Decoding the RAS Maze: Microscopic Complexity Meets Verification

Shubham Mathur, Nimesh Kharat

Hadmath Singh, Darshan Hadadi

tenstorrent

accellera SYSTEMS INITIATIVE™

2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Understanding Faults, Errors, and RAS Concepts

Faults vs. Errors

- A **Fault** is an incorrect physical or logical state, which can be permanent (e.g., a manufacturing defect) or transient (e.g., a cosmic ray).
- An **Error** is the indication or detection of a fault.
- A **Service Failure** is when the system's behavior deviates from its specification due to an error.

Core RAS Techniques

- The RISC-V RERI architecture provides a framework to support three main techniques for dealing with system issues:
- **Fault Prevention:** Using high-quality design and manufacturing to prevent faults from occurring in the first place.
- **Error Detection and Correction:** Using mechanisms like **Error Correcting Code (ECC)** to detect and fix errors. A key concept here is the **Corrected Error (CE)**, which the hardware can recover from autonomously.
- **Error Prediction:** Using a history of corrected errors to predict future uncorrectable failures. This allows for proactive measures, such as taking a failing memory page offline before it causes a critical system crash.

RAS Overview

What is RERI?

The **RISC-V RERI (RAS Error Record Register Interface)** is a specification that provides a standardized, memory-mapped register interface for reporting errors. Its primary goal is to augment **Reliability, Availability, and Serviceability (RAS)** features within a RISC-V System-on-a-Chip (SoC).

Key Functions

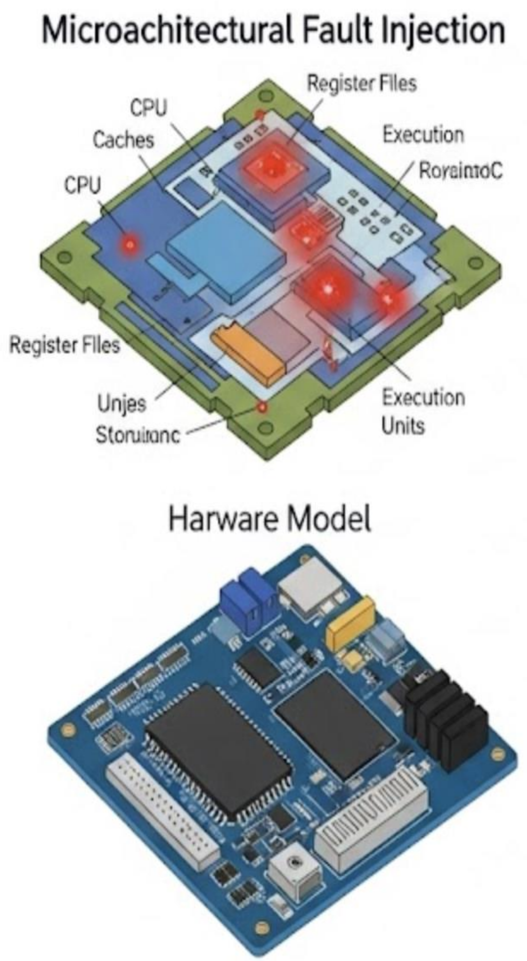
Error Reporting: Provides a consistent mechanism for hardware to log detected errors.

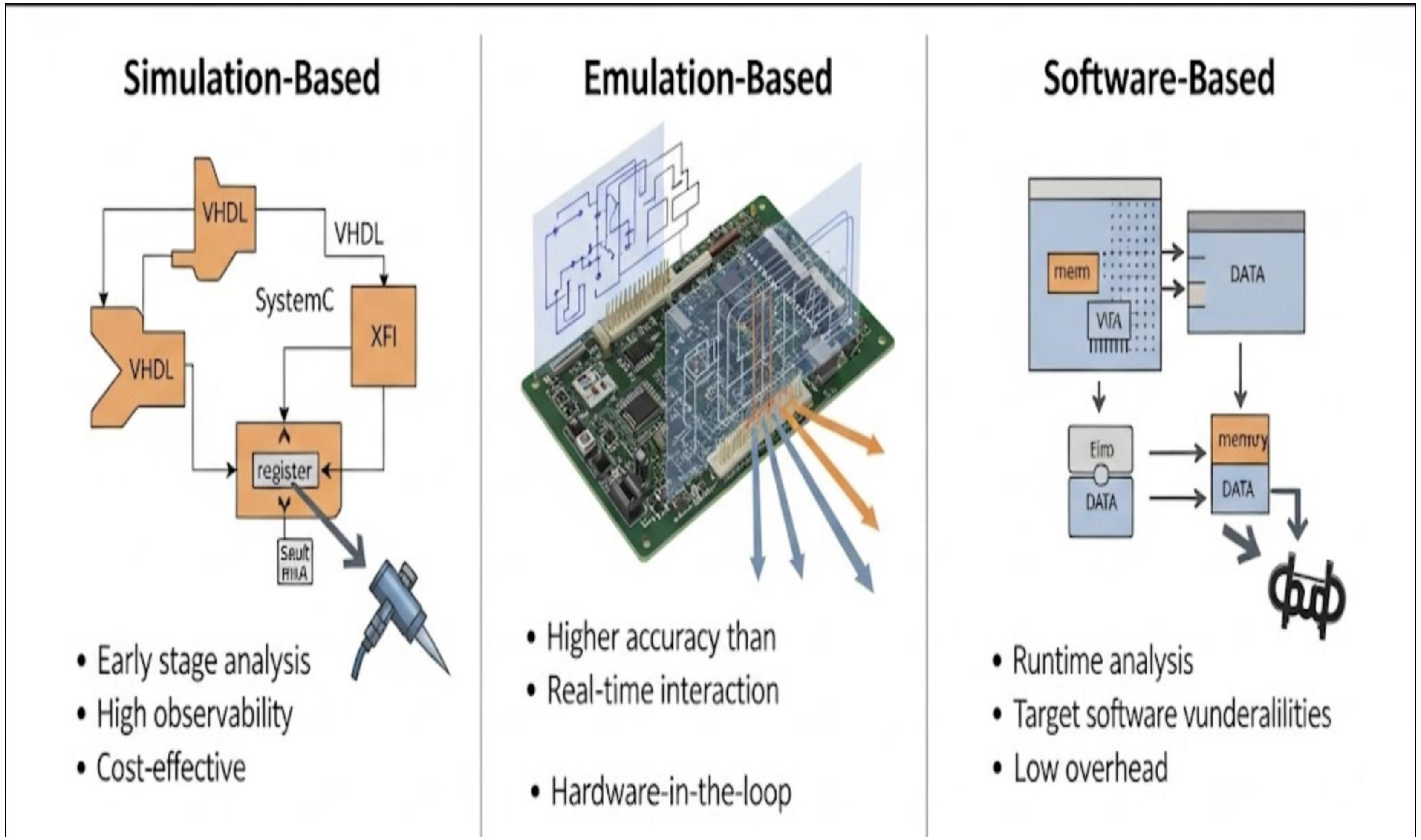
Information Logging: Logs crucial details about errors, including their severity (e.g., Corrected vs. Uncorrected), nature (e.g., permanent vs. transient), and location.

RAS Handler Communication: Enables the system to signal a RAS handler (a piece of software or a dedicated hardware component) so it can take appropriate recovery actions.

This specification is designed to be highly flexible, allowing it to be used in a wide range of systems, from high-end servers to low-power embedded devices. It also works alongside other error reporting standards like PCIe and CXL.

Fault Injection Techniques





Simulation-Based

- Early stage analysis
- High observability
- Cost-effective

Emulation-Based

- Higher accuracy than
- Real-time interaction
- Hardware-in-the-loop

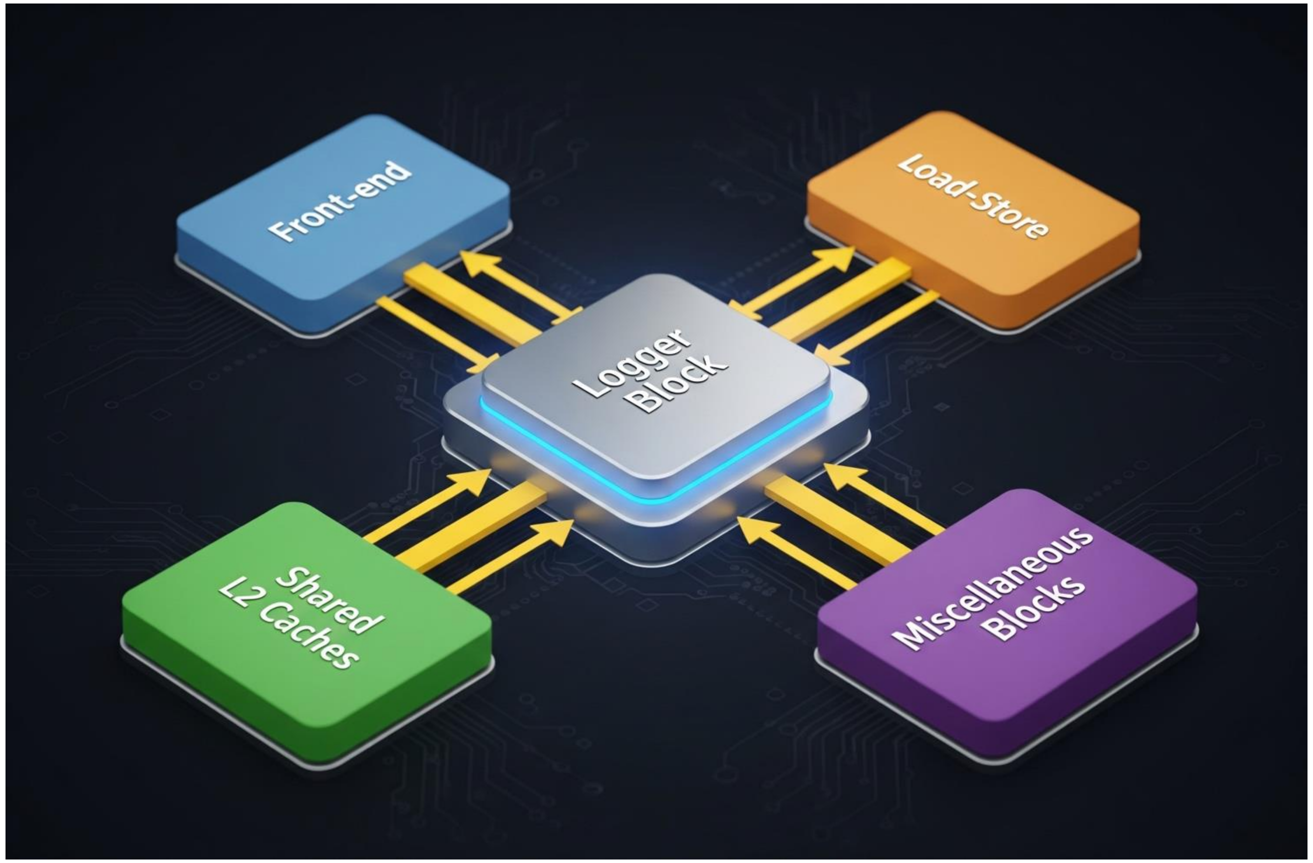
Software-Based

- Runtime analysis
- Target software vunderalilities
- Low overhead

Micro-arch Challenges in RAS verification

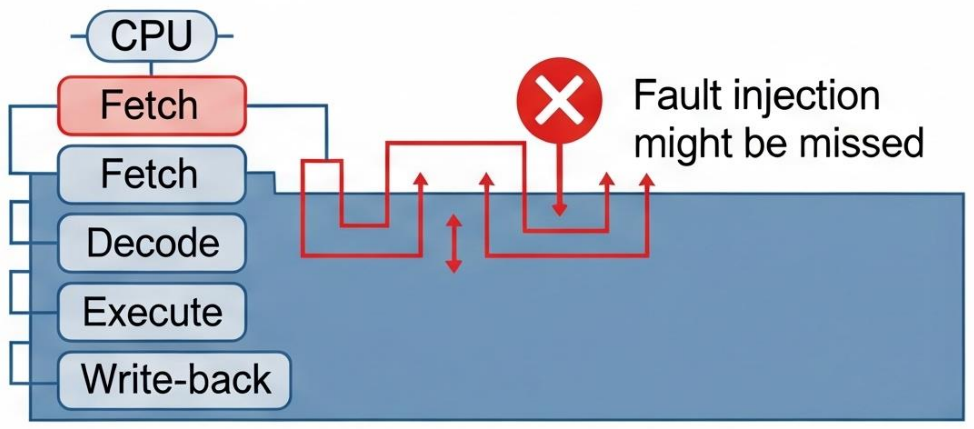
Sub Blocks

- Front-End
- Load-Store
- Shared L2 Cache
- Miscellaneous block
- Common RAS logger

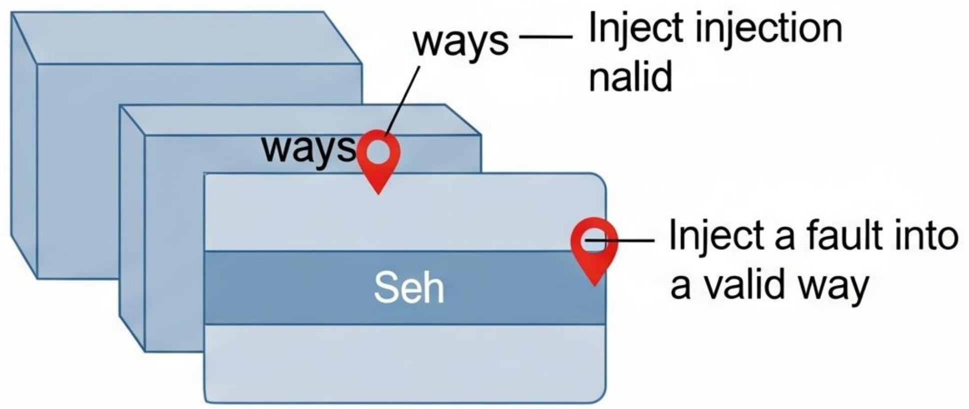


Front-end RAS Challenges

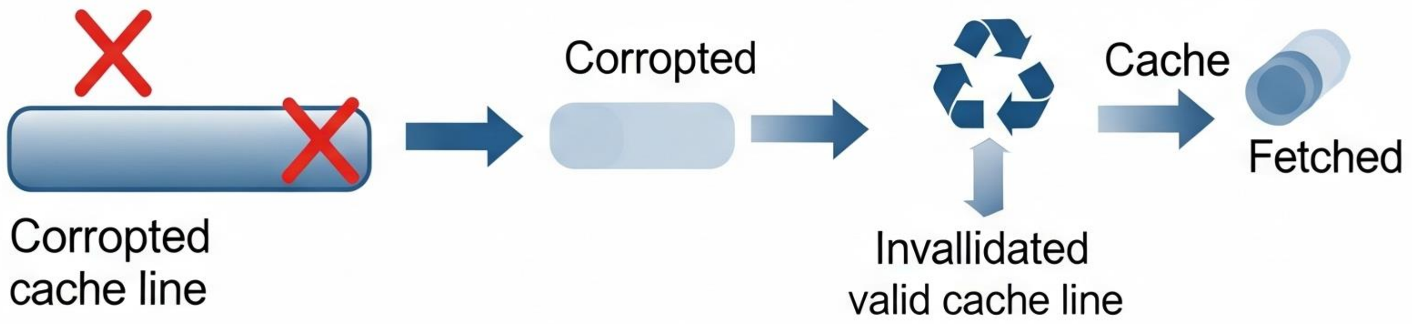
Speculative Execution



Power Optimization Challenges


ways — Inject injection nalid
ways
Seh
Inject a fault into a valid way

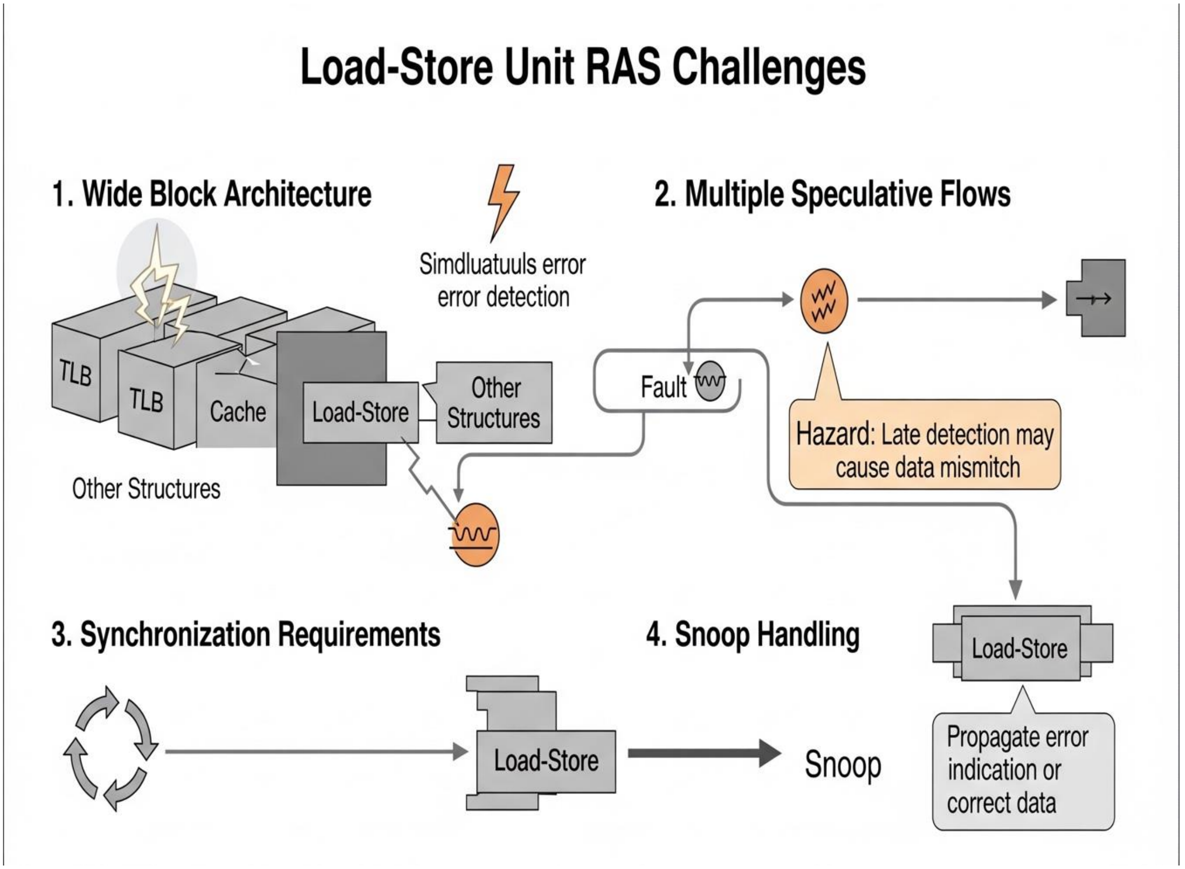
Complexity of Self-Correction



Bookkeeping

Load-Store Challenges

- **Wide Block Architecture**
  - Parallel TLB, cache, and structural operations increase potential failure points. Error detection must cover all sub-blocks concurrently, raising concurrency risks in error handling.
- **Multiple Speculative Flows**
  - Multiple instructions can be in progress at the same time.
  - If an error is found late in one path, others might miss it, causing mismatched results.
- **Synchronization Requirements:**
  - Coordinate error detection, handling, and logging so it's accurate, complete, and happens only once.
  - Maintain forward progress and align with snoops to prevent hangs, deadlocks, or stale data.
- **Snoop Handling:**
  - Handle snoops during error events
  - Ensure snoops get correct data or propagate error indication
  - Maintain system-level forward progress

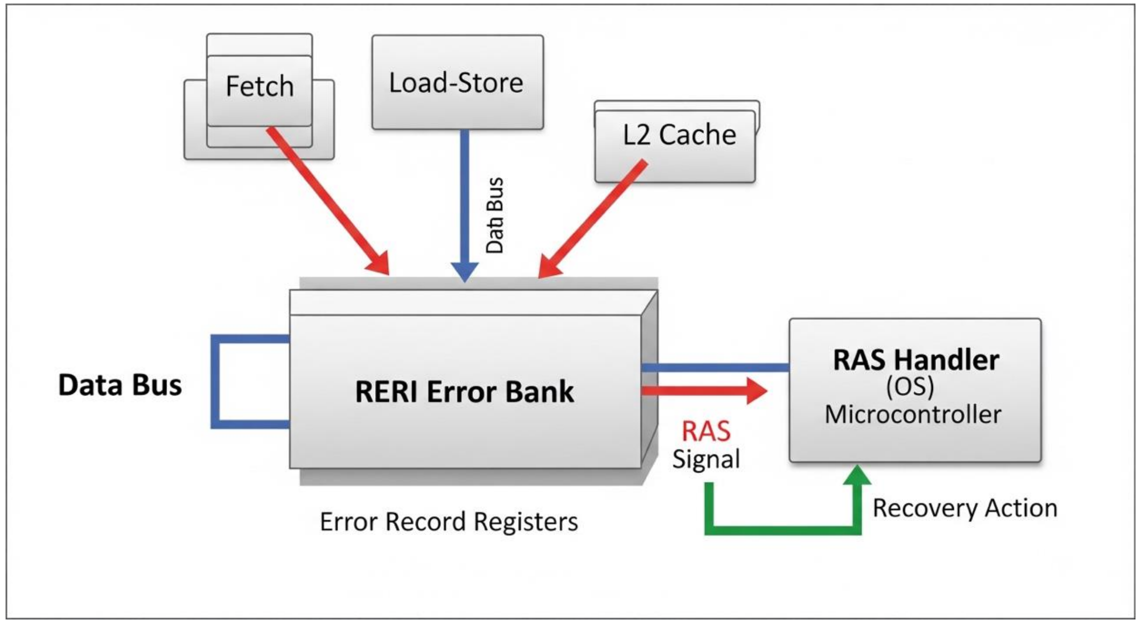

Load-Store Unit RAS Challenges
1. Wide Block Architecture
Simdluatuuls error error detection
2. Multiple Speculative Flows
TLB
TLB
Cache
Load-Store
Other Structures
Other Structures
Fault
Hazard: Late detection may cause data mismitch
3. Synchronization Requirements
4. Snoop Handling
Load-Store
Load-Store
Snoop
Propagate error indication or correct data

Shared Cache Challenges

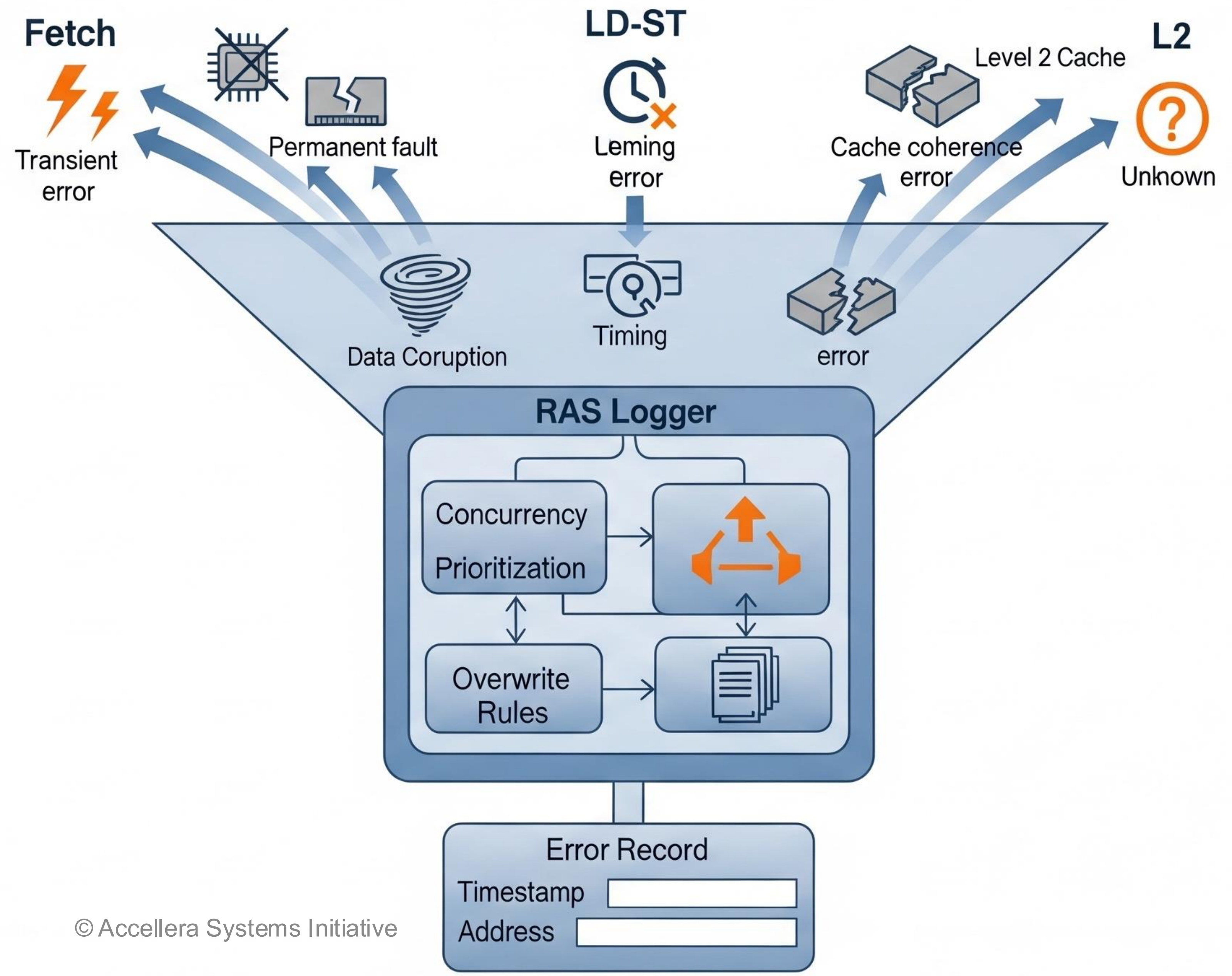
- **Severity Mapping Complexity** – Uncorrected error severity varies by memory array type *(e.g., 2B ECC on replace = UED/DATA CE, while SNP_Filtor/Tag/State = UEC)*
- **Late-Manifesting Bugs** – L2 home node issues often appear after high transaction counts; requires relaxing checkers for affected UEC addresses.
- **Bus Error Propagation** – Ensuring CHI-E DERR/NDERR errors travel correctly from fabric to LS and are properly recorded in L2 (and vice versa).
- **Mixed Transaction Scenarios** – Handling a mix of no corruption, memory corruption, and bus errors while ensuring complete error recording.
- **Checker Relaxation** – Adjusting data and cache coherency checkers to avoid consuming poisoned data.
- **Victim ECC/Bus Error Injection** – Validating IP's generated severity type matches expectations.

RAS Logging Challenges (RERI)

- **RERI Error Bank:** The central element of the architecture, serving as the logger for all detected errors. It contains a set of memory-mapped registers to store error records.
- **Micro-architectural Blocks:** Components like the Fetch, Load-Store, and L2 Cache units detect errors and report them to the RERI Error Bank.
- **Data Bus:** Errors are communicated from the micro-architectural blocks to the RERI Error Bank via a shared bus.
- **RAS Signal:** The Error Bank signals a RAS Handler when an error is logged.
- **RAS Handler:** A component, such as an operating system (OS) or a microcontroller, that receives the RAS signal and processes the error information.
- **Recovery Action:** Based on the error information received, the RAS Handler determines and initiates appropriate recovery actions, which can range from terminating a process to restarting the entire system.

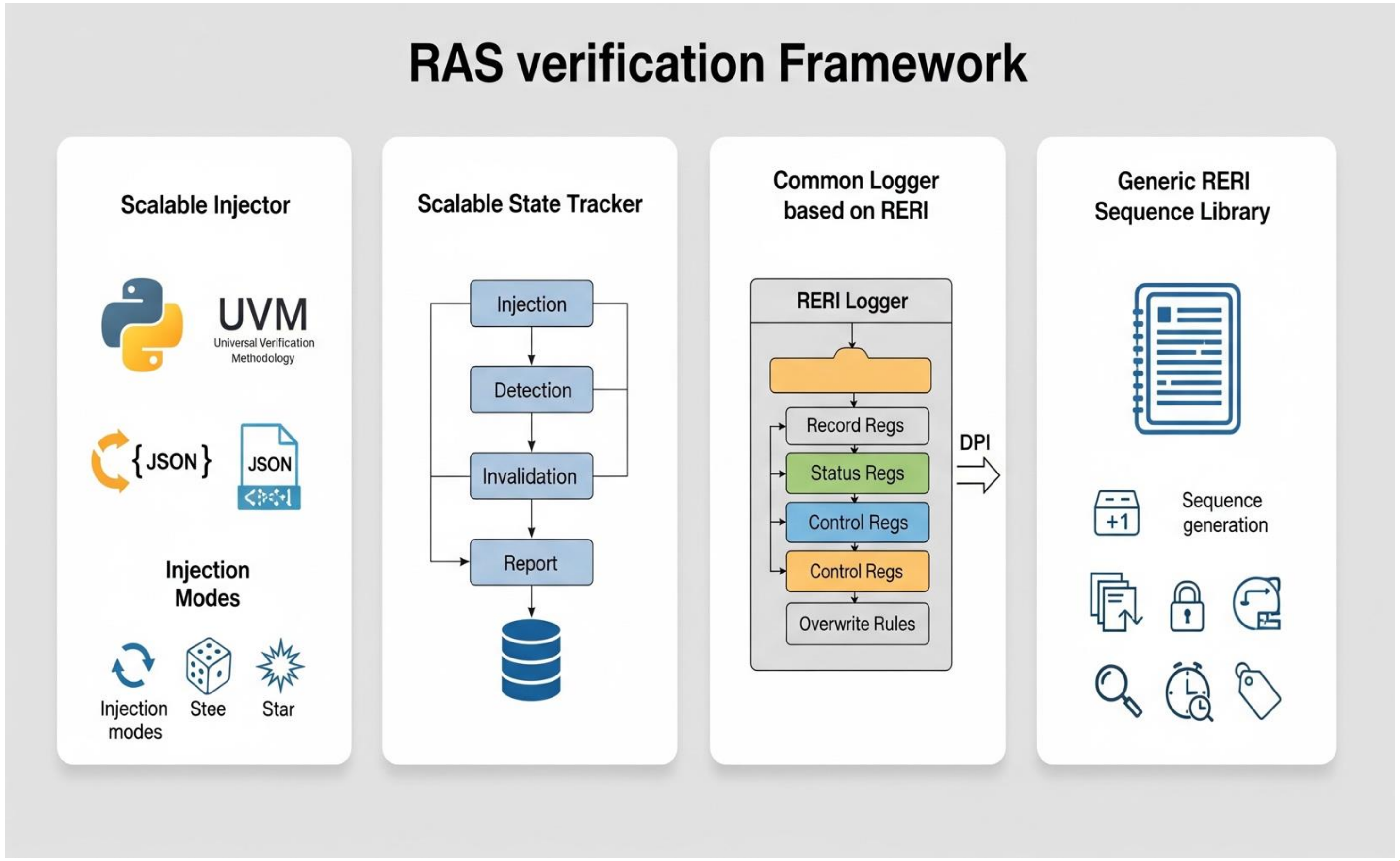


Verification Challenge Funnel

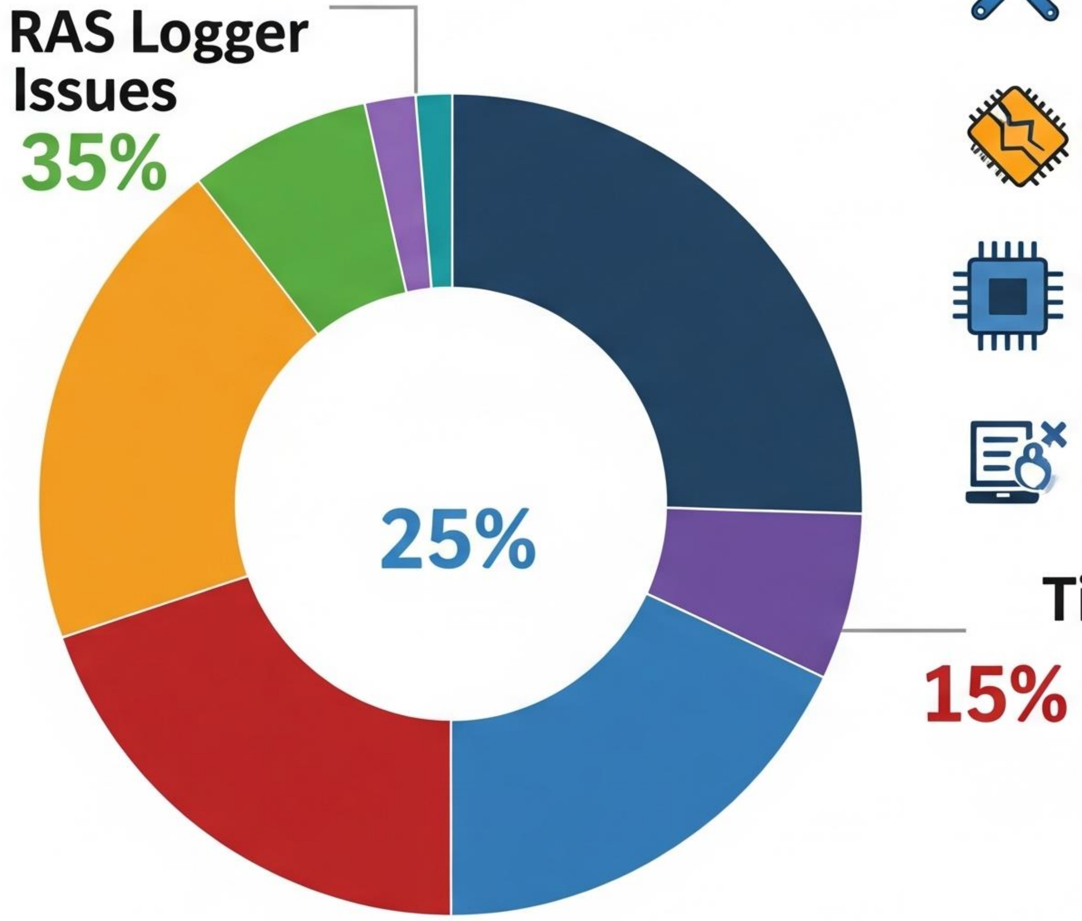


Scalable UVM Framework

- The UVM-based RAS verification framework is founded on four core pillars.
  - **Scalable Injector :**
    - Python based Injector
    - UVM based injector
      - uvm agent created per structure
  - **Scalable state tracker** :
    - Central database to track various states
  - **Common logger based on RERI :**
  - **Generic RERI sequence library :**
    - Correctable counter
    - Record override/retain
    - Read-in-progress
    - Timestamp
    - Locator/descriptor



Key RAS Bug Categories



Status signal mismatches

Status signal mismatches
Descriptor comparision failures

Parity/ECC Eerorrs

Cache parity issues
Memory ECC handing
Error correction failures

Timeout/Hang Issues

Unresolved memory operations

Logging not updading correctlys

Error Handling

DE timeout conditions

Data mismatches,
Unresolved memory operatio

Data Integrity

Coruption scenarios
Bypass failurs

Bypass failurs

Interrupt processing
Exception handling

Error response issues

Key Takeaways

- **Standardized Logging is Essential:** The **RISC-V RERI specification** provides a standardized, memory-mapped interface for a central logger to capture errors from various micro-architectural blocks like the Fetch, Load-Store, and L2 Cache. This allows a RAS handler to receive a signal and initiate a **Recovery Action**.
- **Challenges are Micro-architectural:** Verification must address the unique complexities of individual CPU sub-units. For example, the **Load-Store unit** faces issues with **Wide Block Architecture**, **Multiple Speculative Flows**, and **Snoop Handling** that can lead to data mismatches, hangs, and deadlocks. The **Front-end** deals with challenges related to **Speculative Execution** and **Self-Correction**.
- **Systematic Verification is Key:** A comprehensive and scalable verification framework is the solution to these challenges. A UVM-based framework is founded on three core pillars:
  – **Scalable Injector:** Uses automated UVM agents and various injection modes (Persistent, Random, Simultaneous).
  – **Scalable State Tracker:** A central database tracks the state of an error from injection through detection and invalidation to reporting.
  – **Common Logger:** An RERI-compliant logger that models the necessary registers and is accessed via DPI.

Conclusion

- Verifying RAS in modern microarchitectures is challenging
- Microscopic complexity meets system-level reliability demands
- Requires a multi-faceted verification approach

Questions

Finalize slide set with questions slide