Hotpatching Embedded Real-time Applications

Patch your program without restarting it

- Arun Krishna AMS (EE19B001)

Overview

Motivation

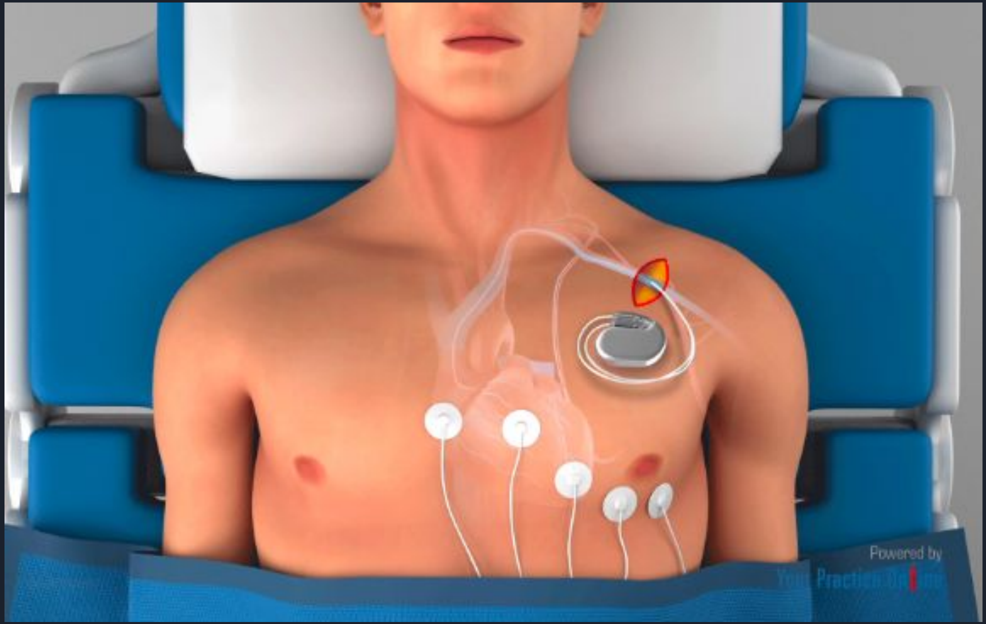
Problem Statement & Challenges

Flash Patch & Breakpoint Unit

Patch Preparation & Switchover

Implementation

Motivation



- Updating vulnerable software - not possible always because of downtime and required reboot
    - Pacemaker, Plant Control IoT systems
- Patch updates introduce downtime
    - Scheduled patching windows
    - Exploit not-yet-updated software even when patches are available
- Hotpatching: update a program when it is being executed without stopping or restarting it

Challenges faced

01 No downtime. Continuous operation required. Cannot be stopped or restarted.

02 Changes in code execution must not violate real-time constraints. Small and predictable delay acceptable. Large and unpredictable are not.

03 Resource and Memory constraints.

Hotpatching Strategies

01

Hotpatching Relocatable Executables
Dynamically linked binaries reference code that is not part of the compiled library and is loaded during run-time. Dynamic linking requires only a symbolic link which can be adjusted during run-time to apply a hotpatch. In Embedded devices, binaries are statically linked. Hence not possible.

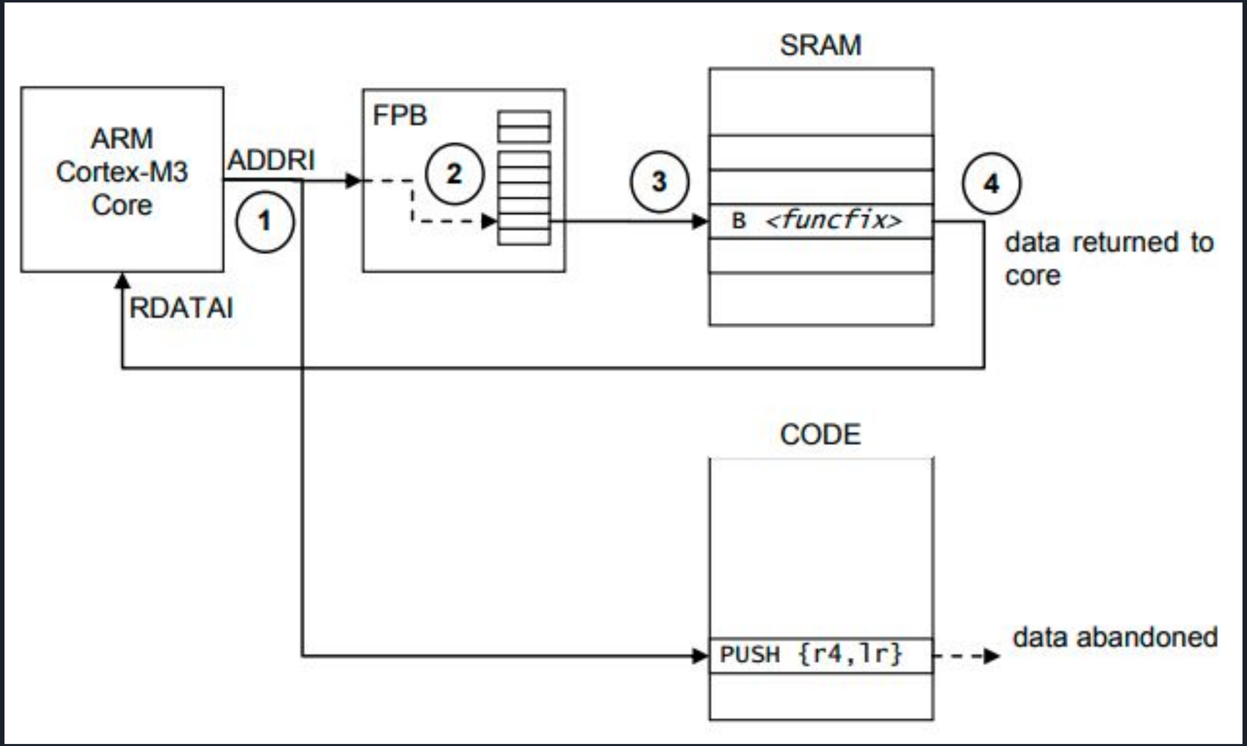
02

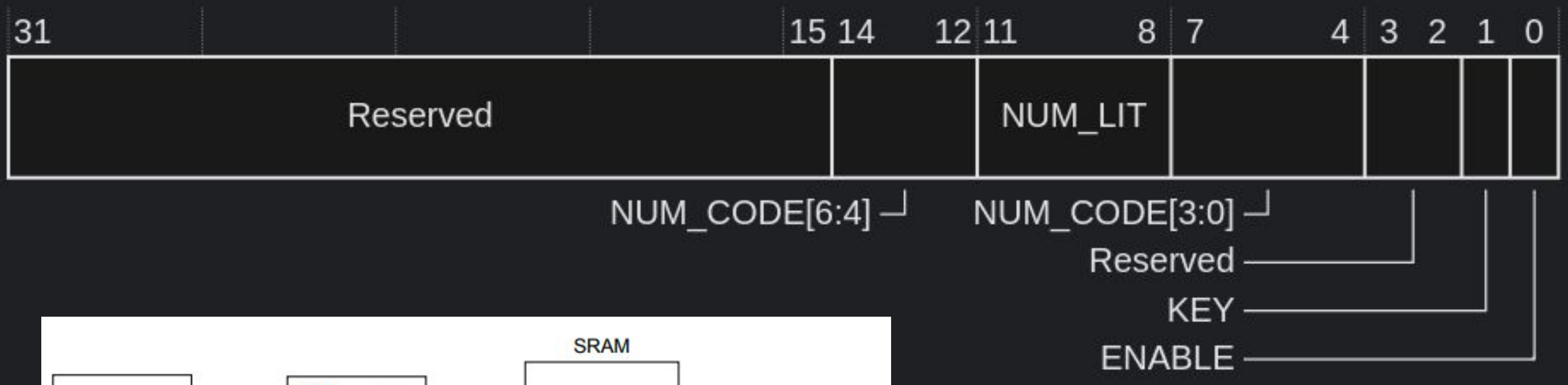
A/B Schemes
Two instances of firmware: One actively running while the other is updated. Switchover later. Problem: Doubling the memory to hold both instances. Dedicated firmware management unit required to update and switch.

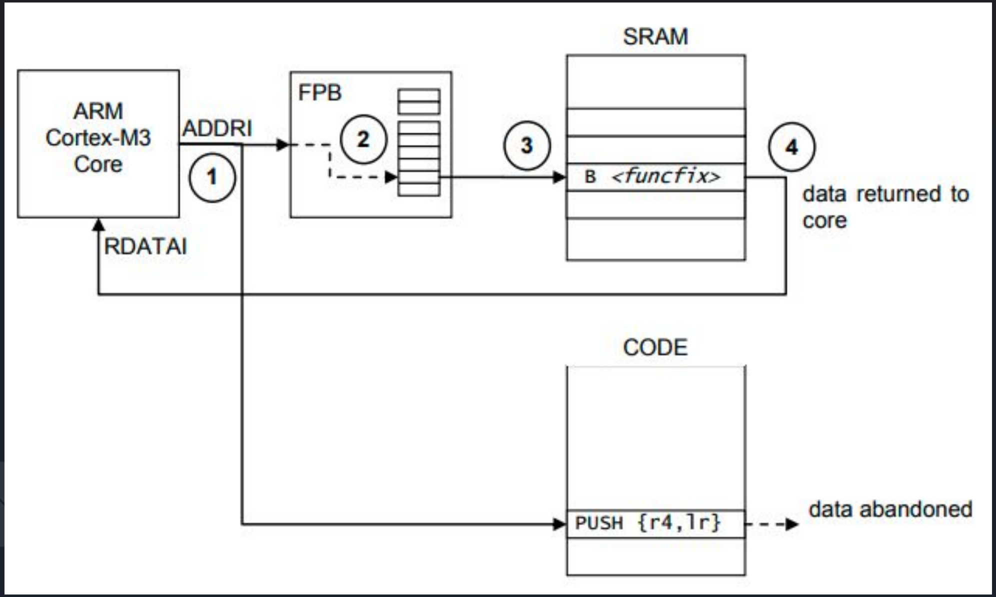
HERA: Key Idea

- Relies on standard debugging unit depending on the architecture
  - Onboard debugging unit with code-remapping unit required
    - temporarily stop the CPU, drop the fetched instruction, and fetch a new instruction
    - leverage this behavior to redirect the control flow to newly deployed hotpatches
  - Example: Flash Patch and Breakpoint in Cortex M3/M4 microprocessors
  - Compatible with existing hardware architectures
- Single trampoline instruction to activate the patch
  - Patch downloaded during IDLE times as per RTOS scheduling
  - Use code-remapping capability to execute a trampoline instead of a processor instruction
  - Trampoline jumps into the updated patch in a single instruction.
  - Execution time deterministic and can satisfy real-time constraints with hard deadlines
- Memory overheads minimal
  - Memory required only to store the patch
  - Couple of memory addresses

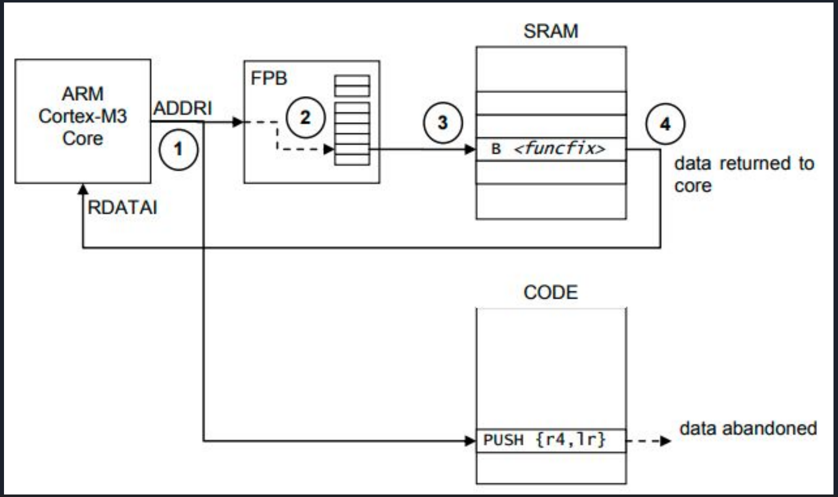
Flash Patch & Breakpoint Unit







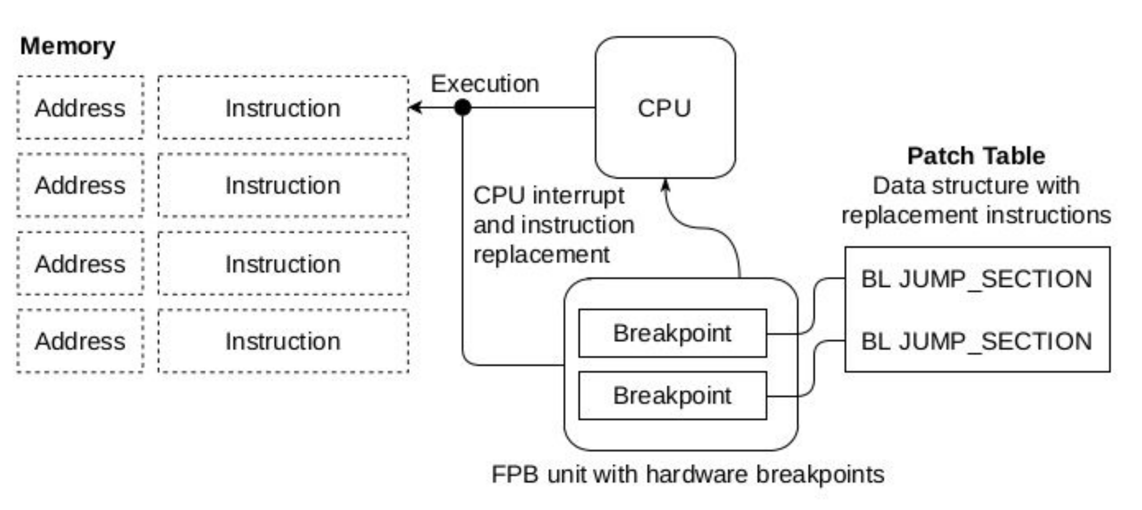
FP_CTRL Register Bit Information



FP_REMAP Register Bit Information

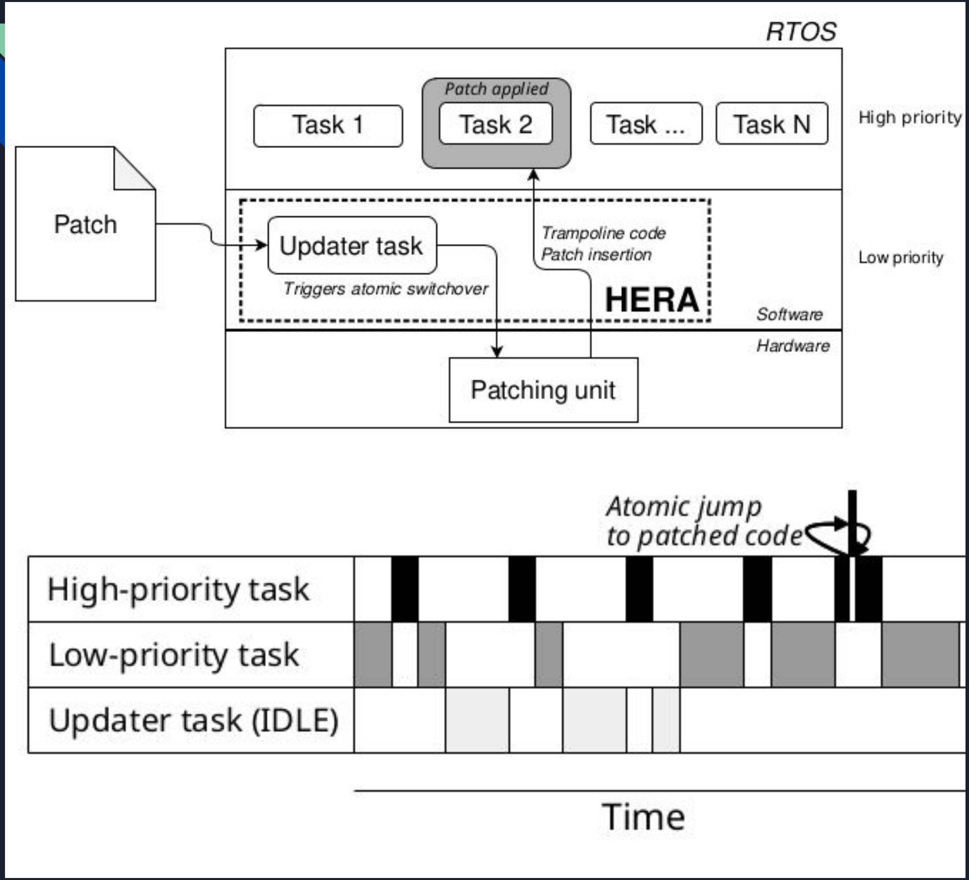
FP_COMPx Register Bit Information

Flash Patch & Breakpoint Unit



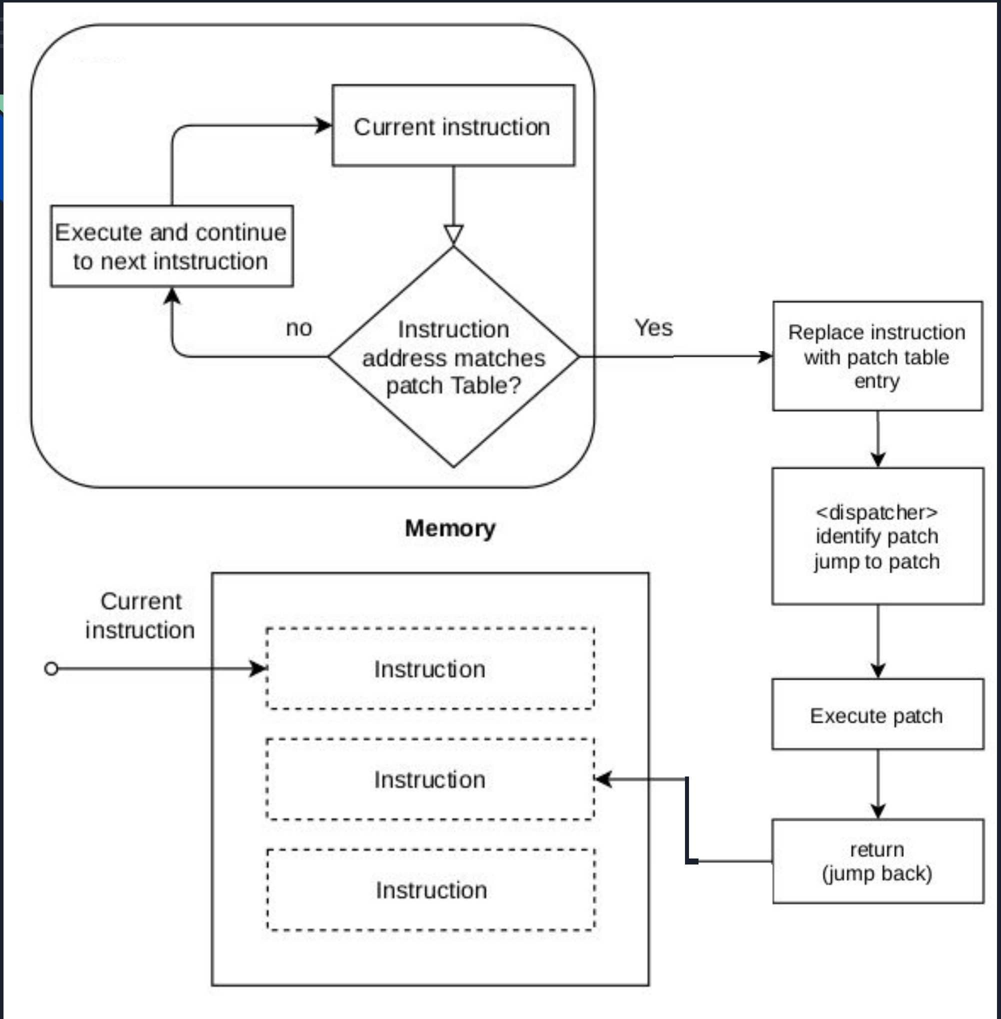
Using FPB,

- Single ‘BL’ function call instruction can be replaced - leaving all other calls to the same function unaffected
- Replace an entire function: remap the first instruction. Note: Returning back is tricky



Patch Preparation

- Receive the patch from an external source,
- Verify the patch by checking its integrity and origin
- Parse the meta-information from the patch format
- Store the patch in selected memory area
- Add a new entry to the dispatcher,
- Configure the patching unit with insertion point and corresponding branching instruction (trampoline configuration)
- Trigger the atomic switchover.



Switchover

- Continuous monitoring of the currently executed instruction
- Instruction replacement on breakpoint hit
- Control-flow redirection to a trampoline
- Trampoline redirection to the appropriate patch
- Patch execution
- Return and continued execution.