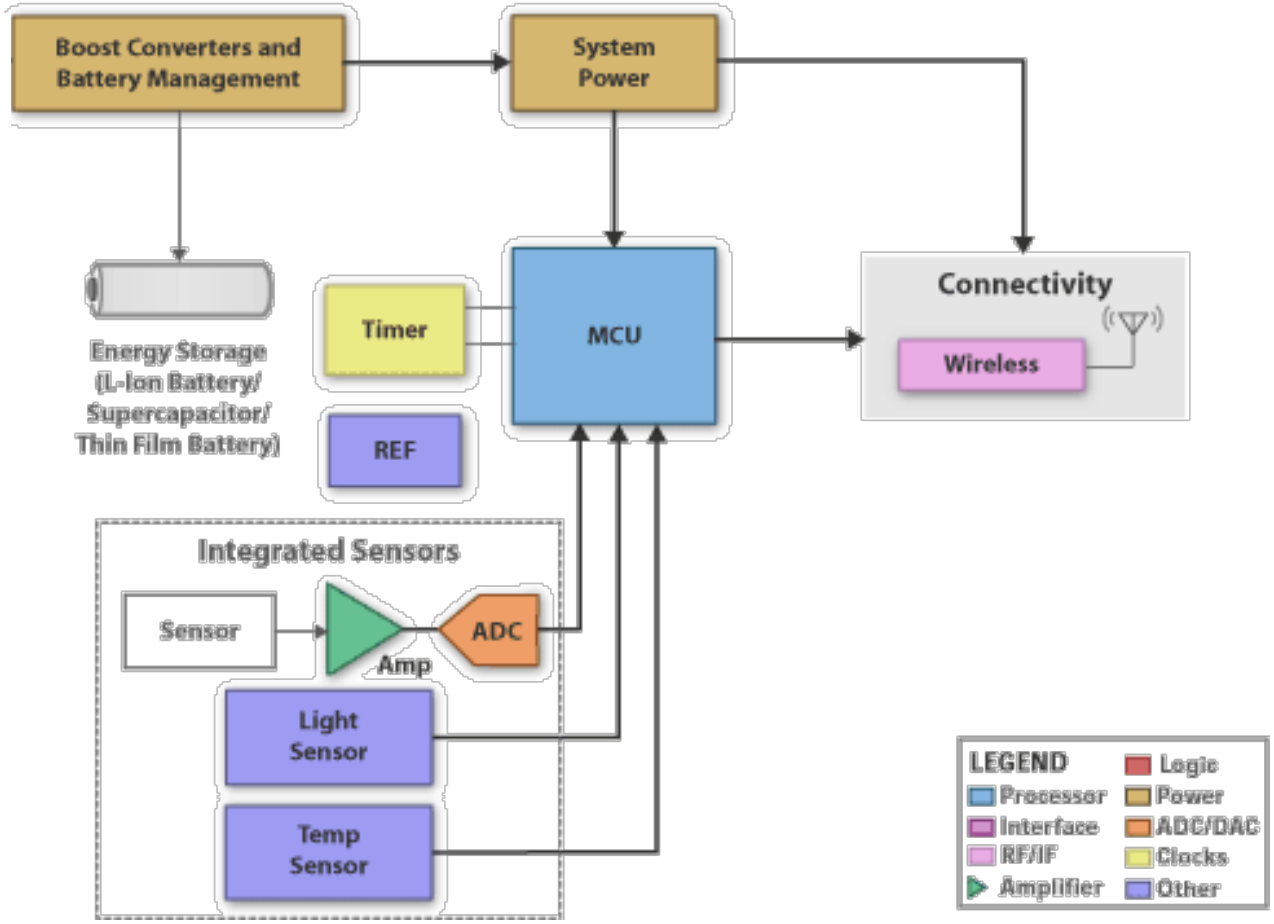
Live Code Update for IoT Devices

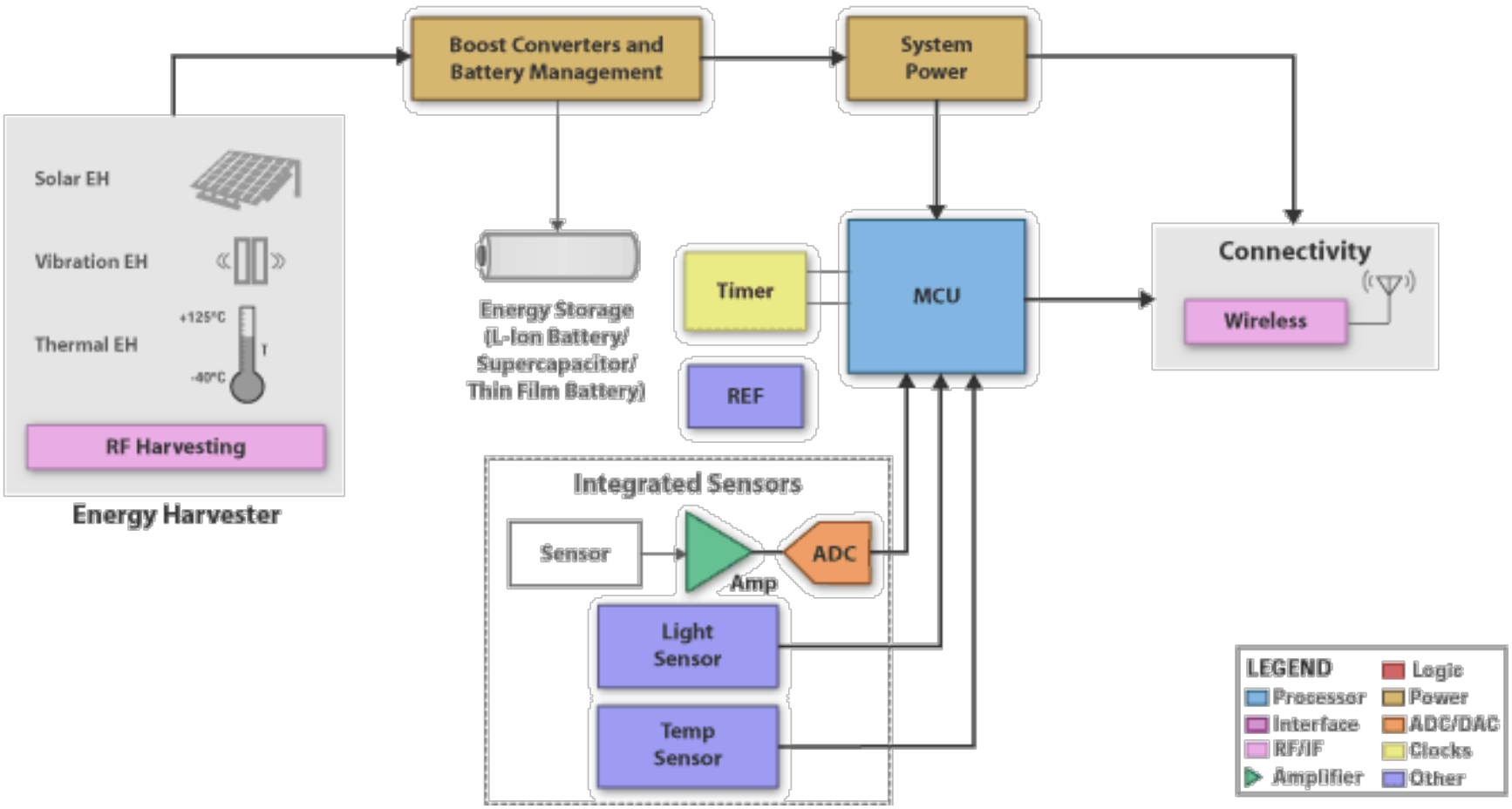
Chi Zhang, **Wonsun Ahn**, Bruce Childers, Youtao Zhang
Department of Computer Science
University of Pittsburgh

NVMSA, August 2016

Battery-Powered IoT Device



Energy-Harvesting IoT Device



Code Update for IoT Devices

- Need for code update post-deployment
  - Debugging: some bugs only manifest in the field
  - Tuning: tune parameters based on feedback
  - New / modified functionality: collect new data
- Code update post-deployment is a challenge
  - Difficult to collect devices scattered in the field
  - Devices often at hard-to-reach locations (e.g. embedded in body, hazardous locations …)

➜ Code update often needs to be performed **wirelessly** in the field, relying on **local power source**

Design Goals for Code Update

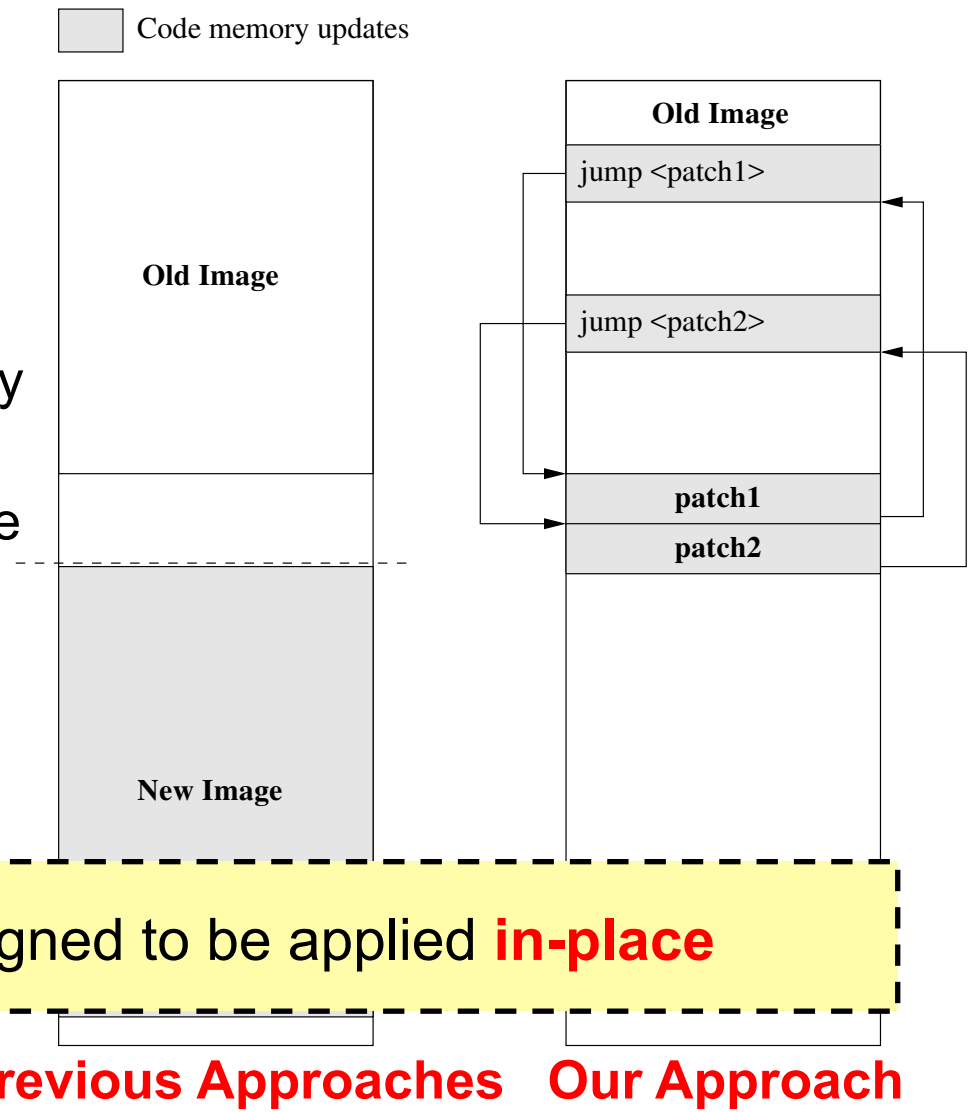
- Minimize device down time
  - Prolonged down time can impact QoS
    (e.g. A sensor device deployment at Reventador Volcano could be up for only 69% of the time due to code updates)
  - Some applications have real-time deadlines
- Minimize reprogramming time
  - Consists of code transfer time + code write time
  - For battery-powered devices: improves lifetime of device
  - For energy-harvesting devices: reduces turn around time
    - Important in iterative debug / tuning scenarios
- Minimize extra memory required for code update
  - Larger memory means more costly devices

Previous Code Update Approaches

- Naïve: Whole image transfer / whole image write
  1. Rebuild code image at base station
  2. Transfer image wirelessly to each IoT device
  3. Write image in separate area in code memory in device
  4. Reboot device from image
- Incremental: Partial image transfer / whole image write
  1. Rebuild code image at base station
  2. Do a diff between old and new image and calculate **delta**
  3. Transfer **delta** wirelessly to each IoT device
  4. Write new image from **delta** and old image in device
  5. Reboot device from new image

Our Code Update Approach

- In-place: Partial image transfer / partial image write
  1. Generate code **patches** at base station
  2. Transfer **patches** wirelessly to each IoT device
  3. Apply patches to **live** image in device **in-place**

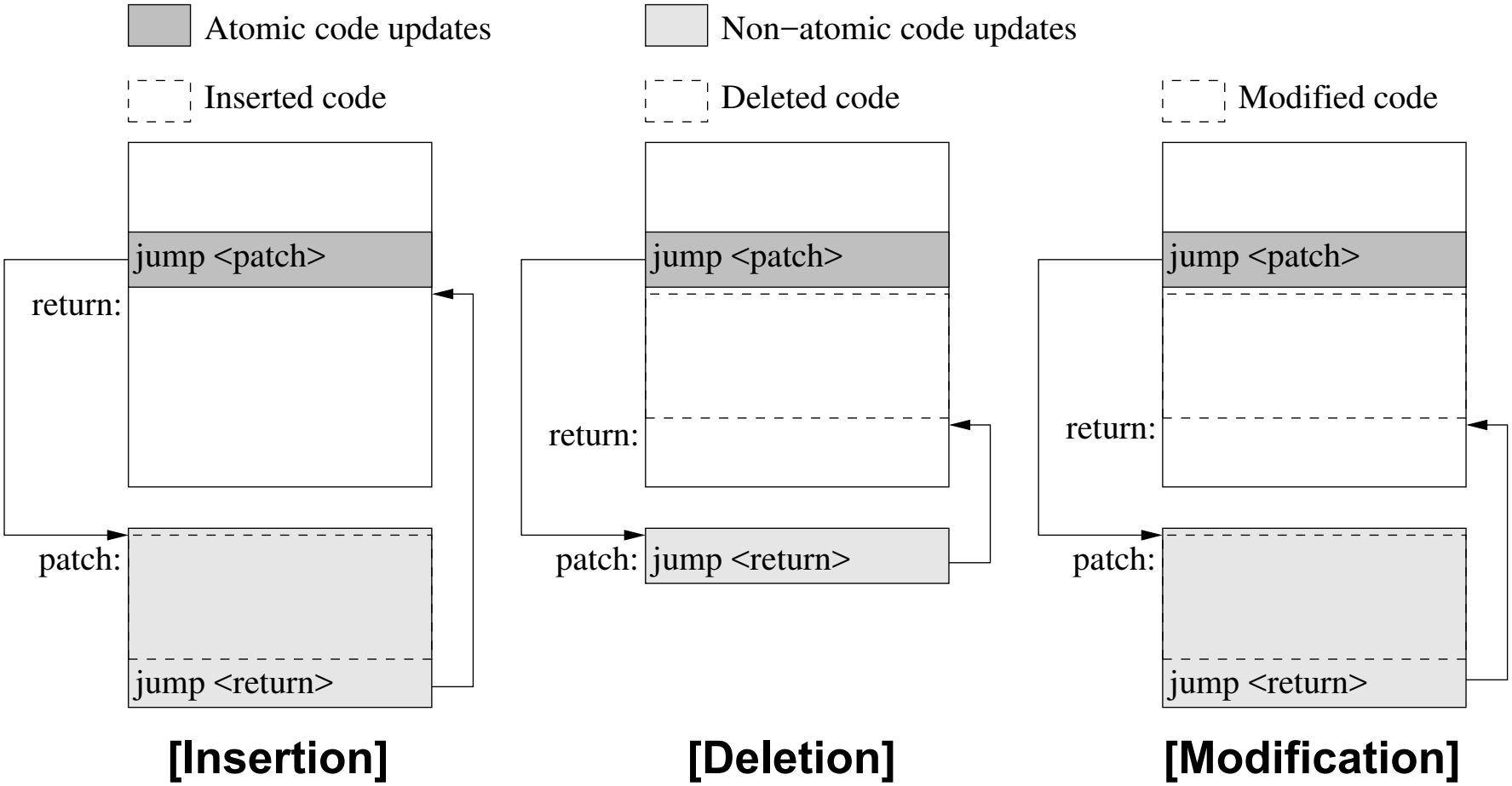


➜ **Patches** are **deltas** designed to be applied **in-place**

Code Update Approach Comparison

	Device Down Time	Reprogramming Energy	Memory Requirement
Naïve	✗ (Need reboot)	✗ (Transfer and write whole image)	✗ (Need twice code image size)
Incremental	✗ (Need reboot)	Δ (Transfer only delta but write whole image)	✗ (Need twice code image size)
In-place	✓ (No reboot needed due to live update)	✓ (Transfer only patches and apply in-place)	✓ (Need only code image size)

Live In-Place Code Patching Scenarios

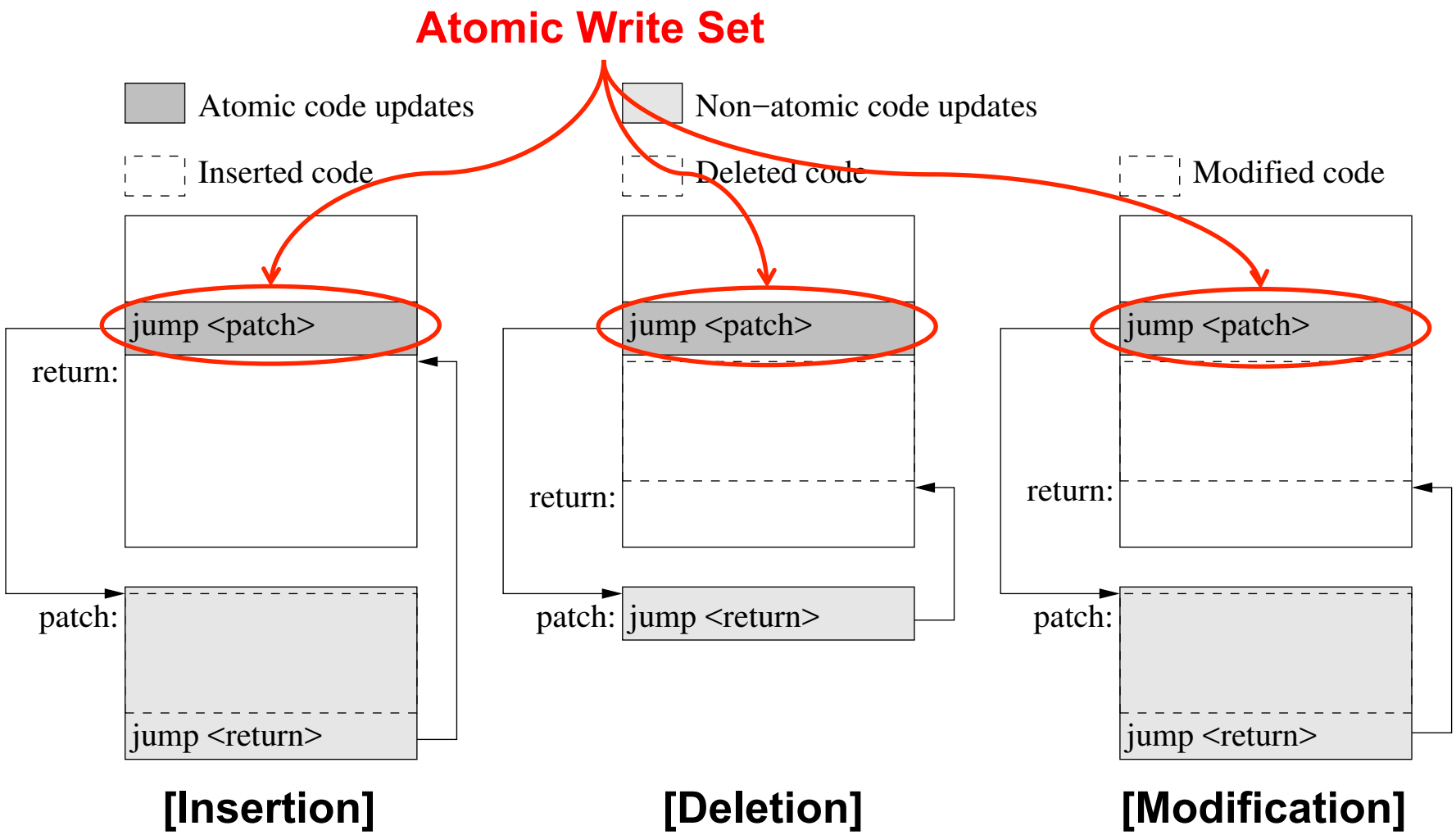


[Insertion] **[Deletion]** **[Modification]**

Code Consistency with Live In-place Patching

- Updating live code while it is executing presents new issues
- What if inconsistent code executes in the middle of update?
- Divide code update into two phases
  - Phase 1: Preparatory phase
    - Write new or modified code to free space in memory without touching the original image
    - Never introduces any inconsistencies
  - Phase 2: Patching phase
    - Write jumps into original image to new / modified code
    - Potential for inconsistencies ➔ patching must be atomic (e.g. by turning off interrupts that may trigger execution)
    - Call set of written jumps the **atomic write set**
    - Size of atomic write set dictates device down time

In-Place Code Patching Revisited

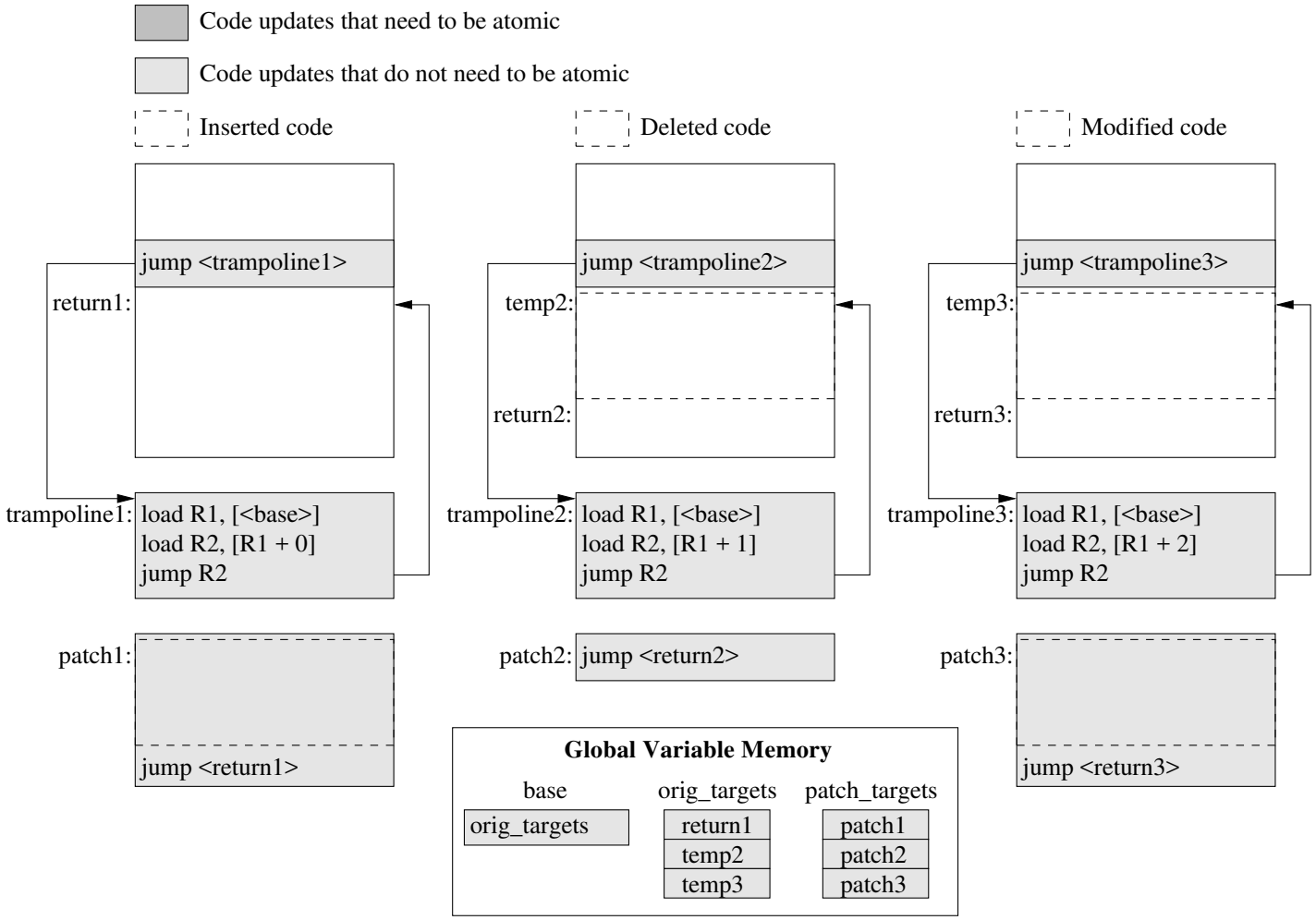


In-Place Code Patching w/ Trampolines

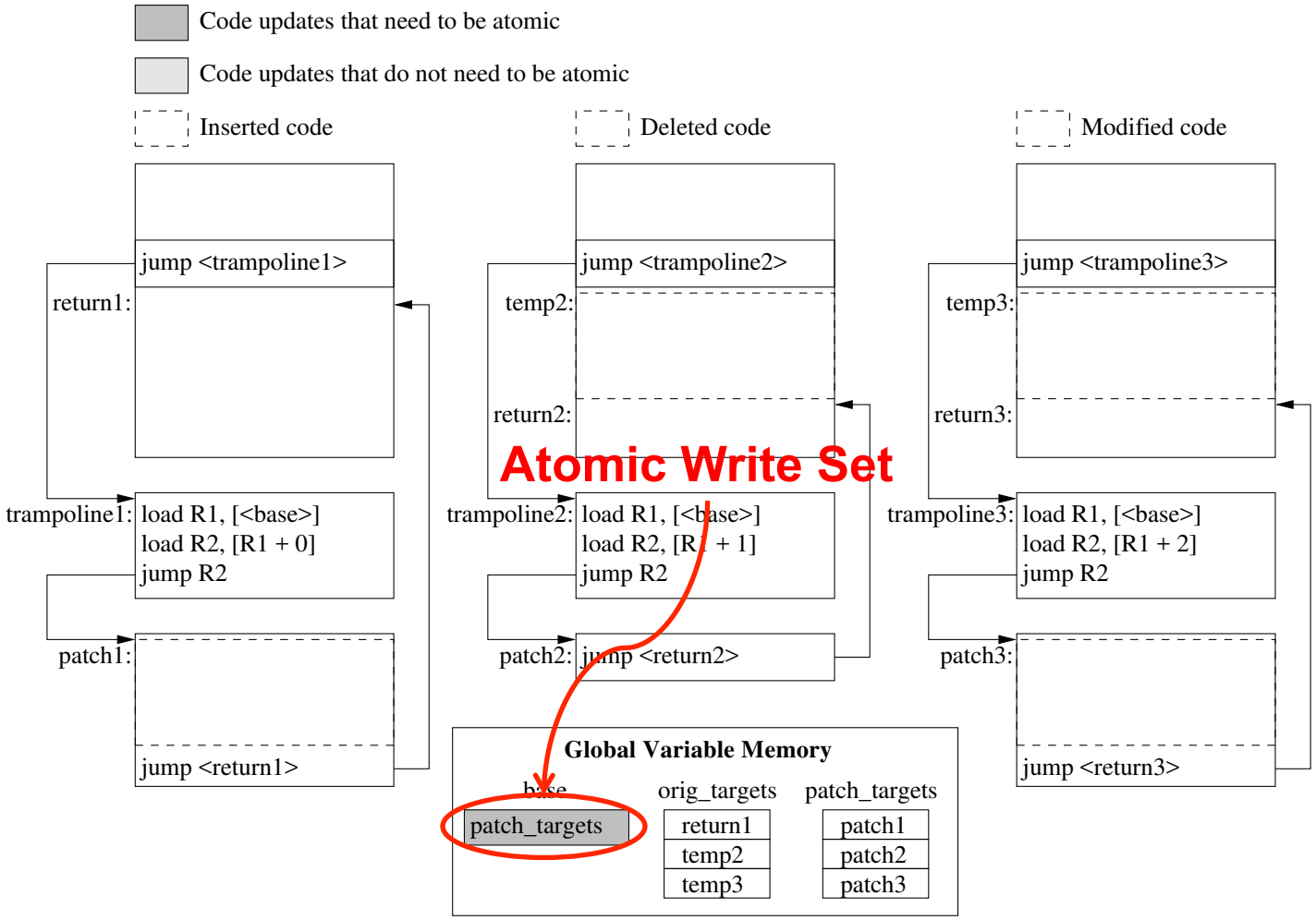
- Still some drawbacks with in-place code patching
  - Still needs updated part of device to be temporarily disabled
  - Also, what if the code update software itself needs to be patched?
    - No way to disable the already running code update software
- Reduce atomic write set to a **single word** using **trampolines**
  - Eliminates all device down time
  - Allows even the code update software to be safely patched
- Trampolines (a.k.a indirect jump vectors)
  - Memory locations holding addresses pointing to pieces of code
  - Used to implement interrupt service routines in OSes

➜ Idea: change functionality instantaneously by switching over to another set of jump vectors atomically

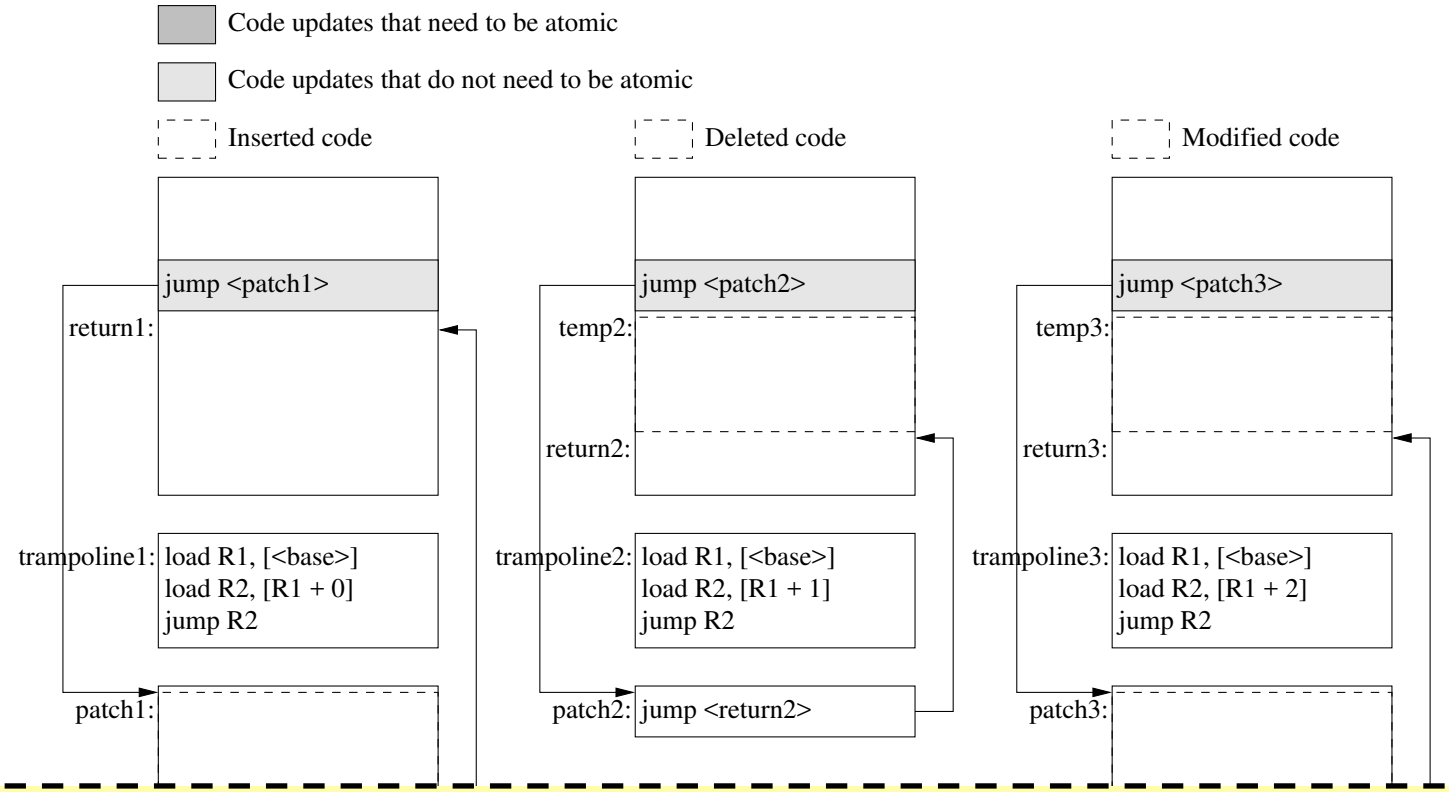
Phase 1: Trampoline Insertion



Phase 2: Trampoline Switching



Phase 3: Trampoline Removal



➜ End result is exactly the same as original in-place patching
➜ Trampolines are recycled to implement next patch

Evaluation Setup

- Tested 4 code update strategies:
  - RSync: Baseline incremental update with diff deltas
  - Zephyr: State-of-art incremental update that minimizes size of delta by mitigating diffs due to “shifts” when inserting or deleting code. Uses function indirection tables.
  - **In-place**: Our baseline in-place update proposal
  - **Trampolines**: In-place update w/ trampolines
- Measured 3 metrics:
  - # of bytes in delta or patch transferred over wireless network
  - # of bytes in atomic write set written to code memory
  - # of bytes in total written to code memory
- Translated metrics to energy numbers for a typical IoT device
  - Energy required to recover from device down time
  - Total energy required for reprogramming

A Typical IoT Device

	
Name	TI-MSP430FR5739
CPU	24 MHz
Execution Memory	1 KB SRAM
Code Memory	**16 KB FRAM** (Ferroelectric)
Write Energy / byte	**4.25 nJ**
Transfer Energy / byte	**104 nJ** (ZL70250 RF Link)
Avg Energy / cycle	330.12 pJ

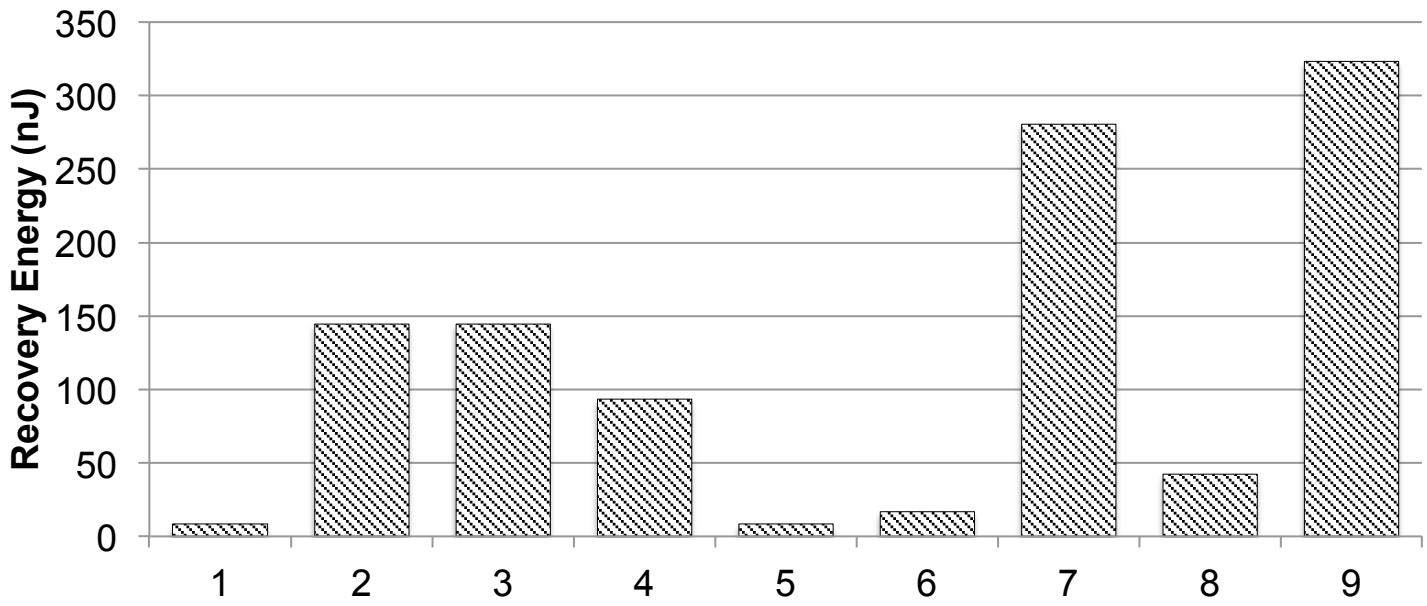
TinyOS Update Cases & Atomic Write Set

Case #	Description	# of Patches
1	In Blink: Change timer0 frequency	1
2	In Oscilloscope : insert one local variable and one use	17
3	n MultihopOscilloscope : insert one local var and use it within a loop	17
4	In RadioCountToLeds : insert one local var and use it inside and outside loop	11
5	In RadioSenseToLeds : Change a + operator to -	1
6	In Blink: remove one function call in NesC code	2
7	In Blink: remove an entire timer	33
8	In RadioCountToLeds : add an else branch	5
9	In Oscilloscope : insert a global var, and use it in three different functions	38

Atomic Write Set for In-place

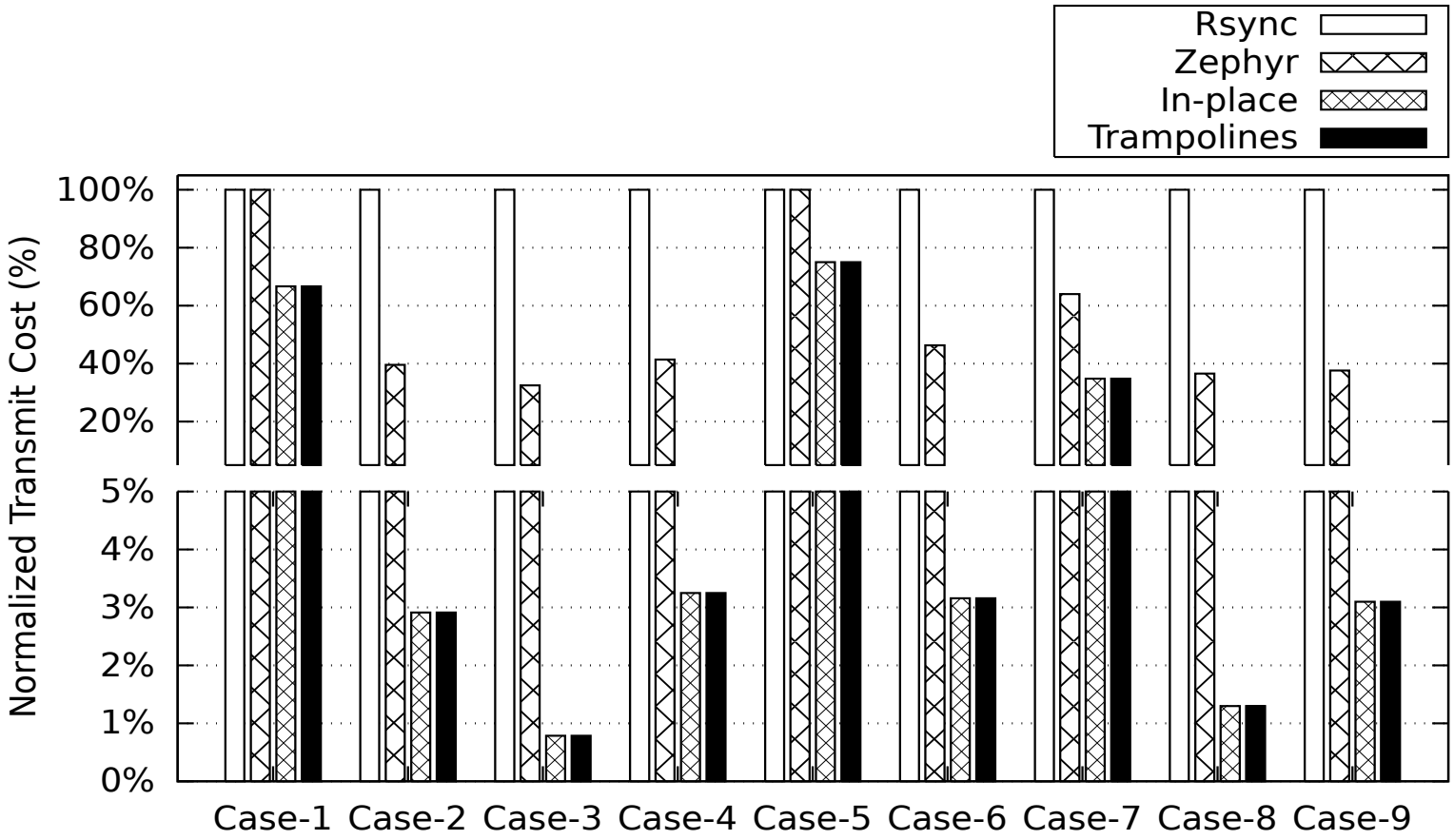
(Atomic Write Set for Trampolines is always 1)

Device Down Time Recovery Energy for In-Place



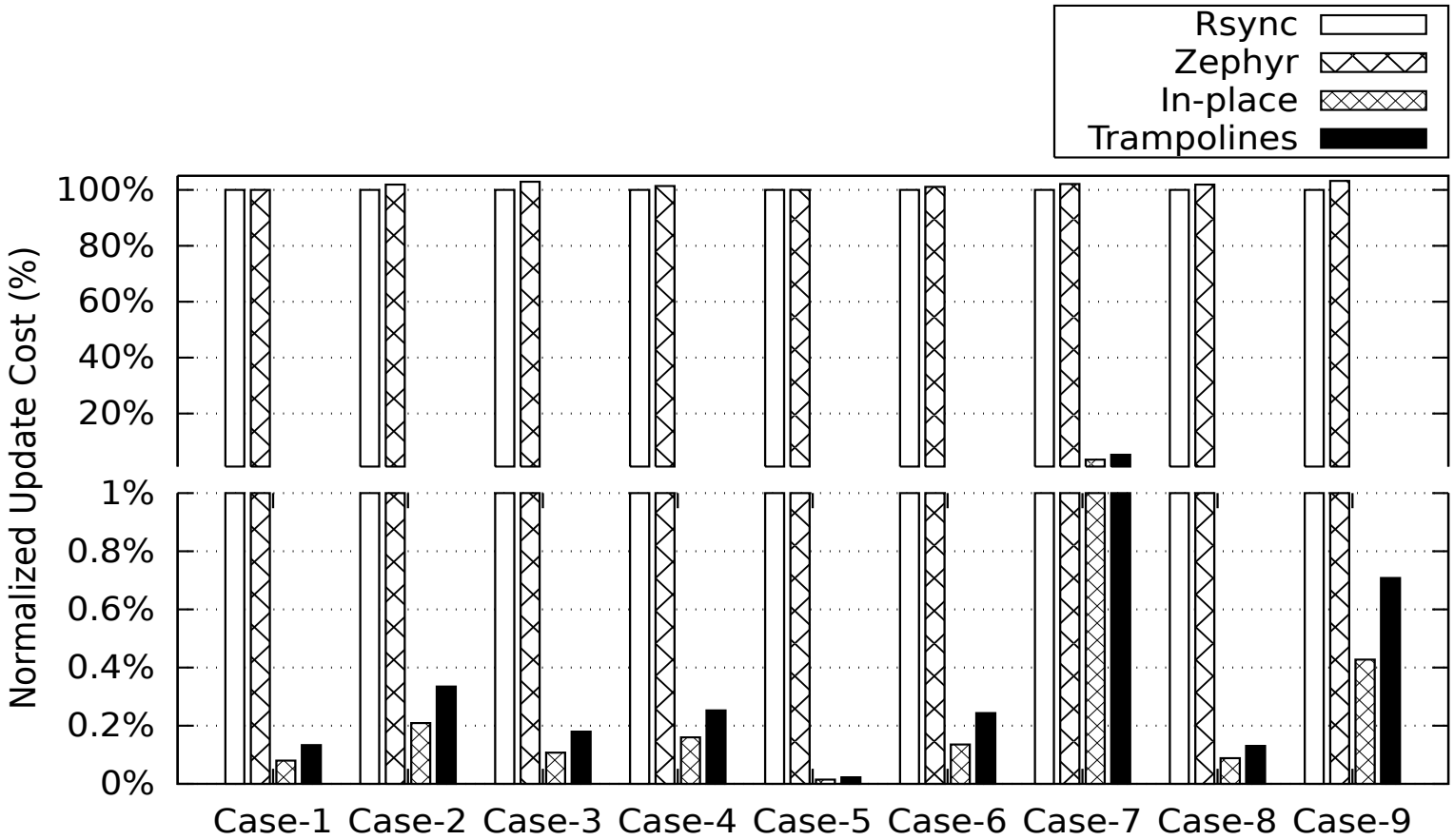
- RSync / Zephyr: 89 uJ for reboot (more than 100 X difference)
- Trampolines: no device down time since atomic write set is 1

Normalized # of Bytes Transferred over Wireless



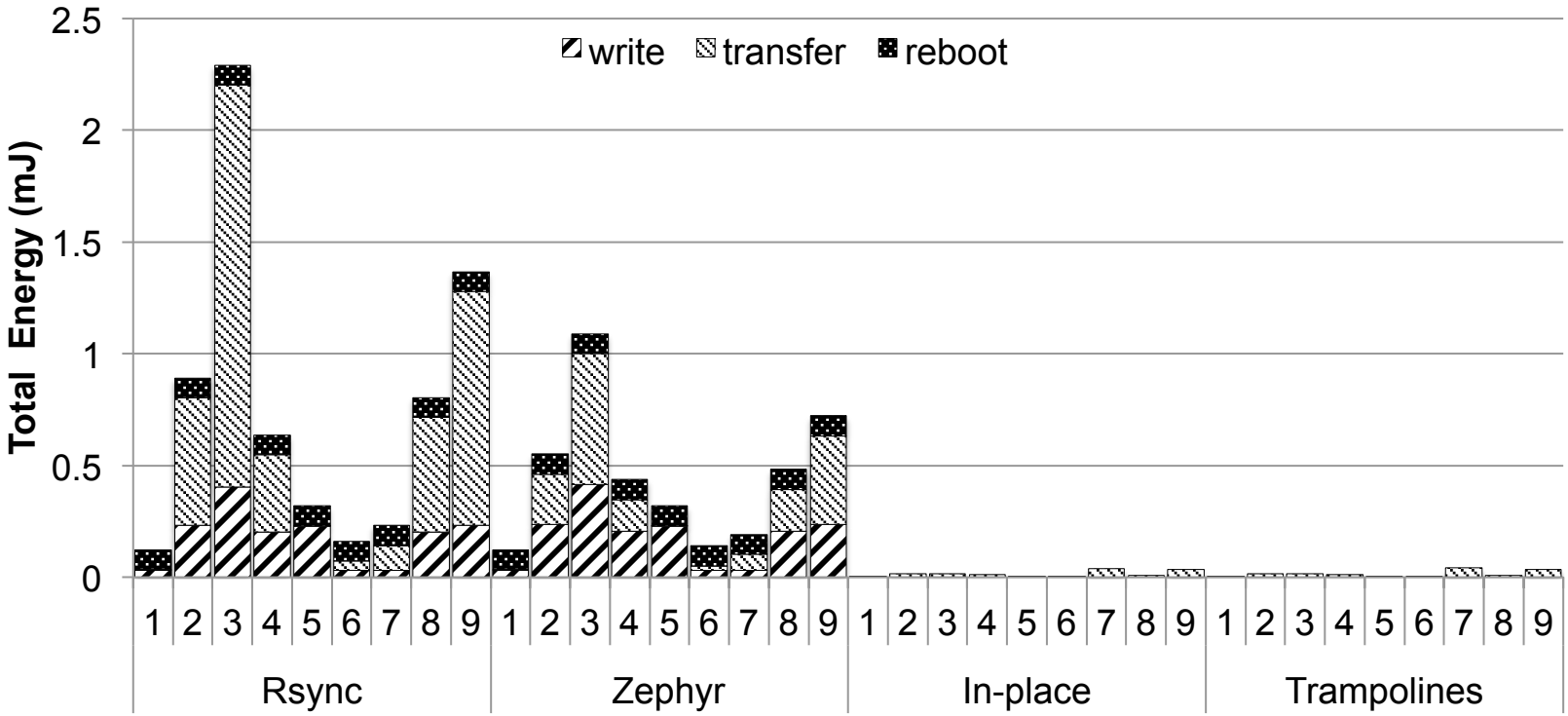
➜ Drastic reduction for in-place due to no code shifting
(Zephyr mitigates shifting but must send function indirection table)

Normalized # of Bytes Written to Code Memory



➜ RSync & Zephyr both must write whole new image
➜ In-place & trampolines need only write patches into original image

Total Reprogramming Energy



- Zephyr: Reduces transfer energy but does not reduce write energy
- In-place & Trampolines: Reduces both transfer and write energy

Runtime Overhead

- Potential for overhead due to jumps to and from patched code
- Our test cases showed negligible overhead
  - At worst 2.7% performance loss for just one case

Summary

- Proposed live code update scheme for IoT devices that can:
  - Improve QoS by eliminating device down time
  - Improve debug / tune / upgrade turn around time by minimizing device reprogramming energy
- Future work
  - How to generate minimal patches from compiler
  - How to deal with fragmentation
    - Frequent patching can leave “holes” in your code
  - How to do updates to data
    - How to modify semantics of variables
    - How to add / delete variables (changing layout)

    ➔ May need to do “data migration”