2026 Design, Automation & Test in Europe Conference (DATE 2026)

AERO: Adaptive and Efficient Runtime-Aware OTA Updates for Energy-Harvesting IoT

Wei Wei*, Jingye Xu*, Sahidul Islam†, Dakai Zhu*, Chen Pan‡, Mimi Xie*
*Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX, USA
†Department of Computer Science, Kennesaw State University, Marietta, GA, USA
‡Department of Electrical and Computer Engineering, The University of Texas at San Antonio, San Antonio, TX, USA
E-mail: {wei.wei2, jingye.xu, dakai.zhu, chen.pan, mimi.xie}@utsa.edu, sislam19@kennesaw.edu

***Abstract*—Energy-harvesting (EH) Internet of Things (IoT) devices operate under intermittent energy availability, which disrupts task execution and makes energy-intensive over-the-air (OTA) updates particularly challenging. Conventional OTA update mechanisms rely on reboots and incur significant overhead, rendering them unsuitable for intermittently powered systems. Recent live OTA update techniques reduce reboot overhead but still lack mechanisms to ensure consistency when updates interact with runtime execution. This paper presents AERO, an Adaptive and Efficient Runtime-Aware OTA update mechanism that integrates update tasks into the device's Directed Acyclic Graph (DAG) and schedules them alongside routine tasks under energy and timing constraints. By identifying update-affected execution regions and dynamically adjusting dependencies, AERO ensures consistent update integration while adapting to intermittent energy availability. Experiments on representative workloads demonstrate improved update reliability and efficiency compared to existing live update approaches.**

***Index Terms*—OTA Updates, EH, IoT, Scheduling, DAG**

I. Introduction

The growing demand for sustainable, low-maintenance computing has accelerated the adoption of EH technologies in next-generation IoT devices. By harvesting energy from ambient sources such as solar, thermal, kinetic, and radio-frequency signals [1], EH devices eliminate the need for frequent battery replacement and enable autonomous operation in resource-constrained environments. These capabilities make EH systems particularly valuable in applications such as agricultural sensing [2], medical implants [3], wildlife tracking [4], and disaster recovery [5]. The global market for EH technologies is projected to reach $6.5 billion by 2028 [6], underscoring their growing role in future sustainable IoT infrastructure.

To ensure the long-term functionality, security, and adaptability of EH devices, OTA updates provide a promising mechanism for firmware maintenance in EH IoT systems. However, frequent power interruptions create unpredictable execution windows, making it difficult to safely and efficiently complete OTA updates. Traditional OTA update mechanisms [7], [8] typically rely on rebooting into a bootloader to apply firmware updates. In this process, update packets are received and stored in non-volatile memory during normal execution, after which the system reboots to install the update. On devices with limited memory, updates must often be processed in smaller chunks, leading to multiple reboots. Each reboot incurs substantial energy overhead, increases latency, and disrupts routine execution. For intermittently powered EH devices, these cumulative costs significantly prolong downtime and increase the risk of incomplete or failed updates.

Live OTA update techniques enable updates to be applied at runtime without rebooting into a bootloader, thereby improving execution continuity [9]. While this approach reduces downtime and preserves system availability, it introduces new correctness challenges. During live updates, updated and non-updated code regions may temporarily coexist, and residual state in volatile memory can become inconsistent with the new logic. Such cross-version execution can compromise correctness and stability, motivating the need for mechanisms that ensure updates are applied only at safe execution points.

Although more recent live OTA update techniques have been proposed for EH IoT devices [10], they still fall short in ensuring correctness during execution. Existing approaches either overlook consistency issues that arise at runtime or avoid them by postponing updates until all routine tasks have completed. This limitation motivates the need for a runtime-aware OTA update mechanism that explicitly accounts for execution state.

To address these limitations, we propose AERO, a runtime-aware OTA update mechanism for EH IoT devices. The key contributions are summarized as follows:

- We propose a lightweight OTA packet format that encodes update task relations, routine task dependencies, and update operations for both update code and the DAG.
- We formally define *mutually dependent update groups* and use them to identify the corresponding *update-affected block* in the DAG, which captures update consistency requirements.
- We devise a runtime DAG adjustment algorithm that integrates update tasks and their dependencies with routine tasks under different update scenarios, enabling coordinated execution during updates.
- We propose a unified scheduling method that jointly schedules update and routine tasks under intermittent energy availability and deadline constraints.

II. Background and Motivation

A. Background

Several studies [7], [8], [10]–[13] have explored OTA update techniques for resource-constrained IoT devices, many of which

978-3-9826741-1-7/DATE26/© 2026 EDAA

are applicable to EH IoT devices. Much of this work reduces transmission and energy overhead through smaller update sizes, improved code similarity, and optimized encoding. Other efforts employ checkpointing strategies [14]–[16] to tolerate power interruptions, or hardware-aware methods [17]–[21] that align updates with available energy and memory constraints.

To provide context for our approach, we review three representative categories of OTA update techniques: whole image, incremental, and live update.

1) Whole Image Update: Whole image update replaces the entire firmware image by transmitting it to the device and overwriting the existing one. MSP430FRBoot [22] implements this approach using dual-flash banks, where the new image is written to an inactive bank and activated upon completion. Light Flash Write [19] improves reliability under intermittent power by lowering the energy cost of flash writes. However, whole image update remains costly for EH IoT devices, as transferring and writing an entire firmware image requires significant energy and memory resources, increasing the likelihood of incomplete updates under intermittent execution.

2) Incremental Update: Incremental update reduces overhead by transmitting only modified firmware portions using delta encoding [8]. For flash-based devices, segment-based designs minimize memory operations and energy usage during intermittent updates [17]. At the network level, multicast and beamforming improve transmission efficiency across multiple devices in LoRa networks [20], [21]. Despite these benefits, incremental update still relies on rebooting into a bootloader, which is problematic for intermittently powered devices.

3) Live Update: Live update applies firmware modifications during runtime without requiring reboots. Trampoline-based patching [9] introduced atomic code replacement, and more recent work extends this capability to intermittently powered devices [10]. While these techniques reduce reboot overhead, they often assume that updates can be safely applied without causing inconsistencies, which is not always valid in real-time or intermittently powered systems.

B. Motivation

Although live OTA updates reduce reboot overhead, they also introduce new consistency risks, highlighting the need for runtime-aware OTA update mechanisms that explicitly manage interactions between update tasks and ongoing execution. During runtime modification, updated and non-updated code regions may temporarily coexist, leading to mixed-version execution. In some cases this has no observable effect, but in others it can cause incorrect computation, execution failure, or unintended interactions, as illustrated in Fig. 1. These risks are amplified in EH devices, where firmware is decomposed into fine-grained tasks to align with short power cycles, sometimes only a few instructions long [23], [24]. Under such conditions, even subtle inconsistencies can propagate into diverse failure types. Although prior work [10] acknowledges these issues, the challenge of handling mixed-version execution remains unresolved.

A further motivation comes from the observation that update tasks do not need to strictly wait until routine tasks complete,

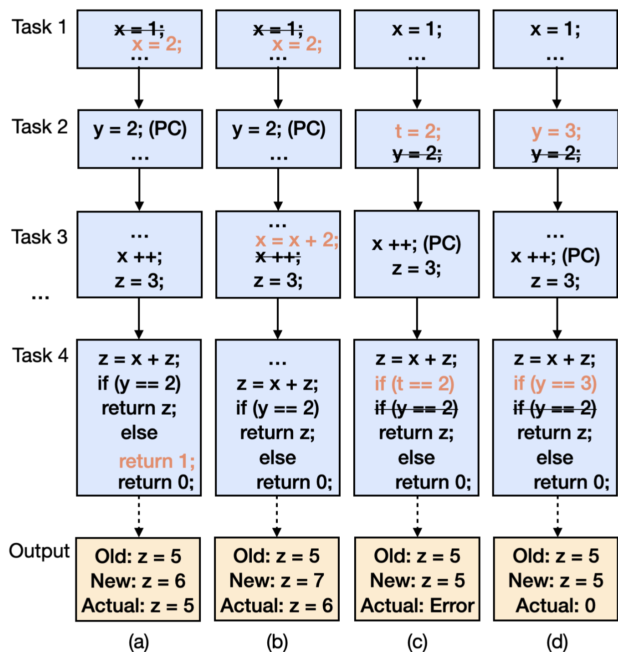

Fig. 1. Motivating Examples of Mixed-Version Execution: (a) No Impact, (b) Incorrect Computation, (c) Execution Failure, (d) Unintended Interaction.

nor should routine tasks be stalled until updates finish. Task execution in EH devices often contains idle periods due to scheduling gaps, during which update tasks can safely proceed. By treating updates as schedulable entities within the DAG, such idle time can be effectively utilized to advance updates without disrupting routine execution.

In addition to scheduling concerns, updates may also require structural changes to the DAG itself. Some updates modify tasks or dependencies, add new tasks, or remove obsolete ones, thereby reshaping the DAG beyond simple code replacement. To the best of our knowledge, no existing OTA update approach for EH IoT devices supports such DAG-level updates.

III. AERO: Runtime-Aware OTA Update Mechanism

We develop AERO, which integrates update tasks and their dependencies into the runtime DAG using a lightweight, dependency-driven packet format. AERO performs runtime DAG adjustment followed by unified scheduling to coordinate update and routine tasks under energy and timing constraints. AERO enforces task-level control-flow and dependency consistency, assuming tasks encapsulate local state and access shared memory, peripherals, and interrupts only at task boundaries.

A. Definition

1) Mutually Dependent Update Group Definition: Let $T = T_{\text{old}} \cup T_{\text{new}}$ denote all firmware tasks, where $T_{\text{old}} = \{t_1, t_2, \dots, t_m\}$ are existing tasks and $T_{\text{new}} = \{t_{m+1}, t_{m+2}, \dots, t_n\}$ are new tasks introduced by updates. Execution is modeled as a DAG $G = (T, E)$, where $E \subseteq T \times T$ and $(t_i, t_j) \in E$ indicates that t_j depends on t_i. Correspondingly, let $U = U_{\text{old}} \cup U_{\text{new}}$ denote the set of update tasks, with each $u_i \in U$ associated with its corresponding $t_i \in T$. Update tasks modify both the code of their associated tasks and any

linked dependencies. An update task u_i depends on another update task u_j if both must be updated in the same process for correctness. A *mutually dependent update group* is a minimal subset $M \subseteq U$ where every $u_i \in M$ depends on at least one $u_j \in M$, and correctness is preserved only when all tasks in M are updated together.

2) Update-Affected Block Definition: The *update-affected block* is the minimal subgraph of the DAG containing all tasks in a *mutually dependent update group*, along with intermediate nodes and edges on the dependency paths connecting them. This block represents the execution region influenced by the update and must be treated as a whole during integration.

B. Dependency-Driven Update Packet

AERO introduces a dependency-driven update packet format, shown in Fig. 2, to enable runtime-aware integration of updates in EH IoT devices. Each packet payload consists of a group field and an update operation. The group field, included only in the first packet, encodes the *mutually dependent update group* with one bit per routine task in the DAG, where 1 indicates association with the update and 0 indicates none. Unused positions are reserved for future tasks. This field allows the system to determine the *update-affected block* early and supports incremental processing, so that updates can progress even when memory cannot hold the entire update at once.

The update operation contains a header and an update block. The header begins with a 2-bit operation code specifying the update type, based on intermittent-aware update operations in [10]. A 1-bit DAG flag indicates whether the update block carries dependency information. If set, the first n bits of the update block form an optional dependency field, with each bit specifying whether the update task depends on the corresponding routine task. The remaining header bits identify the task ID, and the update block concludes with an m-bit code segment delivering the update content.

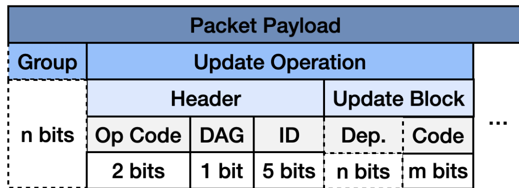

Fig. 2. AERO Update Packet Structure

C. Runtime DAG Adjustment

To support runtime-aware updates without interrupting execution, AERO introduces a dedicated update task chain that is triggered upon receiving an update notification. As shown in Fig. 3, the chain inserts the update tasks together with their dependency adjustments into the DAG.

1) Modifying Existing Tasks: The framework supports updating both tasks and their dependencies when modifying existing tasks. In the example of Fig. 4(a), update tasks modify t_4 and t_8. Integration depends on the current execution. If runtime is within the *update-affected block* (e.g., at t_4, t_5, t_6, or t_8), updates are deferred until the corresponding tasks complete

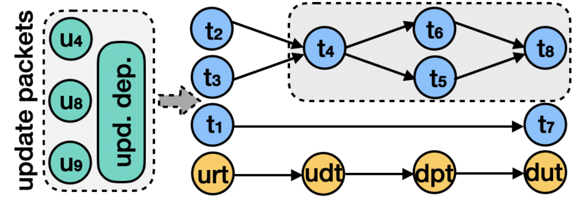

Fig. 3. Overview of the AERO Update Process.
Update Receiving Task (urt): collects incoming packets; Update Decoding Task (udt): decodes packets and reconstructs update tasks; Dependency Processing Task (dpt): identifies the *update-affected block*; DAG Updating Task (dut): inserts update tasks and adjusts affected dependencies in G.

so the current cycle finishes under the old version. Otherwise, updates are inserted before their associated tasks and the system adopts the new version in the same execution. For clarity, this example shows only task updates without dependency changes.

2) Inserting New Tasks: The framework also supports inserting new tasks into the DAG. In Fig. 4(b), update task u_9 creates a new task t_9. Because t_9 does not exist beforehand, u_9 must execute first regardless of runtime to ensure the task is available for scheduling. In practice, new tasks may introduce dependencies with existing tasks, requiring joint updates to preserve consistency. For clarity, this example shows only insertion of t_9 without dependency changes.

3) Removing Obsolete Tasks: The framework also supports removing obsolete tasks from the DAG. In Fig. 4(c), update task u_8 removes task t_8 and all linked dependencies. If runtime is within the *update-affected block*, removal is deferred until t_8 completes; otherwise it occurs before t_8 begins. In practice, a task may have dependencies with others, and its removal requires updating those tasks to maintain consistency. For clarity, this example shows only removal of t_8 without additional dependency changes.

4) Runtime DAG Adjustment Algorithm: Based on the illustrated cases, we now present AERO's algorithm for runtime DAG adjustment in EH IoT devices. The procedure first inserts a virtual start node s connected to all original sources, ensuring that even if the *update-affected block* begins at the graph entry, execution can still be temporarily blocked before entering the block. The block B is then identified, and all incoming edges into B are removed to prevent new execution paths while updates are pending. This blocking is essential when memory cannot hold all update packets, since without it routine tasks inside B could execute under outdated code, leading to inconsistencies. By deferring entry into B until update tasks are completed, AERO preserves correctness while allowing unaffected tasks outside the block to continue. Following prior intermittent update designs [10], AERO checkpoints both update and routine tasks in non-volatile memory, allowing partially applied updates to be safely recovered and re-applied after power loss.

D. Unified Scheduling Algorithm

Once update tasks are integrated, AERO employs a unified scheduler to coordinate routine and update tasks while adapting

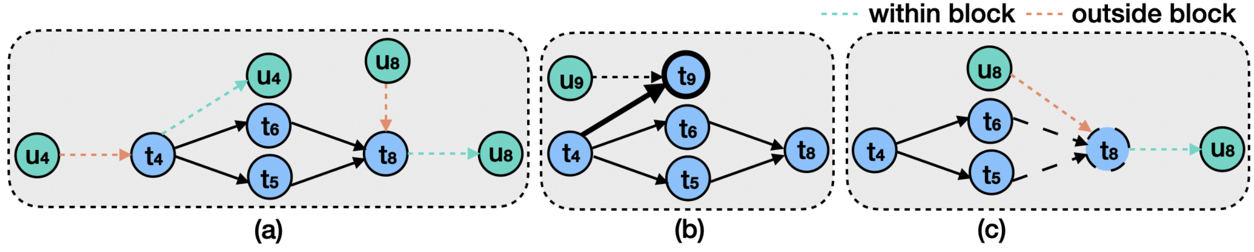

Fig. 4. AERO Update Cases: (a) Modifying Existing Tasks t_4 and t_8; (b) Inserting New Task t_9; (c) Removing Obsolete Task t_8

Algorithm 1 Runtime DAG Adjustment Algorithm

1: **Input:** $G = (T_{\text{old}}, E)$; update group M; current task t_{curr}
2: **Output:** Modified DAG G', blocked edges E_{blk}
3: $T' \leftarrow T_{\text{old}} \cup \{s\}, \quad E' \leftarrow E \cup \{(s, v) \mid v \in \text{sources}(T_{\text{old}})\}$ // add virtual start node s
4: $B \leftarrow f(M, (T', E'))$ // identify update-affected block
5: $E_{\text{blk}} \leftarrow \{(x, y) \in E' \mid y \in B\}$ // backup blocked edges
6: $E' \leftarrow E' \setminus E_{\text{blk}}$ // block entry into B
7: **for** each $u_i \in M$ **do**
8: let t_i be the task associated with u_i
9: **if** $t_i \notin T'$ **then**
10: $T' \leftarrow T' \cup \{u_i, t_i\}$ // add u_i and placeholder t_i
11: $E' \leftarrow E' \cup \{(u_i, t_i)\}$ // add edge (u_i, t_i)
12: **else**
13: **if** $t_{\text{curr}} \in B$ **then**
14: $T' \leftarrow T' \cup \{u_i\}$
15: $E' \leftarrow E' \cup \{(t_i, u_i)\}$ // defer until t_i completes
16: **else**
17: $T' \leftarrow T' \cup \{u_i\}$
18: $E' \leftarrow E' \cup \{(u_i, t_i)\}$ // update before t_i
19: **end if**
20: **end if**
21: **end for**
22: $G' \leftarrow (T', E')$
23: **return** G', E_{blk}

to energy and deadline constraints. Figure 5 provides a graphical overview of the task selection process, while Algorithm 2 presents the detailed procedure. Together, they show how updates are applied safely and efficiently without compromising real-time performance or exceeding energy budgets.

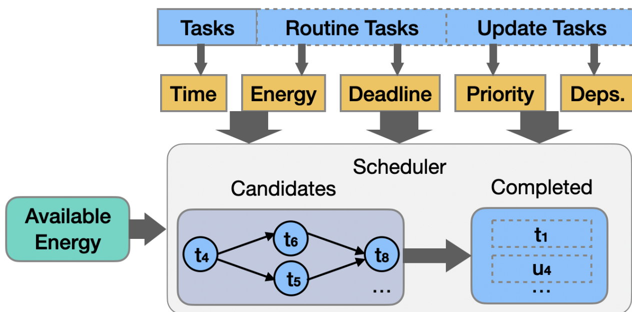

Fig. 5. Graphical Illustration of Tasks Selection Process

Task selection is handled with a priority queue ordered by deadline, then priority. Routine tasks execute directly, while update tasks are applied according to their operations. If the DAG bit in an update operation is set, the corresponding dependency edges are added to the DAG before the update is applied. When all update tasks in the current group M are completed, the scheduler restores the entry edges of the *update-affected block* B removed in Algorithm 1. It then cleans the graph by removing update tasks, helper tasks, and the virtual start node along with their connected edges. The complete procedure is summarized in Algorithm 2.

Algorithm 2 Unified Scheduling Algorithm

1: **Input:** $G' = (T', E')$; blocked edges E_{blk}
2: **Output:** Executed task t^* in each scheduling step
3: $Q \leftarrow \emptyset$ // priority queue by deadline→priority
4: **while** true **do**
5: **for** each $t \in T'$ with $(\forall p \in \text{pred}(t) : p \text{ completed})$ **do**
6: $Q \leftarrow Q \cup \{t\}$
7: **end for**
8: **if** $Q = \emptyset$ **then**
9: **continue** // no schedulable task, recheck
10: **else**
11: $t^* \leftarrow \text{pop}(Q)$ // earliest deadline, then priority
12: **if** $t^* \in T$ **then**
13: $\text{exec}(t^*)$ // execute routine task
14: **else**
15: **if** $\text{header}(t^*).\text{DAG} = 1$ **then**
16: $E' \leftarrow E' \cup \Delta E(t^*)$ // add new edges
17: **end if**
18: $\text{apply}(t^*)$ // apply update operation
19: **end if**
20: $\text{status}(t^*) \leftarrow \text{completed}$
21: **end if**
22: **if** $M \subseteq \{\text{completed tasks}\}$ **then**
23: $E' \leftarrow E' \cup E_{\text{blk}}$ // restore blocked edges into B
24: $S \leftarrow M \cup \{urt, udt, dpt, dut, s\}$
25: $T' \leftarrow T' \setminus S$ // remove update, helpers, virtual node
26: $E' \leftarrow E' \setminus \{ (u, v) \in E' \mid (u \in S) \vee (v \in S) \}$
27: $G \leftarrow (T', E')$ // store updated DAG
28: **break**
29: **end if**
30: **end while**

IV. Experiment

A. Experimental Setup

All experiments are conducted on the TI MSP430FR5994 microcontroller [25], a resource-constrained device featuring ultra-low power operation and embedded FRAM, making it suitable for EH IoT systems. Execution time and energy consumption are obtained using TI EnergyTrace [26] in Code Composer Studio (CCS) [27] for standalone profiling of benchmark tasks. Update tasks are assigned execution time and energy

values based on their respective update sizes. These profiled results are later used to evaluate task execution and update performance under AERO's unified scheduling.

B. Benchmark Design

To evaluate the effectiveness of AERO, we use four benchmarks that represent diverse application types and DAG structures. For each benchmark, the capacitor size is set so that the stored energy is just sufficient to execute the task with the highest energy demand. This configuration drives operation near the system's energy limits, providing realistic conditions for assessing runtime-aware updates. Table I summarizes the benchmarks and their configurations.

The four benchmarks capture different aspects of runtime-aware updates relevant to AERO. Quick Sort (B1) [28] illustrates fine-grained scheduling with very small tasks. AES Encryption (B2) [15] exercises update behavior across parallel but functionally distinct hardware and software execution paths. LeNet-5 (B3) [29] provides a strictly linear task pipeline, suitable for analyzing update propagation under ordered execution. Heart Rate Monitor (B4) [30] combines parallelism and recombination in a fork-join DAG, enabling evaluation of consistency under partially parallel execution.

C. OTA Update Scenarios

Together, these benchmarks provide the basis for evaluating runtime-aware updates across diverse application types and DAG structures. Building on this foundation, we design six update scenarios derived from the benchmarks in Table I. The scenarios cover two categories of updates. Functional modifications change task behavior, such as adjusting sorting routines, replacing the final stage of a neural network, or introducing new signal-processing steps. Parametric modifications change configuration parameters without affecting task behavior, for example increasing AES key or block sizes, updating model weights, or changing UART settings.

Each scenario introduces one or more update tasks that form a *mutually dependent update group*, which must be integrated at runtime without violating DAG dependencies or compromising consistency. Update sizes span from small task-level changes to large firmware updates in EH IoT devices. Table II summarizes the six scenarios and their update sizes.

We do not define separate scenarios for full image update or DAG update. Full image update can be performed by any approach and does not differentiate them, while DAG update is uniquely supported by AERO and is evaluated through feature comparison in Section V.

V. Evaluation

Evaluation is conducted through simulation using execution time and energy values obtained from standalone task profiling. Available energy is updated continuously using real-world solar traces from an EH IoT device [31], and update arrivals follow a pseudo-random process to emulate stochastic behavior. Task priorities follow the DAG, where earlier tasks have higher priority, later tasks have lower priority, and update tasks always run last. We assume sufficient non-volatile storage and that all update packets arrive before integration. Although AERO can handle multi-packet, incremental updates, this assumption removes communication cost as a variable and enables a fair comparison focused on runtime execution. For comparison, we evaluate AERO against two representative baselines: the live update baseline corresponds to the trampoline-based live update approach in [9], while the intermittent update baseline corresponds to the checkpoint-assisted OTA framework in [10].

A. Feature Comparison

We first compare the features of AERO against the baselines. The baselines offer established capabilities such as live update without reboot and intermittent execution with consistency mechanisms, and both can perform progressive full image update. However, neither supports runtime integration or DAG update. AERO adds these capabilities by integrating update tasks into the DAG and scheduling them alongside routine tasks, enabling task dependency updates that are not possible under existing approaches. Figure 6 summarizes these differences.

Feature	Live	Intermittent Update	AERO
Live Update Support	✓	✓	✓
Progressive Full Image Update	✓	✓	✓
Intermittent Update Support	✗	✓	✓
Consistency Guarantee	✗	✓	✓
Runtime Integration	✗	✗	✓
DAG Update Support	✗	✗	✓

Fig. 6. Feature Comparison of Update Approaches

B. Update Error Rate

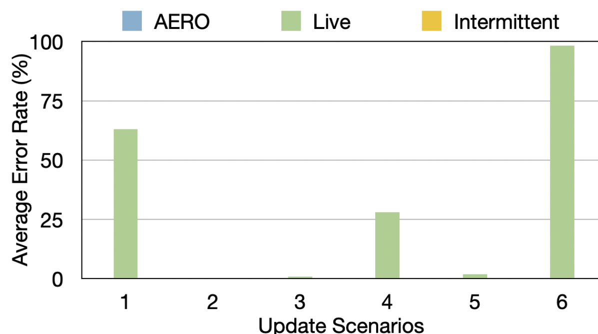

Fig. 7. Comparison of Average Error Rate

As shown in Fig. 7, AERO maintains a zero error rate across all scenarios. In contrast, the live update approach frequently introduces inconsistencies when updates occur in computation-intensive regions of the DAG; for example, in Scenarios 4 and 6, updates overlap with high-cost tasks, leading to error rates as high as 98.3% in Scenario 6. Even Scenario 1, which consists of lightweight tasks, shows high error rates under live update because uniformly short execution times increase the chance of interrupting an active task. By comparison, Scenarios 2, 3, and 5 indicate that live update can succeed for small, isolated updates, though additional safeguards are needed to ensure correctness. Intermittent update also maintains a zero error rate

TABLE I
EVALUATION BENCHMARKS

Benchmark	Size (B)	DAG Type	Cap. (mF)	Description
B1: Quick Sort [28]	1,696	Linear	0.02	Four sequential quick sort operations
B2: AES Encryption [15]	3,975	Parallel	0.2	AES encryption/decryption using HW/SW paths
B3: LeNet-5 [29]	50,632	Linear	10	Inference using an optimized LeNet-5 model
B4: Heart Rate Monitor [30]	46,050	Fork-Join	1	PPG-based heart rate estimation

TABLE II
OTA UPDATE SCENARIOS

ID	Description	Size (B)
1	B1: Modify 2nd sort task	280
2	B2: AES key to 192-bit (acc/DMA)	130
3	B2: AES block to 256-bit	162
4	B3: Final layer to FC	292
5	B4: Add sensor support	702
6	B4: Update HR model & UART	6246

by design, but this guarantee comes at the cost of long update delays, which may be unacceptable for time-sensitive updates such as security patches.

C. Update Completion Time

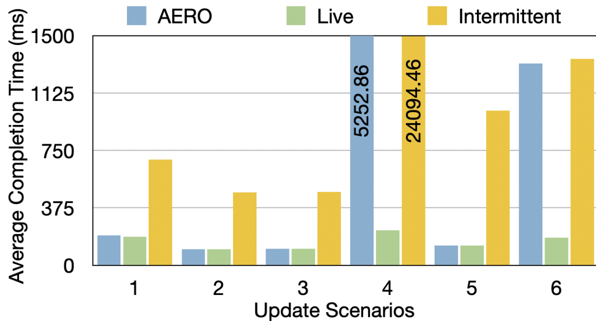

Fig. 8. Comparison of Average Update Completion Time

Figure 8 shows the average update completion time for all approaches. AERO consistently outperforms intermittent update in Scenarios 1–5 by applying updates more quickly through direct integration of update tasks into the DAG, and even in Scenario 6, which involves the largest update size, it still achieves slightly shorter completion time. Compared to live update, our approach shows similar completion times in Scenarios 1, 2, 3, and 5. Small delays occur only when the *update-affected block* is executing, since updates are deferred to preserve correctness, which aligns with the zero error rate results in Fig. 7. The trade-off is most evident in Scenarios 4 and 6, where live update completes faster but suffers from very high error rates, as shown in Fig. 7.

D. Deadline Miss Rate (DMR)

Figure 9 compares the DMR across the three approaches. Deadlines are defined only for routine tasks as the profiled execution time plus a 0.5× margin to account for scheduling overhead, while update tasks are not assigned deadlines and always execute after routine tasks. AERO achieves results similar to or slightly higher than intermittent update in all

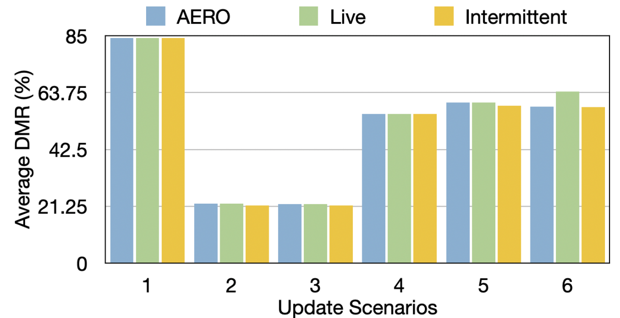

Fig. 9. Comparison of Average DMR

scenarios, showing that updates can be integrated at runtime without excessive overhead. Compared to live update, AERO yields similar DMR in Scenarios 1–5. In Scenario 6, where the update size is relatively large, AERO achieves a lower DMR because live update applies updates immediately without considering the current execution state, delaying routine tasks and increasing DMR.

Although our experiments use MSP430FR5994 hardware and solar energy traces, AERO is not limited to this platform or energy source. Its DAG-level design enables broad applicability across diverse EH IoT devices and environments, while introducing overhead only from runtime DAG adjustments triggered upon update arrivals. This overhead primarily incurs non-volatile memory writes for DAG modifications and is negligible in practice for our evaluated scenarios, without affecting the observed evaluation trends.

VI. CONCLUSION

This paper presented AERO, a runtime-aware OTA update mechanism for intermittently powered IoT devices. By integrating update tasks into the executing DAG, AERO preserves correctness while enabling updates during normal operation. Simulations using real-world energy traces and hardware profiles show that AERO achieves zero update errors, shorter update completion times, and competitive DMR compared with live and intermittent baselines. Beyond single-device evaluations, AERO also enables future work on coordinated updates in EH IoT networks. Overall, AERO delivers a practical and reliable OTA update solution, with a DAG-level design that generalizes across hardware platforms and EH conditions to support sustainable, long-lived IoT deployments.

VII. ACKNOWLEDGMENT

This work was supported in part by the U.S. National Science Foundation under Grant CNS-2443885 and CNS-2318641.

REFERENCES

[1] Yuehang Sun, Yun-Ze Li, and Man Yuan. Requirements, challenges, and novel ideas for wearables on power supply and energy harvesting. *Nano Energy*, 115:108715, 2023.

[2] Kapil Aggarwal, G Sreenivasula Reddy, Ramesh Makala, T Srihari, Neetu Sharma, and Charanjeet Singh. Studies on energy efficient techniques for agricultural monitoring by wireless sensor networks. *Computers and Electrical Engineering*, 113:109052, 2024.

[3] Md Maruf Hossain Shuvo, Twisha Titirsha, Nazmul Amin, and Syed Kamrul Islam. Energy harvesting in implantable and wearable medical devices for enduring precision healthcare. *Energies*, 15(20):7495, 2022.

[4] Sivan Toledo, Shai Mendel, Anat Levi, Yoni Vortman, Wiebke Ullmann, Lena-Rosa Scherer, Jan Pufelski, Frank Van Maarseveen, Bas Denissen, Allert Bijleveld, et al. Vildehaye: A family of versatile, widely-applicable, and field-proven lightweight wildlife tracking and sensing tags. In *2022 21st ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*, pages 1–14. IEEE, 2022.

[5] Xiuchuan Sun, Jian Chen, Haoran Zhao, Wen Zhang, and Yicheng Zhang. Sequential disaster recovery strategy for resilient distribution network based on cyber–physical collaborative optimization. *IEEE Transactions on Smart Grid*, 14(2):1173–1187, 2022.

[6] Global markets, technologies and devices for energy harvesting. https://www.bccresearch.com/market-research/energy-and-resources/global-markets-technologies-and-devices-for-energy-harvesting.html#:~:text=What%20is%20the%20projected%20market,11.4%25%20during%20the%20forecast%20period, Jul 2023.

[7] Wei Dong, Yunhao Liu, Chun Chen, Jiajun Bu, Chao Huang, and Zhiwei Zhao. R2: Incremental reprogramming using relocatable code in networked embedded systems. *IEEE Transactions on Computers*, 62(9):1837–1849, 2012.

[8] Wei Wei, Sahidul Islam, Jishnu Banerjee, Shanglin Zhou, Chen Pan, Caiwen Ding, and Mimi Xie. An intermittent ota approach to update the dl weights on energy harvesting devices. In *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, pages 1–6. IEEE, 2022.

[9] Chi Zhang, Wonsun Ahn, Youtao Zhang, and Bruce R Childers. Live code update for iot devices in energy harvesting environments. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6. IEEE, 2016.

[10] Wei Wei, Chen Pan, Sahidul Islam, Jishnu Banerjee, Shyamala Palanisamy, and Mimi Xie. Intermittent ota code update framework for tiny energy harvesting devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[11] Joel Koshy and Raju Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks, 2005.*, pages 354–365. IEEE, 2005.

[12] Chih-Chieh Han, Ram Kumar Rengaswamy, Roy Shea, Eddie Kohler, and Mani Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, volume 10, pages 1067170–1067188. Citeseer, 2005.

[13] Rajesh Krishna Panta and Saurabh Bagchi. Hermes: Fast and energy efficient incremental code updates for wireless sensor networks. In *IEEE INFOCOM 2009*, pages 639–647. IEEE, 2009.

[14] Priyanka Singla and Smruti R Sarangi. A survey and experimental analysis of checkpointing techniques for energy harvesting devices. *Journal of Systems Architecture*, 126:102464, 2022.

[15] Shyamala Palanisamy, Wei Wei, and Mimi Xie. Energy-efficient persistently secure block-based differential checkpointing for energy harvesting devices. In *2025 26th International Symposium on Quality Electronic Design (ISQED)*, pages 1–6. IEEE, 2025.

[16] Sahidul Islam, Wei Wei, Jishnu Banerjee, and Chen Pan. Energy-adaptive checkpoint-free intermittent inference for low power energy harvesting systems. In *2025 26th International Symposium on Quality Electronic Design (ISQED)*, pages 1–7. IEEE, 2025.

[17] Wei Wei, Jishnu Banerjee, Sahidul Islam, Chen Pan, and Mimi Xie. Energy-aware incremental ota update for flash-based batteryless iot devices. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 51–56. IEEE, 2024.

[18] Jishnu Banerjee, Sahidul Islam, Wei Wei, Chen Pan, Dakai Zhu, and Mimi Xie. Memory-aware efficient deep learning mechanism for iot devices. In *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 187–194. IEEE, 2021.

[19] Songran Liu, Mingsong Lv, Wei Zhang, Xu Jiang, Chuancai Gu, Tao Yang, Wang Yi, and Nan Guan. Light flash write for efficient firmware update on energy-harvesting iot devices. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[20] Zehua Sun, Tao Ni, Huanqi Yang, Kai Liu, Yu Zhang, Tao Gu, and Weitao Xu. Flora+: Energy-efficient, reliable, beamforming-assisted, and secure over-the-air firmware update in lora networks. *ACM Transactions on Sensor Networks*, 20(3):1–28, 2024.

[21] Sukanya Jewsakul and Edith CH Ngai. Fiora: Energy neutrality-aware multicast firmware distributions in energy-harvesting lora networks. In *Proceedings of the 11th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pages 88–98, 2024.

[22] Ryan Brown, Katie Pier, and Gary Gao. Msp430frboot: Main memory bootloader and over-the-air updates for msp430 fram large memory model devices. https://www.ti.com/lit/an/slaa721e/slaa721e.pdf, March 2020.

[23] Shuo Xu, Wei Zhang, Mengying Zhao, Zimeng Zhou, and Lei Ju. Cache-aware task decomposition for efficient intermittent computing systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

[24] Hehe Li, Yongpan Liu, Chenchen Fu, Chun Jason Xue, Donglai Xiang, Jinshan Yue, Jinyang Li, Daming Zhang, Jingtong Hu, and Huazhong Yang. Performance-aware task scheduling for energy harvesting non-volatile processors considering power switching overhead. In *Proceedings of the 53rd Annual Design Automation Conference*, pages 1–6, 2016.

[25] Texas Instruments. Msp430fr5994 microcontroller. https://www.ti.com/product/MSP430FR5994, 2024.

[26] Texas Instruments. EnergyTrace. https://software-dl.ti.com/ccs/esd/documents/xdsdebugprobes/emu_energytrace.html.

[27] Texas Instruments. Code composer studio. https://www.ti.com/tool/CCSTUDIO.

[28] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[29] Sahidul Islam, Shanglin Zhou, Ran Ran, Yu-Fang Jin, Wujie Wen, Caiwen Ding, and Mimi Xie. Eve: Environmental adaptive neural network models for low-power energy harvesting system. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9, 2022.

[30] Jingye Xu, Yuntong Zhang, Mimi Xie, Wei Wang, and Dakai Zhu. Real-time intelligent on-device monitoring of heart rate variability with ppg sensors. *Journal of Systems Architecture*, 154:103240, 2024.

[31] Jishnu Banerjee, Sahidul Islam, Wei Wei, Chen Pan, and Mimi Xie. Autotile: Autonomous task-tiling for deep inference on battery-less embedded system. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, pages 323–327, 2024.