A Domain-Specific Language for Energy Management in Edge Computing Systems

Victor de Oliveira Vidal
Universidade Federal do Amazonas
Manaus, Brazil
victor.vidal@icomp.ufam.edu.br

Horácio Oliveira
Universidade Federal do Amazonas
Manaus, Brazil
horacio@icomp.ufam.edu.br

Raimundo da Silva Barreto
Universidade Federal do Amazonas
Manaus, Brazil
rbarreto@icomp.ufam.edu.br

Abstract

Energy consumption remains a major challenge in edge computing and IoT systems, especially in resource-constrained embedded platforms such as ESP32-based devices. Although Dynamic Frequency Scaling (DFS) and power-saving mechanisms can reduce energy consumption, integrating these mechanisms directly into embedded applications is complex and error-prone. This paper presents a declarative approach for energy-aware management in ESP32 edge applications using a Domain-Specific Language (DSL), automatic code instrumentation, and a lightweight runtime. The proposed approach allows developers to describe energy policies at a high level while the infrastructure automatically inserts runtime hooks and applies DFS-related decisions during execution. Preliminary results using ten FreeRTOS workloads indicate an average energy reduction of 22.40% while preserving application responsiveness.

Keywords

Energy Management, Edge Computing, ESP32, FreeRTOS, Domain-Specific Language, Dynamic Frequency Scaling

1 Introduction

Edge computing has become increasingly important in Internet of Things (IoT) applications due to the growing demand for low latency, local processing, and reduced dependency on cloud infrastructures [1, 3]. Platforms such as ESP32 are widely adopted in edge applications because they combine wireless communication, low cost, and embedded processing capabilities. However, these platforms are highly constrained in terms of energy consumption, processing power, and memory.

Energy management methods such as Dynamic Voltage and Frequency Scaling (DVFS) have been extensively explored to reduce energy consumption in embedded and edge systems [8, 9]. Nevertheless, integrating such mechanisms into embedded software remains difficult. Developers usually need to manually insert energy-related code, monitor runtime metrics, and coordinate power-saving states with application logic.

Recent works in fog and edge computing have investigated resource allocation, offloading, and energy-aware scheduling [4, 5]. However, most approaches focus on network-level optimization or distributed infrastructures, while fewer studies explore declarative mechanisms for local energy management directly inside resource-constrained embedded devices.

This paper presents a declarative infrastructure for energy-aware management in ESP32 edge applications. The proposal combines a Domain-Specific Language (DSL), automatic code instrumentation, and a lightweight runtime capable of applying DFS-related policies during execution. The main goal is to reduce the complexity associated with energy management while preserving application performance and responsiveness.

The contributions of this paper are a declarative DSL for energy policy specification, a policy model with semantic validation rules, an automatic code instrumentation mechanism for FreeRTOS-based applications, a lightweight runtime for DFS-oriented energy management on ESP32, and preliminary experimental results using representative embedded workloads.

2 Related Work

Shi et al. [1] and Satyanarayanan [2] discuss edge computing as a paradigm that brings computation closer to data sources in order to reduce latency and improve responsiveness. Mao et al. [3] further discuss resource allocation and energy management challenges in Mobile Edge Computing (MEC), highlighting the importance of adaptive mechanisms capable of balancing performance and energy consumption. Kopras et al. [4] present a survey on energy reduction techniques in fog networks, discussing communication and computation costs in edge infrastructures. The authors emphasize that energy management depends not only on local computation but also on communication overhead and workload distribution. Panda et al. [5] propose an energy-efficient computation offloading strategy combining DVFS and deep reinforcement learning for latency-sensitive IoT applications. Although the approach achieves promising results, it relies on computationally expensive learning techniques that may be unsuitable for low-cost microcontrollers.

Regarding declarative abstractions, Mernik et al. [6] discuss how Domain-Specific Languages can improve expressiveness and reduce development complexity. Van Deursen et al. [7] reinforce that DSLs can facilitate code generation and separation between domain logic and implementation details. Lu et al. [8] propose a compiler-guided DVFS mechanism in which program structure is analyzed to identify regions where frequency adjustments may reduce energy consumption. This work inspires the automatic instrumentation strategy adopted in the present proposal.

Unlike previous approaches, this work focuses on lightweight DFS-oriented energy management directly inside ESP32 embedded applications, combining DSLs, automatic instrumentation, and runtime adaptation.

3 Proposed Method

3.1 System Architecture

The proposed architecture is divided into two main stages: design-time and runtime execution. During design-time, developers describe energy policies using a declarative DSL. The DSL specification is processed by a parser and semantic analyzer, which validate

the policies and generate an Abstract Syntax Tree (AST). A code generator then analyzes the application structure and automatically inserts runtime hooks into the original C/FreeRTOS application. At runtime, the generated application interacts with a lightweight energy management runtime responsible for monitoring execution metrics, evaluating policies, and applying DFS-related decisions.

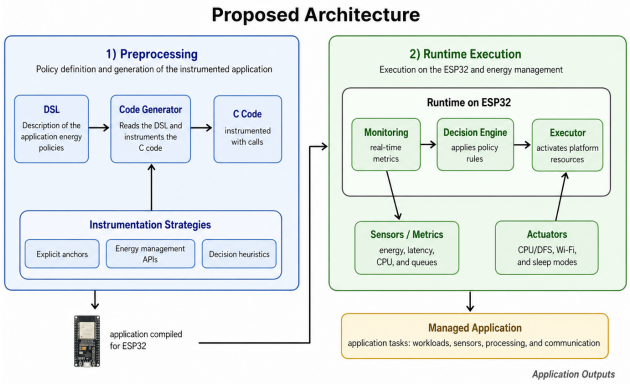

Figure 1: Overview of the proposed architecture.

3.2 DSL Design

The proposed DSL allows developers to describe energy-related policies using a high-level declarative syntax. Policies are composed of conditions, thresholds, events, and actions. Conditions can be associated with metrics such as CPU utilization, idle time, queue size, latency, or communication events.

```
policy energy_policy {
  when cpu_usage > 80 then set_state(performance);
  when idle_time > 5000 then set_state(eco);
}
```

The DSL grammar supports conditional structures, loops, and event-driven rules, allowing developers to express energy management logic without directly modifying runtime internals. The goal is not to replace C or FreeRTOS APIs, but to isolate energy policy definitions from functional application behavior.

3.3 Policy Model and Semantic Validation

The DSL represents objectives, states, events, metrics, and transition rules. A policy can be seen as a tuple $P = (S, E, M, R, A)$, where S is the set of energy states, E is the set of application or platform events, M is the set of monitored metrics, R is the set of guarded rules, and A is the set of actions available to the runtime. This representation is intentionally simple, but it is expressive enough to separate energy decisions from application-specific C code.

Figure 2 shows an input-to-output example. The upper side part is a DSL policy for a periodic sensor that processes data and then publishes it through MQTT. The lower side part shows the code generated by the toolchain: policy constants, state identifiers, and runtime hooks inserted into the original FreeRTOS task. In this example, the rule associated with `tx_burst` is translated into calls that signal an upcoming radio transmission and acquire a performance lock. The idle-time rules become guarded checks that may request the economic state or allow light sleep only when the observed metrics satisfy the policy thresholds. The critical-region restriction is translated into a safety condition that prevents sleep-oriented transitions while the corresponding runtime flag is active.

DSL input

```
policy edge_sensor {
  constraint latency <= 120ms;
  state perf { freq = 240MHz; }
  state eco  { freq = 80MHz; wifi = ps; }
  event tx_burst;

  when tx_burst then lock(perf, 1500ms);
  when idle_time > 800ms and cpu_usage < 25
       then set_state(eco);
  when idle_time > 5000ms and queue_size == 0
       then allow_sleep(light);
  forbid sleep during critical_region;
}
```

Generated C/FreeRTOS output excerpt

```
#define POLICY_TX_LOCK_MS    1500
#define POLICY_IDLE_ECO_MS   800
#define POLICY_IDLE_SLEEP_MS 5000
#define POLICY_CPU_LOW       25

typedef enum { ST_PERF, ST_ECO } policy_state_t;

for (;;) {
  policy_tick();
  read_sensor();
  process_sample();

  policy_signal_radio_burst();
  policy_lock_perf_for(POLICY_TX_LOCK_MS);
  mqtt_publish(topic, payload);

  if (metrics_idle_ms() > POLICY_IDLE_ECO_MS &&
      metrics_cpu_usage() < POLICY_CPU_LOW)
    policy_request_state(ST_ECO);

  if (metrics_idle_ms() > POLICY_IDLE_SLEEP_MS &&
      metrics_queue_size() == 0)
    policy_allow_light_sleep();
}
```

Figure 2: Example of a DSL input and a representative generated output.

The semantic analyzer checks whether all referenced metrics, states, and events are declared or supported by the runtime, whether units are compatible in comparisons, and whether timing parameters such as dwell time, cooldown, and performance-lock duration are positive. It also rejects rules that may lead to unsafe behavior, such as enabling sleep inside critical regions or requesting a low-frequency state while a non-interruptible communication burst is active. After validation, the compiler does not generate a full replacement for the application. Instead, it produces a compact policy descriptor and a set of hook bindings. The descriptor is loaded by the runtime, while the bindings guide the instrumentation phase.

3.4 Runtime and DFS Heuristics

The runtime continuously monitors application metrics and applies energy management decisions. Its implementation is lightweight so it can operate inside resource-constrained ESP32 applications without introducing significant execution overhead. The runtime periodically evaluates application behavior through the `policy_tick()` function, which acts as the central policy-evaluation mechanism. During each evaluation cycle, the runtime collects metrics associated with CPU utilization, idle time, queue growth, communication activity, and execution latency.

The decision process combines feedforward and feedback strategies. Feedforward decisions are used when the application signals predictable bursts of computation or communication. In such situations, the runtime temporarily enables higher-performance states using performance locks. Feedback decisions are based on metrics such as CPU usage, idle time, queue growth, and latency. The runtime periodically evaluates these metrics and decides whether to increase or reduce execution frequency.

DFS decisions are mapped to energy states defined by the DSL policies. A performance state corresponds to higher CPU frequencies, while balanced and economic states correspond to progressively lower energy consumption profiles. To avoid excessive oscillation, the runtime uses hysteresis, dwell-time intervals, and cooldown windows. Hysteresis avoids immediate transitions caused by small metric fluctuations, dwell time ensures that a state remains active for a minimum duration, and cooldown prevents repeated performance-lock activations in short intervals.

The runtime also manages synchronization between DFS transitions and communication subsystems. This aspect is important because Wi-Fi activity and frequency transitions may interfere with timing-sensitive communication operations. As a result, temporary performance locks may remain active during communication bursts to preserve responsiveness.

```
void policy_pm_init(void);
void policy_tick(void);
void policy_lock_perf_for(uint16_t ms);
void policy_signal_radio_burst(void);
void policy_critical_begin(void);
void policy_critical_end(void);
```

3.5 Metrics and Policy Parameterization

The metric layer provides the values used by the runtime to evaluate policy rules. In the current prototype, the main metrics include CPU utilization, idle time, queue length, radio activity, execution latency, and elapsed time since the last state transition. These metrics are exposed as DSL constructs, allowing developers to specify energy policies without directly accessing FreeRTOS counters or ESP-IDF power-management APIs.

Policies can also be parameterized according to application QoS requirements, including CPU thresholds, latency targets, idle intervals, queue limits, performance-lock duration, hysteresis, and cooldown windows. For example:

```
parameters {
  latency_target = 120ms;
  cpu_high = 75; cpu_low = 25;
  idle_sleep = 800ms;
  deep_sleep_idle = 5000ms;
  queue_high = 10;
  tx_lock = 1500ms;
  cooldown = 2000ms;
}
```

These parameters explicitly define the energy–performance tradeoff. At runtime, the policy prioritizes performance constraints and critical regions before transitioning to lower-power states.

3.6 Code Generation

The code generator operates over the AST produced during syntactic and semantic analysis. It transforms validated DSL policies into integration points inside the application source code, connecting declarative policy descriptions to the execution behavior of the embedded application.

The generator supports two complementary instrumentation strategies. The first uses explicit annotations inserted by the developer. These annotations act as anchors that indicate where the runtime should evaluate policies or signal events. This approach gives developers fine-grained control over critical application regions.

```
// @dfs:tick
// @dfs:tx_burst
// @dfs:critical_begin
// @dfs:critical_end
```

The second strategy uses heuristic-based automatic instrumentation. In this case, the generator analyzes structural patterns commonly found in FreeRTOS applications, such as task loops, communication routines, periodic execution regions, blocking operations, and computation-intensive sections.

For example, loops containing periodic sensor acquisition and communication calls may automatically receive runtime evaluation hooks. Communication APIs associated with MQTT publishing or Wi-Fi transmission may trigger automatic insertion of event signaling calls.

```
for (;;) {
  policy_tick();
  read_sensor();
  process_data();
  publish_data();
}
```

The instrumentation stage also inserts synchronization hooks associated with critical regions. During these regions, DFS transitions may be temporarily restricted to avoid instability caused by frequency transitions during time-sensitive execution.

The generated code preserves compatibility with existing C and FreeRTOS applications while minimizing developer effort. This allows the proposal to be integrated incrementally into existing embedded systems without requiring major modifications to the original application structure.

Table 1: Preliminary Results for Representative Workloads

Workload	Baseline (J)	Proposal (J)	Reduction
CPU Loop	12.8	10.9	14.84%
FFT 1024 pts	18.4	13.8	25.00%
Sensor I2C	8.9	6.9	22.47%
MQTT Send	16.2	12.9	20.37%
Wi-Fi Scan	19.0	16.8	11.58%
Edge ML	21.5	15.9	26.05%
AES Encryption	17.6	13.1	25.57%
LZ4 Compression	18.0	13.6	24.44%
Deep Sleep 10s	6.4	4.6	28.13%
Mixed Workload	24.7	18.4	25.51%
Average	–	–	22.40%

3.7 Generated Artifacts and Integration Workflow

The toolchain follows a source-to-source strategy. Instead of rewriting the complete application, it produces an instrumented version of the source code, a policy descriptor used by the runtime, a header file with generated constants and state identifiers, and an optional report listing the inserted hooks. The workflow has five steps: parsing the DSL file, validating the policy model, scanning the C/FreeRTOS code for anchors and structural patterns, binding DSL events to runtime calls, and compiling the resulting application with the runtime library. This design supports partial adoption. Developers can start with explicit anchors, which are safer for critical code, and later enable heuristic instrumentation for less sensitive parts of the application. The generated report improves auditability by showing where energy-related behavior was inserted and which DSL rule motivated each hook.

4 Preliminary Experimental Evaluation

4.1 Experimental Setup

The experimental evaluation was performed using an ESP32 development board running FreeRTOS and ESP-IDF. The experiments compared a baseline implementation against the proposed runtime-enabled approach. The preliminary evaluation considered ten embedded workloads: a lightweight CPU loop, FFT computation, periodic I2C sensor acquisition, MQTT-based communication, Wi-Fi scanning, small edge-ML inference, AES cryptography, LZ4 compression, periodic deep sleep cycles, and a mixed workload combining sensing, computation, and communication. Current measurements were obtained during execution using external monitoring infrastructure. The evaluation considered energy consumption, execution time, and average power. Average power was computed from the relation between energy and execution time.

4.2 Preliminary Results

Table 1 presents preliminary results for representative workloads. The CPU Loop and Sensor I2C workloads benefited primarily from idle detection and reduced-frequency execution between stable or periodic execution intervals. Since sensor readings were separated by inactive periods, the runtime maintained the processor in lower-power states for longer durations.

The MQTT workload benefited from event-driven signaling before radio transmission. In this scenario, the application signaled communication bursts in advance, allowing the runtime to temporarily activate higher-performance states before packet transmission and then return to more economical states after communication completion. The Wi-Fi scan workload obtained smaller gains because the radio remained active for a larger fraction of the experiment, limiting the available opportunities for DFS and sleep-oriented actions.

The FFT, Edge ML, AES, and LZ4 workloads benefited from performance-lock mechanisms during short computation-intensive execution windows. These workloads illustrate the advantage of temporarily increasing performance only during bursty execution phases, followed by a quick return to economic states. The deep sleep workload obtained the largest reduction because the policy identified long idle intervals and allowed sleep only when the expected idle duration compensated for the wake-up cost. The mixed workload combined sensor acquisition, CPU processing, and Wi-Fi communication, representing the most complete use case for the proposed combination of semantic events, feedback metrics, and runtime DFS control.

Preliminary measurements were also conducted to assess wake-up behavior from deep sleep. During transitions, the ESP32 showed current peaks of approximately 40–100 mA, occasionally reaching 200 mA. The average wake-up duration was about 4 ms.

Although the experiments remain preliminary, the results suggest that declarative energy management combined with lightweight runtime adaptation can reduce energy consumption while preserving responsiveness in representative ESP32 edge workloads.

5 Conclusion and Future Work

This paper presents an energy-management approach for ESP32 edge applications based on DFS. The proposal uses a DSL, automatic code instrumentation, and a lightweight runtime to adjust execution frequency and platform resources according to application behavior. As a result, energy policies are separated from functional logic, making decisions clearer, reusable, and easier to modify. Preliminary results indicate that the approach can reduce energy consumption while preserving responsiveness. The savings come from coordinating CPU frequency changes with application events, communication bursts, idle periods, and sleep opportunities, highlighting the value of combining feedback monitoring with feedforward application signals.

The main contribution is a DSL that abstracts energy-policy specification in resource-constrained edge systems. It lets developers define states, events, thresholds, and transitions at a higher level, while generated instrumentation connects these policies to the C/FreeRTOS runtime.

As future work, we plan to expand the experimental evaluation with longer executions and more realistic IoT scenarios involving sensing, local processing, and wireless communication. We also intend to measure runtime overhead in more detail, refine the semantic validation rules of the DSL, and investigate lightweight adaptive heuristics for adjusting policy parameters on ESP32-based platforms.

Artifact Availability

The prototype implementation, workloads, and experimental scripts used in this work will be made available upon request.

References

[1] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[2] M. Satyanarayanan. The Emergence of Edge Computing. *Computer*, 50(1):30–39, 2017.

[3] Y. Mao, C. You, J. Zhang, and K. B. Letaief. A Survey on Mobile Edge Computing: The Communication Perspective. *IEEE Communications Surveys & Tutorials*, 19(4):2322–2358, 2017.

[4] B. Kopras, F. Idzikowski, and H. Bogucka. A Survey on Reduction of Energy Consumption in Fog Networks. *Sensors*, 24(18):6064, 2024.

[5] S. K. Panda, M. Lin, and T. Zhou. Energy-Efficient Computation Offloading With DVFS Using Deep Reinforcement Learning. *IEEE Internet of Things Journal*, 2023.

[6] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 2005.

[7] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, 2000.

[8] T. Lu, A. Kandemir, N. Vijaykrishnan, M. J. Irwin, and B. Shirazi. A Dynamic, Compiler Guided DVFS Mechanism to Achieve Energy-Efficiency in Multi-Core Processors. *Sustainable Computing: Informatics and Systems*, 12:1–9, 2016.

[9] J. Zidar, T. Matic, I. Aleksic, and Z. Hocenski. Dynamic Voltage and Frequency Scaling as a Method for Reducing Energy Consumption in Ultra-Low-Power Embedded Systems. *Electronics*, 13(5):826, 2024.