribution function (CDF) by integrating the PDF that represents path energy using Simpson's method in SciPy [21]. CleanCut uses the CDF to determine a path's failure likelihood for a given device energy capacity C by finding the probability value at the closest index below C in the CDF's array representation. The same CDF can be used to validate for a range of capacities.

We implemented the placer (Algorithm 1) in an LLVM pass that incorporates the path energy model. The pass selects the blocks at which to place task boundaries according to the traversal of the CFG in Algorithm 1. The placer invokes the DINO [30] LLVM passes to insert checkpointing and versioning code at each boundary marker.

8 Evaluation

In this section, we evaluate CleanCut to show that the checker's validations are useful, the placer is flexible and its task decompositions efficient, and analysis time is practical for a real developer. We applied CleanCut to real code on real energy-harvesting hardware. We used the WISP [39] energy-harvesting device, which has an 8MHz MSP430FR5969 MCU with 64KB of non-volatile memory and a 47μF capacitor. We powered the WISP wirelessly using a ThingMagic Astra-EX RFID reader at 16 dBm. We fixed the WISP 45 cm from the power antenna, parallel to its surface.

8.1 Benchmarks

We evaluated CleanCut on four energy-harvesting applications from prior work [10, 30]. Activity Recognition (AR) classifies 8 windows with 8 accelerometer samples each into two activity classes based on a pre-trained model. RSA encrypts an 11-character plaintext with a 32-bit public in non-volatile memory. Cuckoo Filter (CF) exercises a Bloom-filter-like set membership structure that supports deletion. CF inserts 64 pseudo-random keys and then queries for each. The Cold-chain Equipment Monitor (CEM) records 64 temperature readings from a sensor, LZW-compresses them, and stores the result into non-volatile memory.

8.2 Placer Evaluation

We evaluated how well CleanCut's placer helps to insert task boundaries into a program to avoid non-terminating paths. The evaluation shows that CleanCut's decompositions are superior to the programs' original, manually placed boundaries and random placements. Our results also show that CleanCut provides flexibility to changing hardware, while avoiding non-terminating placements.

8.2.1 Performance

The main result of our placer evaluation is that CleanCut produces higher-quality, more efficient placements than a modular, manual decomposition strategy and a large number of randomly-generated potential boundary placements. We assess the quality of a decomposition by measuring the run time of the decomposed application on the real device. Modular decomposition is an intuitive manual approach of placing a task boundary at the entry of each major function or outer loop, and is currently the main approach to task definition. Random decompositions place boundaries at basic blocks chosen uniformly at random from the CFG. Random decompositions systematically quantify the missed opportunity for performance improvement due to poor task boundary placement. We generated 10 random decompositions for each possible boundary count between 1 and 10, for a total of 91 distinct decompositions (there is only one one-boundary placement because CleanCut requires a boundary at the top of main). We measured execution time by wrapping the main function with EDB watchpoints that collect timestamps when hit.

Figure 6 compares run times for CleanCut, manual modular, and random decompositions. CleanCut consistently outperforms the modular decomposition, with a harmonic mean speedup of 2.45x. CleanCut also outperforms all *terminating* random decompositions for AR and CEM, and is slower only than 2 out of 78 random placements for RSA and 7 out of 68 for CF. Several of the random decompositions that are slower than CleanCut are slower by an order of magnitude or more. The placer's decompositions are more efficient, because they contain fewer boundaries than manual and random decompositions, incurring less checkpointing overhead. The low boundary count is a benefit of CleanCut's energy model: the placer's algorithm splits the path with the highest energy cost maximally amortizing boundary cost across the largest available span of code. In contrast, manual decompositions have many boundaries, because the authors of these applications were conservative and relied on intuition alone to estimate task energy cost. The overly conservative assumptions lead to the high overhead in Figure 6.

8.2.2 Adaptation to Changing Hardware

CleanCut is parameterized by the energy storage capacity of the target device. This flexibly lets the user apply CleanCut as hardware specifications change. The manual decomposition strategy lacks flexibility: A decomposition that is valid on one device, may not terminate on a device with a smaller energy buffer, and may be inefficient on a device with a larger buffer. Figure 7 shows how CleanCut selects a different boundary count for different target energy capacity. The counterintuitive increase in boundary count with capacity is a result of the placer 's greedy algorithm.

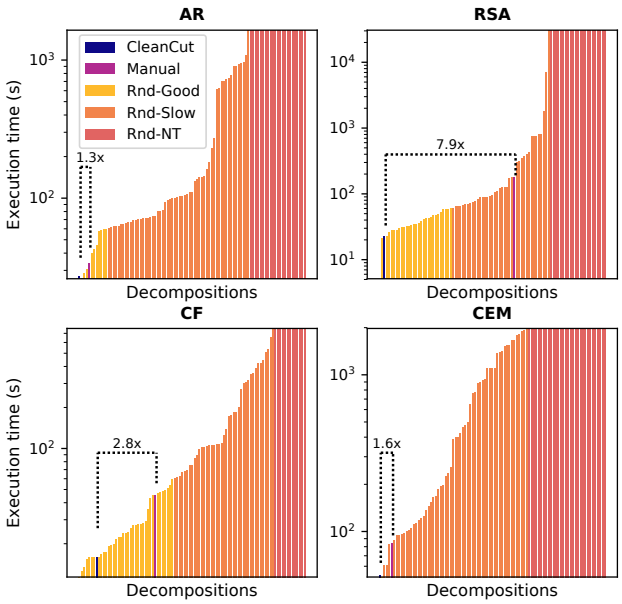

Figure 6. Application execution time when decomposed by CleanCut, manually, or randomly. Random decompositions are grouped into completing within one minute (**Rnd-Good**), completing after a long time (**Rnd-Slow**), and not completing (**Rnd-NT**). The speedup of CleanCut relative to the manual strategy is shown in the annotations.

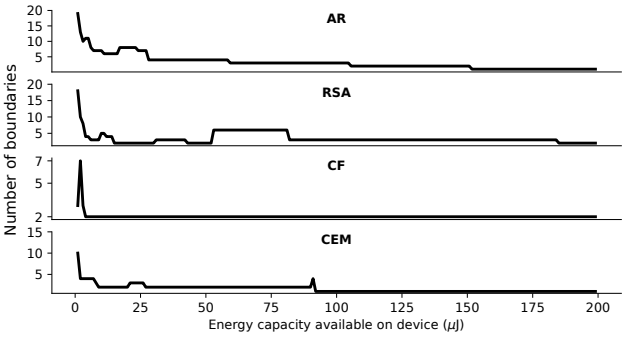

Figure 7. Number of task boundaries in CleanCut decompositions. As opposed to a manual decomposition, CleanCut adapts its decompositions to the energy capacity available on the device.

8.3 Checker Evaluation

We evaluated CleanCut's checker by using it to identify non-terminating path bugs in the same pool of random decompositions used in Section 8.2. The goal of this evaluation is to show that the checker reliably reports non-terminating paths and rarely reports that a path is non-terminating when it is terminating. After obtaining the checker 's predictions for each path in each decomposition, we executed the decomposition on the WISP energy-harvesting device on RF power. During execution some (unknown) subset of the program's paths executed, depending on the real experimental input from the sensors. The outcome of each execution is either that the decomposition terminated, implying that no path *that executed* had a non-termination bug, or that the decomposition did not terminate, implying that a non-terminating path executed. A non-termination prediction for a *program*

may not match the observed behavior because not all *paths* execute in all runs as a result of input variation.

Figure 8 plots the predicted energy for every path in any decomposition that terminated (left plots) and that did not terminate (right plots). Groups of paths from the same decomposition are adjacent, of the same color, and sorted by energy. The horizontal line indicates the measured energy capacity of the device. The plot does not show which paths did not terminate, because our measurement setup does not trace individual path executions while running on intermittent power (a task for which there exists no simple methodology today). For a terminating decomposition, we expect that *for all* paths that executed – and for most paths that did not execute – the checker predicts the energy to be below the capacity; i.e. the adjacent vertical bars of the same color should be below the red line if the paths that they represent executed during the trial run. Paths in a terminating decomposition that were predicted to be above capacity either did not execute, or their energy was overestimated by the model (i.e. a false-positive). For a non-terminating decomposition, we expect that *there exists* at least one path for which the checker predicts the energy to be in excess of capacity; i.e. from each group of bars of the same color, *some* vertical bars (corresponding to the paths that executed) are above the red line. Paths in a non-terminating decomposition that were predicted to be below capacity are expected, because it only takes a single non-terminating path to prevent an execution from terminating. However, if a non-terminating decomposition has no path whose energy was predicted to be above capacity, then CleanCut underestimated the energy cost of at least one non-terminating path (i.e. a false negative).

These expected trends are visible in Figure 8. Every non-terminating decomposition had at least one predicted non-terminating path in CF and CEM; and all but one non-terminating instance had such a path in AR. The results for these benchmarks show that CleanCut successfully identified non-terminating paths. In RSA, all but seven of the non-terminating decompositions had at least one path predicted not to terminate. For the remaining seven, we identified the source of the underestimate to be inaccurate loop bounds for some dynamically-bound loops (e.g. division, container search) that we obtained by profiling on fixed inputs. CleanCut is likely to perform better with more representative profiling and on applications with statically-bound loops.

For terminating decompositions, CleanCut correctly identified that a majority of paths do not exceed the energy capacity of the device. In CEM, CF, and RSA, only a few instances include paths that exceed the capacity. Such paths either did not execute during the trial run or were overestimated by the model. In AR, at least one terminating distribution has all paths predicted as non-terminating (middle of the X-axis). Since the energy of the paths in this group is similar and above capacity, they likely share a common prefix that was overestimated. This overestimate is likely due to overly

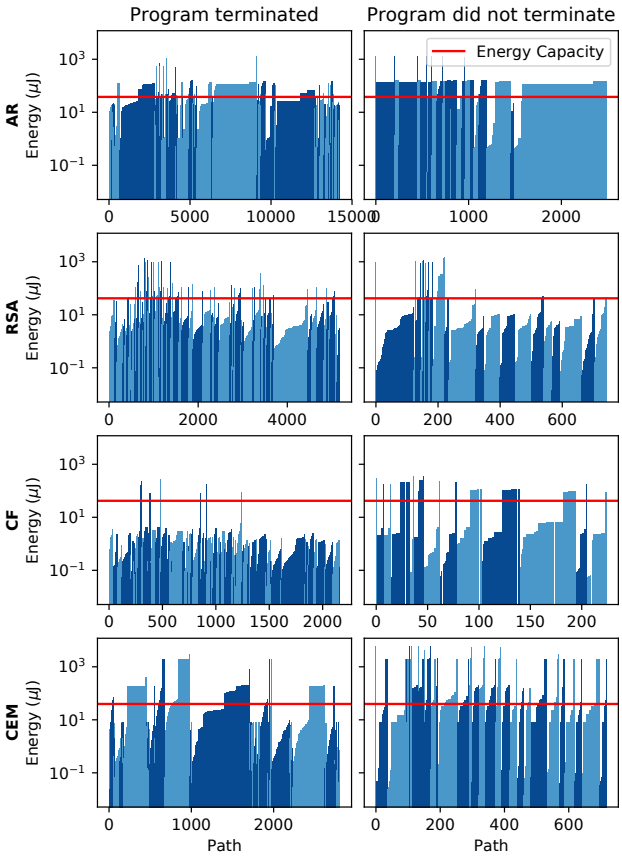

Figure 8. Predicted and observed non-termination for random boundary placements.

conservative loop bounds on the loops that implement arithmetic operations for the pattern matching. The presence of some paths predicted non-terminating in a terminating decomposition, is evidence of CleanCut's conservatism, because these paths are reported to the programmer.

8.4 Characterization

We evaluated the practicality of using CleanCut by measuring the time to validate a program decomposition using the checker and find a decomposition using the placer. Table 1 summarizes the complexity of each application, showing block count, call depth, and maximum path count across all evaluated decompositions. The table also reports the execution time of CleanCut's costliest phases. The block profiling cost varies with block count and averages 36 minutes. CleanCut incurs this cost only the first time it runs; after incremental changes, only changed blocks are re-profiled.

We measured the time CleanCut takes to check the manual placement and to place boundaries in the uninstrumented code. Evaluating energy expressions occupies the majority of the run time due to numerical operations on distributions. The time cost increases with the application size and number of paths. For example, CleanCut takes longest on RSA, which has 1.5-2.4x lines of code of the other applications. The

Table 1. Benchmark and analysis time characteristics. Total basic block counts, maximum path count across all decompositions studied (including random), and the maximum call depth are listed. Times are for one-time block profiling (**BB Prof**), checking a decomposition (**Chkr**), and finding a valid decomposition (**Plcr**).

	App. Characteristics			Analysis Time		
App.	BBs	Paths	Call Depth	BB Prof.(m)	Chkr. (s)	Plcr. (s)
AR	187	298	5	45	30	24
RSA	197	326	5	51	57	56
CF	91	217	2	23	36	8
CEM	70	80	2	24	54	11

checker takes longer than the placer, because it computes more detailed information for the bug report, such as the CDF of the energy distributions for each path. The running time of the placer increases with the number of path splits it has to perform, which decreases with capacitor size.

9 Related Work

We present prior work that relates to CleanCut in the context of energy-harvesting systems, intermittent execution models, and energy-aware program analysis.

9.1 Energy-harvesting and Intermittent Computing

Intermittent computing originates with energy-harvesting hardware. Energy-harvesters can extract energy from e.g., radio waves [15, 28, 36, 39, 42, 46], interaction [22, 35, 44], or light [24, 27]. Fully non-volatile processor architectures [29] support computing through power failures. Federated power system [18] improves flexibility by partitioning energy storage. Mayfly [19] improves programability by providing a notion of time across power failures.

Early energy-ware systems [27, 41] addressed computation under energy constraints but not intermittence. Emerging energy-harvesting platforms have lead to systems for *intermittent computing*. Dewdrop [7] scheduled short computations to maximize throughput of important tasks and avoid failures. Mementos [38] preserved progress in long-running programs on intermittent devices by dynamically checkpointing volatile state. Dynamic checkpointing was explored further in later work [3, 4, 20, 32]. Dynamic checkpoints that are inserted in advance but collected conditionally [3, 4, 20, 32, 38] may generate tasks that are too large (i.e., exceed device energy capacity) or too small (i.e., overprovision checkpoints) — problems that CleanCut addresses. Checkpoints conditioned on energy level [3, 4] require checking the voltage on the capacitor with an ADC or a comparator, which consumes energy, time, and board space.

Static task systems [10, 30, 37, 47], which CleanCut targets, keep memory consistent and allow a programmer control over where computation resumes after a reboot. DINO [30, 37] was the first static task system to observe that checkpointing volatile state alone is inadequate for correctness and versioned non-volatile state. In Chain [10] application is written as a graph of static tasks that communicate over statically multi-versioned channels. Alpaca [31] privatizes variables shared across tasks and atomically commits modified variables at task boundaries. Ratchet [47] combined checkpointing and idempotent processing [12, 13] under the assumption that *all* device memory is non-volatile. Similarly to CleanCut, Ratchet statically puts boundaries into an application to keep state in an intermittent execution consistent. However, a task in Ratchet can end up arbitrarily long and may exceed the energy capacity of the device. The lack of task-sizing in Ratchet makes the work complementary to CleanCut, which could guide Ratchet's boundary placement. Unlike CleanCut, Ratchet does not give the programmer control over boundary placement, which leaves the risk of a boundary splitting an atomic operation in the application. CleanCut is instrumental to adoption of static task systems, since they may encounter non-termination.

9.2 Energy-aware Compilation and Modeling

Prior work has examined the feasibility of estimating energy consumption statically [33] and proposed methods based on constraint satisfaction, e.g. Implicit Path Enumeration Technique, [45], symbolic representation of input-dependent code [26], instruction statistics with one-time profiling [23]. The energy model in CleanCut, most closely follows the approach in [26] in its choice of block granularity and profiling, but diverges in the choice to expose the full distribution instead of a worst-case scalar.

Architecture simulators [5, 6, 25, 40] model power dissipation in architectural structures. When hardware description source for the processor is available, core power (but not full system power) can be obtained from application-specific symbolic simulation at the RTL level [8]. Unlike CleanCut, simulation is a dynamic analysis, applying to an execution, not a program. Work on energy-aware compilation for energy-harvesting systems [34] scheduled short "one-shot" tasks to not exhaust buffered energy. CleanCut, by contrast, is a compiler analysis for computations that span power failures.

10 Conclusion

This work is the first to identify and address the problem of validating and generating task decompositions of programs written for an intermittent execution model. Our system, CleanCut, builds a statistical model of the energy cost of paths through a program. CleanCut's checker uses this energy model along with a model of the energy supply of the target device to report non-terminating paths in a program decomposed with task boundaries. CleanCut's placer iteratively generates a task decomposition for a program, inserting task boundaries to prevent non-termination. Having evaluated our CleanCut prototype on a real energy-harvesting device powered by radio waves, we showed that its checker is accurate and its placer quickly identifies high performance, valid task decompositions.

References

[1] Proteus Digital Health. http://www.proteus.com/, 2015.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *MICRO*, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[3] D. Balsamo, A. Weddell, A. Das, A. Arreola, D. Brunelli, B. Al-Hashimi, G. Merrett, and L. Benini. Hibernus++: A self-calibrating and adaptive system for transiently-powered embedded devices. *CADICS*, 2016.

[4] D. Balsamo, A. Weddell, G. Merrett, B. Al-Hashimi, D. Brunelli, and L. Benini. Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *Embed. Sys. Let.*, 2014.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[6] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *ISCA*, 2000.

[7] M. Buettner, B. Greenstein, and D. Wetherall. Dewdrop: An energy-aware task scheduler for computational RFID. In *NSDI*, Mar. 2011.

[8] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori. Determining application-specific peak power and energy requirements for ultra-low power processors. In *ASPLOS*, pages 3–16, New York, NY, USA, 2017.

[9] A. Colin, G. Harvey, B. Lucia, and A. P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *ASPLOS*, pages 577–589, New York, NY, USA, 2016. ACM.

[10] A. Colin and B. Lucia. Chain: Tasks and channels for reliable intermittent programs. In *OOPSLA*, 2016.

[11] G. Contreras and M. Martonosi. Power prediction for intel xscale®processors using performance monitoring unit events. In *ISLPED*, pages 221–226, New York, NY, USA, 2005. ACM.

[12] M. de Kruijf and K. Sankaralingam. Idempotent processor architecture. In *MICRO*, pages 140–151, New York, NY, USA, 2011. ACM.

[13] M. A. de Kruijf, K. Sankaralingam, and S. Jha. Static analysis and compiler design for idempotent processing. *ACM SIGPLAN Notices*, 47(6):475–486, 2012.

[14] D. C. D'Elia and C. Demetrescu. Ball-larus path profiling across multiple loop iterations. In *OOPSLA*, New York, NY, USA, 2013.

[15] A. Dementyev, J. Gummeson, D. Thrasher, A. Parks, D. Ganesan, J. R. Smith, and A. P. Sample. Wirelessly powered bistable display tags. In *ubiComp*, pages 383–386, New York, NY, USA, 2013. ACM.

[16] P. F. Dubois, K. Hinsen, and J. Hugunin. Numerical python. *Computers in Physics*, 10(3), May/June 1996.

[17] N. Grech, K. Georgiou, J. Pallister, S. Kerrison, J. Morse, and K. Eder. Static analysis of energy consumption for LLVM IR programs. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, pages 12–21. ACM, 2015.

[18] J. Hester, L. Sitanayah, and J. Sorber. Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, SenSys '15, pages 5–16, New York, NY, USA, 2015. ACM.

[19] J. Hester, K. Storer, and J. Sorber. Timely Execution on Intermittently Powered Batteryless Sensors. In *SenSys*, 2017.

[20] H. Jayakumar, A. Raha, and V. Raghunathan. QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers. In *Int'l Conf. on VLSI Design and Int'l Conf. on Embedded Systems*, Jan. 2014.

[21] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2016-08-14].

[22] M. E. Karagozler, I. Poupyrev, G. K. Fedder, and Y. Suzuki. Paper generators: Harvesting energy from touching, rubbing and sliding. In *UIST*, pages 23–30, New York, NY, USA, 2013. ACM.

[23] S. Kerrison and K. Eder. Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor. *ACM Transactions on Embedded Computing Systems*, 14(3):1–25, Apr. 2015.

[24] Y. Lee, G. Kim, S. Bang, Y. Kim, I. Lee, P. Dutta, D. Sylvester, and D. Blaauw. A modular 1mm3 die-stacked sensing platform with optical communication and multi-modal energy harvesting. In *ISSCC*, 2012.

[25] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. Gpuwattch: Enabling energy optimizations in gpgpus. In *ISCA*, pages 487–498, New York, NY, USA, 2013. ACM.

[26] U. Liqat, Z. Bankovic, P. López-García, and M. V. Hermenegildo. Inferring energy bounds via static program analysis and evolutionary modeling of basic blocks. In *Proceedings of the International Symposium on Logic-Based Program Synthesis and Transformation*, LOPSTR, 2017.

[27] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with impala and zebranet. In *MobiSys*, pages 256–269, New York, NY, USA, 2004.

[28] V. Liu, A. Parks, V. Talla, S. Gollakota, D. Wetherall, and J. R. Smith. Ambient backscatter: Wireless communication out of thin air. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 39–50, New York, NY, USA, 2013. ACM.

[29] Y. Liu, Z. Li, H. Li, Y. Wang, X. Li, K. Ma, S. Li, M.-F. Chang, S. John, Y. Xie, J. Shu, and H. Yang. Ambient energy harvesting nonvolatile processors: From circuit to system. In *DAC*, pages 150:1–150:6, 2015.

[30] B. Lucia and B. Ransford. A simpler, safer programming and execution model for intermittent systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 575–585, New York, NY, USA, 2015. ACM.

[31] K. Maeng, A. Colin, and B. Lucia. Alpaca: Intermittent execution without checkpoints. In *OOPSLA*. ACM, 2017.

[32] A. Mirhoseini, E. M. Songhori, and F. Koushanfar. Idetic: A high-level synthesis approach for enabling long computations on transiently-powered ASICs. In *PerCom*, Mar. 2013.

[33] J. Morse, S. Kerrison, and K. Eder. On the infeasibility of analysing worst-case dynamic energy. *CoRR*, abs/1603.02580, 2016.

[34] C. Moser, J.-J. Chen, and L. Thiele. Power management in energy harvesting embedded systems with discrete service levels. In *ISLPED*, pages 413–418, New York, NY, USA, 2009. ACM.

[35] J. A. Paradiso and M. Feldmeier. A compact, wireless, self-powered pushbutton controller. In *UbiComp*, pages 299–304, 2001.

[36] A. Parks, A. Sample, Y. Zhao, and J. R. Smith. A wireless sensing platform utilizing ambient rf energy. In *(WiSNET*. IEEE, 2013.

[37] B. Ransford and B. Lucia. Nonvolatile memory is a broken time machine. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 5:1–5:3, New York, NY, USA, 2014.

[38] B. Ransford, J. Sorber, and K. Fu. Mementos: System support for long-running computation on RFID-scale devices. In *ASPLOS*, 2011.

[39] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, Nov. 2008.

[40] A. Sinha and A. P. Chandrakasan. Jouletrack: A web based tool for software energy profiling. In *DAC*, pages 220–225, 2001.

[41] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: A language and runtime system for perpetual systems. In *SenSys*, pages 161–174, New York, NY, USA, 2007. ACM.

[42] V. Talla, B. Kellogg, B. Ransford, S. Naderiparizi, S. Gollakota, and J. R. Smith. Powering the Next Billion Devices with Wi-Fi. *ArXiv*, 2015.

[43] TI Inc. Overview for MSP430FRxx FRAM, 2014.

[44] N. Villar and S. Hodges. The Peppermill: A human-powered user interface device. In *Conference on Tangible, Embedded, and Embodied Interaction (TEI)*, Jan. 2010.

[45] P. Wagemann, T. Distler, T. Honig, H. Janker, R. Kapitza, and W. Schroder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *ECRTS*, page 105, 2015.

[46] A. Wickramasinghe, D. Ranasinghe, and A. Sample. Windware: Supporting ubiquitous computing with passive sensor enabled rfid. In *IEEE RFID*, pages 31–38, April 2014.

[47] J. V. D. Woude and M. Hicks. Intermittent computation without hardware support or programmer intervention. In *OSDI*, 2016.