

Simulation Techniques For Rapid Software Development and Validation

Mohammadreza Amel Solouki, and Massimo Violante,

Department of Control and Computer Engineering,
Politecnico di Torino, Italy

Motivation

Embedded systems, notably in safety-critical fields like **automotive applications**, are on the rise. Their reliable implementation hinges on factors such as hardware and software design, fault-tolerance mechanisms, programming language choice, and rigorous testing.

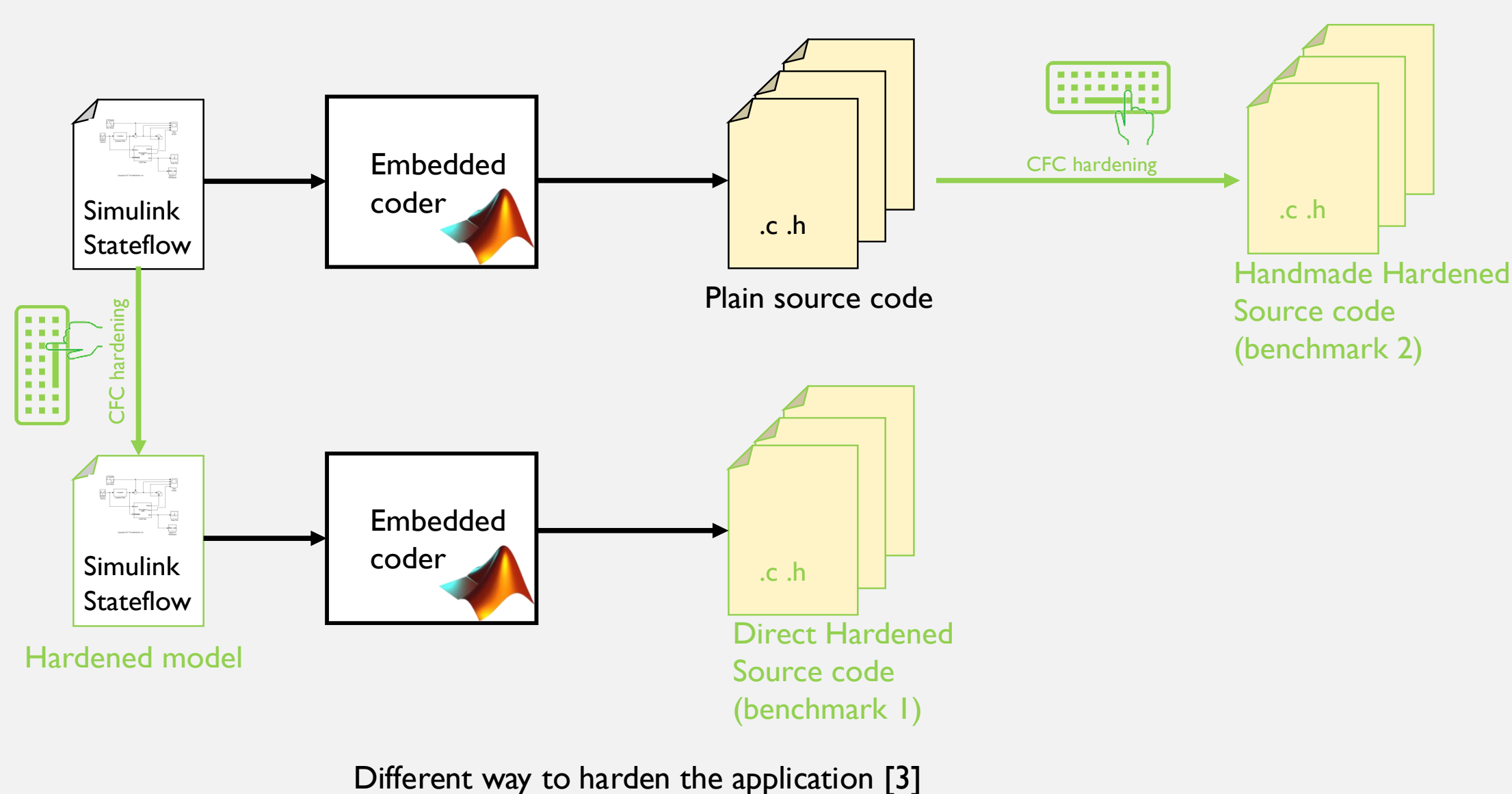
Of particular concern are **Random Hardware Failures (RHF)**, which demand mitigation strategies. We explore the use of **Software-Implemented Hardware Fault Tolerance (SIHFT)** methods, specifically **Control Flow Checking (CFC)** in C language application code.

Motivation stems from the scarcity of guidelines for CFC in **high-level languages**, as most existing proposals focus on lower-level languages like assembly.

Our research aims to address this gap and offer valuable insights into CFC implementation in C.

Proposed Methodology

We adopted the two CFC methods, (1) **YACCA**, and (2) **RACFED**. We chose these methods because they are based on **different** philosophies (bit mask vs. random numbers) and have different detection capabilities (inter-block vs. intra-block).



TESTING AND VALIDATION APPROACH

We have implemented both methods in C and run them on two benchmarks:

- (i) Timeline Scheduler (TS) and (ii) Tank Level Controller (T).

The benchmarks were compiled using the GNU RISC-V tool-chain and simulated at the instruction set level using QEMU. DCs computed for all the failure modes represented by a single-bit stuck-at failure of the Program Counter register.

For a real-world application, it is needed to compute it for each one of the FM's indicated in **part 11 of the ISO26262**.

The analysis is not limited to the effectiveness of the techniques, we also provide results about the overheads each approach which is crucial for real-time applications.

Experimental Results- RAW Classifier output Diagnostic Coverage (DC) for C Approach

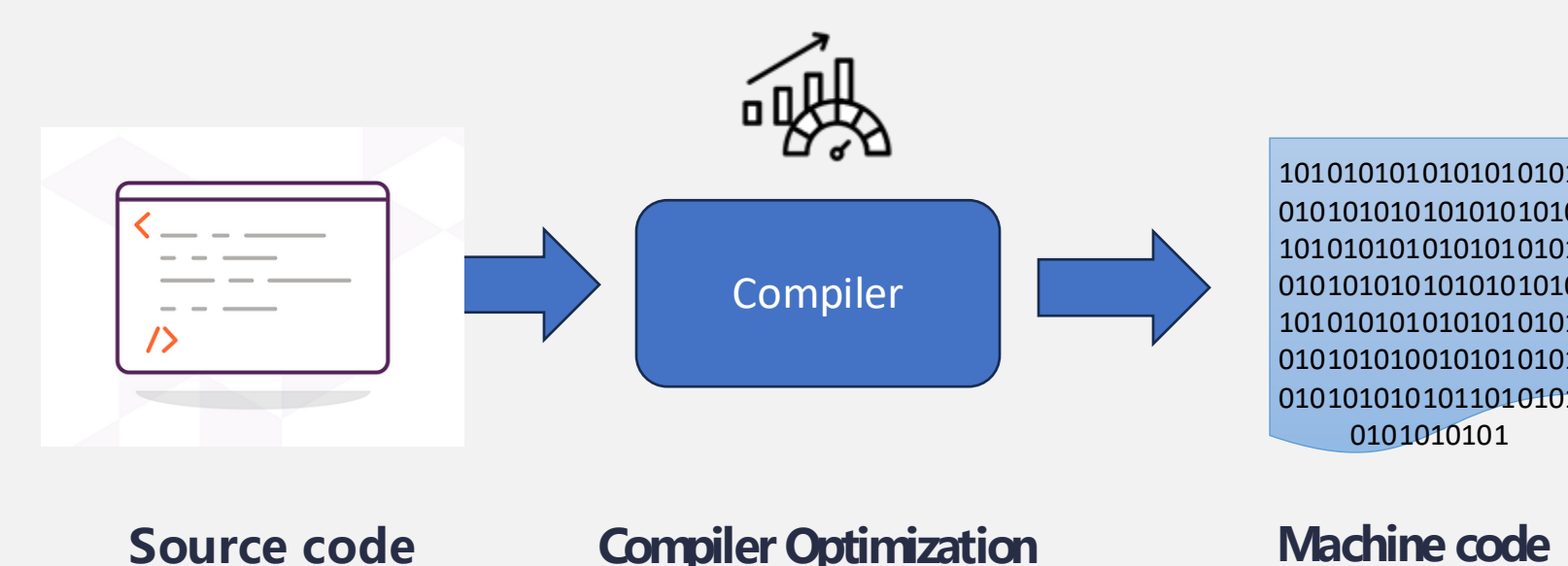
| Classification results | YACCA | | RACFED | |
|--|--------------|-------------|--------------|-------------|
| | TS Benchmark | T benchmark | TS Benchmark | T benchmark |
| Latent after injection | 226 | 883 | 230 | 945 |
| Erratic behavior | 0 | 29 | 0 | 0 |
| Infinite loop or Stuck at some instruction | 408 | 20 | 1,066 | 0 |
| Detected by SW | 253 | 66 | 255 | 52 |
| Detected by HW hardening | 1,063 | 2 | 1,449 | 3 |

| CFC Method | Benchmark | Detected | | Undetected | | |
|------------|-----------|----------|----------|------------|-----------|----------------|
| | | Safe | Detected | Latent | Dangerous | False Positive |
| YACCA | TS | 0.00% | 67.49% | 11.59% | 20.92% | 0.00% |
| YACCA | T | 4.00% | 2.80% | 88.30% | 4.90% | 0.00% |
| RACFED | TS | 0.00% | 56.80% | 7.67% | 35.53% | 0.00% |
| RACFED | T | 5.20% | 0.3% | 94.50% | 0.00% | 0.00% |

Limitation: Compiler Optimization Challenges

Compiling and hardening the assembly code for applications written in high-level languages like C introduces significant overhead, especially in terms of execution time, which is a primary concern for real-time applications.

Our approach harden individual statements in the high-level code before compilation, but the compiler may **remove** or **reorder** the added instructions to optimize the code.



Experimental Results –Diagnostic Coverage (DC) for C with optimization Approach

| CFC Method | Compiler Optimization | Detected | | Undetected | | |
|------------|-----------------------|---------------|----------|------------|-----------|----------------|
| | | Safe Detected | Detected | Latent | Dangerous | False Positive |
| YACCA | 00 | 0.00% | 62.40% | 11.00% | 26.60% | 0.00% |
| YACCA | 01 | 0.00% | 60.70% | 39.30% | 0.00% | 0.00% |
| YACCA | 02 | 0.00% | 56.70% | 43.30% | 26.60% | 0.00% |
| YACCA | 03 | 0.00% | 47.90% | 52.10% | 0.00% | 0.00% |
| RACFED | 00 | 0.00% | 52.90% | 8.60% | 38.50% | 0.00% |
| RACFED | 01 | 0.00% | 77.10% | 22.90% | 0.00% | 0.00% |
| RACFED | 02 | 0.71% | 81.01% | 18.28% | 0.00% | 0.00% |
| RACFED | 03 | 0.00% | 78.28% | 21.72% | 0.00% | 0.00% |

Conclusion and Future Work

My work presented a methodology for evaluating different SIHFT methods, which consists of selecting a set of representative applications and hardening them according to SIHFT methods.

Implementing CFC techniques produces overhead in **text segment size**, **execution time**, etc. We propose to employ **Approximate Computing (AxC)** techniques to **trade off accuracy** for **gains in performance**: power, area, execution time, etc. AxC techniques are employed in applications that are intrinsically tolerant to some accuracy loss.