Automatic Energy Analysis Using Types

Sai Divvela
sdivvela@terpmail.umd.edu
University of Maryland
College Park, Maryland, USA

1 Problem and Motivation

As IoT devices become increasingly prevalent, embedded systems operating in resource-constrained environments must carefully account for their energy requirements. These requirements are typically expressed as the *worst-case energy consumption* (WCEC), the maximum possible energy consumed across any possible execution, and are obtained either through measurement-based profiling or static path enumeration based analysis.

Both approaches face significant challenges when peripheral devices such as sensors, radios, and actuators are involved. Peripherals maintain internal state that evolves based on the operations they perform, making them difficult to reason about. Measurement-based approaches, while realistic, can be difficult to perform and might not always generalize [1, 4]. Static analyses offer stronger guarantees but demand accurate modeling of the device.

The stateful nature of peripherals is a particular obstacle for static analysis. Existing static analyses are insufficient in two key ways: (1) peripherals are either unmodeled or modeled under overly restrictive assumptions such as only turning on or off [6], limiting applicability to a narrow class of devices and programs; and (2) existing tools impose significant workflow burden, requiring external tool integration or custom peripheral implementations [7].

To address these limitations, we make use of *typestate* for peripherals by encoding a type-level representation of a peripheral's modes of operation. While typestate is traditionally used to encode valid operations for a stateful object, it is also a natural choice for energy analysis as the power draw for a given peripheral depends on it's internal state. We combine it with prior work on automatic resource analysis in type systems [2] resulting in, to our knowledge, the first type system to unify these two ideas, enabling symbolic energy bounds for embedded systems without invasive changes to programmer workflow. By combining both typestate and automatic cost analysis, we model a broad class of peripherals without restrictive assumptions. Since the analysis is part of the type system, programmers need not integrate external tools or provide custom peripheral implementations to obtain energy bounds.

2 Background and Related Work

We organize related work around three pillars: WCEC analysis, peripheral typestate, and resource-aware type systems, and an approach combining some of these ideas.

Our goal is to provide a tight, peripheral-aware bound on the WCEC of embedded systems. Recent prior work for this problem has typically followed the same high level structure: enumerate all possible runtime behaviors, and then translate to a formal constraint-based problem that can be solved with knowledge about the specific hardware that is being executed. [6, 9–11] Typically this enumeration is done via a static analysis. An approach that stands out is *WoCA* [7] which enhances this enumeration by integrating peripherals and their modes of operation. In order for the analysis to be sound and provide accurate bounds, code must be written in a way that only activates one peripheral device at a time. Additionally, *WoCA* requires custom device implementations for using their analysis, and does not consider processor energy costs. This means that programs that rely on the processor and do not use peripherals extensively will have incorrect bounds.

Another approach for analyzing resource consumption is through the usage of type systems. An established technique used is known as Automatic Amortized Resource Analysis (AARA) [2].The analysis associates costs to types that are found in the program, and derives resource bounds from type inference by solving a linear program based on the potential method of amortized analysis. The soundness of these bounds is proven via a cost semantics that defines program resource consumption. This has been used to obtain bounds on the consumption of a variety of resources such as time and memory usage for free, motivating our idea of building a type system for energy analysis. Such a type system can overcome the challenge of integrating a static analysis, while also providing robust WCEC bounds, which is why we use it as a foundation for cost analysis. However, AARA alone is insufficient. The authors of [2] note that a major weakness of using this approach is that it lacks knowledge of peripheral behavior, which leaves it unable to address the first challenge of many prior WCEC works.

This motivates our use of typestate. AARA and similar approaches lack knowledge of the internal state of peripherals, which is highly complex and directly affects their behavior and energy consumption. An existing notion for capturing exactly this kind of behavior is *typestate*: a type-level representation of the internal state of a stateful object. The Abacus framework [5] applies this notion directly to peripherals, providing a Rust-embedded DSL that lets programmers specify peripheral states and valid operations, then automatically generates peripheral typestates enforced by the compiler. By incorporating the idea of typestate into our type system, we can address the modeling gap that AARA alone leaves open. Rather than just having costs associated to types, we instead have costs associated to peripheral states, which we can represent in the type system by making use of typestate.

Our goal is to derive tight symbolic energy bounds for embedded systems by tracking peripheral state directly within a type system in the style of AARA, obtaining these bounds for free. Since typestate embeds state in the type level, and AARA is able to derive bounds through type inference, we are able to combine the two ideas effectively together. To our knowledge, the closest prior work to this idea is Performal [12], which analyzes latency in distributed systems by modeling nodes as state machines. Like peripherals, distributed nodes have complex internal state whose behavior and transition costs depend heavily on current state, and Performal

derives latency bounds by tracking both. However, rather than adopting an external proof framework like what Performal does, we integrate this analysis directly into a type system. The key insight is that we can use typestate to provide precise peripheral state information, and AARA to provide precise cost information. By combining the two, we can then accurately report cost bounds.

3 Approach and Uniqueness

Our type system has two main goals: keeping track of peripheral state as accurately as possible, and using this information to provide symbolic energy cost bounds. Our type system achieves these goals through a novel approach that combines typestate with cost analysis to provide precise bounds.

3.1 Approach Overview

We have 3 main costs to consider: costs from executing statements of code on the processor, costs from using peripherals, and costs from active peripherals in the background. The usage of typestate helps us to reason about the second and third cost categories in particular, as peripherals use a different amount of power based on the current mode of operation. A naive approach that assumes worst-case power consumption throughout would produce bounds that are sound but far too conservative to be useful. Precise symbolic bounds therefore require knowing exactly which state each peripheral occupies at each program point. We model peripheral state as a lattice with downgrading policies. In our lattice structure, the partial order $s_1 \leq s_2$ encodes that a peripheral in state s_1 can transition to the state s_2. However peripherals do not solely move up the lattice. Commonly, peripherals return to a prior state upon completing an operation. We model these "backwards transitions" via downgrade policies, $s_2 \hookrightarrow s_1$. This is essentially like a state machine, but we use the lattice structure for ease of modeling.

To accurately report costs, we need to determine what cost is associated to the peripheral and what is associated to the main board. To do this, we broadly have 3 types of cost expressions. $C(p : st)$, represents the cost that depends on a peripheral p existing in a concrete state. $C(p : (st, st'))$ represents a cost that depends on a peripheral p transitioning from one state to another. Additionally, we represent fixed costs are by C in our cost syntax, adding a subscript if necessary. We use the max operation to represent the cost over branches where we cannot determine statically what will be taken. Our language always requires an explicit state for the target of a transition, so we always know at least the intended destination state. We will illustrate this with a running example.

Running Example. The program in the left column of Figure 1 models a simple task using a temperature sensor as a peripheral. The function `log_data` takes a temperature sensor, and then transitions to the `Measure` state. Afterwards, it performs a measurement and conditionally writes to a variable. Finally, it powers the sensor down before returning. The lattice for the temperature sensor is $\langle$`Off`, `Idle`, `Measure`$\rangle$ where `Off` $\leq$ `Idle` $\leq$ `Measure`. It also has downgrade policies `Measure` $\hookrightarrow$ `Off` and `Idle` $\hookrightarrow$ `Off`. In the example, we represent transitions by `transition`(temp, `Measure`). Downgrading is represented by `downgrade`(temp, `Off`). We analyze each statement individually, and add background peripheral costs. Then, we combine all of the constraints for a total function cost.

	Code	Cost Expression
1	`fn log_data(temp) -> u64 {`	
2	`transition(temp, Measure)`	$C(\text{temp} : (\alpha_0, \text{Measure}))$
3	`let res = temp.measure()`	$C_m + C(\text{temp} : \text{Measure})$
4	`let t = 0`	$C_v + C(\text{temp} : \text{Measure})$
5	`if(res == 65) {`	
6	`let t = 1`	$\max(C_v, 0) + C(\text{temp} : \text{Measure})$
7	`}`	
8	`downgrade(temp, Off)`	$C(\text{temp} : (\text{Measure}, \text{Off}))$
9	`return t`	
10	`}`	
`log_data` cost = $C_I = C(\text{temp} : (\alpha_0, \text{Measure})) + 3C(\text{temp} : \text{Measure}) + C_v + \max(C_v, 0) + C(\text{temp} : (\text{Measure}, \text{Off}))$		

Figure 1: Running Example with Cost Analysis

State Inference. In the example, we use $C(\text{temp} : (\alpha_0, \text{Measure}))$ as the first cost expression, which is not something that was mentioned previously. The issue is that when we analyze `log_data()` on it's own, we have no knowledge about the state of the temperature sensor. The cost of this function therefore depends on the entry state of the temperature sensor, which we only know when `log_data()` is actually used in the program. One way to provide this insight is to provide annotations for peripherals. While the lattice gives us the structure to reason about peripheral states, requiring explicit annotations at every function would be burdensome and error-prone. Instead, we use a variation of type inference that we call *state inference*. To motivate our usage of state inference, we will consider the examples in Figure 2.

```
fn main() {
  transition(temp, Idle);
  let t = log_data(temp);
}
```

```
fn main() {
  transition(temp, Idle);
  transition(temp, Measure);
  let m = temp.measure();
  let t = log_data(temp);
}
```

Figure 2: Usage example

In this example, we use `log_data()` in two different ways. On the left hand side, we first transition the temperature sensor from `Off` to `Idle` and on the right, we have an additional transition from `Idle` to `Measure`. Both possible use cases of `log_data()` are valid, since a programmer may want to take additional measurements before calling `log_data()`. Requiring programmers to annotate the entry state of peripherals on functions would force a programmer to choose between whether a peripheral should be in either the `Idle` or the `Measure` state at the start of `log_data()`. While having some sort of state polymorphism can help with this, it can lead to programmers being overly conservative causing imprecise bounds.

We use Figure 3 as our running example for state inference.

`fn main() {`	temp = $[\alpha'_0]$
`transition(temp, Idle);`	temp = $[\alpha'_1 :: \alpha'_0]$, $\Phi = \alpha'_1 = \texttt{Idle} \wedge \alpha'_0 \leq \alpha'_1$
`let t = log_data(temp);`	solve$(\alpha'_1, \Phi \wedge \alpha'_1 \leq \texttt{Measure}) = \texttt{Idle}$
`}`	temp = [`Idle` :: `Measure` :: `Off`]

Figure 3: Concretization and Inference Example.

$$
\begin{array}{rcl}
\text{Cost expression } \varepsilon & ::= & C \\
& | & C(p : st) \\
& | & C(p : (st, st')) \\
& | & C(p : (\alpha, st)) \\
& | & C(p : \alpha) \\
& | & C[\alpha \mapsto st] \quad | \quad \max(\varepsilon_1, \varepsilon_2) \quad | \quad \varepsilon_1 + \varepsilon_2
\end{array}
$$

Figure 4: Symbolic Energy Cost Syntax

Because we do not know the state of a given peripheral on function entry, we start by assigning it a state variable α—analogous to type variables. In Figure 3, the initial state variable for `temp` is α'_0. Every time we encounter a `transition` statement, we create a new state variable α and push it onto a stack associated with that peripheral. We represent this stack with $[\alpha]$. The creation of new state variables maintains an SSA-like property and the stack allows us to understand how the peripheral state evolves through time.

Because we transition to the `Idle` state, we add the constraint that $\alpha'_0 \leq \alpha'_1$ since that is required for the transition to be valid. When we encounter the `log_data` function call, we combine the constraints we have collected thus far with any constraints known within `log_data`. In this case, we know that `temp` will transition into the `Measure` state, so we have the constraint that $\alpha_0 \leq$ `Measure`, where α_0 is the state of `temp` on entry to `log_data`. We rename α_0 to be α'_1 since α'_1 refers to the current state of the temperature sensor. Then, we take all of those constraints and solve for them giving the solution $\alpha'_1 =$ `Idle`. We refer to this process as *lazy* concretization of α_0 since we chose not to solve for it until `log_data` was called and we had additional information about the temperature sensor's state. On the flip side, we were immediately able to solve for the state of the peripheral after line 8 in `log_data` since the peripheral's state only depends on the state of the peripheral after line 2 which is fully known. This is what we refer to as *eager* concretization.

To represent the cost of peripherals in an unknown state prior to concretization, we have two additional cost expressions. We use $C(p : \alpha)$ represents the cost of a peripheral being in an unknown state α and $C(p : (\alpha, st))$ to represent the cost of transitioning from an unknown state to a concrete one. Using this information, it allows us to derive the following bound on the energy consumption of `main`: $C(\text{temp} : (\texttt{Off}, \texttt{Idle})) + C_l[\alpha_0 \mapsto \texttt{Idle}]$ where C_l refers to the cost bound for `log_data` derived earlier. At the very end of `main`, the temperature sensor stack is given by [`Idle` :: `Measure` :: `Off`], where the middle state comes from analyzing `log_data()`.

3.2 Type System.

Guided by our examples in Figures 2, 3, we present a core calculus to formalize this reasoning. Our syntax is shown in Figure 5. We include explicit `transition` and `downgrade` statements so that programmers can control when peripherals perform a state transition. To support dynamically inspecting peripheral state and performing actions based on it, we include a form of state "pattern matching" via `match` p `with` $st_1 \mid \cdots \mid st_n$.

Our type system uses additional constructs detailed in Figure 6. We include the type ρ to refer to peripheral devices, so that we are able to determine when we are performing operations that involve them. We assume a rust-like ownership model, so aliasing is not a

$$
\begin{array}{rcl}
\text{Peripheral Declarations: } p & ::= & \text{id}\{\langle s_0, s_1, \ldots, s_n\rangle, <, st, D\}, p \mid \cdot \\
\text{Declarations: } d & ::= & \text{fn } f(e, p) : \tau_1 \to \tau_2 \ \{c; ret := e\}, d \mid \cdot \\
\text{Commands: } c & ::= & \text{let } x = e \\
& | & \text{let } x = f(e, p) \\
& | & \text{if } e \text{ then } c_1 \text{ else } c_2 \\
& | & \text{transition}(x, \text{st}) \\
& | & \text{downgrade}(x, \text{st}) \\
& | & \text{clock}() \\
& | & \text{match } p \text{ with } \text{st}_1 \to c_1 \mid \ldots \mid \text{st}_n \to c_n \\
& | & c_1; c_2 \\
\text{Expressions: } e & ::= & x \mid v \mid e_1 \text{ binop } e_2
\end{array}
$$

Figure 5: Syntax

$$
\begin{array}{ll}
\textit{Typing Context} & \Gamma ::= x \to \tau, \Gamma \mid \cdot \\
\textit{Types} & \tau ::= \text{int} \mid \text{bool} \mid \text{unit} \mid \rho \mid \tau_1 \to \tau_2 \\
\textit{State Constraints} & \varphi ::= \alpha = \vec{st} \mid \alpha \leq st \mid \alpha \subseteq \vec{st} \mid \\
& \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \\
\textit{Constraints} & \Phi ::= \varphi, \Phi \mid \cdot \\
\textit{Peripheral Context} & \Delta ::= p \to [\alpha], \Delta \mid \cdot
\end{array}
$$

Figure 6: Constructs

concern. Our typing judgment for commands takes the following form: $\Gamma, \Delta, \Phi, C \vdash c \to \Gamma', \Delta', \Phi', C'$. The judgment means that given a typing context, peripheral context, a set of current constraints and a current cost, evaluating c can update all four of these constructs. The typing rules for `transition` and `downgrade` maintain the SSA-like property and add fresh state variables to the peripheral's variable stack. Additionally, they add relevant constraints to Φ so that inference can be performed. At function declarations, the type system checks the body, performs eager concretization where possible, and collects necessary constraints to be used at function calls. At the function call site, constraints from the declaration are combined with current peripheral constraints to perform lazy concretization. Finally, the sequencing rule combines costs together, as well as adds the result of the background cost of any active peripherals that may exist.

4 Results and Contributions

Combining everything together gives a type system that automatically derives symbolic cost expressions for embedded systems with stateful peripherals, requiring no programmer annotations beyond peripheral lattice definitions. We are currently implementing this type system within the Abacus framework using Rust's proc-macros system [8]. We aim to integrate with standard peripherals and the Tock Operating System [3] to test our approach.

Our next steps also include proving a soundness theorem. We are inspired by soundness theorems from previous AARA works. The goal is to show that any cost bounds our system provides is a true upper bound to the cost of executing the program. We report all of our costs symbolically to programmers reason about the energy behavior of their code in a hardware-agnostic manner. Given a pre-existing energy model for an embedded device, programmers may instantiate these cost expressions for concrete bounds.

Acknowledgments

The author thanks his research advisor, Milijana Surbatovich, for all of the guidance, feedback and support throughout this project.

References

[1] Carlo Brandolese, Simone Corbetta, and William Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 333–338, 2011.

[2] Jan Hoffmann and Steffen Jost. Two decades of automatic amortized resource analysis. *Mathematical Structures in Computer Science*, 32(6):729–759, 2022.

[3] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, New York, NY, USA, October 2017. ACM.

[4] Jeremy Morse, Steve Kerrison, and Kerstin Eder. On the limitations of analyzing worst-case dynamic energy of processing. *ACM Trans. Embed. Comput. Syst.*, 17(3), February 2018.

[5] Tyler Potyondy, Anthony Tarbinian, Leon Schuermann, Eric Mugnier, Adin Ackerman, Amit Levy, and Pat Pannuto. Rage against the state machine: Type-stated hardware peripherals for increased driver correctness. In *Proceedings of ASPLOS '26, March 21-26*, 2026.

[6] Phillip Raffeck, Christian Eichler, Peter Wägemann, and Wolfgang Schröder-Preikschat. Worst-Case Energy-Consumption Analysis by Microarchitecture-Aware Timing Analysis for Device-Driven Cyber-Physical Systems. In Sebastian Altmeyer, editor, *19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, volume 72 of *Open Access Series in Informatics (OASIcs)*, pages 4:1–4:12, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[7] Phillip Raffeck, Johannes Maier, and Peter Wägemann. Woca: Avoiding intermittent execution in embedded systems by worst-case analyses with device states. In *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES 2024, page 83–94, New York, NY, USA, 2024. Association for Computing Machinery.

[8] The Rust Project Developers. Procedural macros, n.d. The Rust Reference.

[9] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. Worst-case energy consumption: A new challenge for battery-powered critical devices. *IEEE Transactions on Sustainable Computing*, 6(3):522–530, 2021.

[10] Peter W"agemann. *Static Worst-Case Analyses and Their Validation Techniques for Safety-Critical Systems*, pages 227–247. Springer International Publishing, Cham, 2022.

[11] Simon Wegener, Kris K. Nikov, Jose Nunez-Yanez, and Kerstin Eder. EnergyAnalyzer: Using Static WCET Analysis Techniques to Estimate the Energy Consumption of Embedded Applications. In Peter Wägemann, editor, *21th International Workshop on Worst-Case Execution Time Analysis (WCET 2023)*, volume 114 of *Open Access Series in Informatics (OASIcs)*, pages 9:1–9:14, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[12] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. Performal: Formal verification of latency properties for distributed systems. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.