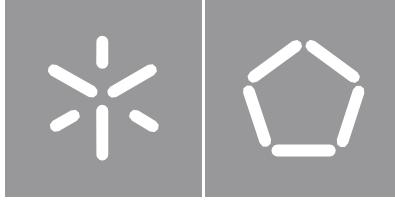




Universidade do Minho
Escola de Engenharia

Nuno Gonçalo Carneiro Teixeira

**Platform-agnostic framework
for fault injection and
response analysis in Mixed-
Criticality Systems**



Universidade do Minho
Escola de Engenharia

Nuno Gonçalo Carneiro Teixeira

**Framework agnóstica à plataforma
para a injeção de falhas e análise
de resposta em Sistemas de
Criticidade Mista**

Dissertação de Mestrado
Mestrado em Engenharia Eletrónica Industrial e
Computadores
Sistemas Embebidos e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Tiago Gomes

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositóriUM da Universidade do Minho.

Agradecimentos

Mais uma etapa da minha vida concluída com muito orgulho. As minhas primeiras palavras de agradecimento vão para as pessoas mais próximas que me acompanharam em todas as fases. Pai e mãe, obrigado pelo sacrifício e todo o apoio que me ofereceram ao longo desta jornada para me verem seguir os meus sonhos. Não há palavras suficientes para descrever o quão grato vos estou. À minha irmã, obrigado pela paciência que tiveste comigo e pela positividade e motivação que me transmitiste nos momentos mais difíceis.

Por outro lado, queria agradecer aos meus colegas de laboratório e em especial um obrigado ao mestre Afonso Santos por toda a ajuda e conhecimento que me disponibilizou para realizar e fortalecer este projeto. Agradeço calorosamente também ao meu orientador Doutor Tiago Gomes por toda a confiança depositada e todos os conselhos partilhados.

Por fim queria agradecer aos meus amigos e colegas que tornaram estes anos académicos incríveis: Carlos, Gonçalo, Fábio, André, Ivan, Vitor, Fato, Italiano, Rocha.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter atuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

*"The greatest glory in living lies not in never falling,
but in rising every time we fall"*

Nelson Mandela

Resumo

A crescente procura por maior desempenho computacional e eficiência de custos tem impulsionado o desenvolvimento de sistemas de criticidade mista e sistemas críticos para a segurança cada vez mais complexos. No entanto, o aumento da complexidade destes sistemas torna-os mais suscetíveis a perturbações e desvios no fluxo de execução. Estes sistemas devem cumprir princípios de fiabilidade e requisitos de segurança funcional que garantam o correto funcionamento do sistema, incluindo a interação adequada com os periféricos, motivando a necessidade de técnicas sistemáticas de validação. Para a sua avaliação, podem ser realizados testes através da introdução intencional de falhas e da observação dos seus efeitos no comportamento do sistema.

A Injeção de Falhas (FI) é uma técnica que consiste na introdução controlada de falhas ao nível do bit em pontos específicos da execução. Estas campanhas de injeção de falhas podem ser realizadas recorrendo a representações virtuais do hardware alvo, permitindo uma avaliação sistemática, repetível e não intrusiva, sem necessidade de hardware. A Injeção de Falhas Baseada em Simulação (SFI) constitui uma abordagem eficaz para a realização de campanhas de injeção de falhas, uma vez que emula de forma próxima o comportamento real do sistema e permite a injeção controlada e a avaliação das respostas do sistema, sem necessidade da produção de hardware.

Esta dissertação propõe uma estrutura de injeção de falhas independente da plataforma e não intrusiva, integrada no emulador QEMU, para a avaliação de mecanismos de tolerância a falhas em sistemas de criticidade mista. O desenho da estrutura facilita a integração de novas arquiteturas de CPU, mantendo simultaneamente um modelo operacional uniforme e independente da arquitetura. Atualmente, suporta arquiteturas ARM e RISC-V. A abordagem proposta explora o mecanismo de tradução binária do QEMU em conjunto com o GNU Debugger (GDB), de forma a permitir a injeção de falhas com elevada precisão ao nível da instrução, proporcionando um controlo preciso tanto sobre a localização da injeção como sobre o instante de ativação durante a execução do programa. Através da manipulação da fase de tradução do QEMU, bem como dos mecanismos utilizados para otimizar a execução em tempo de execução, a estrutura permite a injeção de falhas em qualquer estado da CPU de forma precisa e controlável, garantindo a repetibilidade e produzindo comportamentos defeituosos que se aproximam de cenários reais.

Palavras-chave: Injeção de Falhas por Simulação, confiabilidade, QEMU, GDB

Abstract

Growing demand for computational performance and cost efficiency has driven the development of increasingly complex safety-critical and mixed-criticality systems. However, the growing complexity of these systems makes such systems more susceptible to disturbances and control-flow deviations. These systems must comply with reliability primitives and functional safety requirements that ensure the correct functionality of the system, including proper interaction with peripherals, motivating the need for systematic validation techniques. To evaluate it, tests can be performed by intentionally introducing faults and observing their effects on system behaviour.

Fault Injection (FI) is a technique that consists of introducing controlled bit-level faults at specific execution points. These fault injection campaigns can be carried out using virtual representations of the target hardware, allowing systematic, repeatable, and non-intrusive evaluation without the need for hardware. Simulation-Based Fault Injection (SFI) is an effective approach for conducting fault injection campaigns, as it closely emulates real system behaviour and enables controlled fault injection and assessment of system responses without the need for physical hardware production.

This dissertation proposes a platform-agnostic and non-intrusive fault injection framework integrated into the QEMU emulator for evaluating fault-tolerance mechanisms in mixed-criticality systems. The framework design facilitates the integration of new CPU architectures while maintaining a uniform and architecture-independent operational model. Currently, it supports both ARM and RISC-V architectures. The proposed approach exploits QEMU's binary translation mechanism in combination with the GNU Debugger (GDB) to enable fault-injection precision at the instruction level, providing fine-grained control over both the injection location and the triggering instant during program execution. By manipulating the QEMU translation phase as well as mechanisms used to optimize the execution at runtime, the framework enables fault injection at any CPU state in a precise and controllable manner, ensuring repeatability and producing faulty behaviors that closely resemble real-world scenarios.

Keywords: Simulation-Based Fault Injection, reliability, QEMU, GDB

Contents

1	Introduction	1
1.1	Motivation and Objectives	2
1.2	Dissertation structure	2
2	State of Art	3
2.1	Mixed Criticality Systems	3
2.1.1	Dependability	5
2.1.2	Threats (Fault, error, and failure)	6
2.2	Fault Injection	7
2.3	Emulators	9
2.3.1	QEMU	11
2.4	Debuggers	18
2.5	Related work	20
3	Framework Design	24
3.1	Fault Library	25
3.2	Fault types	27
3.3	Golden and Faulty Runs Components	30
3.4	Framework Initialization	33
3.5	Breakpoint management	34
3.6	Triggering	47
3.7	Data collector	64
4	Evaluation	68
4.1	Validation	68
4.1.1	Precise Triggering Tests	68
4.1.2	Injection Tests	71

4.1.3	TB break mechanism	75
4.1.4	TB flush mechanism	78
4.1.5	TB cache flush	81
4.1.6	Access Injection	84
4.2	Performance Evaluation	88
4.2.1	Single versus multiple instructions TBs	88
4.2.2	Optimizations	89
4.2.3	Triggering overhead	91
5	Conclusion	94
5.1	Future Work	95

List of Figures

The dependability and security tree.	5
Possible evolution of a fault in a vulnerable system.	7
Advantages and disadvantages of the different fault injection techniques.	9
Target code translation and host code generation steps.	12
Translation Block generation process.	15
CPU execute loop function flowchart.	16
Translation Block chaining procedure.	17
Single-threaded TCG (A) and multi-threaded TCG (B) differences and context switching.	19
ESIFT Overall Framework.	22
Framework Overview.	24
Faulty Library parsing flowchart.	26
Generic fault example in a CPU register.	28
CPUArchState struct definition for Aarch64 and RISCv architectures, with the Register base and PC address being located in different positions.	29
Fault example in a memory position.	29
Fault injection flowchart.	31
Golden Run and Faulty Run overview.	32
Collector and injector functions in golden and faulty run.	33
Handler callback flowcharts	35
Problem with injecting the error in the translation moment without the proper mechanisms.	38
Different approaches to inject a fault precisely: A) QEMU configured for one instruction per TB; B) Dividing the TB in question into several TBs with one instruction; C) Breaking the TB so that the moment for injection, stay on top of the second divided part.	39
TB breaking in Translator Loop flowchart.	40
Break TB and TB intersection state machine.	41
TB cache saving mechanism.	42

CPU Execute Loop flowchart.	43
TB re-translation behavior.	44
TB flush mechanism introduced in CPU Execute Loop.	44
Breakpoint placement for injection control.	45
Example of flow context switch between cores.	46
TB cache flush according to context switch.	46
TB cache flush callback flowchart.	47
Difference between Intermittent and Transient triggering.	48
Fault lifetime management function flowchart.	49
Triggering types breakpoint placements in <i>cpu_exec_loop</i>	50
Procedure of triggering and TB breaking for a defined PC address.	51
Approach of TB flushing to perform triggering in a repetitive PC address.	52
PC triggering callback flowchart for the host code breakpoint placing approach.	53
Corner case in PC triggering for the host code breakpoint placing approach.	54
PC triggering optimization overview.	55
Triggering backend for 25 instructions count.	56
TB flush conditions between application and hypervisor level.	57
Instruction triggering callback flowchart.	58
Access trigger type example.	59
Access trigger mechanism flowchart.	60
Callbacks used in the timer trigger type.	61
catch_instruction_cb enabler mechanism.	62
catch_instruction_cb callback enabling for different values of ERROR TUNING.	63
Peripheral triggering callback flowchart.	64
Collecting logs in golden and faulty run example.	65
Info collect function flowchart.	66
Log comparator function flowchart.	67
Description of the steps for triggering precision evaluation.	69
Fault injection in a register causing bad output on the terminal.	73
Output on the terminal after the register fault injection during the UART printing.	73
Fault in a memory causing a faulty output on the terminal.	74
Result in the terminal after memory fault injection.	74

Behavior of register injection during the translation phase in QEMU standard.	75
Behavior of register injection during the translation phase with TB breaking mechanism enabled. .	76
TB tracing in Baseline versus with Breaking Mechanism on.	77
Executed code and their respective TBs.	78
QEMU logs result with active TB flush mechanism.	79
Log output for TB flushing and brekaing.	80
Validation of full cache flush mechanism.	83
Register accesses inside a TB.	86
Register accesses for TBs with one instruction only.	87
Execution time comparison between the flush mechanism and the breakpoint PC triggering approaches.	90
Comparison of the number of breakpoint stops between optimized and non-optimized faulty runs in the worst-case scenario for PC and time triggering.	91
Performance time for each trigger in the same moment for different benchmarks.	92

List of Tables

Comparison of most used software emulators.	10
QEMU-based fault injection methods characteristics.	23
XML tags used to configure fault injection parameters.	25
Details of breakpoints with associated callbacks used for each triggering type.	50
Performance Results	81
Number of trigger possibilities with framework mechanisms for different algorithms.	81
Output results for injection in x7 accessing.	87
Comparison of execution times and TBs for AArch64 and RISC-V architectures with QEMU configured to generate single and multi-instruction TBs.	88

List of Listings

CPUARMState struct code	13
AArch64 target instruction	13
Micro-operations relative to target instruction	14
Generated code to host machine	14
Fault class	27
IntersectPoint class	36
Breaking TB function	41
CPU and Memory Log	66
PC triggering configuration	69
Tracing output of PC triggering moments in golden run	70
Instruction Triggering configuration	70
Tracing output of Instruction triggering moments in golden run	71
Simple print code on terminal	72
Simple addition operation between registers	75
Log outputs after each run (A) <i>golden_out_finish.log</i> and (B) <i>faulty_out_finish.log</i>	77
Code for printing the <i>var</i> variable in the terminal	82
Fibonnaci code example with C-ASM integration	85

Acronyms

ABI	Application Binary Interface
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARM	Advanced RISC Machine
BIT	Built-In Test
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DBT	Dynamic Binary Translation
DUE	Detected Unrecoverable Error
EEA	Execution Environment Abstraction
EFI	Emulation-based Fault Injection
ESIFT	Emulation-based Software Implemented Fault Tolerance
FAIL	Fault Injection Framework
FI	Fault Injection
GDB	GNU Debugger
HFI	Hardware-based Fault Injection
HLE	High-Level Emulation
IoT	Internet of Things
ISA	Instruction Set Architecture
ISO	International Organization for Standardization
LLE	Low-Level Emulation
MCU	Microcontroller Unit

MMIO Memory-Mapped I/O

MMU Memory Management Unit

MTTCG Multi-Threaded Tiny Code Generator

OS Operating System

PC Program Counter

QEFI QEMU Fault Injector

QEMU Quick Emulator

RISC-V Reduced Instruction Set Computer V

RTOS Real-Time Operating System

SBST Software-Based Self-Test

SDC Silent Data Corruption

SER Soft Error Rate

SFI Simulation-based Fault Injection

SUT System Under Test

SWFI Software-based Fault Injection

TB Translation Block

TCG Tiny Code Generator

vCPU Virtual CPU

VHDL Very High-Speed Integrated Circuit Hardware Description Language

XML eXtensible Markup Language

1 Introduction

The world is experiencing rapid technological advancement, with a growing reliance on modern systems that are becoming increasingly complex. Relying on electrical components, such as transistors that are even more reduced in size, higher clock frequencies, and lower operating voltages, these are exposed to disturbances that may alter their expected operation [1]. In safety-critical domains such as automotive and aerospace, several requirements have to be followed in a way that external factors do not affect the system flow or output, maintaining the application deadlines and correctness, in adverse conditions.

Safety-critical systems must not only provide fast response times to certain events but also ensure system integrity, safeguarding users from potentially dangerous situations. To ensure that the system is adapted to the reliability conditions required to operate in its environment, it must follow a standard for that domain. In the automotive domain, for example, the ISO 26262, where functional safety requirements for electrical and electronic systems are defined, should be complied with.

A proper method to validate a system's resilience is by conducting a series of tests to identify vulnerabilities at the hardware or software level. That is possible by deliberately submitting the system under test (SUT) in conditions that can cause faults in real-life scenarios. Fault injection allows the possibility to analyze the fault effect in the system by showing how those can propagate or evolve into errors or failures. With this information, the system complies with the standards required for safety-critical operation, and quantifying the system's reliability [2].

Fault injection is a method where the SUT is forced to unfavorable scenarios close to reality, assessing the system's ability to detect, locate, and/or mitigate fault occurrences [3]. There are several approaches for fault injection, such as by hardware (HFI), by software (SWFI), simulating (SBI), or emulating (EFI).

In this domain, emulating the system becomes a good alternative as it enables repeated testing without the need for physical hardware, while offering good real-time observability and controllability [4]. Emulators are software that simulate a full system (hardware and software), allowing portability and modularity. Considering this, the SFI approach also gives results and behaviors most closely to real-life scenarios, giving more trustworthy results in comparison with other methods. QEMU is an open-source tool that supports emulation across various processor architectures, to enable the injection of soft errors with high repeatability and low overhead across a wider range of systems [3] [5]. To observe the system's fault tolerance under various scenarios defined by the standards, it is necessary to have a comprehensive set of faults with different characteristics, ensuring that the system complies with these requirements. With this in mind, the present dissertation proposes a platform-agnostic fault injection framework integrated into QEMU, fulfilling all the needs for mixed-criticality systems evaluation.

1.1 Motivation and Objectives

Fault tolerance testing has been a central focus of research, leading to the development of a wide range of evaluation methods designed to assess system robustness and reliability [3]. Such evaluation methods, including fault injection techniques, support the identification of effective approaches to prevent anomalies and improve the efficiency of injection approaches.

The objectives of this dissertation are the following: (i) Analyze and investigate non-intrusive fault injection techniques; (ii) Design and implement a fault injection mechanism capable of injecting controlled and reproducible faults into the SUT; (iii) Assess the impact of injected faults on the SUT, and integrate the developed mechanism with the framework for simulation and testing; (iv) Evaluate the performance of the proposed method through system testing.

1.2 Dissertation structure

This document is divided into five chapters: in Chapter 1, a brief introduction is provided, giving the contextualization of this dissertation, the motivation and the objectives to be achieved.

Chapter 2 describes different key concepts and topics that will serve as the foundation of this work. By introducing the concept of a critical system, it lists the characteristics required for its proper operation. The chapter then explores various fault injection methods, justifying the approach adopted in this work. It also elaborates on the different types of fault models, detailing the expected outcomes that can be obtained for each. The section concludes by explaining the approach used to inject faults in the binary translation process in QEMU and the reason for using this platform emulator to deploy the SUT.

Chapter 3 presents the design and implementation of the proposed framework. It introduces the library with the definition of the available fault models, the different fault targets, and how the fault injection is performed, and the different necessary mechanisms that support the injection and data collection.

Chapter 4 presents tests conducted to evaluate the framework's efficacy. The chapter is divided into two main sections. The first subsection presents the validation results, demonstrating the functioning of fault injection with examples of use and justifying the use of different approaches adopted. The second subsection evaluates the performance results of the framework in different conditions and configurations for different benchmarks.

Finally, Chapter 5 presents the results and conclusions and suggests future work to extend the framework and address its limitations.

2 State of Art

This chapter seeks to provide an overview of the current technological environment and outline the state-of-the-art in the relevant technologies and theoretical foundations that shaped it. The goal is to clarify the rationale behind the dissertation's conception.

This dissertation focuses on fault injection techniques to evaluate fault tolerance mechanisms in MCS. A brief explanation of the characteristics, uses, and requirements of this type of system is presented, including dependability needs, giving its definition and the categories evaluated for dimensioning it, like means, attributes, and threats.

Different fault injection techniques are discussed, followed by an introduction to system emulation and different emulators with their characteristics. QEMU is then selected as the target emulation platform due to its architectural support, which makes it particularly suitable for precise and controlled fault injection. Its internal architecture is therefore analyzed in detail to support the proposed approach. Additionally, the debugger used to control and modify the emulator execution is identified and described, ensuring non-intrusive implementation of the proposed extensions. Finally, a review of related work is provided to contextualize the proposed approach within existing research.

2.1 Mixed Criticality Systems

A Mixed Criticality System (MCS) can be defined as an integrated suite of hardware, operating system and middleware services, and application software with two or more distinct criticality levels [6] [7]. These levels define how the applications and activities on the systems should be organized. These constraints are required to ensure system security, reliability, and quality of service (QoS), allowing the maximization of hardware resource utilization while reducing the cost of extra computational power and energy. The primary concern with the implementation of an MCS is the separation of tasks, which guarantees the isolation of the different components, preventing interference between them.

An MCS task can be defined with parameters such as activation period, deadline, worst-case computation/execution time, and criticality level [8]. Tasks with high criticality level have more conservative (i.e., larger) worst-case execution time (WCET) estimates due to stricter assurance and verification requirements. The deadline times may vary depending on whether it is necessary to conclude a safety-critical job faster or delay it to guarantee the QoS of the system.

This concept has been increasingly adopted in domains such as automotive and avionics, replacing the old-fashioned way of associating each Electronic Control Unit (ECU) with only one subsystem [9]. This change was fed mostly by the increasing number of ECUs per vehicle and dependencies between them [10]. The transition to Industry 4.0 and the evolution of advanced driver assistance systems (ADAS) were also reasons to search for these types of systems, since they enable the consolidation of applications with different levels of criticality on a shared hardware platform, improving hardware utilization and reducing costs [11].

A versatile solution for these types of systems is the use of hypervisors. Hypervisor is a software layer that abstracts hardware, allowing multiple independent operating systems to run their applications simultaneously on a single hardware platform [12], [9], [11], [13].

Before the introduction of these systems in the market or for use, they must follow some specific requirements for hardware and software to justify their reliability. In the case of automotive, the *ISO 26262* is a safety standard that manages safety in road vehicles, stipulating a critical role in system behavior analysis and recommending simulation at all development phases. It also recommends the use of a Hazard Analysis and Risk Assessment (HARA) method to identify hazardous events in the system.

The main goals of this safety standard are:

- Providing an automotive safety lifecycle (management, development, production, operation, service, decommissioning), adapting each task to proper work.
- Covering functional safety aspects of the entire development process (including such activities as requirements specification, design, implementation, integration, verification, validation, and configuration).
- Providing requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety is being achieved.

An example of an MCS is an autonomous car system [14] that presents a group with different operations that are differentiated in high, medium, and low criticality. Since tasks like anti-lock braking system (ABS) and electric steering wheel control can lead to disasters for both the system and the user if they fail or do not meet their deadlines, they are considered highly critical. Then, a task such as medium-range obstacle detection is also critical, but does not demand the same level of precision in meeting deadlines as the previously mentioned tasks. Finally, a task like temperature monitoring is of low criticality, as it poses no risk even if it completely fails to operate.

2.1.1 Dependability

Dependability refers to a system's ability to perform a correct operation in adverse situations, taking into account a group of metrics that promote trust in the service delivered to the user [15]. The dependability characterization of a system, as shown in Figure 2.1.1, can branch on three main categories: means, attributes, and threats.

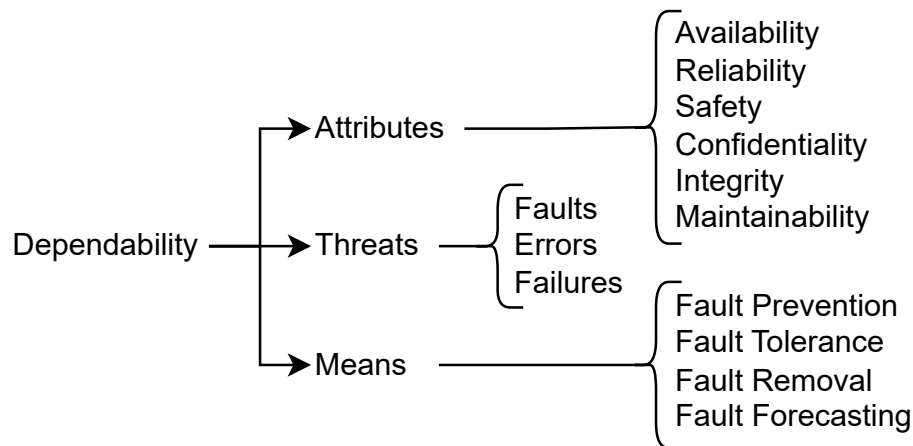


Figure 2.1.1: The dependability and security tree (adapted from [16]).

Means represent the methods used to achieve dependability. These can be grouped into four categories: (i) **fault prevention**, which is the ability to anticipate the occurrence or introduction of faults; (ii) **fault tolerance**, which is the avoidance of service failures in the presence of fault, through error detection or system recovery; (iii) **fault removal**, which is capacity to reduce the number and severity of faults; (iv) **fault forecasting**, which is the estimation of the present number, the future incidence and the likely consequences of faults. While fault tolerance (ii) and removal (iii) focus on a trusted service delivery, fault prevention (i) and forecasting (iv) aim to increase confidence in the system's ability to operate as intended. This confidence is achieved by verifying that the system's functionality, dependability, and security requirements are well defined and realistic.

The attributes are desired properties that characterize a system's dependability. These can be distributed in [15]:

- **Reliability:** the ability to provide a continuous correct service.
- **Availability:** readiness for correct service.
- **Safety:** prevention against catastrophic scenarios from the user and environment perspectives.

- **Confidentiality:** preservation of information against unauthorized entities.
- **Integrity:** absence of improper system alterations.
- **Maintainability:** ability to undergo repairs after a malfunction.

Each metric is established based on the system application and purpose. In the case of mixed-critical systems, as mentioned in 2.1, these systems are applied in scenarios where, if the output is incorrect or delayed, it can expose the user's integrity at risk, leading to life loss in the worst case. Therefore, MCSs place a stronger emphasis on safety and reliability.

2.1.2 Threats (Fault, error, and failure)

Threats are the risks that can compromise the system's dependability by affecting its correct operations, internal state, or service delivery. In dependable systems, undesired behavior typically follows a causal chain composed of faults, errors, and failures.

A fault can be described as an unexpected event, often external, that impacts a component and can cause it to behave incorrectly [16]. Faults can be categorized according to the following parameters: (i) phase of creation or occurrence, if a fault is introduced during the system development phase, for instance due to poor design, or operational if it occurs during the usage period, caused, for example, by hardware degradation; (ii) system boundaries, if the fault is internal, caused by system components, or external, originating from phenomena such as radiation or electromagnetic interference [17]; (iii) domain, specifies whether the fault occurs at the hardware or software level; (iv) phenomenological cause, indicating whether the fault originates from natural or human-made influences; (v) intent, distinguishing between accidental and deliberate causes; and (vi) persistence, which can be classified as transient, if the fault occurs only in a single cycle without causing physical damage, or permanent, if the faulty behavior persists in the system, typically due to physical hardware damage. A transient fault can also be classified as intermittent when its manifestation is not systematically reproducible.

Faults can occur in various system components, such as data logic, registers, and memory. Temporary faults caused by voltage variations, magnetic fields, or radiation can lead to errors in the CPU, affecting components like data registers, address registers, data-fetching units, control registers, and arithmetic logic units (ALUs). If not detected or mitigated, such faults can cause errors. The error phase is defined as a deviation from the expected execution flow, which may stem from a change in a variable or the hardware state. If a fault results in an error, the service delivery may be compromised, thus leading to an incorrect response that will generate a failure.

A failure occurs when a system is unable to perform its intended function due to deviations in its operation, resulting in incorrect service [18]. Both error and failure can be classified at the domain level as content errors/failures if the delivered information is wrong, timing errors/failures if the information is delivered too early or too late, and halt errors/failures if the service is halted, i.e., the system service is no longer perceptible to the users. The diagram on Figure 2.1.2 shows how a fault on a component could evolve to a failure on the target system. A fault inside a component can cause the system to enter a wrong internal state (error). If that problem isn't handled, it leads to a failure, where the system behaves incorrectly.

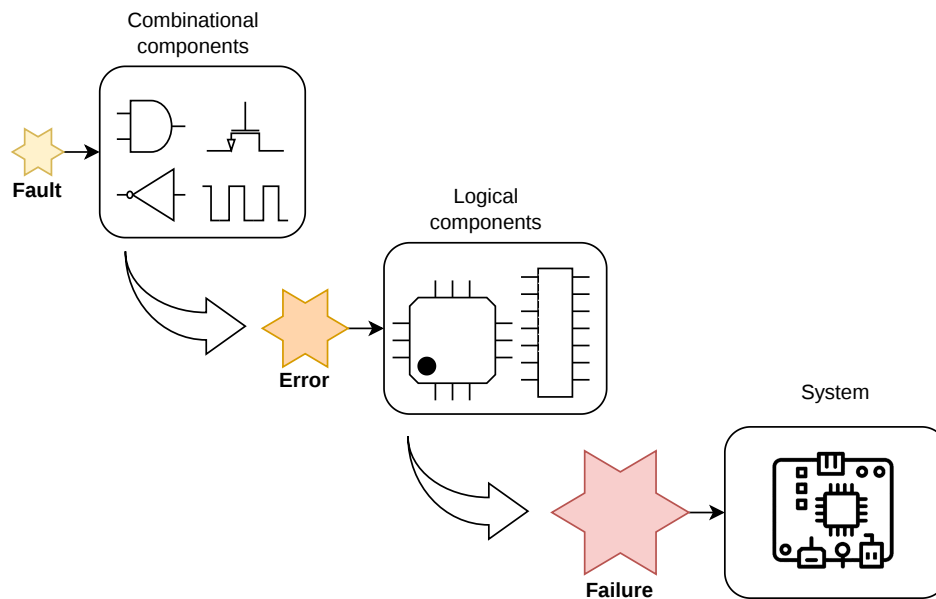


Figure 2.1.2: Possible evolution of a fault in a vulnerable system.

Following a fault, system behavior in the presence of soft errors (SER) can be classified into four categories [19]: Silent Data Corruption (SDC), characterized by incorrect results are produced without error detection; Detected Unrecoverable Errors (DUE), corresponding to situations in which errors are detected but cannot be corrected, typically triggering fail-stop mechanisms; Hang, occurring when execution fails to complete within expected time bounds; and Masked (or No Effect), where the fault has no observable impact on system behavior.

2.2 Fault Injection

There are several fault injection techniques that can be applied to evaluate the fault tolerance introduced in the system under test (SUT). Fault injection is a technique used to evaluate the SUT'S

dependability by intentionally introducing faults. Through this process, it is possible to analyze the effects of realistic fault scenarios under different workloads and operating conditions, and assess the effectiveness of error detection and recovery mechanisms [20]. There are several approaches to perform it. Below are enumerated the different techniques [3, 4, 17]:

- **Hardware-based Fault injection (HFI):** It is achieved at the physical level by disrupting the hardware directly. It can be classified into two categories: (i) fault injection without contact, such as heavy ion radiation, electromagnetic interference, or by using a laser; (ii) with contact, for instance, through methods that include injecting sags on the power rails of the hardware (power supply) or altering the values of the circuit's pins.
- **Software-based Fault injection (SWFI):** A technique that works by altering the software running on the target system, introducing faults either at compile time, by modifying the contents of registers and memory.
- **Simulation-based Fault injection (SFI):** It is based on the construction of a simulation model of the SUT, where there is information about the processor and other components. The simulation models are developed using a hardware description language such as the very high-speed integrated circuit hardware description language (VHDL).
- **Emulation-based Fault injection (EFI):** In this case, the technique combines hardware and software, allowing the designer to study the actual behavior of the system, taking into account real-time interactions.

Figure 2.2.1 presents the main advantages and drawbacks of each approach. From all potential approaches, the HFI brings more accuracy to the injection results. However, it lacks controllability and observability. It can be expensive, whereas some tests can cause permanent damage to the hardware. SWFI gives more controllability and allows for better observability, but the injection can only affect the area accessible to software. Then EFI and SFI are characterized as having the advantage of providing injection at different abstract levels, even as they test different architectures and system models. EFI techniques give higher observability and controllability. For these reasons, an emulation-based approach is adopted in this work [17].

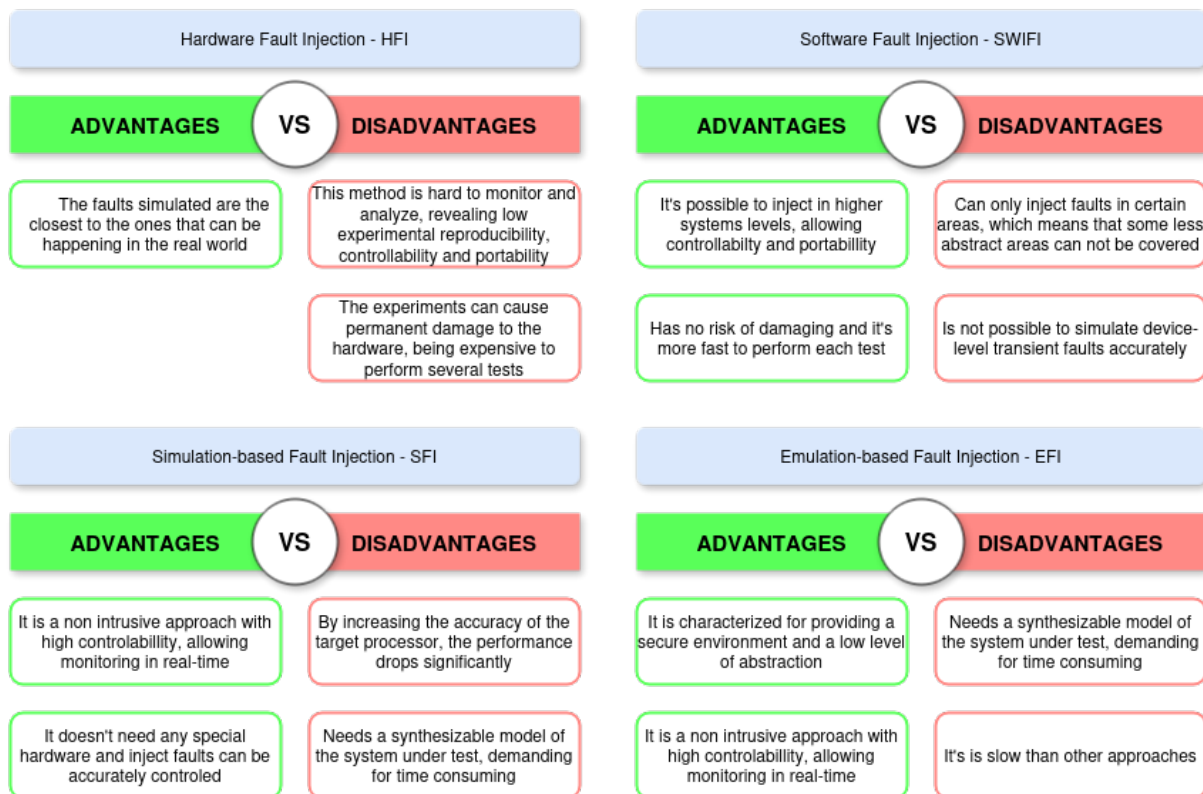


Figure 2.2.1: Advantages and disadvantages of the different fault injection techniques.

Emulators [21,22] are virtual platforms that enable a host system to replicate the behavior of a specified target system by the translation of compiled software from the target on a different host ABI. The emulator constructs each system component and subsequently interconnects them, similarly to how hardware components are linked. The achievable accuracy and execution speed depend on whether low-level or high-level emulation techniques are employed.

basic block accuracy: emulators that use binary translation, to optimize the TB in more efficient versions;

(v) high level emulation accuracy: deals with replacing larger blocks of code that have specific high-level functionality, often related to tasks like audio or video output.

Emulators can also be characterized according to how memory management is handled by the guest system. In High-Level Emulation (HLE), host resources are mapped to the guest system, prioritizing performance over accuracy, while in Low-Level Emulation (LLE), the Memory Management Unit (MMU) is fully emulated. That includes virtual address translation, which provides higher accuracy at the cost of performance. The most commonly used open-source emulators are OVPsim, Unicorn, Gem5, and QEMU. Table 2.1 presents a comparison between different nuances and characteristics.

Table 2.1 Comparison of most used software emulators (adapted from [24]).

SW Emulator	Accuracy Level	Emulation type	Speed	Platforms Supported
OVPsim (Imperas)	Instruction	HLE	●○○○○	●●●○○
QEMU-DBT	Instruction	HLE	●●●●○	●●●●●
Gem5	Cycle	LLE	●○○○○	●○○○○
Unicorn	Instruction	HLE	●●●○○	●●●○○

Open Virtual Platform (OVP) is a virtual platform released by Imperas [21], characterized for giving a set of APIs and a fast Just-In-Time code morphing simulator engine (OPVsim). The APIs provided allow the creation of multiprocessor platforms with processors, memories, and arbitrary topologies, as well as methods for acceleration in the JIT compiler. Finally, it offers the instantiation of behavioral hardware models, while also enabling the modeling of interfaces for external connections, such as bus and network ports. Despite everything, this will not be such a reliable approach, since its configuration is more complex and slower compared to other alternatives.

Then the Unicorn [25] is a lightweight QEMU-based emulator, where some unused functionalities are removed, keeping the Dynamic Binary Translation (DBT), allowing a faster instruction process. However, it is not updated according to QEMU's newest version and does not provide a full hardware representation, being limited to emulating certain abstractions of CPU and memory. [26]

Gem5 [27] is a virtual platform originating from the M5 simulator [28], which was designed for full-system, focusing on peripherals and I/O devices, and GEMS [29], a simulator dedicated to memory systems, i. e., the group of software and hardware mechanisms responsible for saving, delivering, and reading data. Like other approaches, GEM5 supports a large variety of ISAs and several simulation capabilities for different CPU models. It presents a cycle-accurate simulation, which benefits for more

detailed analysis in lower levels of hardware description language models. However, this makes the simulation slower, making it worse for checking extensive benchmarks, in contrast to instruction-accurate models [24, 25, 30, 31].

QEMU is an emulator that provides a user-friendly configuration interface, allowing flexible modification of various aspects of the emulated system components as well as the emulation of complete peripherals. In addition to that, is supported by a large community that contributes regularly with updates. It also offers good execution speed while supporting a wide range of architectures.

2.3.1 QEMU

QEMU (Quick Emulator) is a versatile, open-source emulator that runs on several host platforms and supports the emulation of multiple target architectures, like x86, x86-64, ARM, SPARC, PowerPC, and MIPS. It is supported by a dynamic binary translator that converts target binary code into host binary code, enabling the compatibility of software compiled for one architecture to another, while applying several optimization techniques.

QEMU supports two operation modes: user mode emulation and full system emulation. In user mode, QEMU launches processes compiled for one CPU on another, translating system calls on the fly, working only at the operating system (OS) level. In full system mode, QEMU replicates a complete system (virtual machine), including peripherals and components, such as memories, registers, and I/O devices. This complete emulation is made possible through the use of static data structures that encode architecture-specific parameters, combined with a generic CPU state that maintains the dynamic execution context and is responsible for enforcing the defined behavior during runtime on the host system. Additionally, it offers support for debugging and program tracing, allowing the capture of system execution flow, which facilitates repeatable testing and system analysis.

To manage multiple emulated hardware devices, QEMU relies on a system bus responsible for interconnecting components and dispatching function calls to the appropriate device models. Each device is implemented as an object class, and its instantiation automatically attaches it to the system bus. The CPU emulation is a process composed of three stages: code translation, code generation, and code execution. Between these stages, several optimization mechanisms are applied to improve execution efficiency and overall performance. The translation stage follows the principles of dynamic binary translation.

a) Dynamic Binary Translation

Dynamic binary translation (DBT) [5, 32, 33] is a technique used to translate guest ABI instructions into equivalent host ABI instructions at runtime. The different process phases are illustrated in Figure 2.3.1.

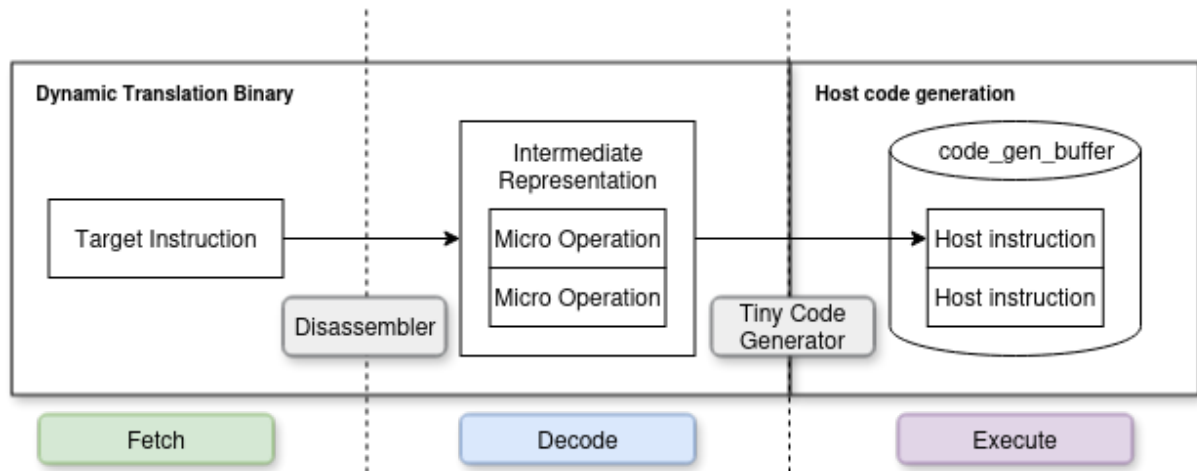


Figure 2.3.1: Target code translation and host code generation steps.

The translation mechanism is triggered when the CPU's program counter (PC) points to a target code that was not translated earlier. In the first stage, the instruction is fetched from the target binary. At the second stage (decode phase), the instruction passes through a mechanism where the operands and target CPU elements involved in this specific operation are read and emitted as an intermediate representation. This representation simplifies complex guest instructions into a set of micro-operations. With this, an abstraction layer between the guest and host architectures is created, enabling the translation of guest instructions into micro-operations. These micro-operations allow for a generic design, facilitating an easier way to add new target and host architectures.

After this, the TCG concatenates the micro-operations, generating the code for the specific host system. The code is first stored and subsequently executed within the QEMU static execution environment, which emulates the target system's specifications. This allows the code to run as if it were on the actual hardware. In the Listing 2.1, the struct that emulates the target system's specifications in the case of an ARM architecture is illustrated.

Listing 2.1: CPUARMState struct code

```
typedef struct CPUArchState {
    uint32_t regs[16];
    uint64_t xregs[32];
    uint64_t pc;
    uint32_t pstate;
    bool aarch64;
    bool thumb;
    ...
    uint32_t usr_regs[5];
    uint32_t fiq_regs[5];
    ...
    uint32_t CF;
    uint32_t VF;
    uint32_t NF;
    uint32_t ZF;
    uint32_t QF;
    uint32_t GE;
    ...
    uint64_t elr_el[4];
    uint64_t sp_el[4];
    ...
} CPUARMState;
```

A simple example of an instruction translation from an AArch64 target to x86 host code is established below. Consider the case of the target instruction presented in Listing 2.2.

Listing 2.2: AArch64 target instruction

```
mov    w0, #0x5
```

After decoding, the micro-operations presented in Listing 2.3 provide a generic, architecture-independent representation of the original instruction. In this case, a MOV operation is performed to load the immediate value into the target register, followed by an increment of the PC to fetch the next instruction.

Listing 2.3: Micro-operations relative to target instruction

```
mov_i64      x0,$0x5
add_i64      pc,pc,$0x4
```

Then this representation is translated into host-specific code, where each micro-operation is lowered to one or more native instructions executed on the host processor, as shown in Listing 2.4.

Listing 2.4: Generated code to host machine

```
movq    $0x5,0x40(%rbp)
mov     0x140(%rbp),%rbx
add     $0x4,%rbx
```

The `$rbp` register points to the CPU's target structure. The values `0x40` and `0x140` are the offsets of the base address of the register array and the program counter, respectively.

In this phase, QEMU introduces a method of grouping sequential instructions in Translation Blocks (TBs) [5, 34].

b) Translation Blocks

The TB generation is an optimization that reduces the number of transitions between the guest code translation and host code execution, improving execution performance. In each TB, a sequence of guest instructions is translated into corresponding host instructions and stored for reuse during subsequent executions. This process is explained in Figure 2.3.2. During the translation loop, instructions are grouped into blocks until a branch condition or jump instruction is encountered. After this loop, the TCG generates host code for the group of TB micro-operations. This generated code is then stored in a buffer and executed. To prevent a block from becoming too large in the absence of branch or jump conditions, a limit is imposed on the maximum number of target instructions a TB can contain. The translated TB is also stored in a cache for possible future use.

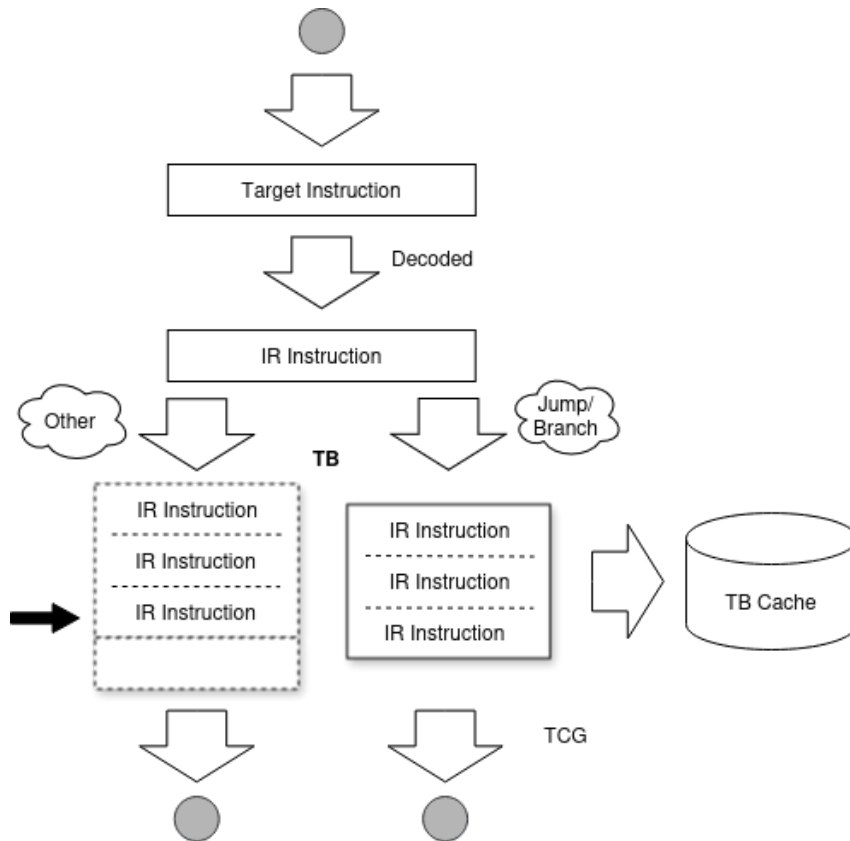


Figure 2.3.2: Translation Block generation process.

Between the TB generation and code execution, QEMU employs different optimizations. Those are managed in a central execution loop function (`cpu_exec_loop`). This loop is invoked whenever execution reaches a code region not present in the TB cache. In Figure 2.3.3, the overall flowchart is demonstrated. It has every step mentioned from lookup to TB execution.

Each TB stores the PC of the first instruction, the execution flags, and a pointer to the translated host code. Whenever a branch or jump occurs in the target system, a lookup is performed to check if a TB associated with the current program counter already exists in the cache. If a matching TB is found, the translation phase is bypassed, and the execution resumes from the cached host code through the pointer stored in the TB. Otherwise, a new TB is generated.

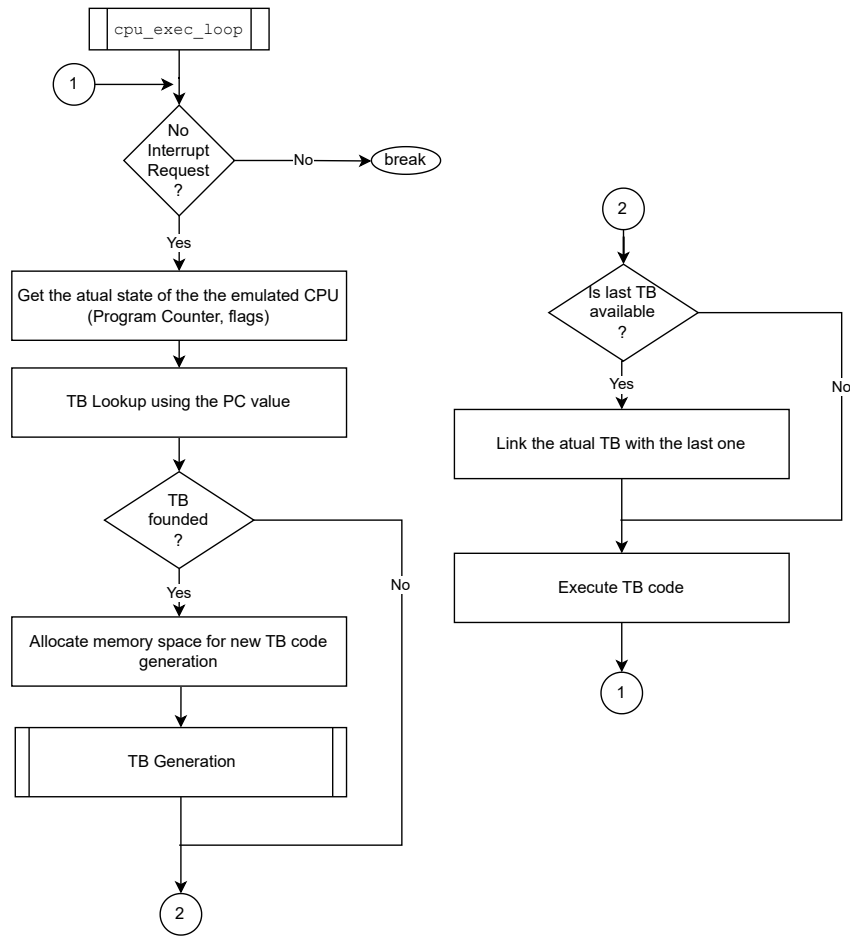


Figure 2.3.3: CPU execute loop function flowchart.

Each TB stores the PC of the first instruction, the execution flags, and a pointer to the translated host code. Whenever a branch or jump occurs in the target system, a lookup is performed to check if a TB associated with the current program counter already exists in the cache. If a matching TB is found, the translation phase is bypassed, and the execution resumes from the cached host code through the pointer stored in the TB. Otherwise, a new TB is generated.

Another mechanism that QEMU assumes is TB chaining. This technique links TBs according to the control flow of the target program. After generation, the TB is linked to the last executed TB. In the TB generation, there is a flow that passes for defining two important standard parts: the prologue, which initializes the TB, and the epilogue, which contains the operations for exiting the TB and jumping back to the emulator control [35]. Linking the TBs prevents the epilogue code from being executed after a TB is linked to another, allowing control to return to the emulator only when execution reaches a region that has not yet been translated.

In Figure 2.3.4, an example of a loop code containing four blocks can be observed. The TBs shown with dashed outlines represent blocks that have not yet been translated. Before each execution phase, a

linking step verifies whether the previous TB can be linked to the current one. Once chaining is completed, execution proceeds through the TBs without returning to the emulator control loop, thereby reducing performance overhead.

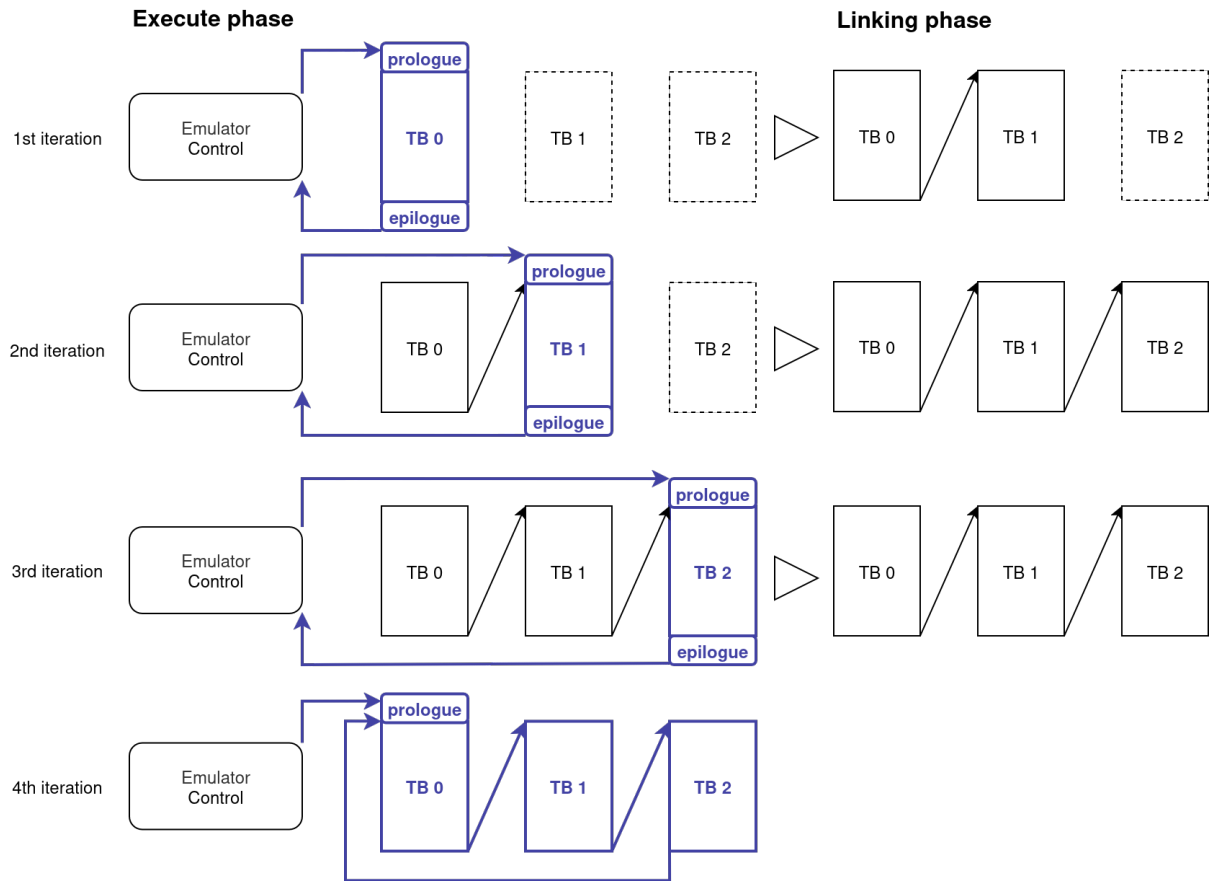


Figure 2.3.4: Translation Block chaining procedure.

Although these optimizations significantly improve execution performance, the grouping of multiple target instructions into TBs, together with caching and chaining mechanisms, makes it difficult to precisely associate a specific execution moment with an individual target instruction. This lack of fine-grained temporal control poses a challenge for fault injection techniques that require deterministic and repeatable injection timing.

c) Additional features

Following multiple updates and community contributions, QEMU has incorporated several features that enhance controllability and execution efficiency, while also improving the observability of system behavior under different operational conditions. One such feature is the support for TCG plugins. These can support different functionalities such as tracing the behavior of and gathering information about TBs,

including translation and execution times, memory and hardware accesses, and the type and number of instructions executed in the timeline. This functionality is accomplished by extending the intermediate representation with micro-operations.

QEMU offers three clock sources: Realtime, Host, and Virtual clocks. The Realtime clock runs even when the emulation is stopped. The Host clock also continues to run while emulation is stopped, but it is sensitive to any changes to the host clock. In contrast, the Virtual clock only advances when the emulation is actively running, offering a high level of precision. The *icount* feature configures the virtual clock to increment with a ratio of 2^N nanoseconds per guest instruction, being N a user-specified parameter.

Additionally, several compatible TCG accelerators enable more efficient use of system resources. A good example is the multi-thread tiny code generator (MTTCG), which is an expansion of TCG that utilizes a thread per virtual CPU (vCPU). The original system-mode TCG implementation was single-threaded and dealt with multiple CPUs with simple round-robin scheduling. The MTTCG allows the translation and generation of host code for each vCPU independently. It also allows running the different emulated cores in parallel, contrary to the conventional TCG. In addition, after a TB has been generated for one vCPU, that translated version can be shared and reused by other vCPUs that need to run the same code, avoiding the need to translate it again [24, 36]. However, doesn't support some plugins neither the *icount* parameter. In Figure 2.3.5, the difference between the conventional single-thread TCG (A) and the MTTCG (B) is demonstrated. It is possible to analyze that the first two threads are standard for the two approaches. While in the single-threaded TCG, all vCPUs are handled by a single thread, in MTTCG, each vCPU is assigned its own thread.

Although MTTCG offers better performance than the single-threaded TCG, its parallel execution model introduces increased complexity and non predictable behavior due to host level scheduling and shared execution resources.

2.4 Debuggers

As previously discussed, QEMU is supported by extensive documentation and an active community, making it one of the most widely used emulators at present. Although implementing this framework requires analyzing the source code in detail for a complete understanding of when the Fault Injector's mechanisms should actuate.

Debugging is the process of finding and fixing errors or bugs in the source code of any software by giving the possibility of stepping through the code instruction by instruction. The debugger can translate memory

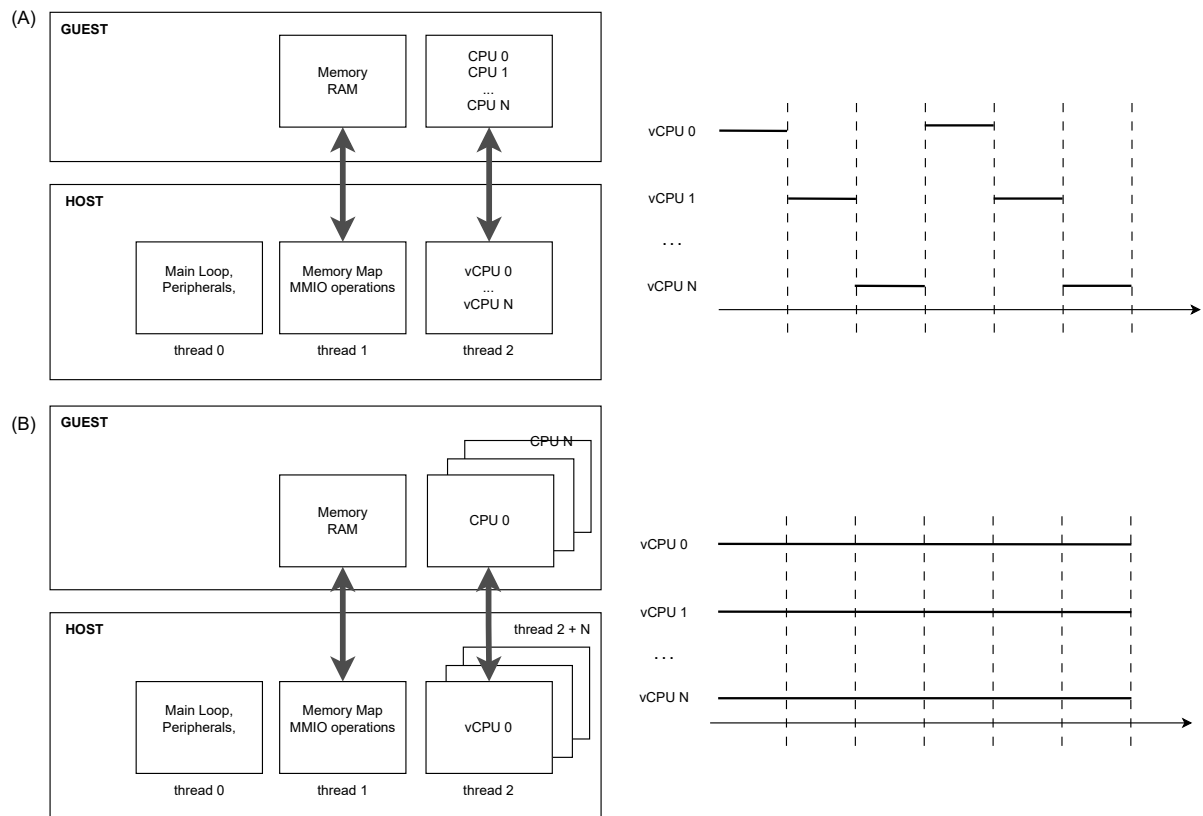


Figure 2.3.5: Single-threaded TCG (A) and multi-threaded TCG (B) differences and context switching (adapted from [24]).

addresses into readable variable names and source code locations using data from image files [37].

GDB is one of the most used debuggers for UNIX-based systems, supporting multiple programming languages, such as C and C++ programs. It provides powerful tools for tracking, inspecting, and changing how programs run. Through GDB, users can observe internal variables, examine memory contents, alter program state, and even invoke functions outside the normal execution flow.

With this debugger, is possible to insert and remove breakpoints and watchpoints. Breakpoints interrupt program execution when a specific PC address or code location is reached. When triggered, the debugger assumes control and the system state can be inspected at this execution point. In contrast, watchpoints interrupt execution when a specified memory location is accessed or modified.

The GDB supports a Python library that allows direct access to GDB features. This enables automated execution of debugging commands without user intervention. This API works within GDB (i.e., Python scripts are executed in the context of GDB), and provides access to objects such as: *Breakpoint class* to create and handle breakpoints.

2.5 Related work

Several research efforts have explored the use of QEMU as a platform for implementing fault injection mechanisms. These solutions leverage the emulator's flexibility to observe and manipulate the internal state of the system, enabling the evaluation of robustness and fault tolerance without requiring intrusive instrumentation of the target hardware or software. Over the years, multiple frameworks and QEMU extensions have been proposed, differing in terms of abstraction level, supported fault models, and the degree of control over the location and timing of fault injection. The following section reviews and discusses some of the most relevant works in this domain, providing context for the approach adopted in this dissertation.

FAIL* [38] is an architecture-level fault-injection (FI) framework for the continuous assessment and quantification of fault tolerance in an iterative software development process. It supplies the developer with FI campaigns that can be reused to ensure repeatability. It also offers pre- and post-processing analyses, which can identify vulnerable spots in the software, well-abstracted backend implementations for several hardware and simulator platforms, and scalability of FI campaigns by providing parallelization. It supports Bochs, QEMU, and Gem5 simulators, as well as a large variety of different host and target CPU architectures. In addition to SFI, it allows two other types of injections, between test-port-based FI (a hybrid technique into JTAG-controlled ARM boards) and SWFI. This allows, the developers introduce more veracity in fault tolerance tests. One of the key components of FAIL* Framework is the Execution-Environment Abstraction (EEA), which provides a backend interface for meta-information like the number of CPU, registers, and platform-independent naming of special registers, fine and coarse-grained state access, i.e., read/write access to CPU registers and memory, and access to the backend's time and backend events. This framework has a highly versatile characteristic, providing ease in modifying and adding new architectures, as well as new types of fault models. However, it ends up being very heavy on performance. This FI is designed for large fault campaigns and is dependent on a set of listeners that capture information about all memory accesses and instruction types. This second component, despite allowing better verification of the most vulnerable locations in the system, introduces more overhead, resulting in longer simulation times.

In the QEMU Fault Injector (QEFI) [39, 40] is possible to perform FI into different paths of a system, such as the CPU, RAM, and peripherals, with a random triggering probability defined by the user. For injection, a fault is controlled with two parameters: location and injection moment. Location is a range of memory addresses from which a random memory cell is selected as an injection target, and the injection

moment can be immediate or delayed [40]. To enable this feature, the authors extended QEMU's Tiny Code Generator (TCG) so that when a trigger occurs, a fake target instruction is emitted, and the TCG generates micro-operations with this into account. It also leverages QEMU's built-in support for the GDB server protocol to integrate fault injection with debugging tasks. This allows it to become possible to inject into specific parts of the code by stopping and continuing the execution.

Frameworks like [41] and [42] also use GDB to perform fault injection. The ESIFT (Efficient System for Error Injection) [41] is based on the Python extended GDB, which makes it portable across a wide variety of systems. It operates at near-native speed, enabling the dependability evaluation of large systems and evaluating, concurrently, both the faulty and the fault-free system behavior. This allows for faster error detection and latency evaluation. The fault injection is performed, running a fault-free application in parallel with the fault-injected application to observe deviations caused by injected faults. These faults are introduced based on execution events, such as the number of variable accesses or the execution count of specific code regions, ensuring injection while remaining independent of the test environment. In Figure 2.5.1, the ESIFT steps are presented. After the workload application load in the target system (1), an associated list of faults (2) is created with each type, location, and trigger condition. Then, two GDB instances are invoked: One will have breakpoints and watchpoints inserted by the fault injection to modify CPU components for fault emulation according to the fault list (3), while the other will be free of faults, for reference (4). This allows for monitoring error propagation and observing the fault injection impact on different variables.

FIG-QEMU [42] is a GDB-extended framework designed to evaluate the robustness of target operating systems and application software against CPU-induced soft errors. It has a main controller module for general management, and a fault library for the specification of the fault to be injected and its location. "It also comprises a trigger module, responsible for determining when to inject the fault; an injection module, which performs the fault injection; and a collector module, which gathers data during the simulation. This is an approach similar to the ESIFT framework, with some differences in the way that the injection is performed. In the FIG-QEMU, the breakpoints for pausing the program are placed in the target emulated system directly

Other frameworks implement modifications in the QEMU source code at the instruction translation level, to perform injection [32, 43–46]. For the MH-QEMU [44], the focus is on the injection of faults at the memory level. This framework provides a memory-state-aware fault injection platform implemented by extending a virtual machine (VM). The main point of performing it is the interception of memory accesses, with features that allow the analysis of memory access behaviors and patterns, such as physical-to-virtual

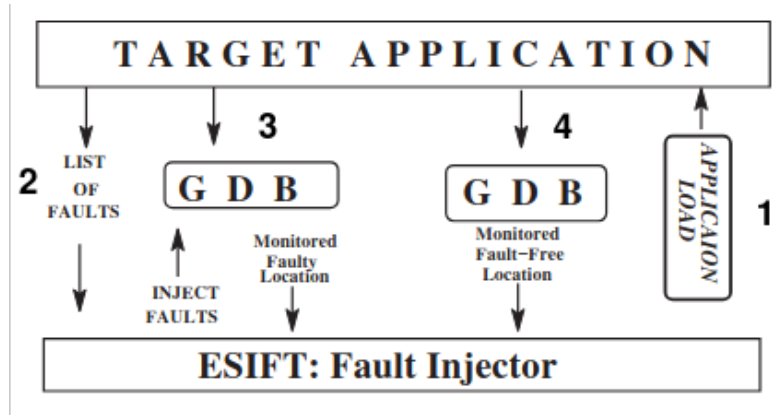


Figure 2.5.1: ESIFT Overall Framework (adapted from [41]).

memory address mapping. This allows the tests performed to be closer to phenomena that occur in the real world, like row-hammer faults, which are obtained in real-world memories that are accessed several times, causing unforeseen changes in content.

Similar to this approach is the F-SEFI [46], designed to take advantage of the QEMU virtualization. It implements a broker that is loaded dynamically after the target system starts. Then it intercepts instructions and corrupts them before sending them to the host system. It allows for injecting faults in the CPU static environment and also gives the flexibility to choose specific micro-operations to modify at random (for coarse-grained) or targeted (for fine-grained) situations. There is also an extended version of P-FSEFI [47] dedicated specifically to High-Performance Computing (HPC) applications, allowing parallel application injection. However, this fault injection mechanism has a significant impact on the QEMU performance.

In the Efficient fault injection in QEMU by Ferraretto et al., [32], the injection is revealed to be non-intrusive. They modify the QEMU DBT mechanism in different phases to perform several fault models and enable them to run in single-step mode. In this mode, each TB in the TB Cache contains only one instruction, so all the instructions can be affected by faults.

In [43] by Amarnath et al., the authors implemented a framework to simulate bit-flips triggered by system calls of Linux, measuring dependability metrics and propagation effects. It modifies the instruction register value before the intermediate DBT stage to inject a specific user or framework-selected fault from the fault library.

A different framework with a built-in test (BIT) is presented in [45] by Li et al. BIT is a method that allows a machine to test itself. It is revealed to be useful for confirming if a fault has been detected during normal operations. This framework uses key-value pair abstractions of XML elements to describe fault modes and then simulates the hardware board with their presence. This framework, similar to [38] and [46],

allows for controllable low-level triggering, enabling the possibility of injecting faults in applications where the source code is not available.

FIES [48] framework works with commonly used embedded commercial off-the-shelf (COTS) processors and provides early design-phase analysis of how well self-tests detect faults. It is developed for the evaluation of software-based self-tests (SBSTs) according to the safety standard IEC 61508. It can simulate faults in the control and execution parts of an ARM processor and also accounts for memory-related issues, such as how different memory cells might interfere with each other. The framework also allows for the interactive specification of automatically inserted defects in XML. This framework was also designed to be easily adaptable to different target architectures, aiming to be as generic as possible.

Table 2.2 provides a brief overview of the different frameworks and their characteristics. It indicates whether the framework modifies QEMU modules, supports multiple architectures, and specifies the fault models and injection locations used.

Table 2.2 QEMU-based fault injection methods characteristics.

Frameworks	Emulator agnostic	Multi architectural framework	QEMU modified modules	Trigger types	Fault Locations
FAIL* (SFI) [38]	No		Non	Number of CPU cycles	CPU; Memory
QEFI [39]	No	No	TCG	PC, Access, Application	CPU; Memory; Peripherals
ESIFT [41]	Yes	Yes	Non	Application (variable access)	CPU; Memory
FIG-QEMU [42]	Yes	No	Non	Random	Registers; Memory; Peripherals
MH-QEMU [44]	No	Yes	TCG	Memory access patterns	Memory
F-SEFI [46]	No	-	TCG	Random, Application, Instruction, Access	Code memory; Register
Ferraretto et.al.; [32]	No	Yes	TCG; DBT Fetching; TB Generator	Random	CPU
Amarnath et.al.; [43]	No	No	DBT Fetching; Disassembler	Syscalls execution	CPU; Memory
BitVaSim [45]	No	Yes	Memory Mapping, Added modules	PC, Time	CPU; Memory; Peripherals
FIES [48]	No	Yes	DBT Fetching	PC, Access, Time	CPU; Memory

Despite the wide range of fault injection techniques and emulation-based frameworks presented in the literature, important limitations remain insufficiently addressed. In particular, many existing approaches focus on the location and type of injected faults while providing limited discussion on how the underlying execution model of dynamic binary translation-based emulators affects injection precision. Moreover, none of these approaches explicitly considers the types of optimizations implemented by QEMU, which directly constrain the possibilities for performing fault injection during the runtime of the target system.

Execution control is only available at TB boundaries rather than at individual instruction boundaries. This makes it difficult to precisely align fault injection events with specific instructions, effectively skipping many potential injection points.

3 Framework Design

This chapter presents the design and implementation of the proposed fault injection framework. In this framework, a GDB interface is used to perform modifications to QEMU during the execution of the target system. The proposed approach will introduce Python scripts that are responsible for carrying out the stop conditions, inspections and modifications in registers, memory, and peripheral states and contents.

Figure 3.0.2 presents an overview of the framework architecture. The framework receives an XML input file that defines the injection points, placing breakpoints according to the faults to be injected, thereby enabling the required modifications. During execution, output log files are generated to record tracing information and the effects caused by each injected fault. Additionally, the framework allows the retrieval of runtime values from the emulated system, supporting detailed post-execution analysis.

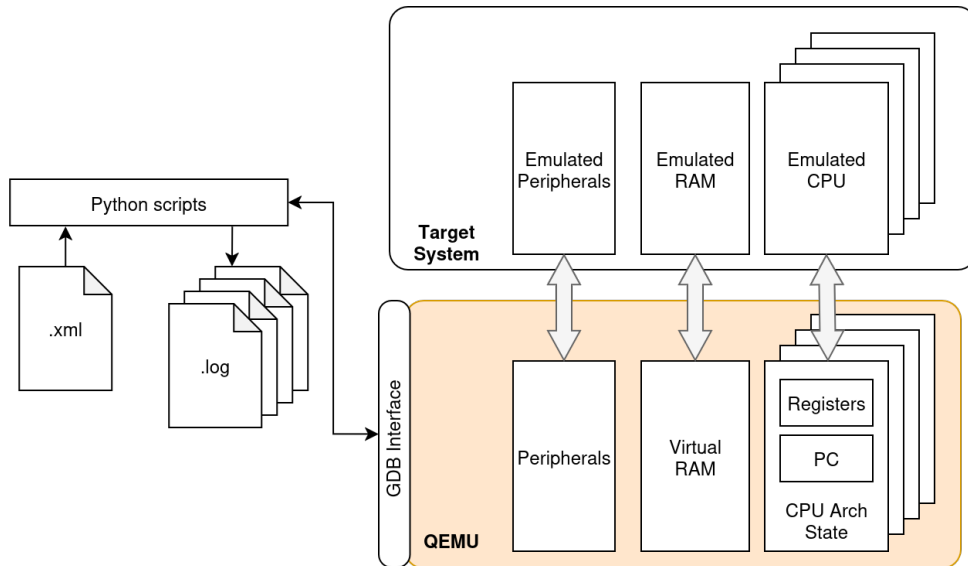


Figure 3.0.2: Framework Overview.

To perform the fault injection analysis, the framework operates in two execution phases. In the first phase, referred to as the golden run, the system is executed without fault injection in order to record its normal behavior. This execution serves as a reference baseline. In the second phase, a faulty run is carried out. This time, the system is executed under the same conditions, but with the fault injection enabled. By comparing the system states between both executions, it is possible to assess the impact of injected faults, verifying possible flow deviations and differences in CPU and memory outputs.

3.1 Fault Library

To specify the parameters of the faults to be injected, a configuration file in XML format is used. This approach was chosen due to its readable format with structured data. Moreover, XML is easy to modify and integrate with various tools throughout the fault injection workflow, since there are already libraries in Python that simplify the implementation process.

The fault description is organized into three main scopes, as displayed in Table 3.3: (i) in the trigger, the user will determine the type, which can be by PC, instruction number, and time deadline, and the associated trigger parameter, which depends on the selected type. For instance, if the chosen type is PC, the value determined is the PC value, where the trigger mechanism will signal to perform the analysis and injection; (ii) the target scope refers to the location where the fault is injected. The index will vary according to the component selected: for register, referring to which number and for memory and peripherals, the address; (iii) model scope, will have the type of fault (see fault type section), the mask that will determine which bits are affected, the duration of the fault and the duration index parameter that change its nature according to the duration parameter specification.

Table 3.3 XML tags used to configure fault injection parameters.

<i>Label</i>	<i>Description</i>
<code><trigger_type></code>	Defines the type of trigger to activate the fault: <i>PC</i> , <i>INSN</i> , or <i>TIME</i> .
<code><trigger_parameter></code>	Defines the variable according to trigger type
<code><component></code>	Identifier of the target component (Registers, Memory, or Peripheral).
<code><index></code>	Identifier of the target component variable (number for Register component and address for Memory and Peripheral components).
<code><type></code>	Type of fault to inject (e.g., BIT-FLIP, BIT-SET, BIT-RESET).
<code><mask></code>	Bitmask applied to select specific bits for the fault.
<code><duration></code>	Indicates whether the fault is TRANSIENT, INTERMITTENT or PERMANENT.
<code><duration_index></code>	Index of the fault to be injected (see section 3.6).

To process the fault description file, an XML parsing library in Python is used as the first step of the framework. Based on the defined fault trigger types, each fault is grouped into a corresponding list, as illustrated in Figure 3.1.1. This division is required because each trigger type is associated with its own specific breakpoint callback functions within GDB, allowing trigger handling to be isolated and managed

independently.

By organizing faults in lists according to their trigger type, the framework enables the association of breakpoints only to a specific callback, avoiding the activation of unnecessary breakpoints and reducing execution overhead. This facilitates the addition of trigger types or fault models that can be incorporated by introducing new callbacks and corresponding lists without affecting existing triggering mechanisms, increasing scalability. If the list associated with a given trigger type is empty, the breakpoints related to that trigger are not placed/enabled.

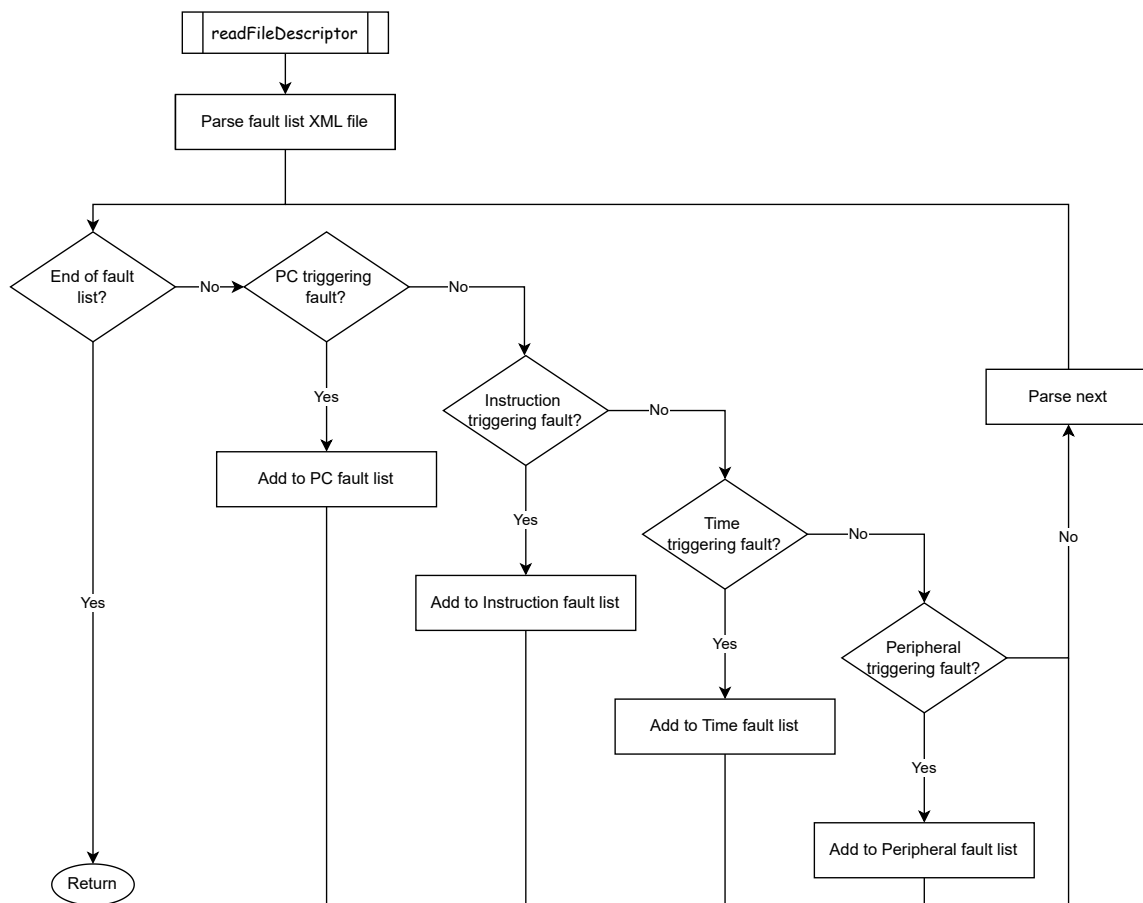


Figure 3.1.1: Faulty Library parsing flowchart.

In Listing 3.5, the fault class is shown, including its internal variables and details. It encapsulates not only user-defined parameters, such as target, trigger, and fault model attributes, but also a set of control parameters required for runtime management. These include a unique fault identifier, a trigger-dependent deadline, and a counter that tracks how many times a specific trigger point has been reached. Those parameters will be dependent on the framework for managing activation and deactivation of each breakpoint, the removal of breakpoints when the defined trigger is completed, and the modification at the defined points.

Listing 3.5: Fault class

```

class Fault:
    id_counter = 0
    def __init__(self, model, duration, mask, component, index,
        trigger_type, t_param, duration_index=None, t_deadline=None):
        self.id = Fault.id_counter
        Fault.id_counter += 1
        #MODEL
        self.model = model
        self.mask = mask
        self.duration = duration
        self.duration_index = duration_index
        #TARGET
        self.component = component
        self.index = index
        #TRIGGER
        self.trigger_type = trigger_type
        self.t_param = t_param
        #CONTROL
        self.t_deadline = t_deadline
        self.t_addr = None
        self.opt = False

```

3.2 Fault types

Some frameworks, such as [44], inject faults by intercepting store or load operations and producing incorrect values. In contrast, the proposed framework allows the injection of faults into CPU registers and memory at any time.

Using a pointer to the *CPUArchState* struct address, it is possible to define in which register the fault will be injected. As demonstrated in Figure 3.2.1, the registers are variables that are saved in memory. By knowing the offset between the *CPUArchState* struct address (*cas_addr*) and the register base address (*rb_addr*), it is possible to determine the addresses of the remaining registers. The offset values are architecture-specific, since for different architectures the *CPUArchState* struct differs, and the variables

corresponding to CPU registers may be located at different offsets within the struct. In the equation presented, the variable w corresponds to the word size, while t represents the index of the register where the fault will be injected. Using this information, a GDB command can be issued to modify the value pointed to by the calculated address.

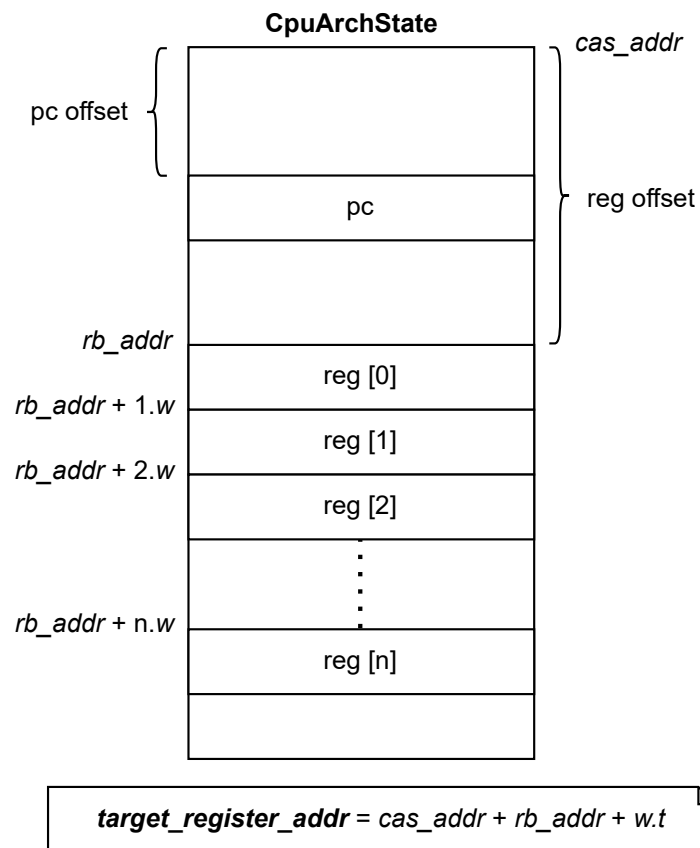


Figure 3.2.1: Generic fault example in a CPU register.

As shown in Figure 3.2.2, for the AARCH64 and RISC-V architectures, the struct will have a different nature in the offsets of each parameter. This approach facilitates the process of adding new CPU architectures and the possibility of extending the injection to other CPU parameters by just adding the offset and associating it with a new fault model.

For the memory case, it is necessary to find in QEMU's map where the associated region is defined by the target virtual address. In Figure 3.2.3, the virtual memory base address **0x40000000** (**vmb_addr**) is mapped to the **0x7fff67e00000** physical memory address (**pmb_addr**), meaning that both addresses point to the same memory data. Consequently, if the width in virtual memory is calculated, from the base address (**vmb_addr**) to the target address (**t_addr**), by adding this result to the physical memory base, it is possible now to perform the injection. In the diagram, it is demonstrated that for virtual address **0x40000018**, the content is the same for the physical address **0x7fff67e00018**. Here, the offset is **0x18**.

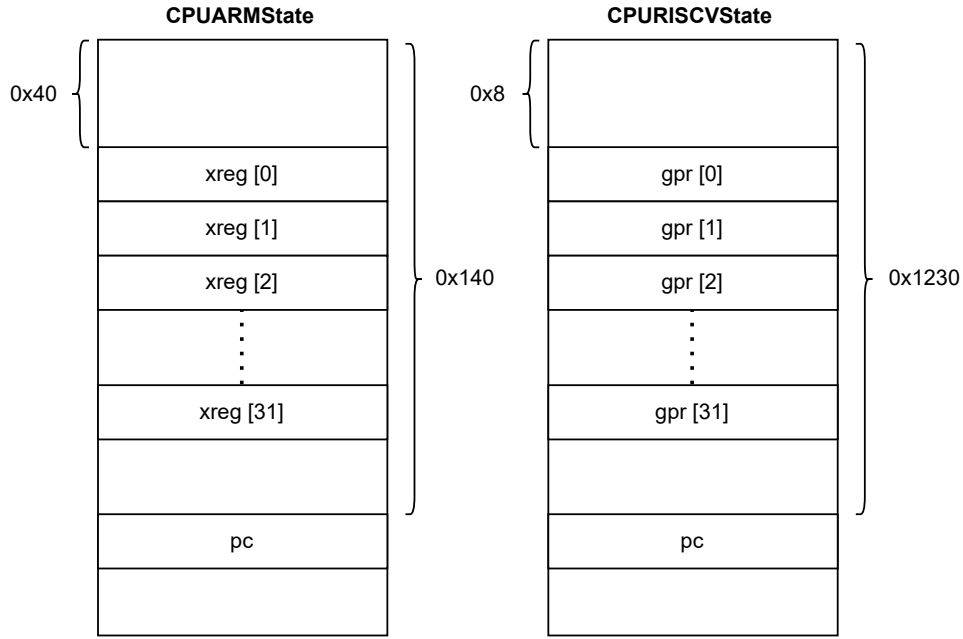


Figure 3.2.2: CPUArchState struct definition for Aarch64 and RISC-V architectures, with the Register base and PC address being located in different positions.

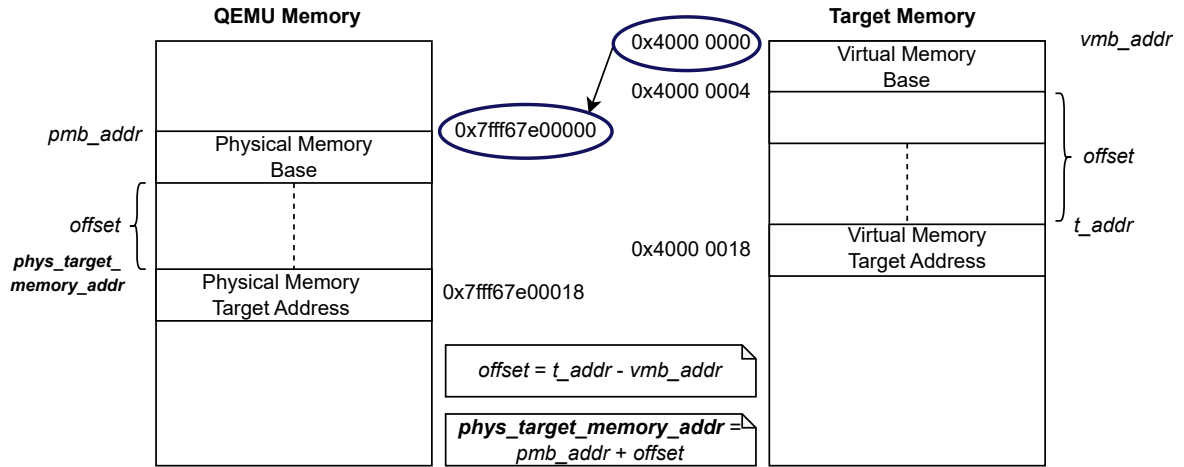


Figure 3.2.3: Fault example in a memory position.

For the device case, the injection is performed by intersecting the *memory_region_read_accessor* and *memory_region_write_accessor* functions. Those have two fundamental parameters: peripheral address and data value. By modifying the data value with a faulty one, the peripheral emulates the faulty behavior. In this way, is possible emulate real hardware faults on peripherals, such as component failure due to power spikes or electrostatic discharges.

Regardless of the injection location, all faults in the proposed framework are injected using a mask.

This approach defines how the original value at the target address is modified according to the selected fault type, which determines the resulting faulty value. The equations below present the method for obtaining the faulty value based on the fault type and the content present at the target address.

$$\text{BitFlip} \rightarrow \text{FaultyValue} = *addr \text{ XOR } mask$$

$$\text{BitReset} \rightarrow \text{FaultyValue} = *addr \text{ NAND } mask$$

$$\text{BitSet} \rightarrow \text{FaultyValue} = *addr \text{ OR } mask$$

With this mask approach, it's possible to emulate SEU by using masks with only one bit set, and MEU with multiple bits set. Then, using a GDB command, the value is modified at the corresponding address, whether it refers to memory, a peripheral, or a CPU component.

In Figure 3.2.4, the flowchart of the *faultInjection* routine, responsible for updating the component values with faulty data, is presented. This function is used on the faulty run when trigger conditions are satisfied. In those moments, the inject function is invoked. First, the component type is evaluated, and the target address is calculated accordingly. Next, the content pointed to by the target address is read, and the corresponding mask is applied. According to the selected fault model, the original value stored is read and combined with the defined mask. In the case of a bit-flip, a XOR operation is performed between the value and the mask. For a bit reset, the masked bits are cleared. For a bit-set, the masked bits are forced to one (OR with the mask). Finally, the modified value is written back to the target address, effectively producing the faulty behavior.

3.3 Golden and Faulty Runs Components

As mentioned earlier, the fault injection technique is applied to evaluate whether the system is capable of detecting and correcting possible errors. This is achieved by comparing the values of different system components across executions, between a fault-free golden run and a faulty run that includes the faults designated in the descriptor file. For this, it is necessary to perform a golden run first, followed by the faulty run. After fault injection, if the values of CPU and memory remain equal to the ones in the golden run, this indicates that the system either successfully handled the fault or it was masked. During each run, several components are responsible for fault injection and analysis. Each run is designed to terminate when all system cores reach an idle state.

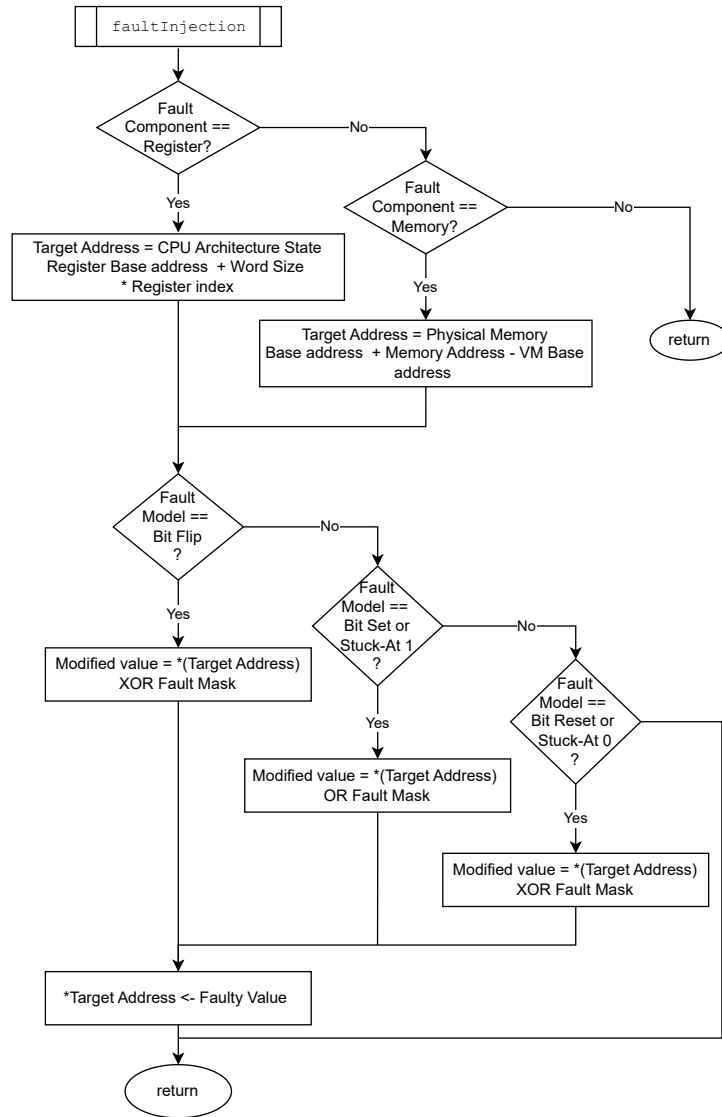


Figure 3.2.4: Fault injection flowchart.

The parser is responsible for processing the XML file and providing the extracted information to both the controller and the injector. The controller defines which system components will be monitored by the collector during the target execution. It also determines when a fault injection should occur, based on the fault descriptions provided in the XML file. The injector modifies the values of the determined target component for the defined core at the points determined by the controller. This module is used exclusively during the faulty run. The collector is responsible for gathering information about the simulation state after each fault injection event. Its main purpose is to generate log files used for subsequent analysis and to identify the outcome of each injected fault.

As shown in Figure 3.3.1, during the golden run, the parser decodes the XML file. Then, according to it, the controller places breakpoints that will be necessary for injecting and collecting results during

execution. Those will be responsible for invoking the routines in the triggering moments. Each run is split into two phases: preparation and execution. The preparation phase corresponds to the initialization of framework parameters and the placement of breakpoints. The execution phase begins when the breakpoints associated with the trigger conditions are enabled, allowing the collector and injector to operate.

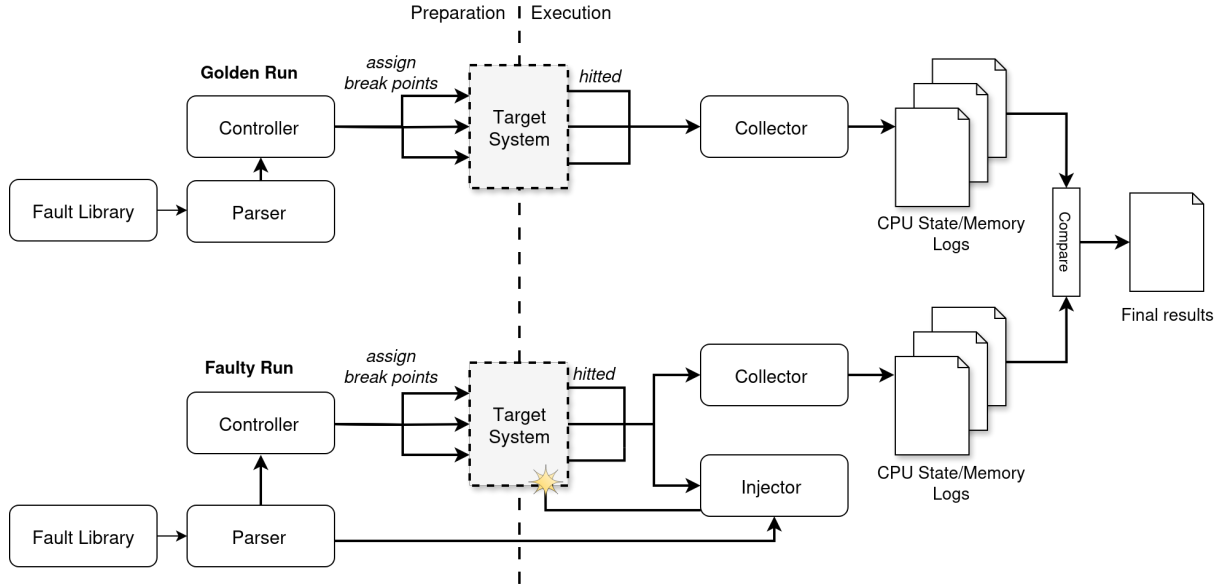


Figure 3.3.1: Golden Run and Faulty Run overview.

During the preparation phase, the controller places breakpoints in specific regions of QEMU based on the fault models defined by the user. These breakpoints are used to satisfy triggering conditions as well as to intercept and store relevant addresses within the emulation environment. These addresses include the *CPUArchState* struct addresses for each core, the memory base address, and additional locations required by the injection mechanisms. Once these variables are stored, the framework becomes ready to perform fault injection and data collection.

By transitioning from the preparation phase to the execution phase through precise control of breakpoint stop conditions and the timely invocation of the associated routines, fault injection and analysis are performed at the defined execution points, ensuring controllability and observability. The collector and injector operate according to the type of run being performed, either golden or faulty.

In Figure 3.3.2 is illustrated how the injector and collector take place at the trigger points. During the golden run, the collector records CPU and memory information at each trigger moment. Then, in the faulty run, the injector is activated at the same trigger moments. In this case, the collector not only records system information but also performs comparisons to identify deviations caused by injected faults.

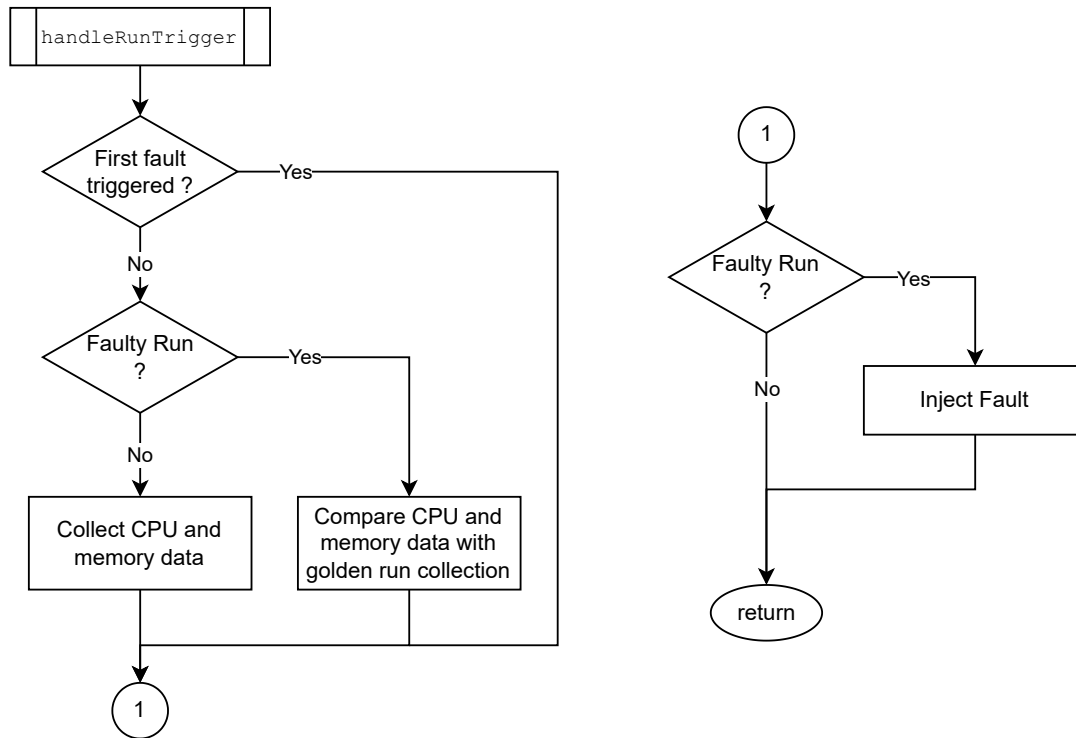


Figure 3.3.2: Collector and injector functions in golden and faulty run.

3.4 Framework Initialization

To initialize an experimental campaign, the framework runs QEMU under GDB control, which provides an external observation and intervention layer for precise fault injection without modifying the system under test. The execution flow is initiated by launching GDB with a set of predefined commands that specify the Python control scripts, the QEMU executable, and the corresponding execution parameters. These scripts are responsible for configuring the fault injection campaign, setting breakpoints and watchpoints, and managing both golden and faulty runs throughout the experiment.

To configure the framework, the user must define a set of options that describe the target architecture, the fault injection, and the data collector scope. These parameters are passed to the Python control script and designate how the framework initializes the emulator and applies the fault models:

- *-root*: Directory path containing the emulation image.
- *-arch*: Target architecture of the system.
- *-core*: CPU core selected for tracing and fault injection.
- *-l-addr*: Lower bound address for tracing memory faults.

- *-h-addr*: Upper bound address for tracing memory faults.
- *-vm-base*: Base address of the virtual machine memory space in the guest system.
- *-hyp-base*: Base address of the hypervisor memory space.
- *-xml*: XML file describing the fault injection campaign.

These parameters are required to configure both Python control scripts. The *root* and *arch* define the system image and target architecture to be emulated. The *core* parameter selects the CPU core on which the golden and faulty runs are performed. *l-addr* and *h-addr* define the lower and upper bounds of the memory region for result tracing. The *vm-base* and *hyp-base* parameters are required to support specific trigger types and to ensure the correct mapping between virtual and physical memory spaces. The *xml* refers to the descriptor file containing fault models to be applied.

3.5 Breakpoint management

The controller manages the breakpoints of each run, assigning and removing them when necessary. As mentioned earlier in section 3.3, each run is split into two phases: preparation and execution. In the preparation, fundamental addresses are intersected: (i) *CPUArchState*: the breakpoint is put in the initialization of the CPUs. For that, QEMU uses functions with the architecture family name (eg *arm* or *riscv*) and *_cpu_realize* suffix. When this breakpoint is triggered, the *handle_read_cpu_struct_addr_cb* routine is executed, as shown in Figure 3.5.1a. For multi-core runs, each emulated core has its associated *CPUArchState*. Therefore, within this routine, a condition is used to verify whether the core being initialized corresponds to the target core (defined in the framework parameters) for fault injection. This verification is possible using the *cpu_index* variable of the *CPUState* struct. If the condition is satisfied, the address of the struct, the PC variable address, and the register base array address are saved. Then the breakpoint is removed; (ii) Start of the physical memory region address: a breakpoint is placed in the *get_page_addr_code_hostp* function. This function retrieves the physical address mapped to a virtual memory address for each translated TB.

By intercepting the TB whose PC corresponds to the beginning of the virtual RAM region (e.g., 0x40000000 for AArch64 systems), as defined by the *VM_BASE* framework parameter, it is possible to determine the base physical address of the RAM region. Once this reference point is obtained, any virtual address within the RAM region can be mapped to its corresponding physical address only using offset values. This provides the reference point for emulating soft errors in memory, as explained in section

3.2. When this breakpoint is triggered *handle_read_virt_memory_addr_cb*, the function will search for the PC corresponding to the first instruction on the application level, being invoked in this location until it finds it, as shown in Figure 3.5.1b. After encountering it, the physical memory address is saved, and the breakpoint is destroyed. With these stored parameters, the framework transitions to the execution phase.

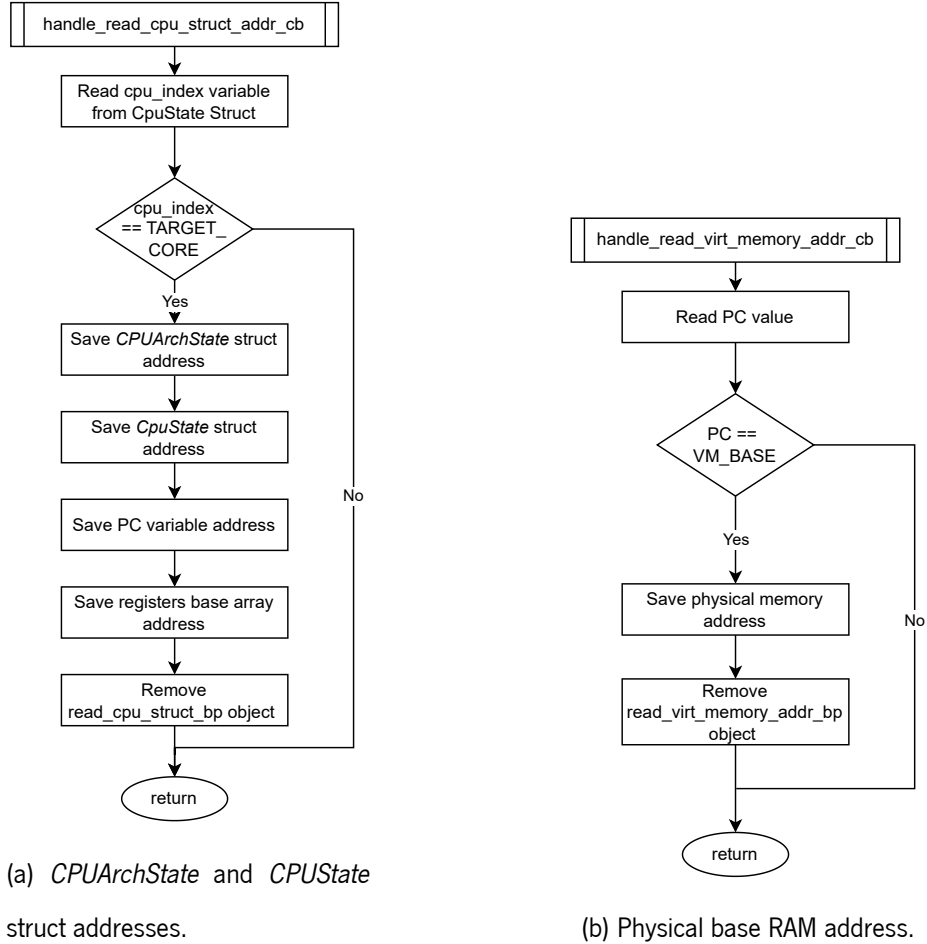


Figure 3.5.1: Handler callback flowcharts

Similar to other frameworks, such as [43], [32], [46], and [39], this framework modifies the translation mechanism to perform injection. The breakpoints are placed during the dynamic binary translation, more specifically, in the translator loop. An alternative approach would be to place it directly on the translated code. The main issue with this approach is that the code translation does not follow a 1-1 instruction translation (one target instruction to one host instruction), meaning that converting a trigger for the target to the host would be a time-consuming task. Another drawback of injecting into the translated code is that during the translation phase, the TCG applies optimizations that further change the match between target and host instructions. As a result, the injection points or monitored instructions may no longer align with their original positions, potentially leading to inaccurate fault placement or inconsistent data collection.

Therefore, taking into account the previously described nuances, the translation phase was chosen as the basis for fault injection.

For breakpoint placement, the `IntersectPoint` class, on Listing 3.6, extends the Python `gdb.Breakpoint` API class, acting as a wrapper that inherits its parameters and methods through the built-in function `super()`.

Listing 3.6: `IntersectPoint` class

```
class IntersectPoint(gdb.Breakpoint):
    def __init__(self, location, stop_action=None, enabled=True,
bp_type=gdb.BP_BREAKPOINT, action_args=(), **kwargs):
        super().__init__(location, bp_type, enabled, **kwargs)
        self.stop_action = stop_action
        self.action_args = action_args
        self.dstr_bp = False
        self.bp_count = 0
        self.enabled = enabled

    def stop(self):
        flag = False
        self.bp_count += 1
        if self.stop_action(*self.action_args):
            return True
        if self.dstr_bp:
            gdb.post_event(self.delete)
        return flag
```

Parameters such as *location*, *bp_type*, and *enabled* are intended for use by the `gdb.Breakpoint` base class. The *location* specifies the address, function, or line number where the breakpoint is set. *enabled* determines whether the breakpoint is initially active, and *bp_type* defines whether it is a standard breakpoint or a watchpoint. Furthermore, the *stop* routine, which is executed when the breakpoint is triggered, is configured to accept a *stop_action* parameter containing the associated routine and its associated arguments (*action_args*). The *stop()* method, used to retrieve the control to the terminal when the simulation finishes, is called when the return value is true.

The *bp_count* parameter tracks how many times the breakpoint was hit, and the *dstr_bp* variable indicates the routine for which the breakpoint is no longer needed and can therefore be disabled. By

default, the class creates an active breakpoint when it is initialized. To remove the breakpoint, it schedules a post-event so that *delete* runs in a safe context, outside the internal GDB code that processes breakpoints. This helps prevent crashes or other unintended behavior. An example to initialize a breakpoint with an associated callback is listed below.

```
bp = IntersectPoint("<qemu_function>", stop_action=foo_cb,  
                   enabled=True, action_args=(a,b))
```

Although this method is favorable, since the breakpoints are placed in generic areas or homologous functions, making it compatible with various types of emulated architectures, it will also present some issues related to the optimizations performed by QEMU.

a) Precise injection

The first challenge is related to the moment of injection itself. In QEMU's workflow, a TB is translated and executed. Intercepting the instruction translator would be sufficient to inject a fault by changing the values. However, the CPU register state at that moment only reflects the state when execution is at the first PC address in the TB. In other words, during the translation phase, the current *CPUArchState* struct variables are synchronized only with the moment the first TB instruction executes. So, if there is an intention to inject a fault at a moment other than the first instruction, this will not be possible.

The example shown Figure 3.5.2 illustrates a 1:1 translation for simplicity. In this case, the fault is injected into register r0 when the PC reaches address 0x4000002c. However, once QEMU begins executing the host code for the corresponding TB, the value of r0 has already been modified. This means that if any instruction within the same TB that corresponds to a PC address before 0x4000002c uses r0, it will observe the faulty value as well. In other words, the error is effectively applied earlier than intended, not only from 0x4000002c onward.

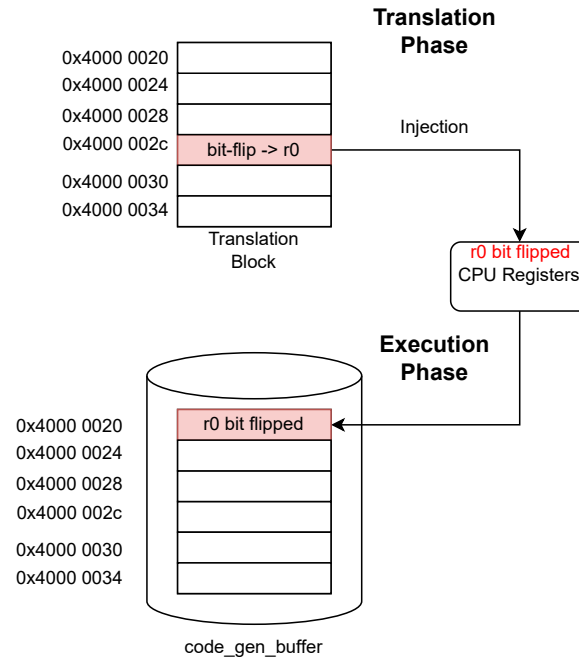


Figure 3.5.2: Problem with injecting the error in the translation moment without the proper mechanisms.

To mitigate this nuance, three approaches, illustrated in Figure 3.5.3, are proposed. In option A, configuring QEMU to generate TBs with only one instruction would be a solution, as every instruction would be associated with the first PC. Nevertheless, this will significantly increase performance degradation. In option B, the controller would modify the TB, which contains the fault in TBs, with one instruction. This would be possible by changing the static flag *one_insn_per_tb* in QEMU to start generating TBs with one instruction. When QEMU reaches the last instruction of the TB, then TBs would be generated normally. The QEMU only updates the nature of the TB generator mechanism during the initialization of a new iteration in the CPU execute loop, meaning that changing this flag will not result in TBs with different morphology immediately during translation. Then, in option C, the TB is divided into two, forcing the instruction of interest to appear as the first instruction of the second TB, thereby enabling precise fault injection at the desired execution point.

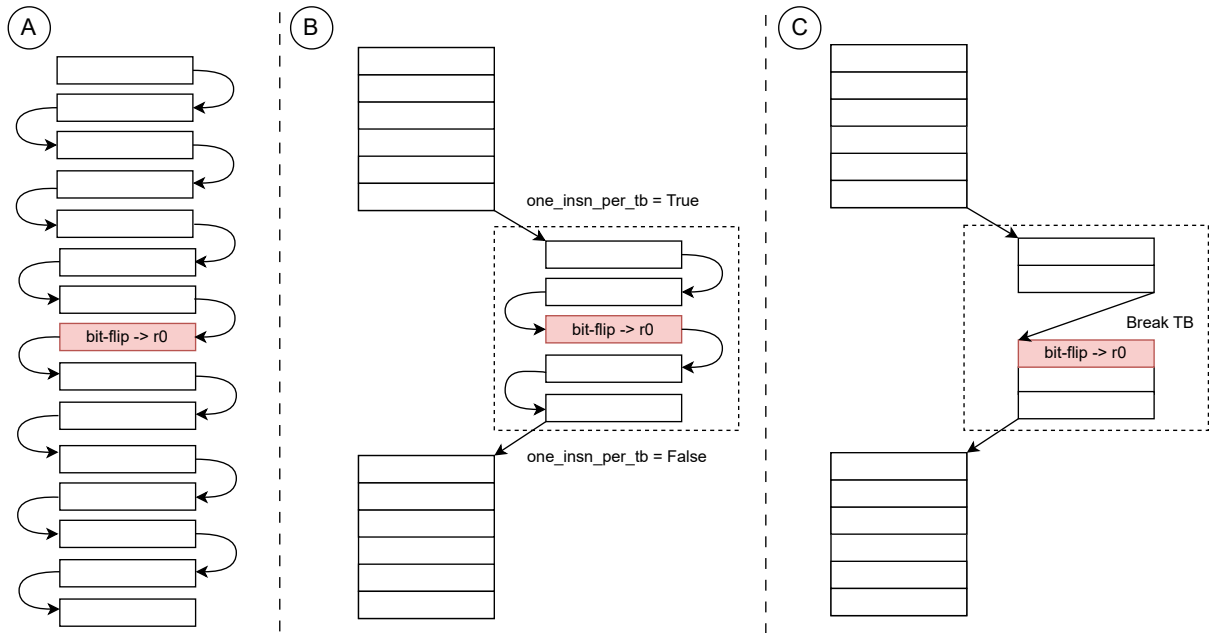


Figure 3.5.3: Different approaches to inject a fault precisely: A) QEMU configured for one instruction per TB; B) Dividing the TB in question into several TBs with one instruction; C) Breaking the TB so that the moment for injection, stay on top of the second divided part.

Option C was selected because it is more direct and requires only the evaluation of the trigger point to initiate the breaking mechanism, while in option B, in addition to this condition, it is necessary to evaluate the moment beforehand during the CPU execute loop (*cpu_exec_loop*), in which the TB, containing the instruction where the injection is performed, is generated. Furthermore, having a large volume of TBs with an instruction will also mean an increase in the execution exchange between the emulator controller and host code, causing an increase in overhead.

To perform this process, a breakpoint was added to the translation phase, where each instruction is disassembled and translated. This breakpoint has an associated callback, which is executed whenever it is triggered. As shown in Figure 3.5.4, this callback is represented by the routine highlighted in green and is responsible for forcing a TB break when the next instruction to be translated matches the selected triggering point.

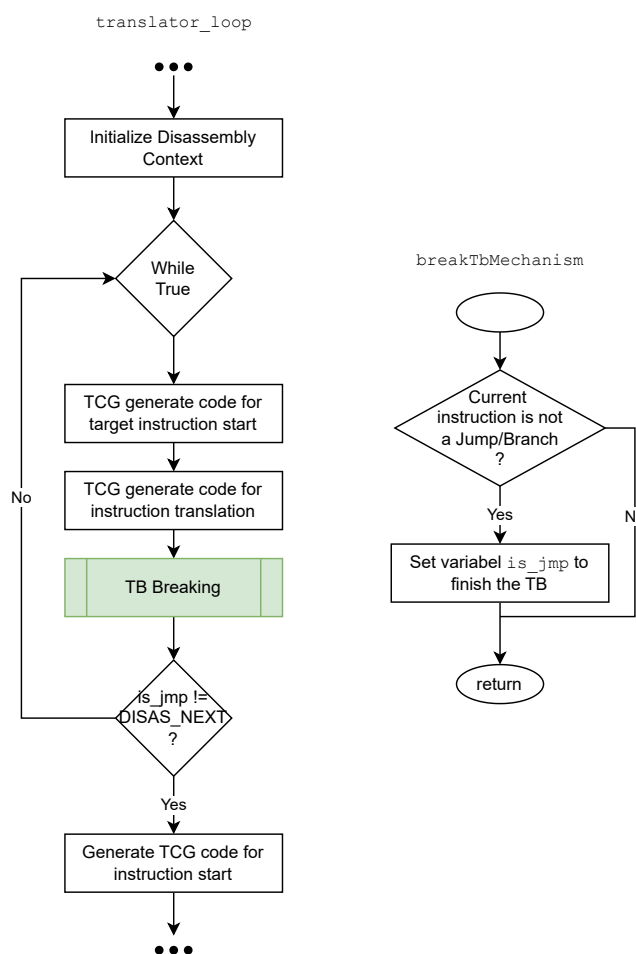


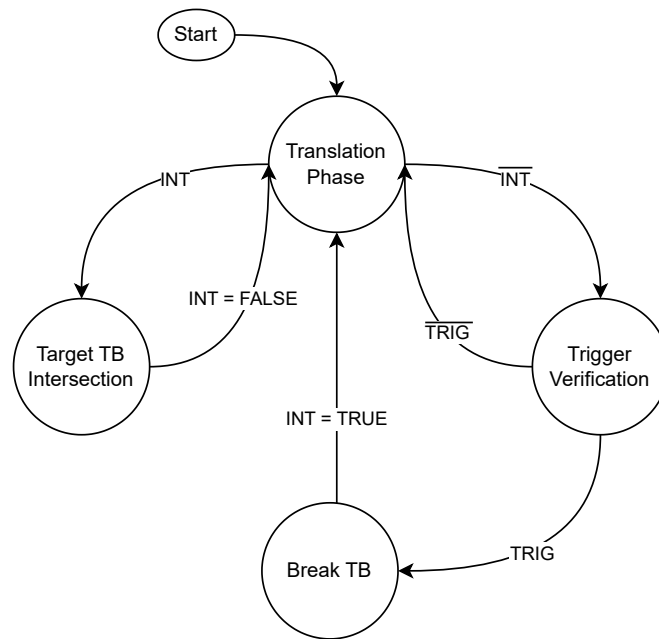
Figure 3.5.4: TB breaking in Translator Loop flowchart.

To guarantee the successful triggering, in Figure 3.5.5, a state machine is presented to illustrate the TB breaking process and how the second part of the split is intersected. During the translation phase, a trigger condition is continuously evaluated. The *TRIG* flag is set to true when the trigger point is reached, signaling that the next instruction to be translated corresponds to the user-defined injection point. When this condition is detected, the flag causes the invocation of the callback responsible for dividing the current TB.

The breaking process is accomplished by reading and modifying the *is_jmp* variable. It can take three values: *DISAS_NEXT* when the next instruction in program order is selected, *DISAS_TOO_MANY* when there are too many instructions translated, and *DISAS_NORETURN* when the next instruction address cannot be determined. As long as the *is_jmp* variable holds the *DISAS_NEXT* value, translation proceeds normally. By forcing the *is_jmp* variable to have the *DISAS_TOO_MANY* value, the instruction addition to

the actual TB will stop, and in the next iteration of the CPU execution loop, a new TB will be generated.

Once the breaking is performed, the next step is to intercept the second part of the TB. At this point, the *INT* flag is asserted to indicate that in the next iteration of the CPU execution loop, the TB where the injection must be performed will be generated. After the intersection of the second TB is completed, the *INT* flag is cleared, and the translation flow resumes normal operation.



TRIG: Trigger point reached signaling
INT: TB intersection signaling

Figure 3.5.5: Break TB and TB intersection state machine.

The code in Listing 3.7 presents the method used. When there is a necessity to break the TB, it guarantees that *is_jump* state causes the TB to be closed. With this, is possible to inject a fault for any moment inside the TB.

Listing 3.7: Breaking TB function

```

def breakTB():
    is_jump = int(gdb.parse_and_eval("db->is_jump"))
    if is_jump == DISAS_NEXT:
        gdb.execute("set db->is_jump = DISAS_TOO_MANY")
    return False

```

b) TB intersection

The next challenge is regarding the TB cache. Every time a TB is translated, it will never be intersected again in the *cpu_exec_loop* (routine responsible for translating the target code). This means that if the SUT has, for example, a loop code, only the first iteration is prone to fault injection.

In Figure 3.5.6, a case is demonstrated where a code section, related to TB 0 is executed by the CPU more than once.

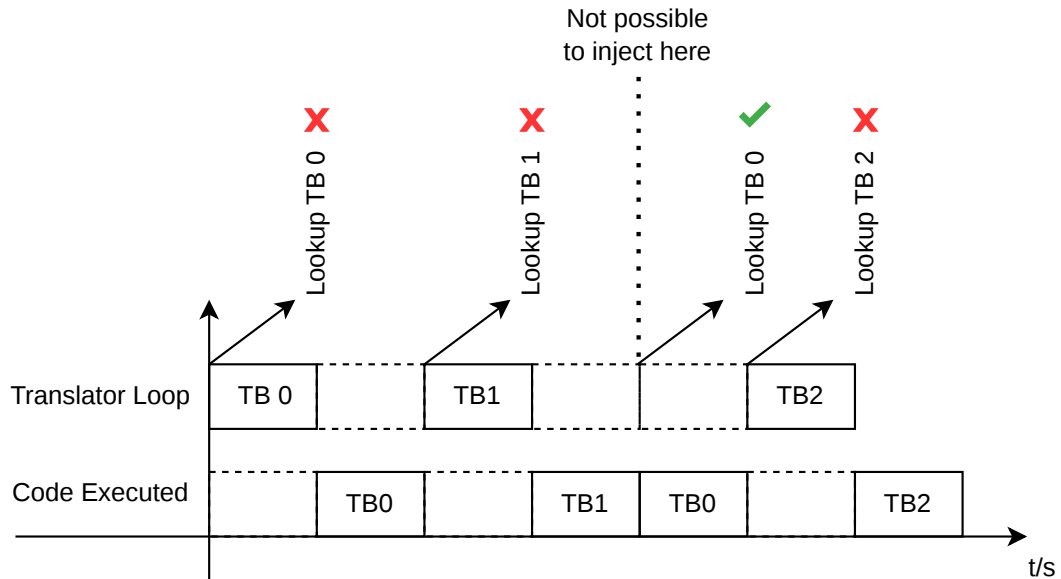


Figure 3.5.6: TB cache saving mechanism.

While in the first time the TB is translated and the associated code is executed, in the second time, the translation phase is bypassed. This happens because, in the translation phase, QEMU checks if TB is already in the cache through the first instruction PC, as shown in Figure 3.5.7 in the orange box. With this, the TB, which was previously translated, is retrieved, avoiding an unnecessary retranslation. The linking process also accelerates the process, as it eliminates the need to perform a lookup in the TB cache, allowing execution to proceed directly to the next block of code. Although this behavior is good for overall emulation performance, it becomes incompatible with the approach adopted by this framework.

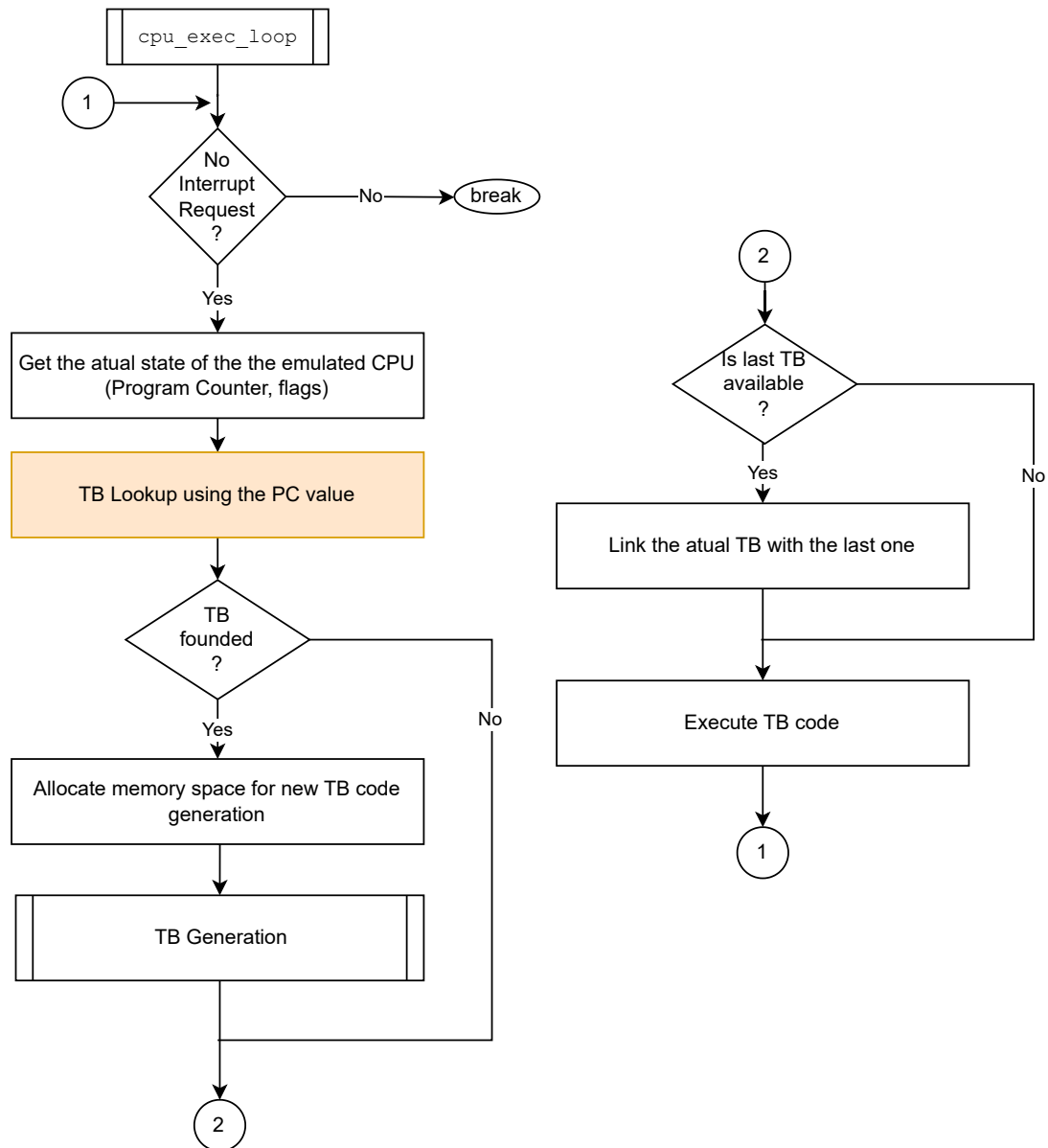


Figure 3.5.7: CPU Execute Loop flowchart.

To mitigate the effects of TB caching, two approaches were considered: the flush TB mechanism or host code execution intersection. In the first approach, TBs are flushed, forcing the code retranslation and consequently allowing an error to be injected at any point during execution. This allows for inject a error in any part of the execution. Figure 3.5.8 shows how this can be applied under the same conditions presented in Figure 3.5.6, where the fault is defined to occur in TB 0 after it has already been executed. In this example, after translation and execution, a flush takes place. As a result, the TB is retranslated, and the injection can then occur successfully. This flush can be performed as many times as needed, which supports cases where the same code region is reached multiple times through jumps while still allowing faulty behavior to be injected.

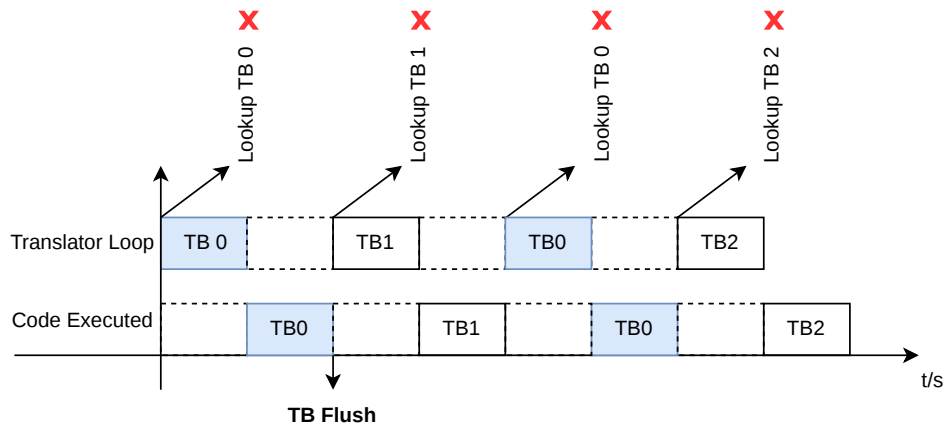


Figure 3.5.8: TB re-translation behavior.

For the flush mechanism, a breakpoint is placed after TB execution. This breakpoint is activated by other callbacks whenever a TB intersection is required. As shown in Figure Figure 3.5.9, the corresponding callback is invoked to flush the TB. This operation is performed by executing QEMU's *tb_phys_invalidate* function, which removes the TB from the cache upon receiving its address. This mechanism allows the user to inject faults multiple times in the same code region. Once the operation is completed, the breakpoint is disabled to avoid unnecessary overhead.

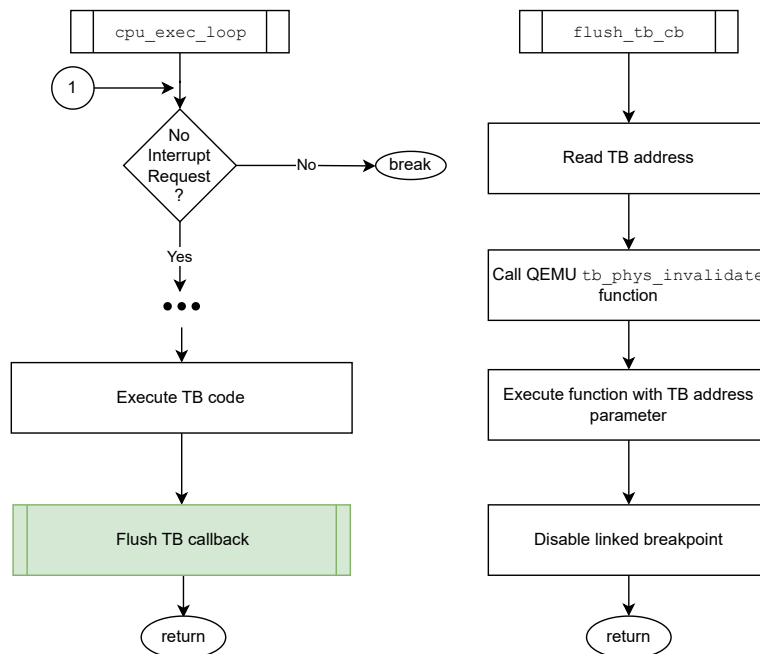


Figure 3.5.9: TB flush mechanism introduced in CPU Execute Loop.

For the host code execution intersection method, a property of TB itself is used. When host code

is generated, the TB associated with that block of host code stores a pointer to the address where the code starts. By placing a breakpoint at that address, it is possible to intercept every entry into the TB during execution, which allows injection at the desired iteration. To accomplish this, it is sufficient to collect that address after TB generation and set a breakpoint on it. Figure 3.5.10 shows a representative diagram of events. The main limitation of this approach is that it relies on hardware breakpoints, which are constrained by the limited number supported by the CPU and provide reduced controllability.

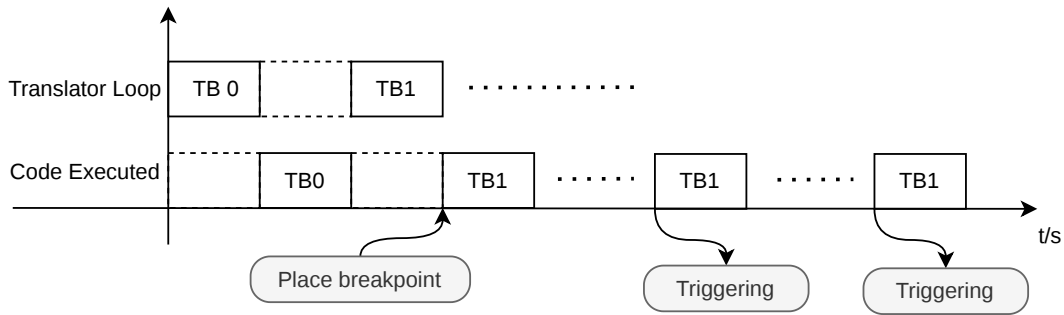


Figure 3.5.10: Breakpoint placement for injection control.

c) Multicore TB Cache Sharing

In QEMU, each emulated core maintains its own execution environment, handling both the translation and execution phases. However, all these environments share the same TB cache. This implies that, in runs where multiple cores execute the same code, only one core performs the TB translation, while the others retrieve the translated TBs from the cache.

In other words, if core A translates and executes code and then core B passes the same code through the *cpu_exec_loop* of the core B environment, the code associated with TBs will be retrieved from the cache, bypassing the translation phase. In Figure 3.5.11, a case is represented where core 0 and N execute the same code. TB 0, 1, and 2 are translated and executed in core 0. When the core N starts execution, the step of TB translation is bypassed. As a result, the translation process is not re-entered, preventing access to the beginning of TB translation in cases where they are already in cache. Consequently, the injection cannot cover all CPU occasions.

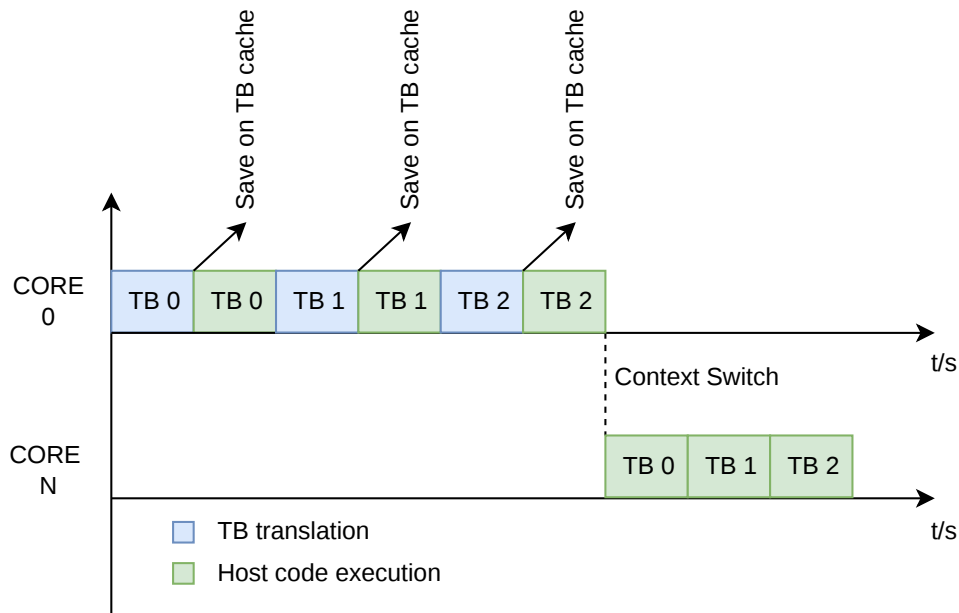


Figure 3.5.11: Example of flow context switch between cores.

To prevent this behavior, Figure 3.5.12 illustrates how TBs are flushed to enable their intersection. In this example, core 2 is the user-defined target. When this core gains execution control, the mechanism flushes all TBs from the cache. When execution control is later transferred to a non-target core, no flush is performed, since TB intersection is no longer required.

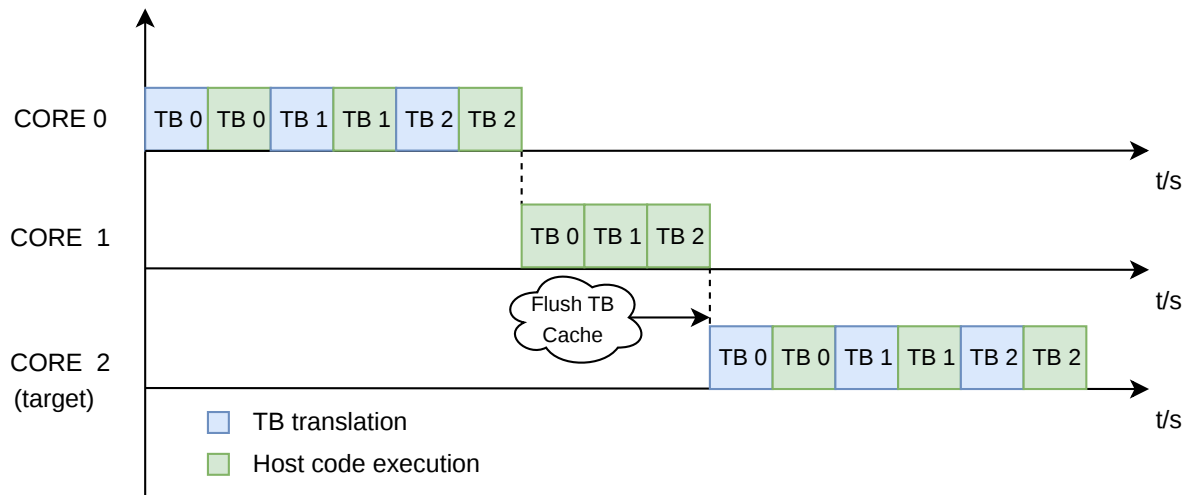


Figure 3.5.12: TB cache flush according to context switch.

QEMU provides the `cpu_exec_start` function, which is invoked whenever a specific core starts its execution. As shown in Figure 3.5.13, the framework takes advantage of this function by intersecting it and verifying if the current core is equivalent to the target one. This is achieved by reading the `cpu_index`

variable of *CPUState* struct. If the condition is satisfied, the flush is performed by calling the *tb_flush* function in QEMU, using the procedure of parsing the function symbol and treating it as a pointer to execute it, as explained in section 2.4.1. This function flushes all TBs stored in the cache without requiring additional information, in contrast to the *tb_phys_invalidate* function described in section 3.5 b). While a TB-to-TB flush would also be sufficient to achieve this effect, intersecting TBs at every execution step requires pausing the program more frequently than when flushing only when the execution control changes. Consequently, the proposed approach is more advantageous in terms of performance for these scenarios.

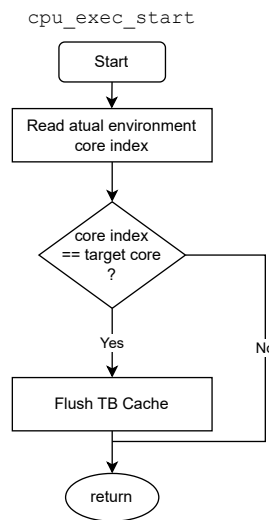


Figure 3.5.13: TB cache flush callback flowchart.

3.6 Triggering

Triggering mechanisms define fault injection moments based on execution events, register values, or memory accesses. When a trigger takes place, the *handleRunTrigger* routine (in Figure 3.3.2 flowchart) is executed according to the run type.

This framework, as explained in section 3.3, is composed of five different trigger types (PC value, number of instructions, time, device, or access) that can have three types of duration: transient, intermittent, and permanent. Each trigger type is associated with a specific parameter whose nature depends on the trigger itself. For example, in the case of a PC type, the associated parameter specifies

the PC address at which the triggering mechanism is activated, whereas for an instruction type, the parameter defines the number of instructions that must be executed before the trigger condition is satisfied.

Each trigger is also associated with a specification referred to as the duration index. This parameter varies according to the selected duration type, as illustrated in Figure 3.6.1. In the presented example, the duration index is set to a value of 4 (one for each fault to be injected). The purple arrow indicates the moment when the run (collector and injector) routines shown in Figure 3.3.2 are executed, corresponding to data collection in the golden run and fault injection in the faulty run. In the case of injecting an intermittent fault, those routines are executed 4 times when the defined trigger takes the specified value. On the other hand, for transient faults, this parameter represents a threshold like approach, i.e., those routines will only be executed in the iteration corresponding to the duration index. Until then, the trigger process is stalled.

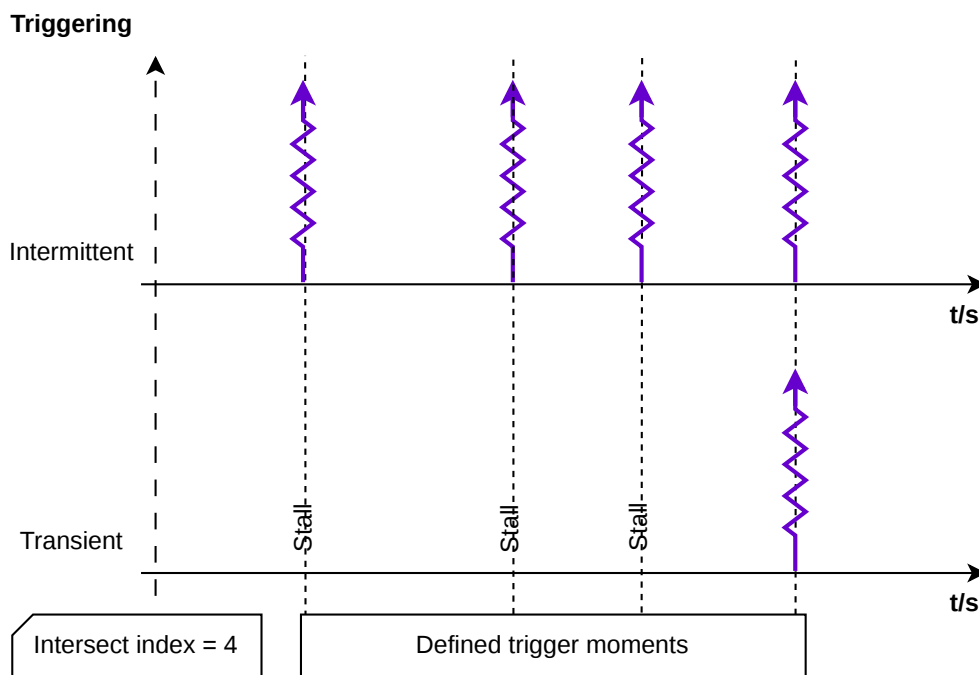


Figure 3.6.1: Difference between Intermittent and Transient triggering.

To evaluate and help control triggering during execution, the *lifetimeManagement* function (shown in Figure 3.6.2 flowchart) is responsible for tracking how many times a fault has been iterated using the *times_passed* parameter and for managing the fault lifetime based on the duration index defined by the user. When the condition is satisfied, the fault is removed from its corresponding list. If the list then becomes empty, the associated breakpoint, which is also responsible for invoking the triggering callbacks, is removed.

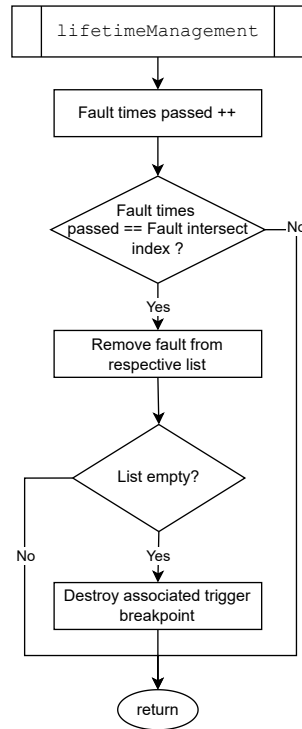


Figure 3.6.2: Fault lifetime management function flowchart.

Table 3.4 summarizes the breakpoints and callbacks associated with each triggering type supported by the framework. For each trigger, the table lists the callbacks involved, the type of interception used (breakpoint or watchpoint), and their activation during the golden and faulty runs. This overview highlights how different triggering mechanisms rely on distinct combinations of callbacks, depending on their semantics and required precision.

The location column specifies the execution stage at which each breakpoint is placed. These locations refer to well-defined points in QEMU's execution flow and are illustrated in Figure 3.6.3. The numbered markers in the figure correspond directly to the location identifiers used in the table, mapping callbacks to stages such as TB lookup, TB generation, TB execution, and post-execution handling within the *cpu_exec_loop*.

Some trigger types, such as PC and time, also have optimizations that can be applied to the faulty run, further reducing overhead. After each golden run, a new XML is created, storing trigger-specific parameters observed during the fault-free execution. Since both runs follow the same execution flow, this information can be used to predict relevant TB properties and bypass unnecessary breakpoint activations in the faulty run. As a result, the number of execution stops is reduced, leading to a more efficient fault injection campaign without compromising correctness.

Table 3.4 Details of breakpoints with associated callbacks used for each triggering type.

Trigger type	Used callbacks	Type	Location	Used in Golden Run	Used in Faulty Run
PC	<i>flush_tb_cb</i>	BP	4	Yes	Yes
	<i>pc_triggering_cb</i>	BP	2	Yes	Yes
	<i>intersect_tb_cb</i>	BP	1	No	Yes
	<i>bp_accessed_cb</i>	BP	3	Yes	Yes
Instruction	<i>flush_tb_cb</i>	BP	4	Yes	Yes
	<i>instruction_triggering_cb</i>	BP	2	Yes	Yes
Timer	<i>timer_triggering_cb</i>	BP	4	Yes	No
	<i>timer_triggering_opt_cb</i>	BP	4	No	Yes
	<i>catch_instruction_cb</i>	BP	2	Yes	Yes
Peripheral	<i>peripheral_triggering_cb</i>	BP	3	Yes	Yes
Access	<i>access_triggering_cb</i>	BP	2	Yes	Yes
	<i>target_accessed_cb</i>	WP	3	Yes	Yes

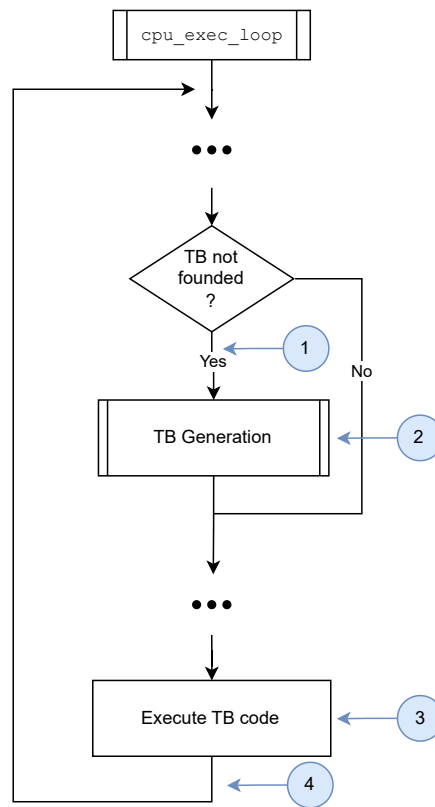


Figure 3.6.3: Triggering types breakpoint placements in `cpu_exec_loop`.

a) Program Counter (PC) Triggering

For this trigger type, activation occurs when the PC reaches the value defined by the user. To cover a wider range of fault injection scenarios at different stages, and unlike the approaches adopted by other frameworks, such as [48], [46], [40], and [32], this framework allows transient faults to be injected even after the first code block translation. As a result, even after the TB has been generated, injection may still take place. In addition to supporting transient fault injection, it also enables activation whenever the specified PC value is reached, thereby emulating an intermittent soft error each time that PC value occurs.

The used method, as explained in section 3.5 b), is breaking the TB when the next instruction PC is equivalent to the trigger targeted PC value. In Figure 3.6.4, a simple transient fault injection where the duration index is 1, for a specified PC value (0x40000078), is presented. When the instruction with PC value of 0x40000074 on TB 1.1 is intersected, the break mechanism comes is activated, causing the next TB to have the defined value as the initial PC. Then, by injecting the error on the CPU/memory data structure before the executing code of the next TB (1.2), is ensured that the faults has reached its target.

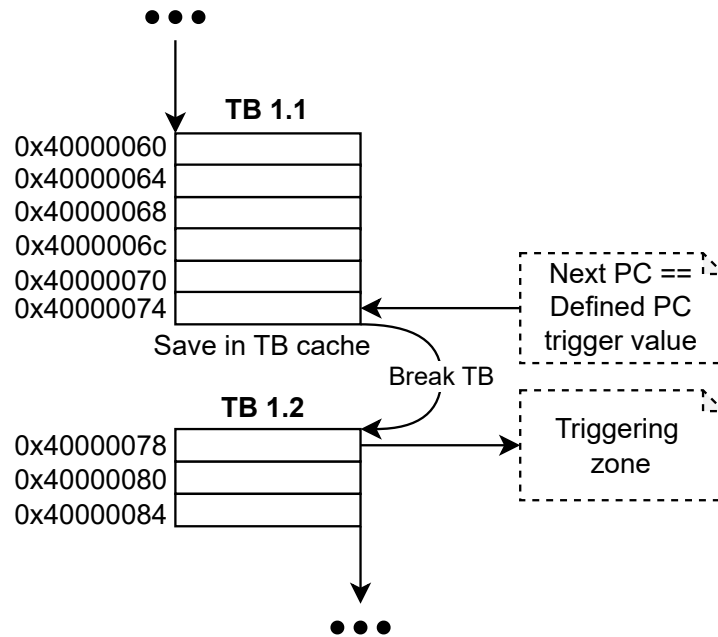


Figure 3.6.4: Procedure of triggering and TB breaking for a defined PC address.

For transient faults with an duration index of one, this alone would be sufficient. However, for models that require intercepting a given TB multiple times, additional mechanisms are needed. Two alternative mechanisms were considered to support this behavior: (i) manipulating the TCG execution flow or (ii) intercepting host code execution.

For the first type, the retranslation method presented in Figure 3.5.8 is used. To enable this mechanism, a breakpoint with the callback *pc_triggering_cb* is placed in QEMU's translation phase. As illustrated in Figure 3.6.5, this callback is responsible for modifying QEMU internal variables according to the parameters defined in the XML fault library.

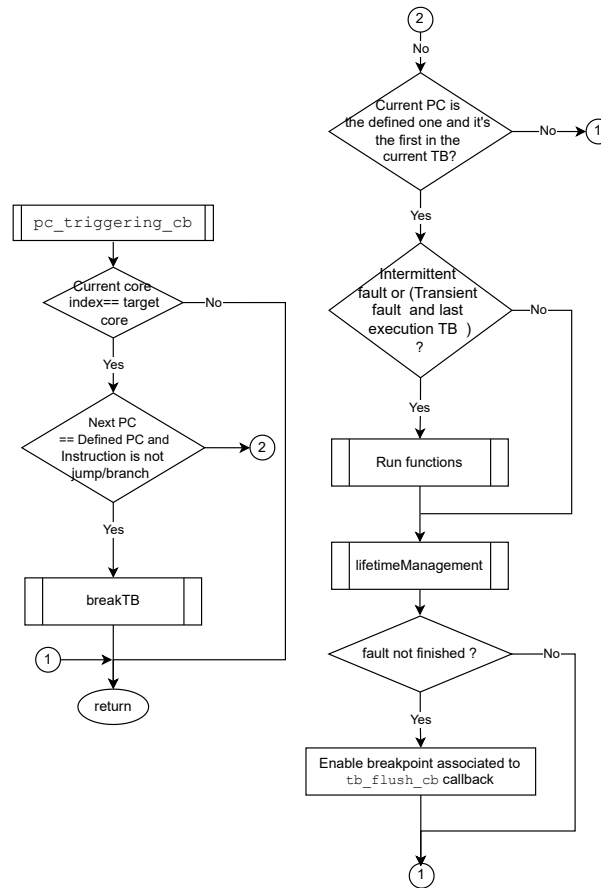


Figure 3.6.5: Approach of TB flushing to perform triggering in a repetitive PC address.

First, the current core index is checked to verify whether it matches the target core. Then, the PC value is checked and, when the next address corresponds to the defined trigger value, the TB breaking mechanism is activated, causing the original TB to be split into TB 1.1 and TB 1.2. Afterward, TB 1.2, which starts with the targeted PC value, is intersected to evaluate the duration type and the duration index in order to determine the correct behavior.

For intermittent faults, the collecting and injecting functions are executed repeatedly until the number of executions of TB 1.2 matches the duration index value. In contrast, for transient faults, these functions are executed only in the last reach of TB 1.2, when the number of executions of this TB is equal to the duration index, as demonstrated in Figure 3.6.1.

In both cases, the breakpoint associated with the *tb_flush_cb* callback is enabled, allowing the

retranslation and repeated intersection of the target TB until the fault model completes its lifetime. The *lifetimeManagement* function, shown in Figure 3.6.2, manages this process by removing the specific fault when its model finishes. While not concluded, the *tb_flush_cb* breakpoint remains enabled to flush the TB after each generation, allowing a new TB with the same properties as TB 1.2 to be generated and intersected again. Once the model is completed, this breakpoint is removed to avoid unnecessary overhead.

In the second approach (intercepting host code execution) elaborated in Figure 3.6.6, the stop condition is placed directly in the host code, using a callback to perform the required decisions, as shown in Figure 3.5.10. Initially, a breakpoint is placed in the translation phase (just as in the previous approach), and the PC address is monitored until the TB breaking mechanism is triggered.

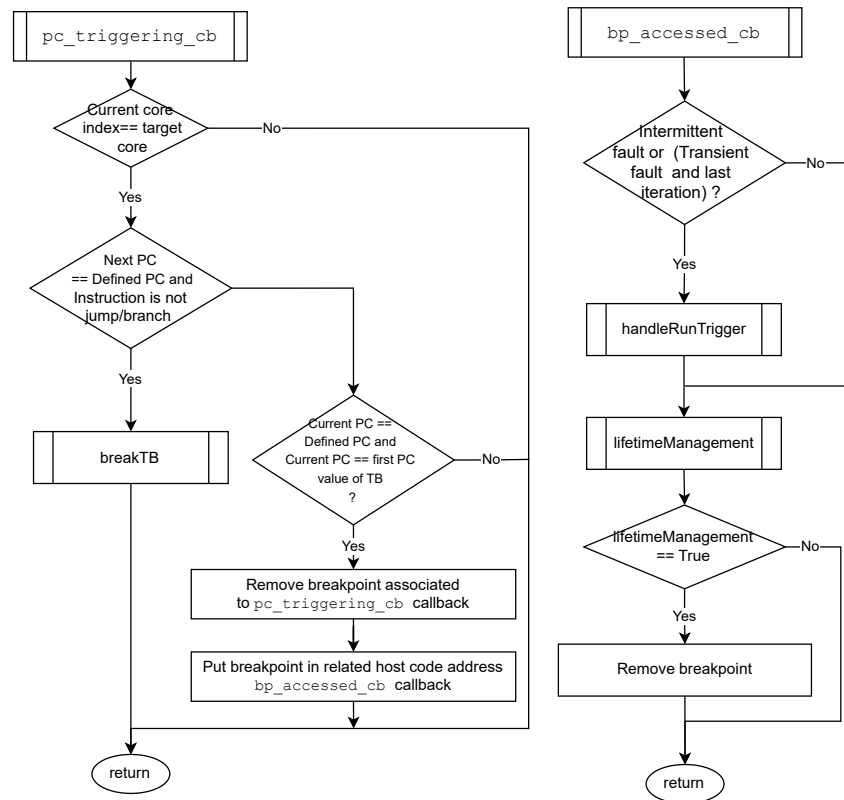


Figure 3.6.6: PC triggering callback flowchart for the host code breakpoint placing approach.

The main difference is that, after this process, the breakpoint in the translation phase is disabled if no other PC remains in the queue. A new breakpoint is placed in the host code corresponding to the second part of the split TB, instead of repeatedly flushing the TB until the model completes. The *bp_accessed_cb* callback performs the same operations, namely the execution of the *handleRunTrigger* routine (in Figure 3.3.2 flowchart) and the invocation of the *lifetimeManagement* routine (present in Figure 3.6.2 flowchart). Once the fault model has finished, the breakpoint placed in the host code is

removed.

Theoretically, the second approach should be more efficient, as it eliminates the need to constantly check the PC for the flushing decision and also bypasses the requirement for code retranslation. However, is not compatible with every case, in contrast to the flush (first) approach.

In Figure 3.6.7, a corner case that may lead to incorrect triggering behavior is illustrated. In this example, a TB is split into two parts: a first TB (shown in red) starting at PC address 0x40000328, and a second TB (shown in green) starting at PC address 0x4000034c. A breakpoint is placed on the host code corresponding to this second TB, allowing the trigger to be activated whenever region 2 of the *code_gen_buffer* is executed. However, if the branch instruction at address 0x40000360 is taken, QEMU generates a new TB whose initial PC is 0x40000340. This occurs because QEMU's TB cache lookup considers only the first PC address of each TB, ignoring the remaining instruction addresses within the block. Since no TB starting at 0x40000340 exists in the cache, a new TB is translated, and the instructions originally present in the green TB are retranslated into a different region of the code generation buffer (region 3). As a result, a breakpoint placed exclusively in region 2 will not intercept all executions of the target instruction sequence. Consequently, some executions associated with the intended PC value may bypass the triggering mechanism, leading to incomplete or incorrect fault injection behavior. Both approaches were implemented, leaving the choice to the user.

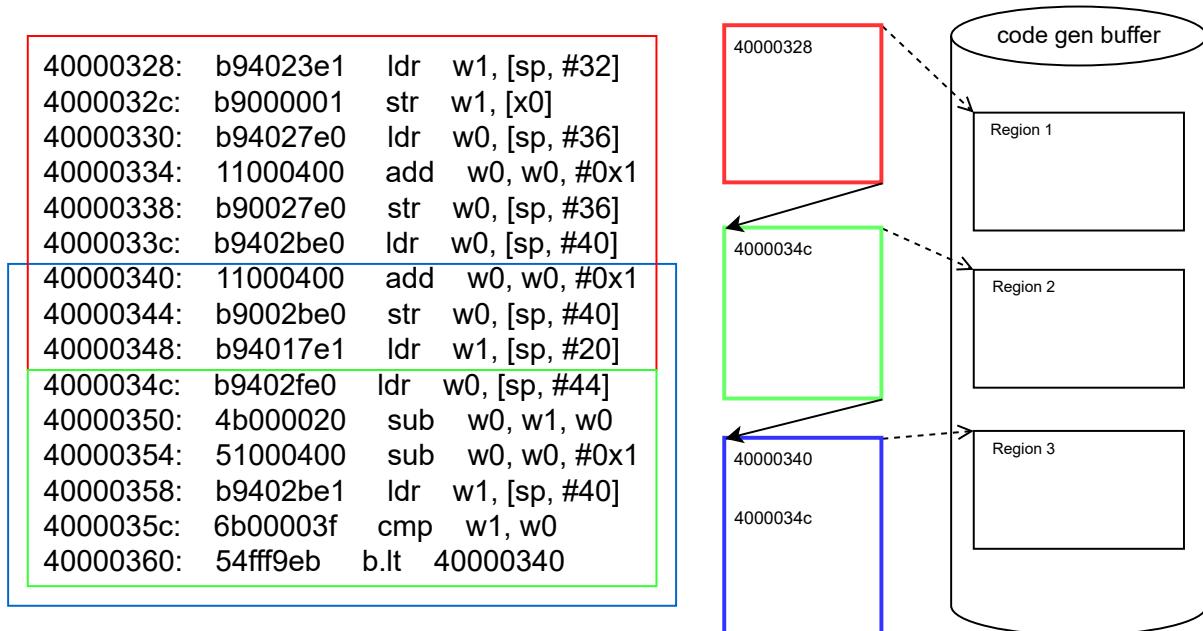


Figure 3.6.7: Corner case in PC triggering for the host code breakpoint placing approach.

To optimize the faulty run, an approach was adopted to bypass the verification instruction by instruction until the TB with the targeted PC address is found, reducing the execution stop moments. For this, the

framework callback (*intersect_tb_cb*) is associated with a breakpoint in the *cpu_exec_loop* routine, where QEMU does the lookup for the actual TB in the cache by using the initial PC value.

In the golden run, in the moment of triggering, it is possible to access the first PC value of the TB. By storing this value, the framework can later use it as a reference during the faulty run. Instead of stopping execution at every instruction translation to check whether the triggering PC has been reached, this optimization allows the framework to enable translation-level interception only when the TB whose initial PC matches the stored value is about to be generated. As a result, unnecessary stop events are avoided, significantly reducing execution overhead during the faulty run.

In Figure 3.6.8, the example illustrates a fault being injected when the PC reaches address 0x40000078. When the break happens in the specified TB 1 during the golden run, the value of its first PC, which is 0x40000060, is saved in the optimization XML. Finally, in the faulty run, the PC associated with the future TB generation is verified outside of *translator_loop* before translation, and only when this stored value matches the current one, the breakpoint associated with *pc_triggering_cb* is enabled. It's also important to understand that in the faulty run, there are fewer break conditions since in the faulty run, the iterations at the translation level (green arrows) in TB 0 are bypassed with only one iteration at the CPU execute level (blue arrow).

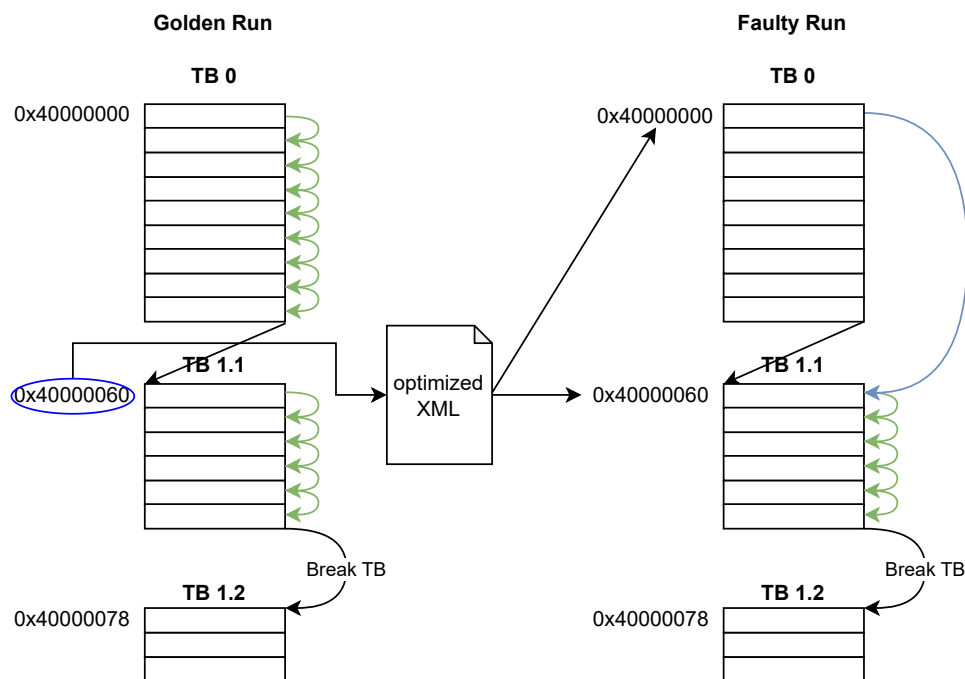


Figure 3.6.8: PC triggering optimization overview.

b) Instruction triggering

To perform triggering based on instruction count, the framework maintains a counter that is incremented after each guest instruction is translated. Similar to PC triggering, the TB is broken once a predefined threshold is reached. In this case, the breaking occurs at the instruction count minus one, ensuring that the target instruction appears at the beginning of the subsequent TB. However, this triggering mechanism has a key difference. In PC triggering, flushing is restricted to the TB whose initial PC matches the trigger value, while all other TBs are ignored. In contrast, instruction triggering requires a more conservative strategy, in which TBs are flushed sequentially as execution progresses. This behavior is necessary to ensure that all possible execution paths are intersected until the trigger condition is satisfied, although it results in additional retranslation overhead. As a consequence, subsequent TBs are verified instruction by instruction until the condition is met.

As demonstrated in Figure 3.6.9, the triggering threshold in this example is set to 25 instructions, and TB flushing continues until the target instruction count is reached. Until the desired value is attained, all generated TBs are immediately flushed, activating the mechanism that splits the TB into two parts at the correct moment.

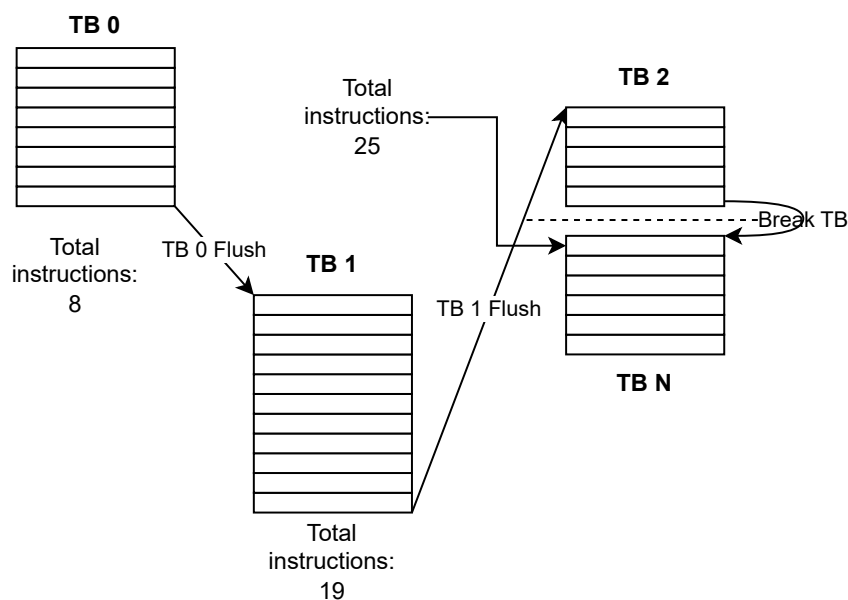


Figure 3.6.9: Triggering backend for 25 instructions count.

Since the framework supports hypervisor emulation, instruction triggering can be configured to count either application-level or hypervisor-level instructions. To minimize overhead, the framework flushes only the TBs associated with the execution level defined by the user. In Figure 3.6.10, an example is shown in

which triggering occurs when the instruction counter reaches 15 instructions at the application level.

When control transfers between TBs, if the previous TB corresponds to application-level execution (shown in red), it is flushed, and the instruction counter is incremented accordingly. Conversely, when the processor is executing at the hypervisor level (shown in blue), the flush operation is ignored and the corresponding instructions are not counted toward the instruction counter. The TB breaking mechanism is triggered when the instruction counter is one unit below the defined threshold. As a result, the target instruction appears at the beginning of the subsequent TB, satisfying the requirements for both the collector and injector to operate correctly.

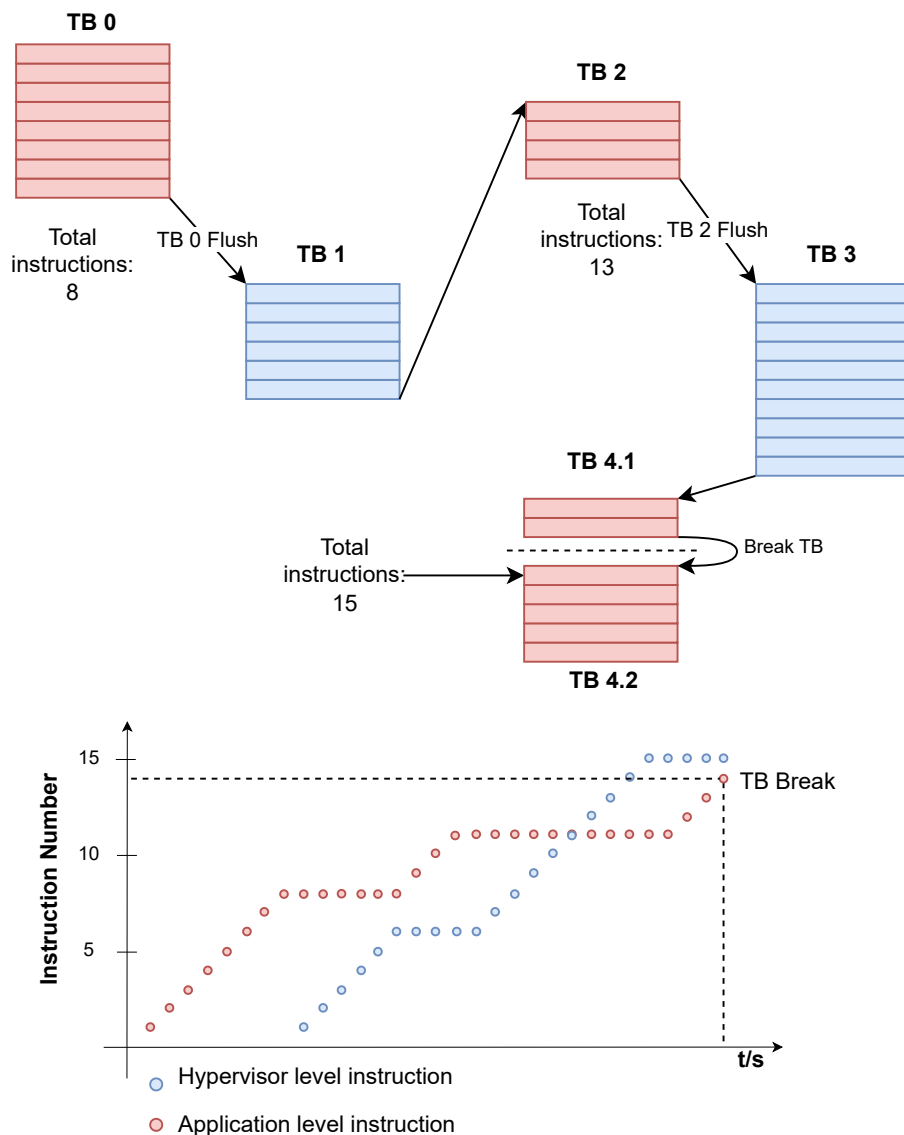


Figure 3.6.10: TB flush conditions between application and hypervisor level.

To apply the cases mentioned above, the framework's callback *instruction_triggering_cb* is used. The breakpoint associated with this callback is placed in QEMU's *translator_loop* routine, after

instruction decoding, like the PC triggering (*pc_triggering_cb*) approach presented in Figure 3.6.5. The flowchart shown in Figure 3.6.11 illustrates the conditions governing the TB breaking and TB flushing procedures. First, the callback evaluates whether the current instruction belongs to the application or hypervisor execution level. If the instruction is associated with the application level, the instruction counter is incremented, and the TB flush callback is enabled. Subsequent conditions then determine whether the TB should be broken or whether the *handleRunTrigger* routine (in Figure 3.3.2 flowchart) should be executed.

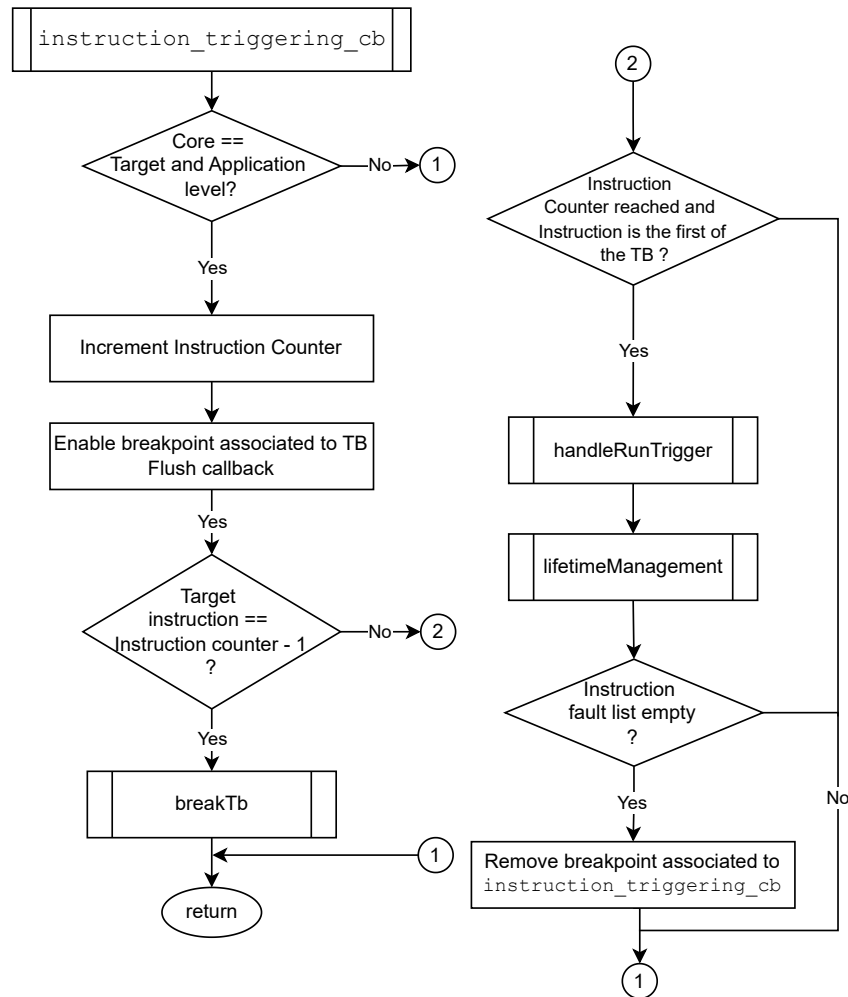


Figure 3.6.11: Instruction triggering callback flowchart.

c) Access triggering

This trigger type is activated whenever a specific register or a defined memory location is accessed. For this model, a user-defined PC value acts as a start point. After this PC is reached, register or memory accesses are monitored to activate the trigger, as shown in Figure 3.6.12. Here the R0 access are taken

into consideration only after the PC value is reached. This behavior is achieved by placing a watchpoint on the corresponding address element within the *CPUArchState* struct. By specifying the PC value at which execution is intercepted, this mechanism provides increased controllability while introducing lower execution overhead, as it avoids unnecessary stop events that would otherwise occur before the CPU reaches the intended fault injection point.

Since write operations are sufficient to cover all relevant fault injection scenarios, the watchpoint is configured to trigger the mechanism only on write accesses. Monitoring read operations is, therefore, unnecessary and would only introduce additional overhead. This restriction further reduces spurious activations by avoiding fault injections that could later be overwritten or masked by subsequent instructions.

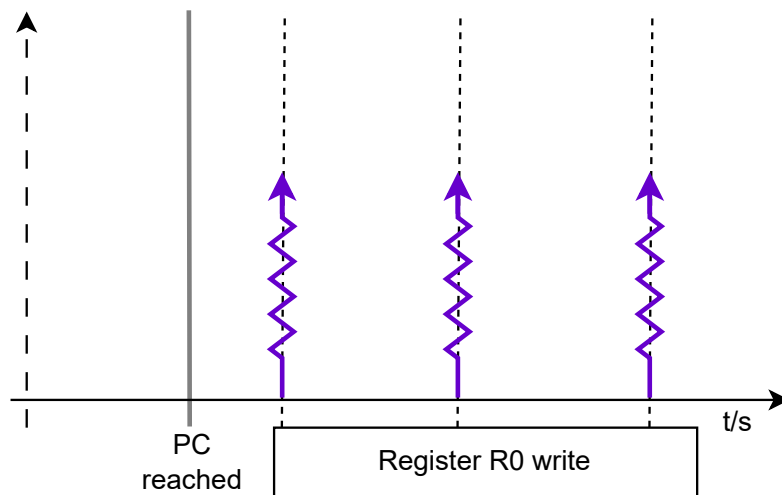


Figure 3.6.12: Access trigger type example.

This trigger type operates at the host code execution level, independently of the underlying translation process. As a result, it does not require TB intersection or flushing mechanisms, making it a less complex approach. However, as discussed in Section 3.5, even when the executable passed to QEMU is compiled without optimizations, the TCG still applies optimization passes during translation. Consequently, the number of register accesses in the target code may not directly correspond to the number of accesses observed in the generated host code.

This triggering mechanism is composed of two steps. The first step corresponds to the *access_triggering_cb* callback, which verifies the PC value and, once it reaches the user-defined trigger point, places a watchpoint on the appropriate address component, as illustrated in the left flowchart of Figure 3.6.13. In terms of PC verification logic, this callback behaves similarly to the behaves similarly to

the logic used on the PC triggering (with the *pc_triggering_cb* routine). The second step involves the *target_accessed_cb* callback. This is invoked every time the defined watchpoint is accessed. At this stage, the *handleRunTrigger* routine (in Figure 3.3.2 flowchart) is executed, fault injection is performed, and breakpoint management is handled, similarly to the other trigger types.

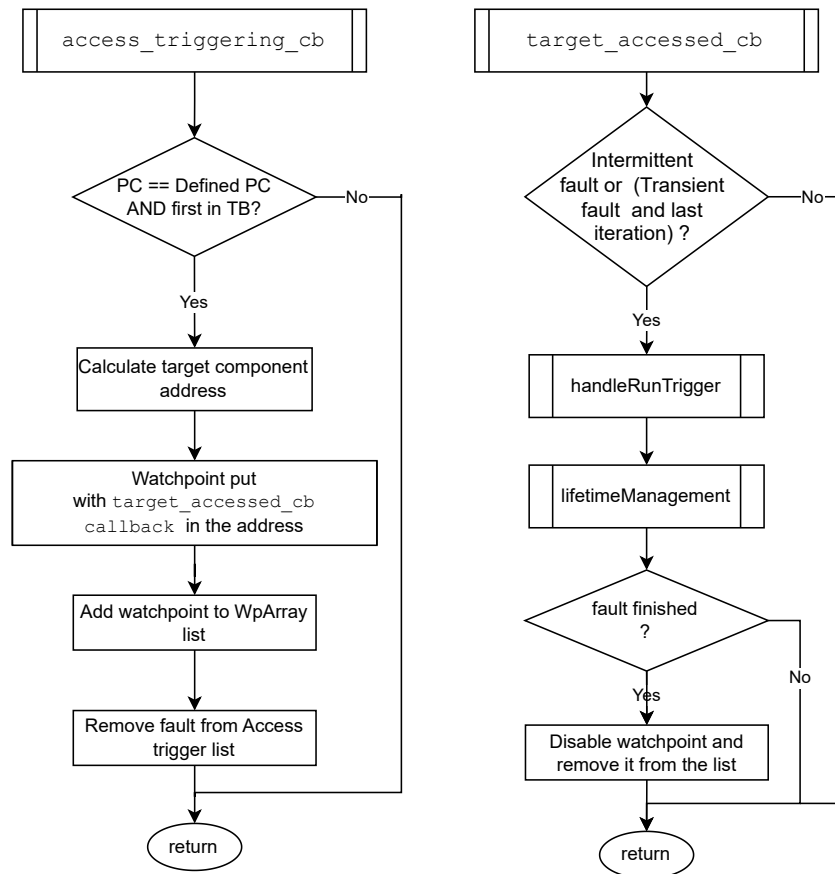


Figure 3.6.13: Access trigger mechanism flowchart.

d) Time triggering

Within the time-triggered approach, QEMU's virtual clock (*icount*) is used. In this context, the user defines a period in nanoseconds, assuming that each nanosecond corresponds to the execution of a single instruction. This trigger type is therefore similar to instruction triggering, as it is effectively based on the number of executed instructions. The difference is that time triggering approach relies on a global virtual clock shared across all cores. As a result, it is not possible to distinguish the number of instructions executed by individual cores, in contrast to the instruction triggering method.

Following this, a breakpoint associated with the timer callback (*timer_triggering_cb*) is initially placed in the *cpu_exec_loop* after TB execution, as shown in Figure 3.6.14. At this stage, TBs are flushed

sequentially, and the virtual clock value is continuously monitored in each CPU loop iteration. Once the virtual clock approaches the defined period, evaluated with a margin defined by *ERROR_TUNING*, a new breakpoint associated with the *catch_instruction_cb* callback is enabled at the translation level, at the same interception point used by the *pc_triggering_cb* described previously. From this point onward, the number of instructions translated within the current TB is monitored, and when the sum of these instructions and the last value of the virtual time clock period equals the defined period, the TB is broken. After this operation, the breakpoint containing *catch_instruction_cb* callback is disabled.

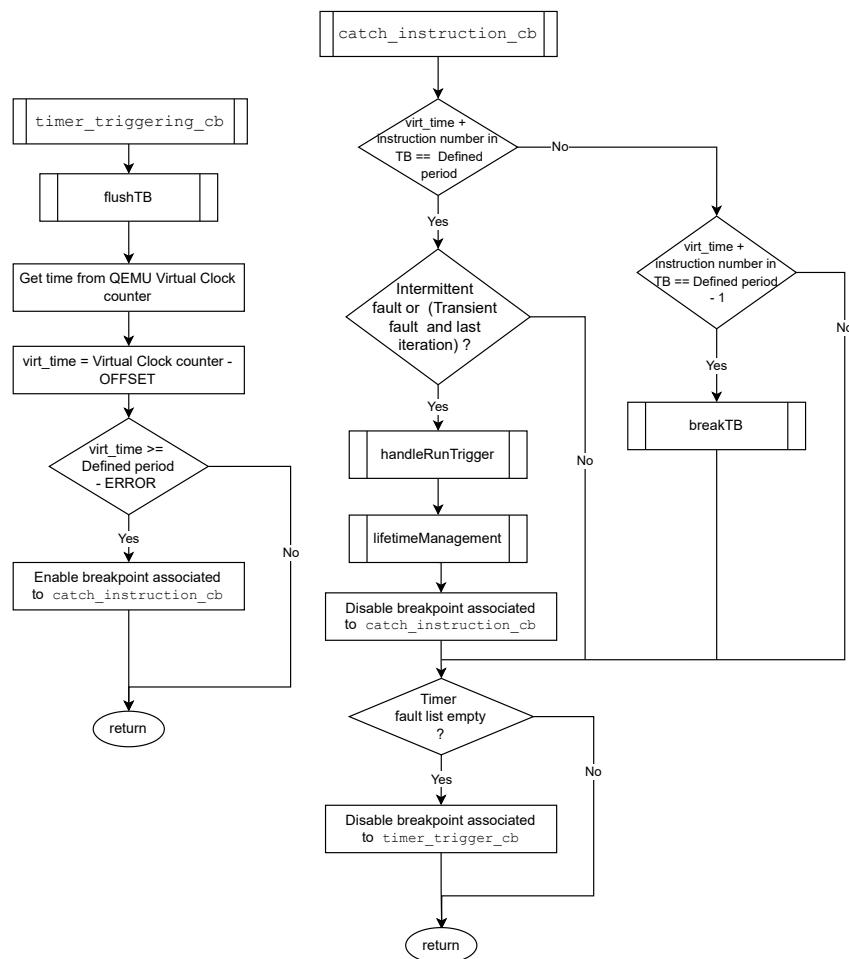


Figure 3.6.14: Callbacks used in the timer trigger type.

To avoid unnecessary overhead, a minimum value for *ERROR_TUNING* is required. In the example shown in Figure 3.6.15, a fault injection with a triggering period of 16 nanoseconds is considered. The virtual clock is updated after each TB execution. As observed, the virtual clock value immediately before TB_T , which spans the interval from the trigger moment to the target moment, is 9 ns. To satisfy the required condition, the minimum value of *ERROR_TUNING* must be 7, ensuring that the *catch_instruction_cb* callback is enabled before the translation of TB_T . If the *ERROR_TUNING* value is

greater than 13 and less than 16, the callback is enabled before TB_1 , resulting in increased overhead.

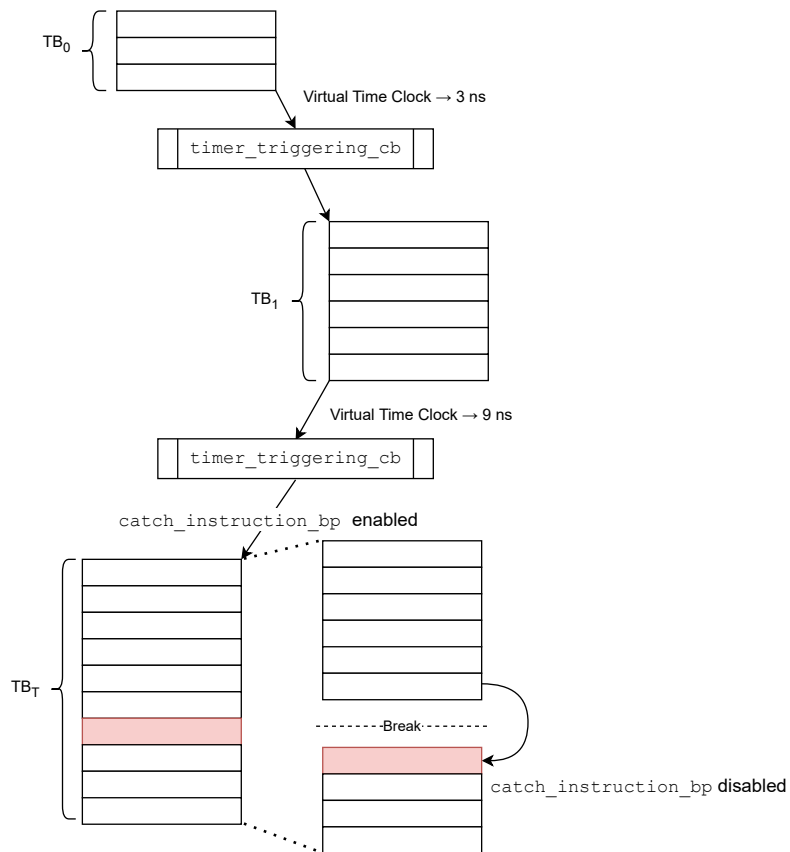


Figure 3.6.15: `catch_instruction_cb` enabler mechanism.

In Figure 3.6.16, the activation of the `catch_instruction_cb` callback is illustrated for different error values using the same example shown in Figure 3.6.15, with green dots indicating the activation points. The blue lines represent the instructions that are intersected until the trigger moment is reached. As the number of intersected instructions increases, the execution overhead also increases. The figure further shows that if the error value is below the minimum threshold (sample 1), the breakpoint in the translation phase is not activated and the trigger does not occur, leading to incorrect behavior.

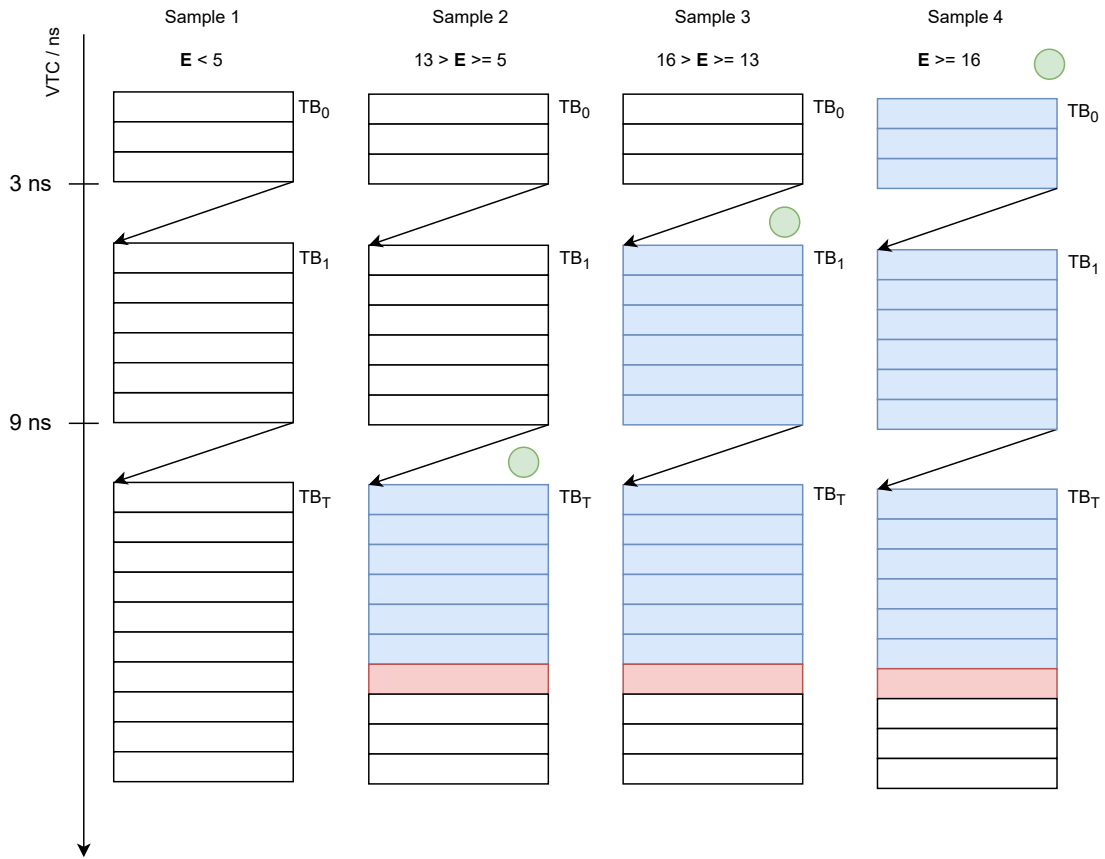


Figure 3.6.16: `catch_instruction_cb` callback enabling for different values of ERROR TUNING.

The best option would be attributing an error in the limit presented in sample 2. However, this condition is dependent on the last virtual clock value and the position where the trigger moment is placed in the next TB. For example, in the same case, if the defined period is 19, here the *ERROR_TUNING* has to be at least 11. Since it is not possible to predict the optimal error value in advance, and given that QEMU limits each translation block to a maximum of 255 instructions, this value is selected as the error threshold. This choice ensures correctness at the expense of a bounded and predictable overhead.

A closer analysis of the flow reveals that enabling the *catch_instruction_cb* callback immediately before the translation of the target TB would further improve performance. To fulfill this nuance, an optimization was also added in this trigger type. It consists of saving the first PC value of the affected TB and the *icount* value before entering it, during the golden run. Then, in the faulty run, the TBs are intersected, and when those values are reached, the *catch_instruction_cb* is enabled, avoiding that unnecessary instruction being intersected. This approach is similar to the PC triggering type optimization presented in Figure 3.6.7.

e) Device Triggering

In the case of a peripheral trigger, QEMU internally relies on `int_st_mmio_leN` routine to perform read and write operations on memory-mapped I/O regions. In Figure 3.6.17, the callback is responsible for intercepting and applying the faults during the execution of this function. With this approach, it is possible to inject faults without requiring TB intersection or flushing mechanisms. The user is required to define the target MMIO address and the fault model on the XML descriptor file. By stopping execution at the beginning of this function and modifying its parameters, the respective effects can be emulated. Since this mechanism is activated exclusively on MMIO accesses, the overhead introduced by the framework is lower than that of other trigger types.

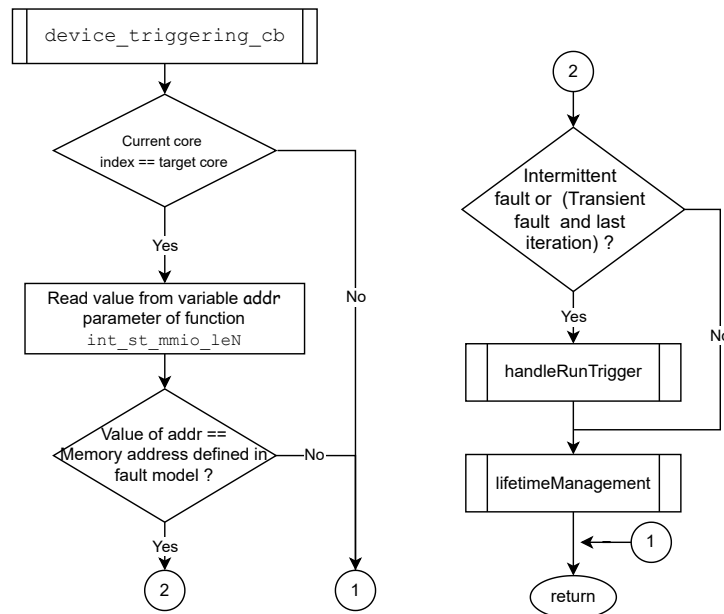


Figure 3.6.17: Peripheral triggering callback flowchart.

3.7 Data collector

The data collector gathers execution data from both golden and faulty runs, enabling a detailed analysis of fault effects across different system components. It traces and records logs containing register values, the PC, and memory contents. The memory region to be traced is defined by the user through the initial configuration parameters, using user-specified lower and upper memory addresses. After the triggering mechanisms are satisfied, additional steps are required to perform data collection. The collector generates a log entry at each trigger point, allowing the outcome associated with each injected fault to be monitored.

Figure 3.7.1 illustrates how logs are retrieved throughout execution. Instead of evaluating only the

final output at the end of the run, logs are extracted at multiple points during its execution.

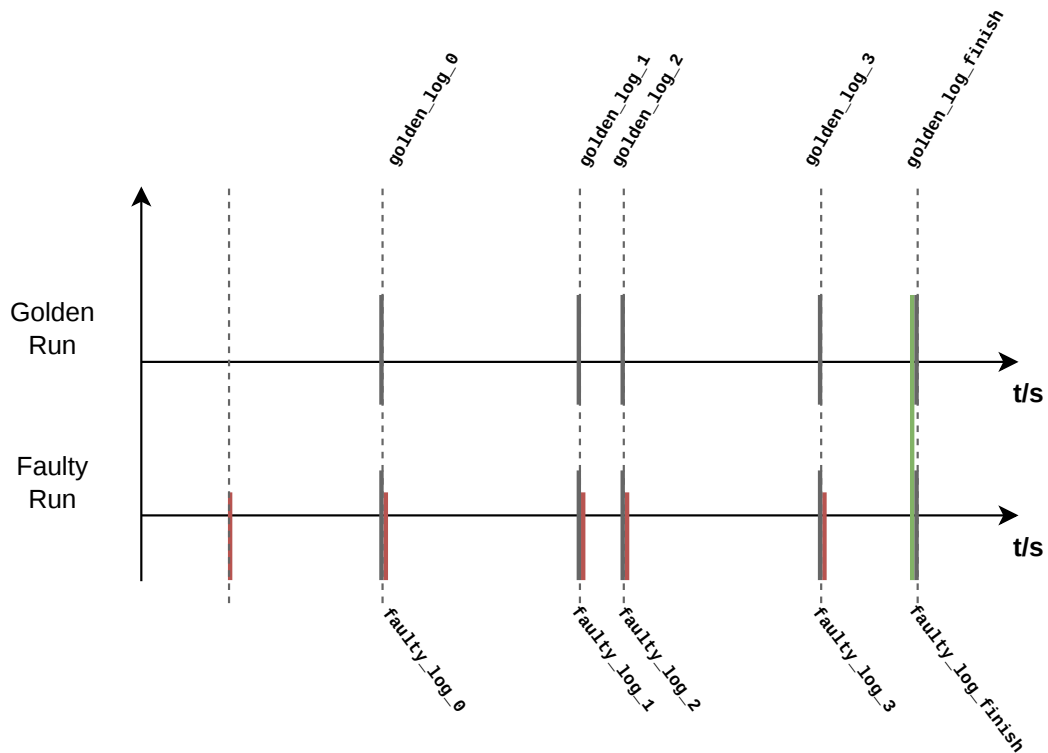


Figure 3.7.1: Collecting logs in golden and faulty run example.

The red traces in Figure 3.7.1 represent fault injection moments, while the dark gray traces correspond to the extracted logs. The end of the run is indicated in green. Logs are labeled according to the run type and their corresponding identifier. To ensure a correct analysis, log collection must be synchronized across both runs. Consequently, during the comparison phase in the faulty run, each log is compared against its corresponding reference log from the golden run.

At the first trigger point, no log is retrieved because fault injection occurs only after the logging. As a result, the comparison of CPU and memory states in faulty and golden run at this position will always have identical values. Depending on the run type, all collected data is stored in either a *golden_log_n.log* or a *faulty_log_n.log* file, where *n* denotes the log index. An example of the resulting log format is shown in Listing 3.8. In this example, the user defined 0x50000000 as the *LOW_ADDRESS* and 0x5000000c as the *HIGH_ADDRESS*

Listing 3.8: CPU and Memory Log

```

PC :
REG 0:
    ...
REG 31:
0x50000000:
    ...
0x5000000c:

```

This approach, which records execution logs at multiple points during execution, is beneficial because it enables a more precise identification of code regions that are prone to vulnerabilities, particularly in scenarios where multiple faulty behaviors are defined.

The collector exhibits different behaviors depending on the run type. During the golden run, the routine illustrated by the flowchart in Figure 3.7.2 is executed to collect diagnostic data from the program counter, registers, and memory. This is accomplished by reading the corresponding values from the *CPUArchState* struct and from physical memory.

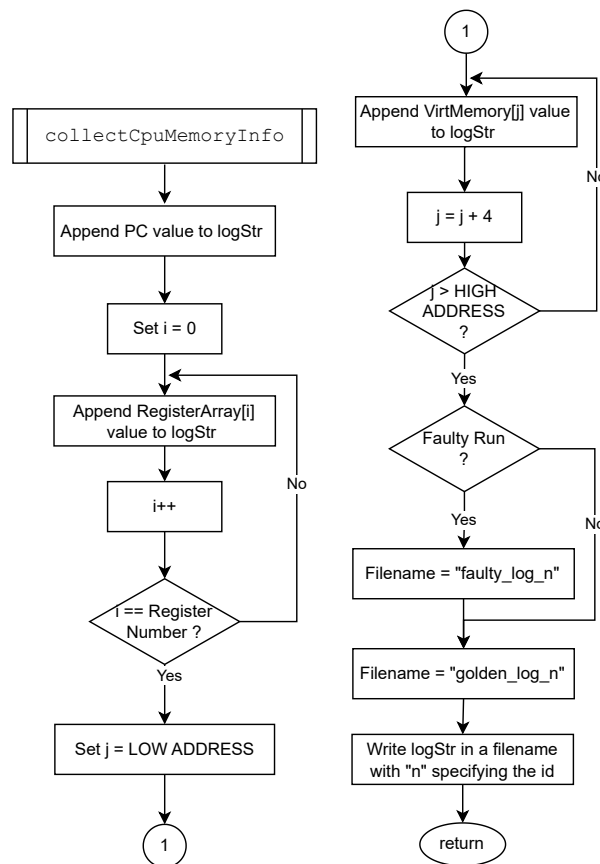


Figure 3.7.2: Info collect function flowchart.

Then, in the faulty run, besides logging the values, a comparison is carried out against the reference logs from the golden run, as shown in Figure 3.7.3. The process begins by checking the PC value. If it differs from the golden run, a control-flow deviation is identified, indicating an unrecoverable error. If the PC matches, the registers and memory region are retrieved and compared. At this stage, two outcomes are possible: either the effect is masked (values are equivalent) or silent data corruption occurs (values differ).

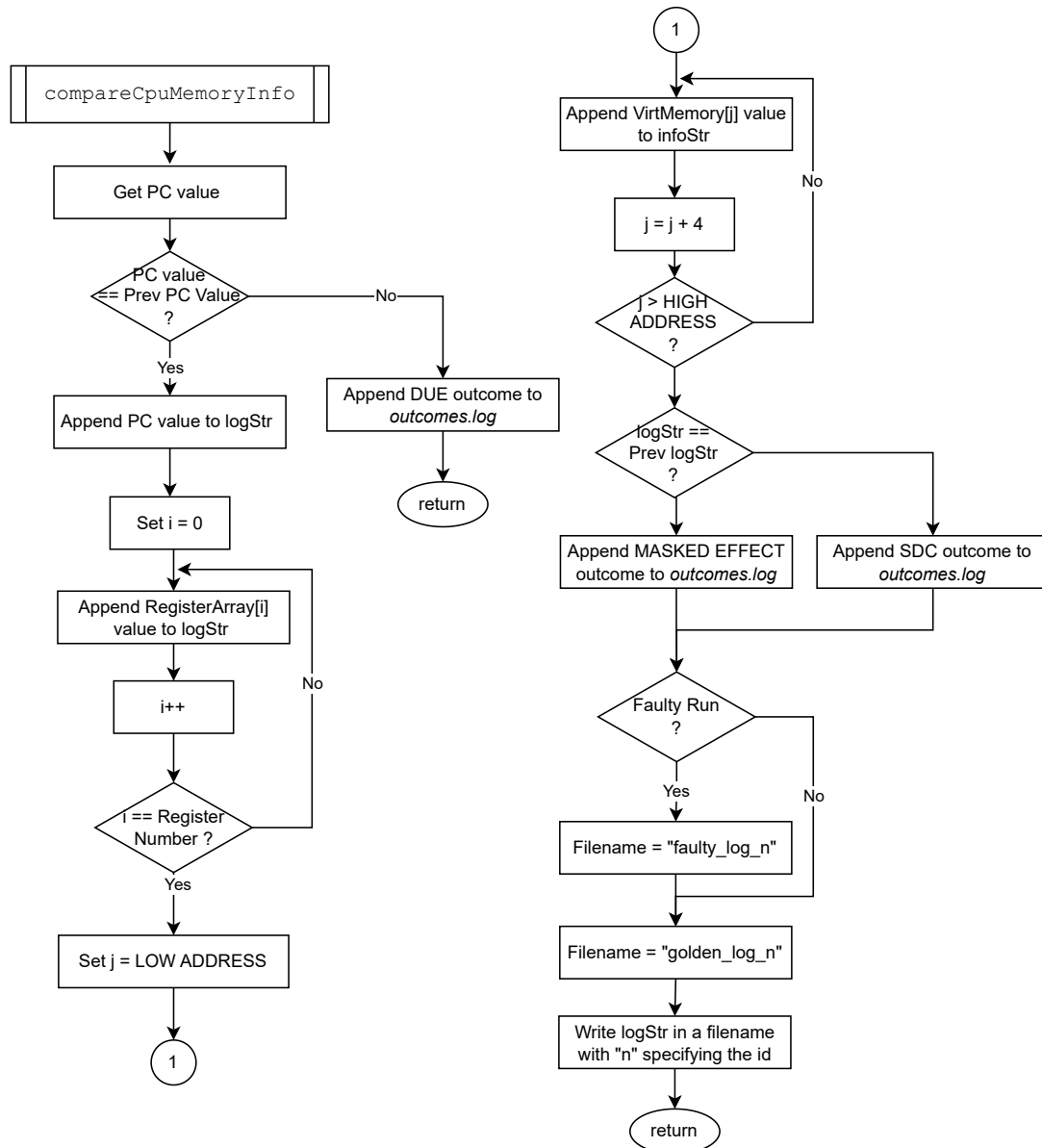


Figure 3.7.3: Log comparator function flowchart.

4 Evaluation

In this section, several experimental runs were performed to evaluate the operation and performance of the proposed framework. The evaluation is divided into two parts. The first corresponds to the validation phase, which demonstrates the correct operation of the framework components and the mechanisms used to support the collection and injection during the emulation. The second part focuses on performance evaluation, overhead analysis, and performance degradation under different workloads and conditions, while motivating the design decisions adopted in the implementation.

4.1 Validation

This section presents the results validating the correct operation of the fault injection process, the triggering logic, and the mechanisms used to support the precise fault injection. The injection tests demonstrate the correct modification of memory and register values, as well as the correct behavior of the triggering logic according to user-defined parameters. To justify the need for the different TB manipulation mechanisms (flush, break, and cache flushing), a set of demonstrations in different workloads was conducted with each mechanism selectively enabled and disabled.

4.1.1 Precise Triggering Tests

As documented in section 3.1, the proposed framework supports different trigger types that can be used to perform fault injection and collect CPU and memory data at distinct points during execution. Those are defined in the XML descriptor file by the user. One of the main goals of this framework is to deliver control and precision, and reproducible fault injection.

The experiments presented in this section aim to demonstrate the synchronization between different trigger types. Consider an example program in which instruction 10 is located at address 0x4000040. If, in one run, the user defines the injection to be triggered at the PC address 0x4000040 and, in another run, defines it to be triggered at the tenth instruction moment, the injection must occur at the same execution point in both runs. Timer triggering was not explicitly evaluated in this synchronization analysis, since the *icount* configuration ensures that each timer unit corresponds directly to one executed instruction. Therefore, timer triggering is inherently synchronized with instruction triggering under the adopted experimental setup.

To perform these results, the steps in Figure 4.1.1 will be followed. In a first golden run, a PC triggering definition is applied. During this run, the triggering moments are tracked, and the number of instructions

executed before each triggering point is recorded. Then, in a subsequent golden run, instruction triggering is configured using the recorded instruction counts. After tracking, the PC value at each triggering moment in the second run should match the one defined in the first golden run. Since the triggering behavior is identical in golden and faulty runs, the triggering tests were performed only for the golden run.

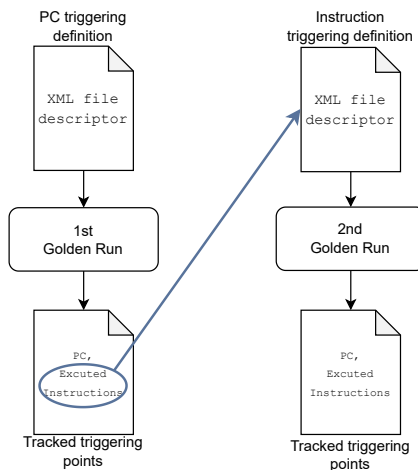


Figure 4.1.1: Description of the steps for triggering precision evaluation.

To evaluate this, a trace of the trigger events defined in the XML was collected during a golden run. This trace lists each moment at which the triggering mechanism is activated, providing information about the TB address, the initial PC address, the number of executed instructions, and the trigger type. The selected algorithm was a bubble sort operating on 400 elements. Consider the XML file in Listing 4.9.

Listing 4.9: PC triggering configuration

```

<injection>
  <fault>
    <trigger>
      <trigger_type>ProgramCounter</trigger_type>
      <trigger_parameter>0x4000037c</trigger_parameter>
    </trigger>
    <model>
      <duration>Intermittent</duration>
      <duration_index>5</duration_index>
    </model>
  </fault>
</injection>

```

For this run, when the PC address reaches the address 0x4000037c for the first five occurrences, register and memory collection, as well as tracing, are performed. Since this test focuses solely on trigger operation, the mask type and target parameters were disregarded. In Listing 4.10, the obtained trace, which contains the TB address value, the PC address, the instructions executed, and the trigger type performed at the trigger moment, is presented. It was observed that, for each instance in which the PC reached the address 0x4000037c, the number of executed instructions at that moment was 99, 149, 199, 249, and 299.

Listing 4.10: Tracing output of PC triggering moments in golden run

0x7fffb0001ac0:0x4000037c	Insns: 99	(PC TRIGGER)
0x7fffb0001dc0:0x4000037c	Insns: 149	(PC TRIGGER)
0x7fffb0001f40:0x4000037c	Insns: 199	(PC TRIGGER)
0x7fffb00020c0:0x4000037c	Insns: 249	(PC TRIGGER)
0x7fffb0002240:0x4000037c	Insns: 299	(PC TRIGGER)

To prove that, for different trigger types, the injection point remains the same, in the next golden run, the numbers of executed instructions obtained in Listing 4.10 (99, 149, 199, 249, and 299) were passed as parameters to a new XML configuration using instruction triggering. By using these values, it is expected that the resulting triggering moments will be identical to those previously obtained, with the PC reaching the address 0x4000037c at each trigger. The Listing 4.11 presents the XML descriptor file for the instruction triggering. The five triggers were specified as transient.

Listing 4.11: Instruction Triggering configuration

```

<fault>
  <trigger>
    <trigger_type>InstructionNo</trigger_type>
    <trigger_parameter>99</trigger_parameter>
    <trigger_parameter>149</trigger_parameter>
    <trigger_parameter>199</trigger_parameter>
    <trigger_parameter>149</trigger_parameter>
    <trigger_parameter>299</trigger_parameter>
    <duration>Transient</duration>
    ...
  </trigger>
</fault>

```

After this run, the log shown in Listing 4.12 was obtained. It can be observed that, compared to the results obtained in Listing 4.10, the triggering takes place at PC address 0x4000037c.

Listing 4.12: Tracing output of Instruction triggering moments in golden run

0x7fffb00023c0:0x4000037c	Insns: 99	(INSTRUCTION TRIGGER)
0x7fffb0003340:0x4000037c	Insns: 149	(INSTRUCTION TRIGGER)
0x7fffb00042c0:0x4000037c	Insns: 199	(INSTRUCTION TRIGGER)
0x7fffb0005240:0x4000037c	Insns: 249	(INSTRUCTION TRIGGER)
0x7fffb00061c0:0x4000037c	Insns: 299	(INSTRUCTION TRIGGER)

With these results, it is possible to conclude that, for different trigger types, when the parameters are configured so that the trigger takes place at the same execution moment, that moment is consistently reached without any deviation. With this, the precision and reproducibility of the proposed triggering approaches are validated.

4.1.2 Injection Tests

To perform error emulation, it is necessary to pause QEMU at specific points and modify the contents of the static CPU structures (*CPUARMState* or *CPURISCVState*) or the physical memory. This framework supports fault injection in both registers and memory, which may lead to faulty outputs and behaviors.

To validate the injection mechanism, two faulty runs were performed for each architecture (AArch64 and RISC-V): one involving fault injection in a CPU register and another involving fault injection in memory. For this purpose, a simple bare-metal program that prints the message “Hello World!” through the UART interface was executed. Faults were injected at specific execution points, causing the output to be altered. The source code used for these experiments is shown in Listing 4.13.

Listing 4.13: Simple print code on terminal

```
void uart_putc(char c) {
    UART0_DR = c;
}

void uart_puts(const char *str) {
    while (*str) {
        uart_putc(*str++);
    }
}

int main(){
    char* buffer = "Hello World!!\n";
    UART0_FCR = UARTFCR_FFENA;
    uart_puts(buffer);
    __asm__ volatile ("wfi");
}
```

In the first example, the adopted strategy consisted of intercepting the *uart_puts* function and manipulating one of the character values loaded from the buffer by altering the register content. As shown in Figure 4.1.2, a bit-flip fault with mask 0xF was applied in each faulty run, at the moment when the UART sends the character 'W' to the terminal.

By analyzing the assembly code, it is possible to observe that, in the AArch64 case, the instruction at address 0x40000040 stores the value of register x1 into the device memory, whereas in the RISC-V case, the instruction at address 0x80000078 stores the value of register x4. Applying the XML containing the information of the instruction address for PC triggering and the index for the target register, it is possible to cause a faulty behavior in the printing message output in the terminal. Since the 'W' character is seventh to be sent, the *duration_index* parameter holds the value 7. The bit-flip will cause the 'W' character to become 'X'. The output result is shown in Figure 4.1.2.

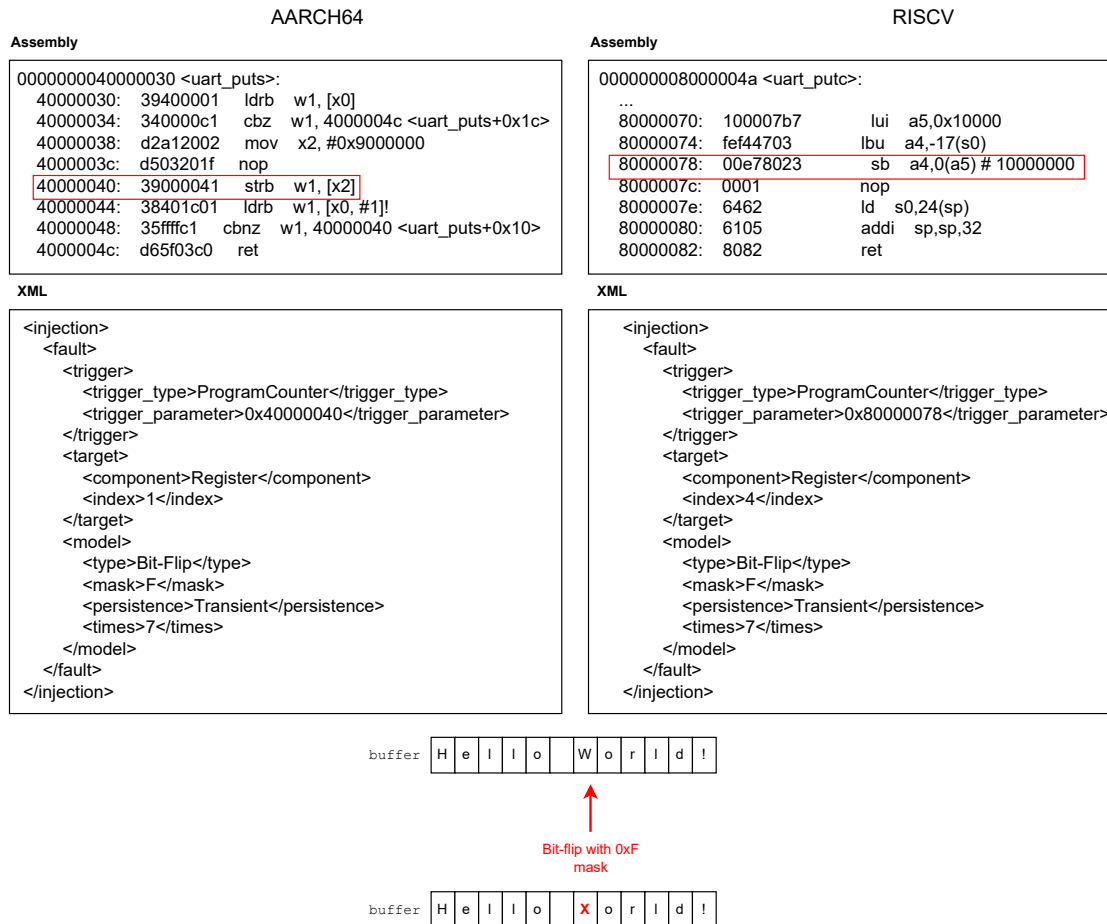


Figure 4.1.2: Fault injection in a register causing bad output on the terminal.

The output result is shown in Figure 4.1.3. This concludes that injecting faults into the register can lead to faulty output, thereby verifying the validity of the proposed approach.

AARCH64

```

Defined Breakpoint at: translator.c:180
Hello Fault Id 0      Bit Flip for bits 0xf in 0x1
Xorld!!

```

RISCV

```

Defined Breakpoint at: translator.c:180
Hello Fault Id 0      Bit Flip for bits 0xf in 0x4
Xorld!!

```

Figure 4.1.3: Output on the terminal after the register fault injection during the UART printing.

In the next example, to validate memory injection, two faulty runs (one for Aarch64 and the other RISC-V) are performed. The fault is injected directly into the buffer that stores the message to be printed. The buffer is located from 0x40000088 to 0x40000094 in the AArch64 virtual memory space and from 0x800000c0 to 0x800000ce in the RISC-V virtual memory space. In Figure 4.1.4, the approach used in each architecture is demonstrated. In contrast to the previous example, in this case, the injection does

not need to be applied at a specific execution moment. The only requirement is that the memory contains the faulty value before the UART starts transmitting the corresponding character. With this in mind, the selected moment for each faulty run was when the PC reached the address corresponding to the first instruction of the *main* function. In the XML descriptor file for each architecture, the virtual memory address containing the 'W' character in the *index* parameter is defined. The fault model type is equal to the last example.

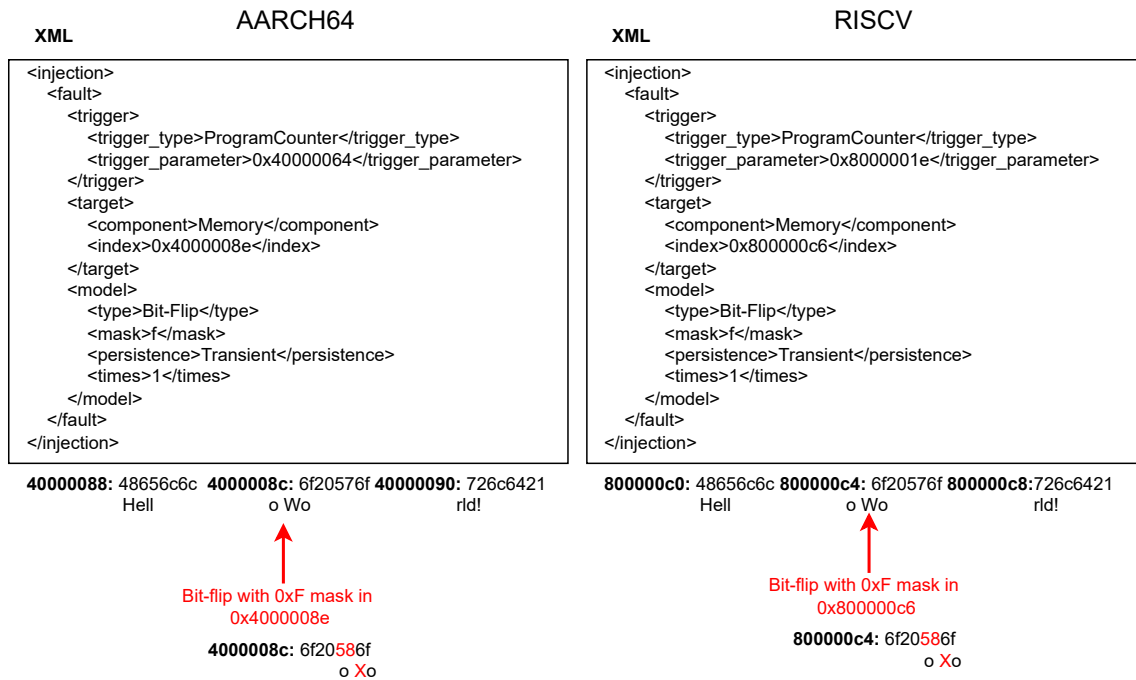


Figure 4.1.4: Fault in a memory causing a faulty output on the terminal.

The output in the terminal is presented in Figure 4.1.5. The result is the same as the register injection demonstration, with the character 'W' changed to 'X'.

```
AARCH64
Defined Breakpoint at: translator.c:180
Fault Id 0      Bit Flip for bits 0xf in 0x4000008e
Hello Xorld!!

RISC-V
Defined Breakpoint at: translator.c:180
Fault Id 0      Bit Flip for bits 0xf in 0x800000c6
Hello Xorld!!
```

Figure 4.1.5: Result in the terminal after memory fault injection.

It is concluded that the injection is being performed correctly for both architectures, targeting both register and memory regions.

4.1.3 TB break mechanism

As discussed in section 3.5 a), modifications in *CPUArchState* or memory do not take effect during the translation phase itself, but only once the corresponding translated code is executed. Consequently, injecting faults directly during translation may lead to premature state modifications that do not accurately reflect the intended execution point. By breaking the TB into two parts and forcing the state where the fault should be injected to appear at the beginning of the second TB, the architectural state is updated only up to that point, ensuring that register and memory values reflect the correct execution context when the injection is performed. To demonstrate the necessity of breaking TB, two golden runs were conducted: one with the mechanism enabled and another without it. The example code shown in Listing 4.14, which performs two MOV operations and one ADD operation, was executed.

Listing 4.14: Simple addition operation between registers

```
0000000040000010 <main>:
    40000010: d2800021  mov x1, #0x1           // #1
    40000014: d2800042  mov x2, #0x2           // #2
    40000018: 8b020023  add x3, x1, x2
    4000001c: 94000001  bl 40000020 <loop>
```

In the first run, the breaking mechanism will remain disabled. In this example, as shown Figure 4.1.6, a fault is injected when PC reaches the 0x40000014 address. When moving from PC 0x40000010 to 0x40000014 during translation, the value of register x1 is masked, since it is only altered when a branch or jump to a new TB occurs. With this in mind, just injecting a fault during instruction translation is not sufficient, as the modified values would be placed at the beginning of the TB, leading to unintended behavior.

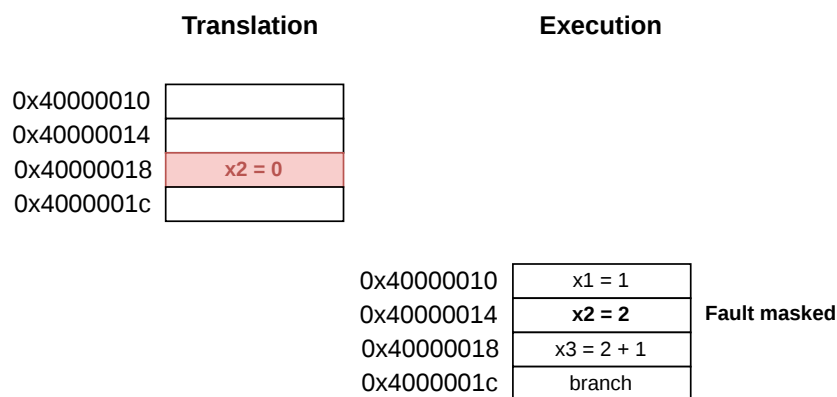


Figure 4.1.6: Behavior of register injection during the translation phase in QEMU standard.

In the second run illustrated in Figure 4.1.7, a break will be applied in 0x40000018, being the instructions of the new TB highlighted in green for this case. Analyzing the register values of this new TB for the same instructions, at the moment of the ADD operation the register is updated. Register x1 holds the value 0, and register x2 holds the value 2, contrary to the last run. Under these conditions, if a fault is injected into either x1 or x2, the resulting value in x3 will differ from that obtained in the normal execution.

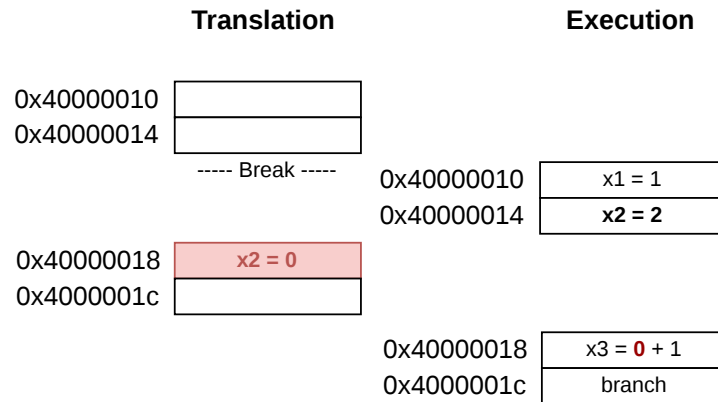


Figure 4.1.7: Behavior of register injection during the translation phase with TB breaking mechanism enabled.

To validate the breaking, Figure 4.1.8 illustrates the QEMU TB logs with the mechanism disabled and enabled. When disabled, the instructions are grouped into a single TB starting at address 0x40000010. When the TB breaking mechanism is enabled, this TB is torn apart, and a new TB is generated starting at address 0x40000018. This behavior confirms that TB breaking is working correctly and allowss to update the CPU registers at any moment of the execution.



Figure 4.1.8: TB tracing in Baseline versus with Breaking Mechanism on.

This behavior confirms that controlled TB breaking allows errors to be injected at arbitrary points within a TB. To validate the injection, the golden and faulty runs were performed for this case. In the faulty, a fault model of a bit-flip was applied in register x2 for the PC previously mentioned. After the execution, Listing 4.1 shows the following results for the golden run (*golden_out_finish.log*) and the faulty run (*faulty_out_finish.log*).

```
PC: 0x40000028
REG 0: 0x48000000
REG 1: 0x1
REG 2: 0x2
REG 3: 0x3
REG 4: 0x0
REG 5: 0x0
...
```

A) Golden Run results

```
PC: 0x40000028
REG 0: 0x48000000
REG 1: 0x1
REG 2: 0x3
REG 3: 0x4
REG 4: 0x0
REG 5: 0x0
...
```

B) Faulty Run results

Listing 4.1: Log outputs after each run (A) *golden_out_finish.log* and (B) *faulty_out_finish.log*.

This behavior confirms that controlled TB breaking together with the fault injector allows to produce errors at any moment of the translation. However, this mechanism alone does not take into consideration moments when the instructions were already translated.

4.1.4 TB flush mechanism

TB flush mechanism, presented in section 3.5 section b), is necessary to force TB retranslation, enabling repeated intersection of the same code regions and allowing fault injection beyond the first execution of a cached TB. To validate the correct operation of the flush mechanism, two runs are performed: a baseline run and a run with the flush mechanism enabled, using a code example that repeatedly executes the same code segment multiple times. The chosen code was a Fibonacci algorithm.

In Figure 4.1.9, the disassembled code is shown on the top rectangle, while the logging output containing TB information is shown below. In the disassembly, the blue box (TB 3) represents the code inside the *for* loop, while others represent the TBs executed only one time. From this, it can be observed that, although the *for* loop executes five iterations, the corresponding TB is generated only once.

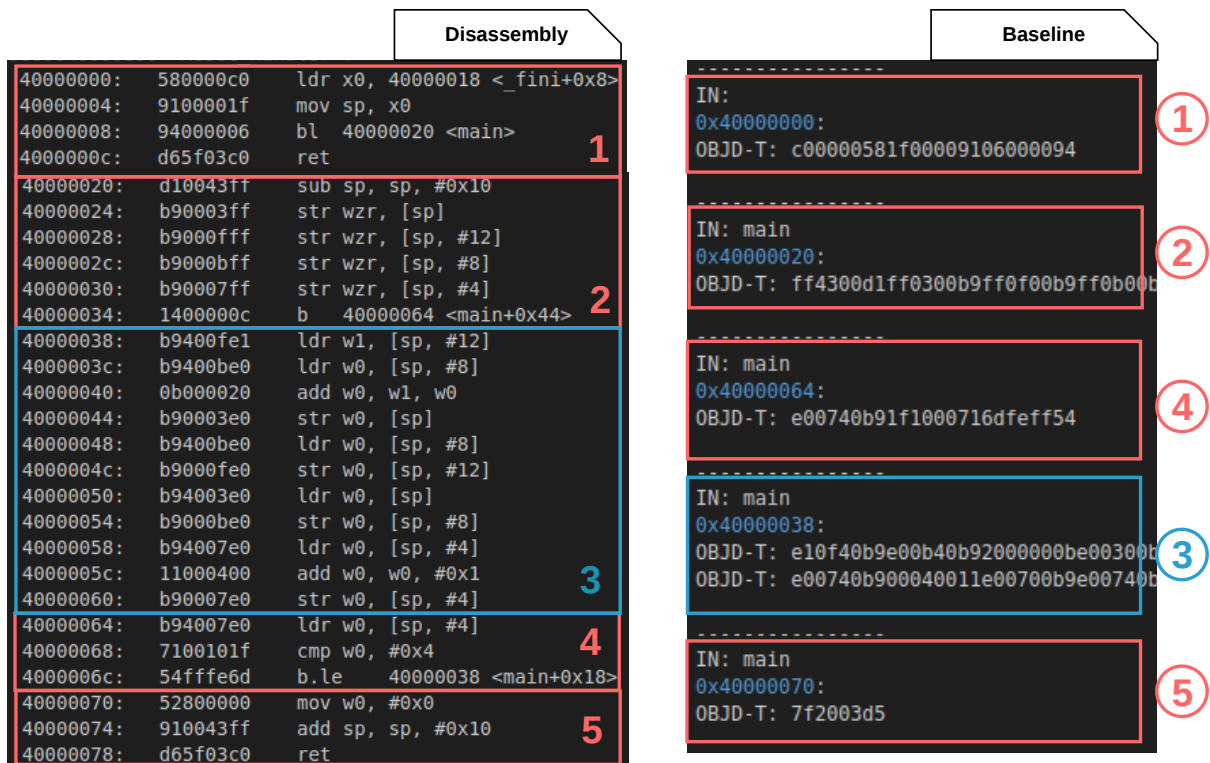


Figure 4.1.9: Executed code and their respective TBs.

With the flush mechanism enabled for TB inside the loop, is forced to be retranslated multiple times. To illustrate this behavior, the mechanism was kept active until the program terminated. In Figure 4.1.10, the new QEMU log is shown under those conditions. In this case, the TB 3 is flushed five times for each iteration of the *for* loop. keeping the translation and allowing injection faults at any iteration.



Figure 4.1.10: QEMU logs result with active TB flush mechanism.

Combining both mechanisms (TB breaking and TB flushing), injection can be performed at any moment of the loop, as demonstrated in Figure 4.1.11. In this example, the breaking is defined to 0x4000004c. The blue highlight shows the same TB with the code inside the loop, present in the example in Figure 4.1.10 (TB 3). TB 3 is divided into two parts: TB 3.1 (highlighted in blue) and TB 3.2 (highlighted in green). The second part starts being generated at the address where the break occurs. As a result, this portion is repeatedly regenerated and flushed in every loop iteration. This behavior enables the injection of faults at any iteration of the loop, since TB 3.2 is continuously retranslated and executed.



Figure 4.1.11: Log output for TB flushing and brekaing.

It can be concluded that, with both mechanisms enabled, fault injection can be performed at any moment of the execution, overcoming the limitations imposed by QEMU's TB caching and translation model.

Other frameworks, such as [48], [39], and [46] that inject faults during the translation phase, do not take into consideration the QEMU TB management. This is a problem because, when ignoring the TB cache principles and TB chaining, the injection occurs only during the first TB generation. With the proposed mechanisms integrated, it becomes possible to cover a larger portion of the execution, enabling a higher number of injections at different execution moments.

To evaluate the increased coverage provided by these mechanisms, four workloads were executed, each consisting of four golden runs: (i) baseline execution; (ii) execution with the flush mechanism enabled; (iii) execution with the breaking mechanism enabled; and (iv) execution with both mechanisms enabled. The mechanisms remained active throughout the entire execution. Since the injection happens only on top of TB, by tracking the number of TBs generated during the execution, the number of possible injection points can be deduced. To quantify the overhead introduced by the proposed mechanisms, a normalized cost metric was defined. This metric relates the execution time of a run to the number of available injection points, allowing a fair comparison between runs with different numbers of trigger opportunities. The normalized overhead (Ov) is calculated : $Norm.Ov. = \frac{M}{B}$

$$M = \frac{runtime}{numberoftriggerpossibilities}$$

$$B = \frac{baselinruntime}{baselinnumberoftriggerpossibilities}$$

Here, B represents the average execution time per injection point in the baseline configuration, while M represents the average execution time per injection point in a run where the proposed mechanisms are enabled. Table 4.6 illustrates the number of trigger possibilities (TP) for each case and the Normalized Overhead (Norm. Ov.).

Table 4.6 Number of trigger possibilities with framework mechanisms for different algorithms.

Algorithm	(i) Baseline	(ii) Flush Mechanism		(iii) Breaking Mechanism		(iv) Both Mechanisms	
	TP	TP	Norm. Ov.	TP	Norm. Ov.	TP	Norm. Ov.
Dhrystone	38	10249	2.5x	119	1.3x	50752	1.4x
Btrees	162	13760	2.4x	847	1.5x	105046	3.3x
Recursive	211	14235	1.6x	7234	1.1x	280559	2.7x
Bubble Sort	46	13326	2.8x	201	1.7x	203878	3.3x

It can be concluded that enabling both mechanisms significantly increases the number of possible trigger points, thereby expanding the fault injection space and precision. However, this increase in controllability comes at the cost of additional overhead introduced by the emulation stops required to apply the proposed mechanisms.

4.1.5 TB cache flush

To ensure correct and controllable fault injection in multi-core scenarios, it is essential to manage the TB cache appropriately. Since the TB cache is shared among all emulated cores, a complete flush of the cache

is required to guarantee that fault injection can be correctly targeted to a specific core. By fully flushing the TB cache when the selected core gains execution control, the retranslation is forced. When execution control is transferred to non-target cores, the cache is preserved, avoiding unnecessary retranslations and the introduction of additional overhead. This method is preferable to the TB flush approach in multi-core cases, since it eliminates the necessity to flush TBs one by one in each *cpu_exec_loop* iteration.

The following demonstration will show how the TB cache flush is necessary to carry out the injection in multi-core scenarios. A baremetal with 4 cores workload was performed on top of the Bao hypervisor [49]. The code shown in Listing 4.15 prints a value to the terminal from each execution core. The result appearing in the terminal should be 0xAA.

Listing 4.15: Code for printing the *var* variable in the terminal

```
void main(void){
    static volatile bool master_done = false;
    if(cpu_is_master()){
        spin_lock(&print_lock);
        printf("Bao bare-metal test guest\n");
        spin_unlock(&print_lock);

        master_done = true;
    }

    while(!master_done);
    spin_lock(&print_lock);
    int var = 0xAA;
    printf("(cpu %d): Value: %x\n", get_cpuid(), var);
    spin_unlock(&print_lock);

    while(1) wfi();
}
```

The objective of this experiment is to induce an incorrect terminal output by injecting a fault into the register that holds the variable being printed. Two types of faulty runs were performed: (i) TB cache flush mechanism disabled; (ii) TB cache flush mechanism enabled. For each type, 4 runs were performed for each core. The objective of this experiment is to induce an incorrect terminal output by injecting a fault into a guest register that holds the variable being printed. The injected fault is a bit-reset with a 0xAA

mask. By injecting this fault in the register that holds the output, the expected faulty output is 0.

In Figure 4.1.12, the results for each target core with the cache flush mechanism enabled and disabled are presented. By disabling the flush cache mechanism, the injection was performed only for core 0. This occurs because this is the first core to execute the code, which means it will intersect the TB generation that contains the code that stores the *var* value in the stack. When the other cores take control, the TBs associated with the printing code are already in the cache. With this, the translation phase is bypassed, and the fault does not take place for these cases. When the cache flush mechanism is enabled, after the target core takes the execution control, the cache flush happens, allowing the repetitive generation of the TB and consequently allowing the injection in each core.

	Flush Cache Enabled	Flush Cache Disabled
Core 0	<pre> Bao Hypervisor Virtual Memory base: 0x7ffe70400000 Bao bare-metal test guest Fault Id 0 Bit Flip for bits 0xaa in 0x2 (cpu 0): Value: 0 (cpu 1): Value: aa (cpu 2): Value: aa (cpu 3): Value: aa </pre>	<pre> Bao Hypervisor Virtual Memory base: 0x7ffe70400000 Bao bare-metal test guest Fault Id 0 Bit Flip for bits 0xaa in 0x2 (cpu 0): Value: 0 (cpu 1): Value: aa (cpu 2): Value: aa (cpu 3): Value: aa </pre>
Core 1	<pre> Bao Hypervisor Bao bare-metal test guest (cpu 0): Value: aa Virtual Memory base: 0x7ffe70400000 Fault Id 0 Bit Flip for bits 0xaa in 0x2 (cpu 1): Value: 0 (cpu 2): Value: aa (cpu 3): Value: aa </pre>	<pre> Bao Hypervisor Bao bare-metal test guest (cpu 0): Value: aa Virtual Memory base: 0x7ffe70400000 (cpu 1): Value: aa (cpu 2): Value: aa (cpu 3): Value: aa Warning: Some faults were not completed! </pre>
Core 2	<pre> Bao Hypervisor Bao bare-metal test guest (cpu 0): Value: aa (cpu 1): Value: aa Virtual Memory base: 0x7ffe70400000 Fault Id 0 Bit Flip for bits 0xaa in 0x2 (cpu 2): Value: 0 (cpu 3): Value: aa </pre>	<pre> Bao Hypervisor Bao bare-metal test guest (cpu 0): Value: aa (cpu 1): Value: aa Virtual Memory base: 0x7ffe70400000 (cpu 2): Value: aa (cpu 3): Value: aa Warning: Some faults were not completed! </pre>
Core 3	<pre> Bao Hypervisor Bao bare-metal test guest (cpu 0): Value: aa (cpu 1): Value: aa (cpu 2): Value: aa Virtual Memory base: 0x7ffe70400000 Fault Id 0 Bit Flip for bits 0xaa in 0x2 (cpu 3): Value: 0 </pre>	<pre> Bao Hypervisor Bao bare-metal test guest (cpu 0): Value: aa (cpu 1): Value: aa (cpu 2): Value: aa Virtual Memory base: 0x7ffe70400000 (cpu 3): Value: aa Warning: Some faults were not completed! </pre>

Figure 4.1.12: Validation of full cache flush mechanism.

For the cases in which the injection takes place, the value printed on the terminal is 0 for the corresponding core. When the cache flush mechanism is disabled, the injection is performed only on core 0. By disabling the flush cache mechanism, the injection was performed only for core 0. This occurs because this is the first core to execute the code, which means it will intersect the TB generation that contains the code that stores the *var* value in the stack. When the other cores (1,2 and 3) take

execution control, the TBs associated with the printing code are already in the cache. With this, the translation phase is bypassed, and the fault does not take place for these cases. When the cache flush mechanism is enabled, after the target core takes the execution control, the cache flush happens, allowing the repetitive generation of the TB and consequently allowing the injection in each core.

With the results presented, it can be concluded that TB cache flush is necessary for ensuring the correct fault injection in multi-core runs. When the mechanism is disabled, fault injection is limited to the first core that executes the target code, due to the reuse of previously translated TBs. On the other hand, enabling the cache flush forces retranslation when the target core takes the execution control, allowing the injection to be applied across all cores.

4.1.6 Access Injection

The access triggering type consists of placing a watchpoint on a defined memory region. In QEMU, placing a watchpoint at the address of an emulated register allows it to stop execution whenever a read or write operation occurs on that register. However, as mentioned in section 3.6 c), this mechanism presents a corner case that must be taken into account. QEMU cannot strictly follow instruction-by-instruction execution during the translation phase, since its execution model is dynamic and based on runtime code generation. During this process, several optimizations may be applied, including the use of temporary registers. Consequently, some accesses that are visible from the guest code perspective may be truncated or eliminated at the emulator level.

Considering behavior can be observed in the code example below in Listing 4.16.

Listing 4.16: Fibonacci code example with C-ASM integration

```

void main(void){
    int t1 = 0;
    int t2 = 1;
    int r;

    for (int i = 0; i < 5; i++) {
        __asm__ volatile (
            "mov x5, %[t1]      \n"
            "mov x6, %[t2]      \n"

            "add x7, x5, x6      \n" // r = t1 + t2
            "mov x5, x6          \n" // t1 = t2
            "mov x6, x7          \n" // t2 = r

            "mov %[t1], x5       \n"
            "mov %[t2], x6       \n"
            "mov %[r], x7        \n"

            : [t1] "+r"(t1),
              [t2] "+r"(t2),
              [r]  "=r"(r)

            :
            : "x5", "x6", "x7", "cc"
        );
    }
    puts_uart0("Value: ");
    print_uint64_uart0(r);
    puts_uart0("\n");
    return 0;
}

```

In the guest code, register x7 is accessed three times: one used as the destination register for the ADD operation and two for MOV operations as an operand register. However, during the translation to host execution code, these operations are optimized away. In Figure 4.1.13, the resulting host code for

the guest code inside the black box is illustrated. In bold are highlighted the instructions where x7 is accessed. QEMU uses temporary registers to optimize operations inside the TB. As can be observed, the ADD operation is performed using temporary registers instead of directly accessing the *CPUArchState* register x7. The intermediate results are also stored in temporary registers and only written back at the TB exit. As a result, a watchpoint placed on x7 will only trigger on the second access. This discrepancy can lead to unexpected fault injection timing from the user's perspective, since an access that logically occurs in the guest code is not observed at the emulator level.

Guest code

```

40000194: a9be7bfd stp x29, x30, [sp, #-32]!
40000198: 910003fd mov x29, sp
4000019c: b9001fff str wzr, [sp, #28]
400001a0: 52800020 mov w0, #0x1
400001a4: b9001be0 str w0, [sp, #24]
400001a8: b90013ff str wzr, [sp, #16]
400001ac: 14000011 b 400001f0
400001b0: b9401fe1 ldr w1, [sp, #28]
400001b4: b9401be0 ldr w0, [sp, #24]
400001b8: aa0103e5 mov x5, x1
400001bc: aa0003e6 mov x6, x0
400001c0: 8b0600a7 add x7, x5, x6
400001c4: aa0603e5 mov x5, x6
400001c8: aa0703e6 mov x6, x7
400001cc: aa0503e1 mov x1, x5
400001d0: aa0603e0 mov x0, x6
400001d4: aa0703e2 mov x2, x7
400001d8: b9001fe1 str w1, [sp, #28]
400001dc: b9001be0 str w0, [sp, #24]
400001e0: b90017e2 str w2, [sp, #20]
400001e4: b94013e0 ldr w0, [sp, #16]
400001e8: 11000400 add w0, w0, #0x1
400001ec: b90013e0 str w0, [sp, #16]
400001f0: b94013e0 ldr w0, [sp, #16]
400001f4: 7100241f cmp w0, #0x9
400001f8: 54ffdcdb b.le 400001b0
400001fc: b98017e0 ldrsw x0, [sp, #20]

```

Host code

```

t1 <- CPUArchState REG 5
t2 <- CPUArchState REG 6
t3 <- t1 + t2
t1 <- t2
t2 <- t3
t1 -> CPUArchState REG 5
t2 -> CPUArchState REG 6
t3 -> CPUArchState REG 7

```



Figure 4.1.13: Register accesses inside a TB.

One way to mitigate this problem is by configuring for one instruction per TB. By doing this, each instruction is independent, meaning that before each operation, if it's necessary to access any register, it has to read from *CPUArchState* struct directly. For the case in Figure 4.1.14 where QEMU is configured to generate TBs with one instruction, the register x7 is accessed three times, synchronized with the guest code accesses. Take into consideration the code Listing 4.16, the register x7 is accessed 15 times during the full runtime.

This configuration, however, has performance costs (see section 4.2). In addition to this, for read accesses, the fault is reflected only in the next instruction. If the trigger occurs during the MOV operation between x6 and x7, the faulty value is not immediately written to x6 but only occurs on the next access to

that register.

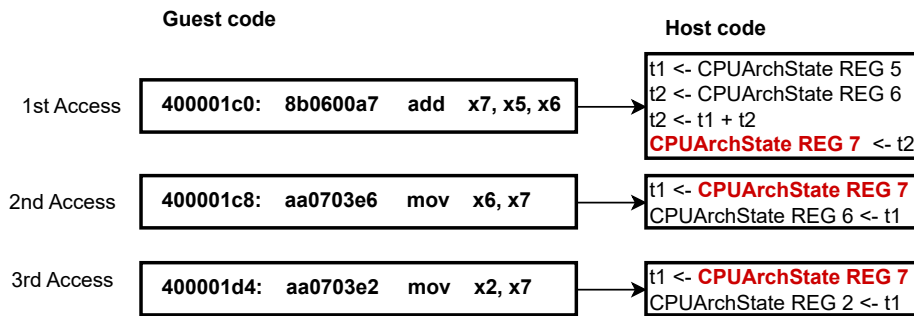


Figure 4.1.14: Register accesses for TBs with one instruction only.

To verify the effect caused by access injection, a faulty run was performed for each access to register x7. Since the loop executes five iterations and x7 is accessed three times per iteration, a total of 15 accesses occur. Consequently, 15 faulty runs were conducted, each injecting a fault into a different access, from the first to the fifteenth. The fault model is a bit-reset that clears the register completely. In Table 4.7, the output result for each access injection in each run is displayed. The baseline output is 8.

Table 4.7 Output results for injection in x7 accessing.

Loop Iteration	1st Access	2nd Access	3th Access
1	3	8	8
2	2	8	8
3	2	8	8
4	3	8	8
5	0	0	8
Baseline	8		

For this example, it can be observed that by injecting in the first access in every iteration, it will cause the output to have a value different from 8 since the value of the ADD operation is modified after the operation. For the second access, the final result only gets a faulty value for the last iteration (14th access in general) where the output takes the 0 value. This occurs because register x2 provides the value for the stack position that is accessed after exiting the loop, which is then used by the variable that stores the output result. For the third access faulty runs, as expected, the output value stayed the same as the baseline in all the cases. This is because the fault only affects the MOV operation of the third access, after

which register x7 is overwritten. In the next loop iteration, this faulty value is replaced by the result of the subsequent ADD operation, effectively masking the fault.

These tests confirm that configuring the QEMU to generate TBs with one instruction solves the problem, but for faults in read data, only takes effect in the next instruction. This occurs because when an instruction reads from a register or a memory location where the fault is injected, the faulty behavior is only manifested when the faulty value is used by the next operation, meaning that the destination operand of the original instruction does not receive the faulty value at that stage.

4.2 Performance Evaluation

As mentioned, implementing a fault injector through the invocation of a Python interface extension in GDB, together with the addition of mechanisms to enable precise injection, introduces execution overhead. Taking this into account, several techniques were integrated to minimize this impact as much as possible. This section presents the optimizations applied to reduce this overhead, the design decisions made to ensure the highest efficiency, and a comparison between the different approaches.

4.2.1 Single versus multiple instructions TBs

As mentioned in section 2.3.1 c), QEMU has a parameter that limits the TBs to a single instruction only. Although these characteristics provide advantages for precise fault injection, they significantly degrade performance, resulting in substantially longer execution times. The table in Table 4.8, shows baseline runs of a bubble sort algorithm for an array of 400 elements in bare metal for both Aarch64 and RISC-V architectures. For each type, 200 samples were performed.

Table 4.8 Comparison of execution times and TBs for AArch64 and RISC-V architectures with QEMU configured to generate single and multi-instruction TBs.

CPU architecture	TB type (ms)	time mean (ms)	TB generations no	host code length (bytes)
AARCH64	single instruction	914.1	129	20600
AARCH64	multi instruction	7.74	19	10888
RISC-V	single instruction	487.71	158	18184
RISC-V	multi instruction	5.85	25	11540

By analyzing the results, it can be observed that limiting each TB to a single instruction not only

significantly degrades performance but also increases the size of the generated host code.

4.2.2 Optimizations

To improve performance, several optimization approaches were adopted to accelerate execution during faulty runs. In this section, the mechanisms associated with each fault model trigger type are described.

As mentioned earlier in Section 3.6, this framework allows fault injection at execution points that may have already been reached by the CPU. This means that, for example, if the user intends to inject multiple faults at a specific PC address, this can be achieved using the proposed framework.

For PC triggering specifically, two approaches were presented in Section 3.6: the TB flush mechanism and host code execution intersection. This evaluation compares the performance impact of both approaches over a fixed number of iterations of a loop workload. To perform this comparison, the mechanisms are enabled after the first translation and remain active until the end of the run.

In each experiment, the intersect index parameter was configured with the values 100, 250, 500, and 1000, respectively. Therefore, each approach was evaluated under four different trigger activation points within the loop. For this workload, the intersect index is synchronized with the loop iterations, meaning that each value directly corresponds to a specific iteration of the loop.

The evaluated workload consists of a Fibonacci algorithm with 1000 iterations, executed on a single-core Armv8-A architecture. The graph in Figure 4.2.1 presents the average results of 20 runs for each experiment. The Y-axis represents the total execution time measured using the wall-clock time at the end of the program, while the X-axis corresponds to the intersect index value defined in the XML configuration.

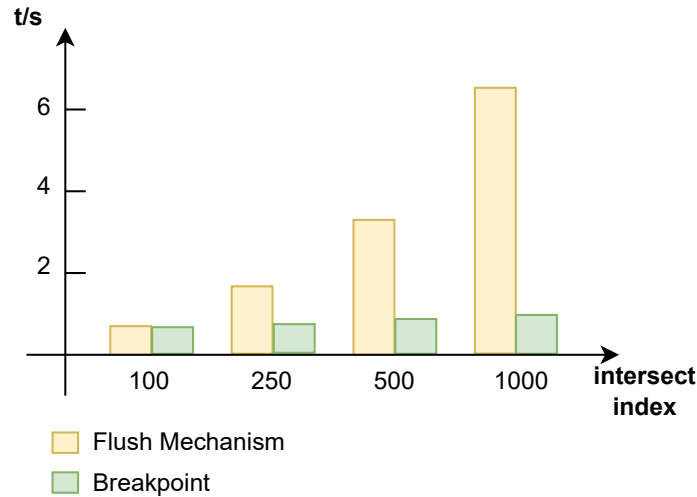


Figure 4.2.1: Execution time comparison between the flush mechanism and the breakpoint PC triggering approaches.

For this experiment, the data collector was disabled. From the results, it can be observed that using host code breakpoint intersection is significantly more efficient, as it avoids forcing QEMU to repeatedly retranslate TBs.

Other optimizations were implemented for the faulty run in the PC (section 3.6 a)) and time (section 3.6 d)) injections. Those are applied by gathering information during the golden run, storing it in an XML file (*optimization.XML*). With these improvements, several breakpoint stops can be bypassed. In the case of PC triggering, successive instruction-by-instruction PC verifications can be avoided, as illustrated in Figure 3.6.8. Similarly, for timer triggering, the proposed optimization bypasses unnecessary TB flushes, reducing the number of breakpoint stops required to perform this action and, consequently, improving overall performance.

To prove that, four faulty runs were performed, for each workload: (i) PC triggering, with optimizations; (ii) PC triggering, without optimizations; (iii) Time triggering, with optimizations; (iv) Time triggering, without optimizations. In each workload, a fault was injected in the last possible trigger, i. e., in the last executed instruction. With this, the trigger mechanisms will cover all the translated instructions until the injection is performed. Then, the number of executions stop were calculated in each scenario. After the run tests, the results in Figure 4.2.2 were obtained.

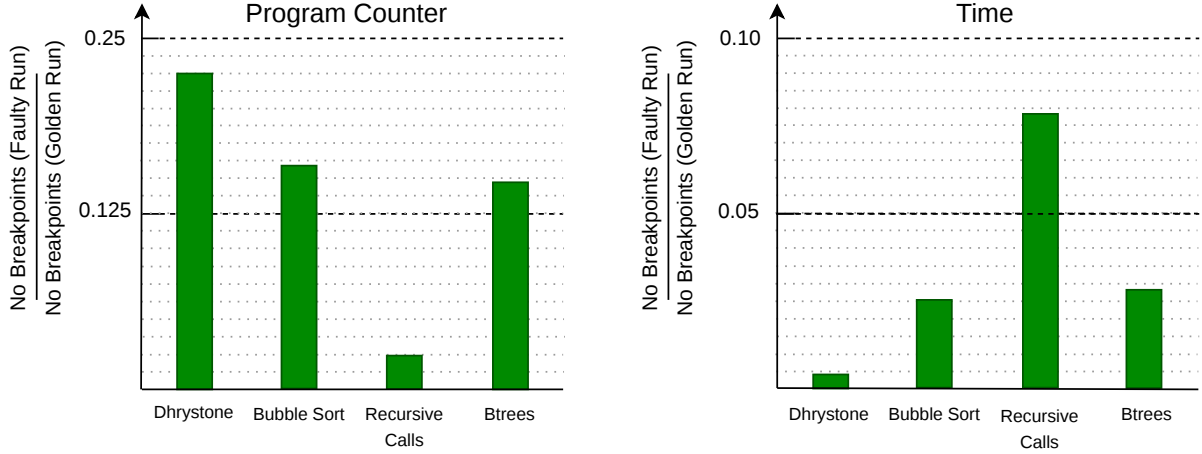


Figure 4.2.2: Comparison of the number of breakpoint stops between optimized and non-optimized faulty runs in the worst-case scenario for PC and time triggering.

In the graph, the proportion of intersected breakpoints between optimized and non-optimized faulty runs for both types of triggering is presented. For the Btrees workload, under PC triggering, the number of breakpoint stops in the optimized run corresponds to approximately 15% of those observed in the non-optimized faulty run. On the other hand, under time triggering, the number of breakpoint stops in the optimized run is reduced to only 3% of those in the non-optimized faulty run. The most significant improvements from PC and time triggering optimizations were observed for the Recursive Calls and Dhrystone workloads, respectively. In PC triggering, the Recursive Calls workload achieved a reduction in breakpoint stops to about 5% of the non-optimized case, while in time triggering, the Dhrystone workload achieved a reduction to less than 1%. From these results, it can be concluded that the optimizations decrease the number of breakpoint stops in faulty runs, leading to a positive impact on performance.

4.2.3 Triggering overhead

Each trigger type has its own implementation strategy, relying on specific mechanisms to ensure precise triggering at predefined execution moments. Trigger types such as instruction and timer triggering tend to require more frequent TB flushes than PC triggering, as discussed in Section section 3.6. This behavior results from their finer-grained control requirements, which demand more frequent synchronization with the translation and execution phases. In this section, the time comparison between these triggers will be performed for distinct execution points in different workloads.

To perform these tests, the method in Figure 4.1.1 was used. For each workload, 4 golden runs for each trigger type were executed to evaluate the time. Each triggering was configured to 2500th, 5000th,

7500th, and 10000th.

To understand the overall results, measurements were collected for each trigger at equivalent execution milestones, i.e., for the same number of instructions executed with different trigger definitions. In Figure 4.2.3, the execution time for each trigger type (instruction, timer, and PC) is presented. To perform these tests, the instruction trigger was first used as a baseline to identify the PC value corresponding to each injection moment, which was subsequently used to configure the PC trigger. For the timer, the parameter was the same as used in instruction triggering. For the timer trigger, the same parameter configuration used for instruction triggering was adopted, since the selected timer period corresponds exactly to the number of executed instructions.

Since experiments were performed in single-core execution, the time parameter corresponds to the same number of instructions executed.

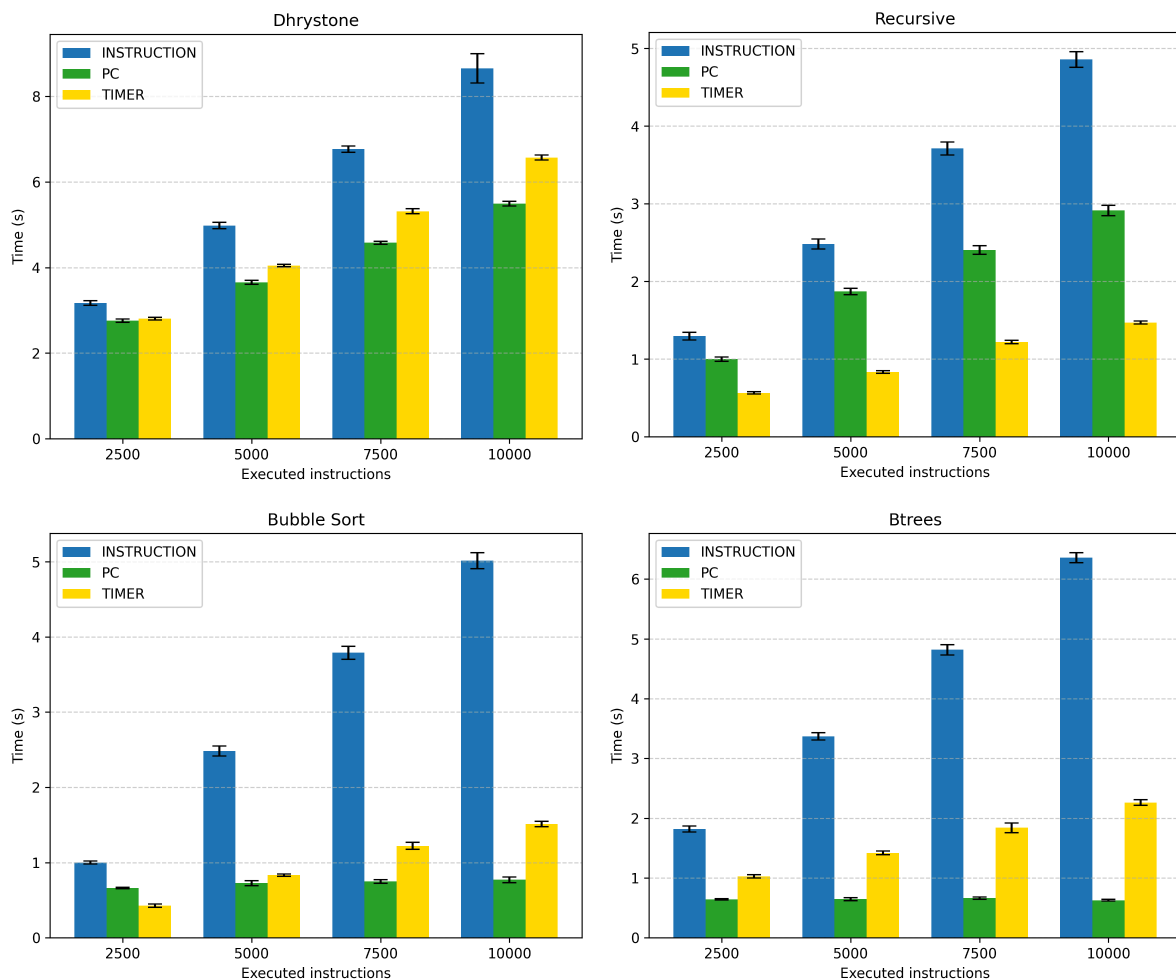


Figure 4.2.3: Performance time for each trigger in the same moment for different benchmarks.

Analyzing the results, it is concluded that overall, the PC triggering introduces the lowest overhead.

However, in scenarios involving recursive calls that generate a larger number of distinct TBs, this approach becomes less effective.

Instruction triggering is the approach that causes the highest overhead in all cases. This is because, for this trigger type, a large number of stop conditions are executed between TB flushes. With this in mind, this trigger type should be used only for multi-core operation, since it can isolate the number of executed instructions for a specified core, unlike the timer type.

Timer triggering also introduces significant overhead, since it conservatively flushes TBs. However, this approach stops execution between TB generations rather than during instruction-by-instruction translation. In addition, optimizations were also applied in the faulty runs.

This framework provides several trigger types, allowing it to be adapted to different reliability standards. However, for some triggers, the overhead introduced is higher, particularly those that require frequent TB flushing or fine-grained execution control. As a result, these mechanisms may impact execution performance more significantly, especially in long-running or computation-intensive workloads. Overall, despite the use of optimizations, the framework still presents higher overhead when compared to baseline execution times.

5 Conclusion

Through the exploration of non-intrusive fault injection techniques and the use of emulation as a viable alternative to testing on real hardware, it was possible to conceive a systematic approach to analyze the resilience of critical systems under realistic disturbances. The focus on reproducibility, observability, and control of fault scenarios enabled the development of the framework to be aligned with the requirements imposed by safety standards such as ISO 26262, thus establishing a solid foundation for the rigorous evaluation of system reliability and behavior under adverse conditions.

This framework enables the injection and analysis of faults in systems emulated by QEMU for AArch64 and RISC-V architectures. By integrating a GDB interface on top of the emulator source code, the proposed solution also allows compatibility with future QEMU updates, offering a scalable sandbox where multiple standards can be applied and modified easily. The fault injection has two phases: the golden run and the faulty run. First, the golden run corresponds to a reference execution of the system without fault injection, in order to obtain a set of logs that represent its correct behavior. Then, the faulty run replicates the same execution conditions but with the controlled injection of predefined faults. The effects of disturbances are then evaluated through a systematic comparison of the logs collected in both runs, enabling the identification of functional, temporal, and internal state deviations, as well as the characterization of the impact of faults on the overall system behavior.

In the proposed implementation, several triggering types were implemented to support different fault injection strategies. The exploitation of multiple QEMU internal mechanisms enabled the development of techniques that have not been previously explored in existing approaches. Two principal controller mechanisms were implemented: TB breaking, which allows for injection in precise points, providing CPU components modification in any defined context; TB flushing, which enables the injection in points already covered by the TCG. The combination of these two main components allows the coverage of all system states during the execution.

Furthermore, the implementation was carried out within the generic code translation and execution paths of QEMU. This facilitates portability to other CPU architectures with minimal integration effort. From a design perspective, the architectural approach complements the previously discussed contributions by ensuring that the implemented mechanisms can be selectively enabled, combined, and extended, reinforcing the framework's role as a flexible and scalable environment.

The combination of these mechanisms and strategies enabled the evaluation of CPU execution at all runtime stages. This allows for more precise identification of potential system vulnerabilities, enabling the systematic definition of diverse triggering conditions and fault injection models, thereby supporting

fine-grained and adaptable fault injection.

Regarding performance, the proposed framework introduces a significant overhead compared to baseline emulation. This is due to the introduction of execution stop conditions caused by the GDB interface, which is an expected consequence of prioritizing precision. Although performance was not the primary optimization target, the obtained results demonstrate that it remains efficient for evaluation and precise analysis purposes.

Although the proposed framework supports a wide range of triggering mechanisms, not all of them are equally recommended for all evaluation scenarios. Some triggers may introduce unexpected behaviors that can lead to misleading results. In particular, PC triggering through breakpoints may cause control-flow jumps to code regions that have already been translated by QEMU, potentially missing synchrony with certain fault models. Similarly, access triggering may rely on temporary registers introduced during translation, which may truncate some expected memory or register access patterns as observed from the guest code perspective. These effects can lead to incomplete or misleading fault coverage and, therefore, require careful consideration when selecting triggering mechanisms.

5.1 Future Work

Although the proposed framework provides a modular and extensible implementation that supports the integration of additional processor architectures, and enables precise and controllable fault injection for the evaluation of system fault tolerance mechanisms, several aspects remain open for improvement. The following future work directions focus on improving result accuracy, reducing execution overhead, and extending the applicability of the framework to more complex execution models and evaluation scenarios.

- **Explicit classification of the fault effect:** The collector of the framework allows for observing multiple logs during the runtime that help to predict possible vulnerabilities. However, the current result analysis is limited to detecting errors based on discrepancies observed in CPU state and memory contents at synchronized points with triggering moments between the golden run and the faulty execution. If a system has a mechanism to recover or to perform rollback execution, the veracity of the results could be compromised. One way to mitigate this problem could be the study of an interface to detect system calls.
- **Golden Run Independence:** To ensure result verifiability, test executions should be deterministic, since comparable runs must exhibit identical control flow and behavior. Traditionally, this requires executing a golden run followed by a faulty run. By removing the need

to execute a golden model beforehand, this dependency is eliminated. As a result, the evaluation can be performed using a single execution, significantly reducing the overall execution time.

- **Optimization of fault models reliable in multiple retranslations:** This framework enables fault injection at execution points that have already been reached during program execution by forcing code retranslation. While this approach is effective in guaranteeing precise injection, it introduces additional overhead due to the need to flush Translation Blocks (TBs). Within the proposed framework, multiple stop and control conditions are applied to manipulate QEMU execution, ensuring accurate fault injection throughout the entire runtime of the application. The achieved precision depends on the correct use of the implemented mechanisms, while certain corner cases, such as excessive TB flushing, may introduce unnecessary overhead. Exploring alternative approaches that provide equivalent fault injection capabilities without relying on TB retranslation represents a relevant direction for future work to further reduce overhead while preserving injection accuracy.
- **Porting for MTTCG:** The proposed fault injection framework was implemented and evaluated with QEMU configured to use the single-threaded Tiny Code Generator (TCG). This configuration guarantees deterministic execution and accurate control over fault injection timing, which are required for reproducible fault injection experiments. The use of a multi-threaded TCG (MTTCG) may introduce non-deterministic behavior due to concurrent execution and host-level scheduling, making it more difficult to precisely correlate injection triggers with specific execution points. Although the single-threaded configuration limits performance, it provides a controlled execution environment suitable for high-precision fault injection. Extending the framework to support MTTCG-based execution models is therefore identified as future work, with the potential to improve performance and scalability.

Bibliografia

- [1] R. Katz, K. LaBel, J. Wang, B. Cronquist, R. Koga, S. Penzin, and G. Swift, "Radiation effects on current field programmable technologies," *IEEE Transactions on Nuclear Science*, vol. 44, no. 6, pp. 1945–1956, 1997.
- [2] J. Knight, "Safety critical systems: challenges and directions," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pp. 547–550, 2002.
- [3] Y. Bekele, D. Limbrick, and J. Kelly, "A survey of qemu-based fault injection tools & techniques for emulating physical faults," *IEEE Access*, vol. PP, pp. 1–1, January 2023.
- [4] H. Ziade, R. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *Int. Arab J. Inf. Technol.*, vol. 1, pp. 171–186, 01 2004.
- [5] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '05*, (USA), p. 41, USENIX Association, 2005.
- [6] A. C. Dimopoulos, G. Bravos, G. Dimitrakopoulos, M. Nikolaidou, V. Nikolopoulos, and D. Anagnostopoulos, "A multi-core context-aware management architecture for mixed-criticality smart building applications," in *2016 11th System of Systems Engineering Conference (SoSE)*, pp. 1–6, 2016.
- [7] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, Nov. 2017.
- [8] A. Burns and R. I. Davis, *Mixed Criticality Systems - A Review: (13th Edition, February 2022)*. February 2022. This is the 13th version of this review now updated to cover research published up to the end of 2021.
- [9] C. Evrpidou and A. Burns, "Scheduling for Mixed-criticality Hypervisor Systems in the Automotive Domain," in *WMC 2016 4th International Workshop on Mixed Criticality Systems*, (Porto, Portugal), Nov. 2016.
- [10] F. Bedulli, R. Leite, F. Freitas, T. Murari, and I. Winkler, "Optimizing automotive product development: Integration of electronic control units and virtual prototypes using virtual reality," 01 2024.
- [11] J. Simo, P. Balbastre, F. Blanes, J.-L. Poza-Luján, and A. Guasque, "The role of mixed criticality technology in industry 4.0," *Electronics*, vol. 10, p. 226, 01 2021.

- [12] S. Lozano, T. Lugo, and J. Carretero, "A comprehensive survey on the use of hypervisors in safety-critical systems," *IEEE Access*, vol. PP, pp. 1–1, 01 2023.
- [13] S. Avramenko and M. Violante, "Rtos solution for noc-based cots mp soc usage in mixed-criticality systems," *Journal of Electronic Testing*, vol. 35, 02 2019.
- [14] R. Martin, "Why mixed-criticality is the future of automotive architectures," 09 2022.
- [15] J. Aizpurua and E. Muxika Olasagasti, "Design of dependable systems: An overview of analysis and verification approaches," 08 2012.
- [16] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.
- [17] N. K. Salih, D. Satyanarayana, A. S. Alkalbani, and R. Gopal, "A survey on software/hardware fault injection tools and techniques," in *2022 IEEE Symposium on Industrial Electronics & Applications (ISIEA)*, pp. 1–7, 2022.
- [18] A. Avizienis, J.-c. Laprie, and B. Randell, "Fundamental concepts of dependability," 09 2001.
- [19] H. Nguyen, Y. Yagil, N. Seifert, and M. Reitsma, "Chip-level soft error estimation method," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 365–381, 2005.
- [20] L. Feinbube, L. Pirl, and A. Polze, *Software Fault Injection: A Practical Perspective*. 06 2018.
- [21] F. Cucchetto, A. Lonardi, and G. Pravadelli, "A common architecture for co-simulation of systemc models in qemu and ovp virtual platforms," *IEEE/IFIP International Conference on VLSI and System-on-Chip, VLSI-SoC*, vol. 2015, 01 2015.
- [22] C. BasuMallick, "Translator internals," 01 2023.
- [23] A. Tijms, "Binary translation : Classification of emulators," 2000.
- [24] E. Diaz, R. Mateos, E. Bueno, and R. Nieto, "Enabling parallelized-qemu for hardware/software co-simulation virtual platforms," *Electronics*, vol. 10, p. 759, 03 2021.
- [25] C. Menard, J. Castrillon, M. Jung, and N. Wehn, "System simulation with gem5 and systemc: The keystone for full interoperability," in *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pp. 62–69, 2017.

- [26] L. Jünger, J. Weinstock, R. Leupers, and G. Ascheid, "Fast systemc processor models with unicorn," pp. 1–6, 01 2019.
- [27] N. Binkert, B. Beckmann, G. Black, S. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib Bin Altaf, N. Vaish, M. Hill, and D. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, pp. 1–7, 08 2011.
- [28] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.
- [29] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Computer Architecture News*, vol. 33, pp. 92–99, 11 2005.
- [30] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of gem5 simulator system," in *7th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7, 2012.
- [31] A. A. Abudaqa, T. M. Al-Kharoubi, M. F. Mudawar, and A. Kobilica, "Simulation of arm and x86 microprocessors using in-order and out-of-order cpu models with gem5 simulator," in *2018 5th International Conference on Electrical and Electronic Engineering (ICEEE)*, pp. 317–322, 2018.
- [32] D. Ferraretto and G. Pravadelli, "Efficient fault injection in qemu," in *2015 16th Latin-American Test Symposium (LATS)*, pp. 1–6, 2015.
- [33] F. Bellard, "Translator internals," 2024.
- [34] J. Lee, X. Mehrenberger, L. Craig, M. Wassermann, and S. Duverger, "Translator internals,"
- [35] M. Marazakis, "Qemu: Architecture and internals lecture for the embedded systems course csd, university of crete," pp. 1 – 30, 04 2023.
- [36] E. G. Cota, P. Bonzini, A. Bennée, and L. P. Carloni, "Cross-isa machine emulation for multicores," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 210–220, 2017.
- [37] M. Prathyusha and C. R. Kumar, "A survey paper on debugging tools and frameworks for debugging real time industrial problems and scenarios," in *2019 International Conference on Vision Towards Emerging Trends in Communication and Networking (ViTECoN)*, pp. 1–4, 2019.

- [38] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, "Fail*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance," 09 2015.
- [39] S. Chylek and M. Goliszewski, "Qemu-based fault injection framework," *Studia Informatica*, vol. 33, pp. 25–42, 01 2012.
- [40] S. Chylek, "Emulation-based software reliability evaluation and optimization," vol. 90, pp. 121–124, 02 2014.
- [41] N. Tian, D. Saab, and J. A. Abraham, "Esift: Efficient system for error injection," in *2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS)*, pp. 201–206, 2018.
- [42] J. An, H. You, F. Xie, Y. Yang, and J. Sun, "Fig-qemu: A fault inject platform supporting full system simulation," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, pp. 275–278, 2020.
- [43] R. Amarnath, S. N. Bhat, P. Munk, and E. Thaden, "A fault injection approach to evaluate soft-error dependability of system calls," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 71–76, 2018.
- [44] H. Jitsumoto, Y. Kobayashi, A. Nomura, and S. Matsuoka, *MH-QEMU: Memory-State-Aware Fault Injection Platform*, pp. 71–85. 04 2019.
- [45] Y. Li, P. Xu, and H. Wan, "A fault injection system based on qemu simulator and designed for bit software testing," *Applied Mechanics and Materials*, vol. 347-350, 02 2013.
- [46] Q. Guan, N. Debardeleben, S. Blanchard, and S. Fu, "F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1245–1254, 2014.
- [47] Q. Guan, N. BeBardeleben, P. Wu, S. Eidenbenz, S. Blanchard, L. Monroe, E. Baseman, and L. Tan, "Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques, SIMUTOOLS'16*, (Brussels, BEL), p. 9–17, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

- [48] A. Hölbling, G. Schönfelder, N. Kajtazovic, T. Rauter, and C. Kreiner, “Fies: A fault injection framework for the evaluation of self-tests for cots-based safety-critical systems,” 12 2014.
- [49] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, “Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems,” 01 2020.