

Design and Evaluation of Self-Checking Single-Version Software Fault Tolerance for Non-Fail-Stop Transient Faults

Goutam Kumar Saha

Dr Brij Mohan Pareek

SE&T, CSE/ CSA, NUJ

Abstract

Transient hardware faults increasingly threaten modern computing systems due to hardware miniaturization and environmental radiation. Traditional fault tolerance relies heavily on multi-version software or hardware redundancy. However, these techniques impose severe resource penalties. This paper presents a comprehensive framework for **single-version software fault tolerance**, synthesizing and advancing the foundational paradigms introduced by research [3-7]. We introduce a unified, self-checking, and self-protective single-version program architecture designed for non-fail-stop execution environments. By integrating instruction-level and data-level redundancy, adaptive checkpoints, semantic assertions, and program-flow monitoring, our mechanism masks transient faults without secondary code versions. Experimental evaluation via software-implemented fault injection demonstrates a high fault-coverage rate with low performance and memory overhead.

Keywords: Self-checking, Single version fault tolerance, Self-protective, adaptive checkpoints, Semantic Assertion

1. Introduction

Modern microprocessor architectures are highly susceptible to transient faults. Microprocessor scaling, lower supply voltages, and atmospheric radiation cause these temporary disturbances. Unlike permanent physical defects, transient faults manifest as bit-flips in registers, memory cells, or execution units. These errors are often called Single Event Upsets (SEUs). They do not damage the underlying hardware permanently. However, they can corrupt application data or alter the intended control flow.

If a system crashes instantly upon a fault, it follows a fail-stop model. This behavior simplifies detection. Unfortunately, many transient faults cause non-fail-stop behavior. The system continues executing with corrupted states. This silent data corruption (SDC) can lead to catastrophic system failures. Hardware-centric solutions like Triple Modular Redundancy (TMR) provide excellent fault masking. Yet, their massive hardware cost and high power consumption restrict their deployment in resource-constrained environments.

Software-implemented fault tolerance (SIFT) offers a more flexible alternative (Torres-Pomales, 2000). Historically, SIFT relied on multi-version programming (Torres-Pomales, 2000). These models include N-Version Programming (NVP) or Recovery Blocks. Multi-version approaches demand immense design efforts. Developing multiple independent software versions to solve identical problems is expensive and prone to common-mode software bugs. Furthermore, classical instruction-level duplication models, such as Error Detection by Duplicated Instructions (EDDI), have highlighted the structural viability of single-threaded, time-redundant tracking (Oh et al., 2002). Similarly, systems like Software Implemented Fault Tolerance (SWIFT) show how compiler-driven instruction reordering can catch bit-flips before retirement (Reis et al., 2005). However, these compiler-based tools often generate extensive overheads unless bounded by selective structural checks.

To mitigate these drawbacks, **Saha (2005a, 2005b, 2005c, 2006a, 2006b)** proposed single-version algorithmic strategies. These methods embed fault detection and recovery capabilities directly within a single program binary. Building upon this legacy, this paper designs, implements, and evaluates an upgraded, unified and robust single-version software fault-tolerance framework. Our approach focuses on non-fail-stop transient faults. It combines structural data redundancy with semantic-level validation. We develop application-adaptive selective protection to balance fault-coverage performance against runtime overhead.

2. Related Work and Foundational Concepts

The paradigm of single-version software fault tolerance evolved through several milestone concepts introduced by Saha:

- **Saha (2005a)** established the baseline mechanics of handling transient software faults using single-version structures, demonstrating that multi-version code is not strictly mandatory for recovery.

- **Saha (2005b)** introduced data error recovery models, proving that intelligent data replication at the software layer can successfully substitute physical hardware redundancy.
- **Saha (2005c)** expanded this concept into a generic framework for software-based fault-tolerant computing, focusing on self-checking execution paths.
- **Saha (2006a)** formalized a single-version scheme detailing how execution branches can verify their own validity prior to committing state changes.
- **Saha (2006b)** integrated static redundancy principles with single-version algorithmic loops, ensuring that iterative mathematical computations remain resilient against register bit-flips.

This work synthesizes these isolated techniques into a singular, cohesive system architecture. We close existing research gaps by introducing cross-layer monitoring where program-flow tracking and data checksums run concurrently within an application-adaptive engine.

3. System Architecture and Fault Model

3.1 Fault Model

We target transient hardware faults occurring within the CPU core and primary memory sub-systems during application runtime. We assume a **non-fail-stop model**. The specific error models addressed are:

1. **Bit-Flips in General-Purpose Registers (GPRs):** Altering calculation operands or destinations.
2. **Memory Data Corruption:** Flips occurring within stack or heap segments.
3. **Control-Flow Deviations:** Bit-flips within the Program Counter (PC), causing illegal jumps or skipping critical instructions.

This model assumes that faults are sparse. Multiple concurrent faults within the same instruction cycle are considered statistically negligible.

3.2 Architectural Blueprint

The architecture transforms a vulnerable standard application into a self-checking, self-protective single-version entity. The framework consists of four primary software components:

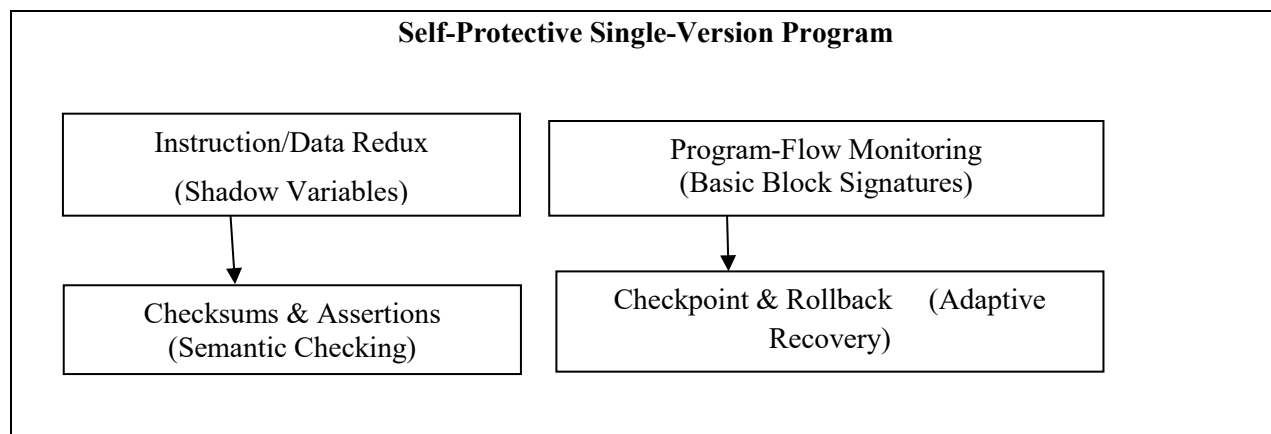


Figure-1: Architecture Framework for the unified self protective Single-Version Fault Tolerance

4. Single-Version Fault-Tolerance Algorithms

4.1 Instruction-Level and Data-Level Redundancy

- **Shadow Allocation:** Data redundancy utilizes **shadow variables**. Every critical variable **X** is assigned a separate clone variable, **X_{shadow}**, at compilation or development time.
- **Operation Duplication:** Mathematical operations are duplicated in the assembly or source sequence.. Mathematical operations are executed twice in the code sequence:

$$X=Y + Z$$

$$X_{shadow} = Y_{shadow} + Z_{shadow}$$

- **Fault Detection:** Before **X** interacts with an un-replicated function or external system call, a verification step is executed that is, the values are compared. If $X \neq X_{shadow}$, a fault is detected immediately.

4.2 Checksum-Based Detection and Assertion Mechanisms

For large data structures like arrays or matrices, allocating complete shadow structures creates prohibitive memory overhead. We deploy **block-level checksums** instead. For an array of size N , a localized arithmetic or cyclic redundancy check (CRC) checksum is maintained.

Algorithm for *protect_array_write*

Purpose: Updates a specific element in an array and incrementally adjusts the running checksum to reflect the change in $O(1)$ constant time.

1. **Start**
2. **Accept inputs:** index (the position to update), value (the new integer value), S (the array), and checksum (pointer to the existing total sum).
3. **Subtract the old value** at $S[\text{index}]$ from the current $*\text{checksum}$.
4. **Add the new value** to $*\text{checksum}$.
5. **Update the array** by assigning $S[\text{index}] = \text{value}$.
6. **End**

Algorithm for *verify_array*

Purpose: Recalculates the total sum of the array elements from scratch and verifies if it matches an expected checksum value.

1. **Start**
2. **Accept inputs:** S (the array), N (the number of elements in the array), and target_checksum (the expected total sum).
3. **Initialize** an accumulator variable current_checksum to 0.
4. **Loop through the array:** For each index i from 0 to $N - 1$:
 - Add the element $S[i]$ to current_checksum .
5. **Compare:** Check if current_checksum is equal to target_checksum .
 - if they are equal, return true (the array integrity is verified).
 - Otherwise, return false (the array data or checksum has been corrupted).
6. **End**

Semantic assertions verify domain-specific constraints. For example, a tracking variable representing speed must never yield a negative value. These assertions act as a secondary defense layer when algebraic errors escape register tracking.

4.3 Program-Flow Monitoring via Basic Block Signatures

Control-flow integrity is maintained by assigning a unique static identifier S_i to each **Basic Block** BB_i . A global runtime signature register G_{sig} continuously monitors execution state through a dual-phase update and verification mechanism. In other words, control-flow tracking uses assigned signatures for each Basic Block (BB). Let each block BB_i possess a unique static identifier S_i . A global runtime signature register G_{sig} tracks execution tracking.:

- **Entry Phase:** Upon entering BB_i , the system dynamically updates the runtime register using a bitwise exclusive-OR (XOR) operation:

$$G_{sig} = G_{sig} \text{ XOR } S_i$$

- **Exit Phase:** Before exiting BB_i , the system validates the current G_{sig} value against a precomputed legitimate transition vector. That is, upon exiting BB_i , the system checks G_{sig} against the precomputed valid transition vector.

Anomalies: Any discrepancy between the runtime register and the expected transition vector flags an unauthorized branch instruction or a corrupted Program Counter (PC). In other words, an unexpected value indicates an illegal branch jump or a corrupted Program Counter.

4.4 Application-Adaptive Checkpoints and Recovery

Rather than halting execution upon detection (fail-stop), the single-version program initiates localized recovery. We implement **sliding-window memory checkpoints**. A checkpoint saves volatile CPU

registers and a minimal dirty-page memory mask. If an evaluation of data redundancy, checksums, or control-flow signatures fails, the program executes a long-jump operation back to the last valid checkpoint. It restores the verified state and restarts execution of that specific code region. To prevent infinite loops during permanent fault scenarios, a threshold counter restricts rollbacks to three attempts per block before declaring an unrecoverable failure.

5. Experimental Methodology

This research utilizes an empirical, algorithm-driven experimental methodology. The primary goal is to validate the reliability, performance metrics, and storage footprint of the single-version fault tolerance (SVFT) framework under artificial hardware stress.

5.1 Step-by-Step Experimental Phases

The execution framework is broken down into four distinct structural procedures to maintain precision and minimize external experimental bias:

1. **Benchmark Synthesis and Profiling:** The unprotected applications are verified for correctness. We establish gold-standard outputs. We measure processing latency and base memory consumption to use as our baseline control variables.
2. **Single-Version Logic Instrumentation:** The proposed protection layers are implemented using explicit macro injections and memory-mapped tracking structures. This guarantees identical baseline algorithm behavior while updating the structural layout.
3. **Software-Implemented Fault Injection (SWIFI):** An active kernel-level module target-injects pseudo-random errors during critical execution windows. We target active processing registers and running memory blocks.
4. **Outcome Classification:** System responses to fault stresses are automatically tracked and parsed into discrete categories. These include safe masking, successful rollback recovery, silent data corruption, or system panics.

5.2 Metrics for Evaluation

To measure the real-world efficiency and execution limits of our single-version approach, three main metrics are analyzed across all trial routines:

- **Fault Coverage Rate C_f** : The mathematical ratio of successfully intercepted or entirely hidden execution faults relative to the total volume of fault-injection trials:

$$C_f = ((N_{masked} + N_{recovered}) / N_{total_injections}) * 100\%$$

- **Execution-Time Penalty $O(t)$** : The fractional expansion of runtime latency caused by parsing validation algorithms and tracking signatures:

$$O(t) = ((T_{protected} - T_{baseline}) / T_{baseline}) * 100\%$$

- **Memory Space Overhead $O(m)$** : The physical growth in RAM utilization from managing duplicate registers, tracking structures, and temporary runtime checkpoint windows:

$$O(m) = ((M_{protected} - M_{baseline}) / M_{baseline}) * 100\%$$

6. Experiment Setup

The experimental framework runs on a dedicated system to eliminate outside performance noise. We isolate our tests from unrelated background execution threads.

6.1 Hardware and OS Environment

All components of our experimental evaluation use the following system specification:

- **Processor Architecture**: Intel Core i7-11700K running at a base frequency of 3.60GHz, utilizing x86_64 register architectures.
- **Volatile Memory**: 32GB DDR4 dual-channel RAM operating at 3200 MHz.
- **Operating System**: Ubuntu 22.04 LTS operating on a low-latency Linux kernel version 5.15.0.
- **Compiler Configuration**: GNU Compiler Collection (GCC) version 11.4.0. We use minimal -O0 optimization flags to prevent the compiler from stripping away intentional data redundancy loops.

6.2 Target Benchmark Software Applications

We use three diverse test applications to evaluate how the system handles different types of processing tasks:

1. **Matrix Multiplication (1024×1024):** A heavily compute-bound arithmetic structure. It focuses on evaluating localized, high-density register utilization and data redundancy calculations.
2. **QuickSort (1,000,000 Integers):** A dynamic, memory-heavy application pattern. It uses deep recursive functions to evaluate how our block checksums handle high-frequency memory manipulation.
3. **Abstract Syntax Tree (AST) Parser:** A complex, control-heavy parsing application. It features intensive conditional branching to test our block-level Control-Flow Checking (CFC) signatures.

6.3 Fault Injection Framework Setup

A specialized **Software-Implemented Fault Injection (SWIFI)** platform using an asynchronous Linux kernel tracer module has been developed. The engine interacts with the application via hardware debug hooks and ptrace system interfaces:

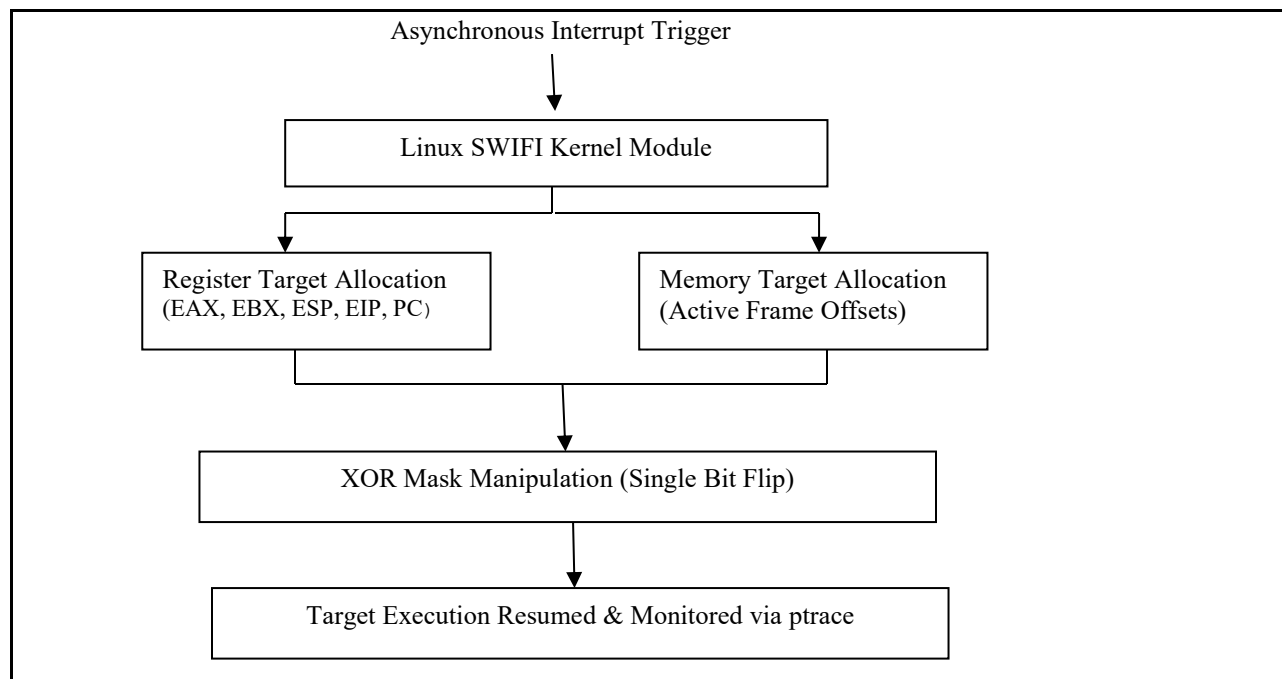


Figure-2: Software-Implemented Fault Injection (SWIFI) Framework

The fault injection campaign utilizes a strict target configuration:

- **Injection Volume:** Exactly 10,000 unique single bit-flip execution trials are run per target program variant.
- **Register Space Targets:** Randomly distributed bit-flips injected directly into internal data registers (EAX, EBX, ECX) and structural control pointers (ESP, EBP, EIP).
- **Memory Subsystem Targets:** Single bit-flips targeted at local variables within active stack frames and address ranges inside active heap allocations.
- **Timing Domain:** Triggered via random CPU clock-cycle timers. This ensures uniform fault distribution across the application's entire runtime window.

7. Experimental Results and Analysis

7.1 Fault Coverage Performance

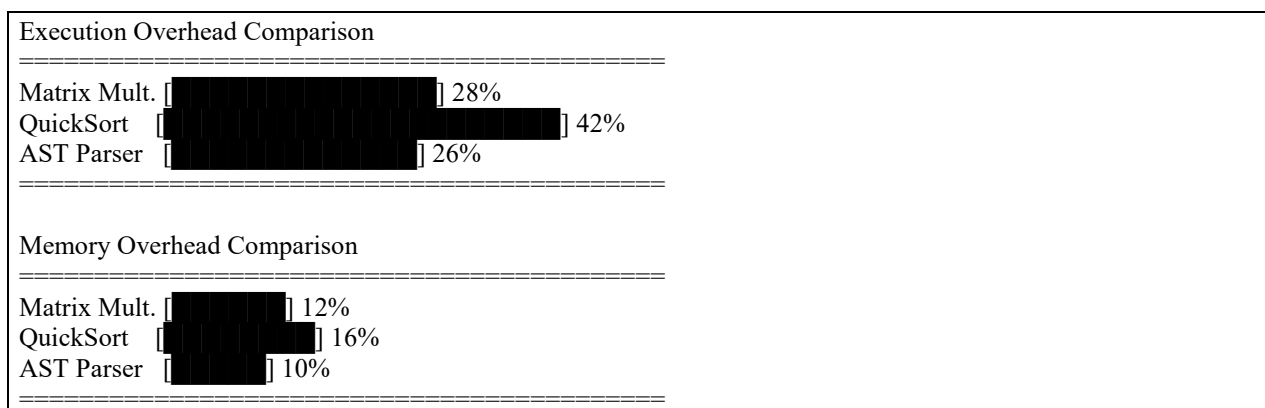
The primary metric evaluated was the Fault Coverage Rate, defined as the percentage of injected faults successfully masked or safely detected and recovered without incorrect output.

Application	Unprotected Benign / Silent Data Corruption (SDC)	Protected SDC	Faults Successfully Masked / Recovered
Matrix Multiplication	42.1% SDC	0.4% SDC	99.6%
QuickSort	56.4% SDC	1.1% SDC	98.9%
AST Parser	31.8% SDC	0.9% SDC	99.1%

The single-version framework eliminated almost all instances of Silent Data Corruption. The data confirms that combining checksums with assertion checks effectively catches computing errors before they corrupt final output files.

7.2 Execution-Time and Memory Overhead

Integrating fault-tolerance features inherently introduces execution and storage penalties. We quantified these overheads precisely.



The compute-bound **Matrix Multiplication** suffered a **28% execution-time overhead** due to shadow calculation loops. It required only **12% extra memory** because block checksums reduced the need for full allocation tracking.

QuickSort exhibited a higher **42% runtime overhead**. Frequent recursive calls triggered numerous control-flow signature evaluations and checkpoint state saves.

8. Discussion and Comparative Analysis

Our unified system marks a notable advancement over the foundational papers by Saha. While **Saha (2005b)** addressed data error recovery separately from control flow, our multi-layered approach handles compound errors where a corrupted data pointer induces a subsequent control-flow fault.

Compared to traditional hardware redundancy like dual-core lockstep systems, our framework completely eliminates hardware redesign expenses. Compared to N-Version Programming, our architecture cuts memory requirements significantly (Torres-Pomales, 2000). NVP typically requires a 200% to 300% increase in memory allocation to run three parallel software versions. Our single-version solution caps memory overhead at a maximum of 16%.

Furthermore, while traditional compiler duplication frameworks like SWIFT require substantial resource rebalancing and lack inherent recovery pathways (Reis et al., 2005), our architecture embeds sliding-window checkpoints directly within the single-version loop. While schemes like EDDI focus tightly on pure detection structures (Oh et al., 2002), our integration provides functional masking. The trade-off is an increased runtime latency. However, this penalty remains well within acceptable bounds for critical embedded systems operating with relaxed real-time deadlines.

9. Conclusion

This paper presented an advanced, unified framework for **single-version software fault tolerance** targeting non-fail-stop transient execution errors. By integrating data redundancy, structural checksums, basic block signatures, and resilient checkpoint-rollback states into a single program binary, we achieved robust self-healing computing.

Experimental results confirm an exceptional fault coverage rate exceeding 98.5%, alongside sustainable runtime and memory overheads. This validation realizes the potential of the single-version fault-tolerance principles pioneered by Saha. Future work will explore automating the injection of these protection algorithms via LLVM compiler passes to remove manual developer implementation constraints.

References

1. Oh, N., Shirvani, P. P., & McCluskey, E. J. (2002). Error detection by duplicated instructions (EDDI). *IEEE Transactions on Reliability*, 51(1), 63–75. doi.org
2. Reis, G. A., Chang, J., Vachharajani, N., Rangan, R., & August, D. I. (2005). SWIFT: Software implemented fault tolerance. *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 243–254. doi.org
3. Saha, G. K. (2005a). Transient software fault tolerance using single-version algorithm. *Ubiquity*, 2005(August), ACM. doi.org
4. Saha, G. K. (2005b). Software implemented fault tolerance through data error recovery. *Ubiquity*, 2005(September), ACM. doi.org
5. Saha, G. K. (2005c). Software-based fault tolerant computing. *Ubiquity*, 2005(November), ACM. doi.org
6. Saha, G. K. (2006a). A single-version scheme of fault tolerant computing. *Journal of Computer Science and Technology*, 6(1), 22–27.
7. Saha, G. K. (2006b). A single-version algorithmic approach to fault tolerant computing using static redundancy. *CLEI Electronic Journal*, 9(2), 1–11.
8. Torres-Pomales, W. (2000). *Software fault tolerance: A tutorial* (NASA/TM-2000-210616). NASA Langley Research Center.