

Design and Experimental Evaluation of a Single-Version Software Fault-Tolerance Algorithm for Transient Execution Errors

Goutam Kumar Saha

Dr Brij Mohan Pareek

SE&T, CSE/CSA, NUJ

Abstract

Transient execution faults can corrupt data and intermediate results without terminating a running application. These faults may cause silent data corruption. Conventional fault-tolerance methods often depend on hardware redundancy, replicated execution, or independently developed software versions. This paper investigates an alternative based on single-version software fault tolerance. Building on prevailing research works on transient software fault tolerance, recovery, data-error recovery, and static redundancy, this study proposes an integrated single-version algorithm. The algorithm combines selective computation redundancy, data-integrity checks, assertions, program-flow monitoring, and checkpoint-based recovery. A small numerical application is used to demonstrate the design. A Python prototype is evaluated against an unprotected baseline using 10,000 integer inputs. The preliminary benchmark uses 50 timed repetitions and compares median execution times. The baseline completes in 0.359 milliseconds, while the protected implementation takes 1.451 milliseconds. This corresponds to approximately 304.3% execution-time overhead. The protected implementation also detects and recovers from a deliberately injected accumulator error in the demonstration. These measurements establish functional feasibility, not general fault coverage. The experiment is limited to a simple workload and one fault-injection pattern.

Keywords: single-version fault tolerance, transient execution faults, software redundancy, data integrity, assertions, checkpoint recovery, program-flow monitoring, fault injection, execution-time overhead

1. Introduction

1.1 Background

Reliable software is essential in embedded systems, scientific computing, industrial control, and other applications where incorrect results can have serious consequences. However, transient faults can affect execution even when the underlying hardware remains operational. A transient fault may change a register value, corrupt an intermediate result, or alter a control-flow decision. The program may continue running. Its final output can nevertheless be incorrect. Fault tolerance addresses this problem through

detection, recovery, masking, or controlled degradation. Established approaches include recovery blocks, N-version programming, algorithm-based fault tolerance, and checkpoint-based rollback (Avizienis, 1985; Huang & Abraham, 1984; Randell, 1975). Saha (2003, 2005a, 2005b, 2005c, 2006a, 2006b) explored software-level fault tolerance, recovery, single-version algorithms, and static redundancy. These publications provide the foundation for investigating whether a single program can protect itself through additional checking and recovery operations. A single-version approach does not prohibit redundant computation. It avoids the requirement for multiple independently developed software versions. The same program can repeat selected calculations, validate intermediate states, and restore a saved state after detecting an error.

1.2 Problem statement

Several protection mechanisms are available, but each has limitations.

Instruction redundancy can compare repeated computations. Checksums can reveal inconsistencies in data. Assertions can test application-specific conditions. Program-flow monitoring can detect unexpected execution paths. Checkpointing can restore a previously valid state.

However, each mechanism introduces cost. Redundant calculations require additional instructions. Checksums require extra operations. Assertions consume execution time. Checkpoints require storage. Recovery may repeat previous work.

The central problem is to integrate these mechanisms into a single-version program and evaluate the trade-off between fault protection and computational overhead.

1.3 Research objectives

The general objective is to design, implement, and evaluate a single-version software fault-tolerance algorithm for selected transient execution faults.

The specific objectives are:

1. Develop a self-checking and self-protective single-version program.
2. Introduce selective computation and data redundancy.
3. Use checksums and assertions to detect selected inconsistencies.
4. Monitor basic program-flow progression.
5. Preserve validated checkpoints and recover from detected errors.

6. Measure code redundancy and execution-time overhead.
7. Demonstrate fault detection and recovery using controlled fault injection.

1.4 Research questions

This study addresses the following questions:

- Can a single-version program detect and recover from a selected transient execution error?
- What additional computation and code are required for protection?
- How much execution-time overhead does the prototype introduce?
- What limitations prevent the prototype from providing comprehensive fault tolerance?

2. Related Work

2.1 Single-version software fault tolerance

These works motivate the use of software-level protection within a single program. The present study develops a compact implementation that combines selected protection mechanisms in one experimental framework. Saha's publications provide the primary conceptual basis for this study. Saha (2003) addressed transient software fault tolerance through recovery. Subsequent publications examined single-version algorithms, data-error recovery, software-based fault tolerance, and static redundancy (Saha, 2005a, 2005b, 2005c, 2006a, 2006b). The relationship should not be interpreted as proof that the proposed combination is novel. Establishing novelty requires a detailed comparison with the original algorithms and other related implementations. Montezanti et al. (2020) investigated soft-error detection and recovery using replication combined with checkpointing. Their work demonstrates how replication and recovery can be integrated to improve resilience in parallel applications.

2.2 Software redundancy

Redundancy adds information or computation that supports error detection or recovery. It may be introduced through repeated instructions, duplicate values, checksums, control-flow signatures, or saved program states.

N-version programming uses independently developed implementations to reduce the risk of common design errors (Avizienis, 1985). By contrast, the present approach uses one primary implementation with selected redundant operations.

This reduces the need for independent program versions. However, it does not eliminate common-mode errors. A fault may affect both the original computation and its checking mechanism.

2.3 Algorithm-based fault tolerance

Algorithm-based fault tolerance embeds mathematical relationships into an application. Huang and Abraham (1984) demonstrated this principle for matrix operations.

A checksum can be calculated before or after a computation. The relationship is then checked at a later point. A mismatch indicates possible corruption.

The usefulness of a checksum depends on its construction. Simple additive checksums can miss certain error patterns. Independent validation and semantic assertions may provide additional protection.

2.4 Checkpoint recovery

Checkpointing saves a representation of the program's state. When an error is detected, the program restores a previously valid checkpoint and repeats the affected computation.

Young (1974) studied the selection of checkpoint intervals. Elnozahy et al. (2002) reviewed rollback-recovery methods in message-passing systems.

Checkpointing is useful only when the saved state is valid and the affected operation can be repeated safely. Therefore, the proposed algorithm validates its state before accepting a checkpoint.

2.5 Research gap

The literature establishes the individual principles of redundancy, checking, and recovery. This study focuses on a small integrated implementation.

Its immediate contribution is an executable demonstration and a measurement procedure. The prototype is not intended to establish universal fault coverage. It provides a basis for larger experiments involving additional algorithms, fault types, and protection configurations.

3. Proposed Algorithm

3.1 Design

The proposed algorithm divides the input into chunks. Each chunk is processed independently.

- Before processing a chunk, the program preserves the accumulated result. It computes the chunk result and repeats the calculation for comparison. It also evaluates selected integrity and semantic checks. A simple control-flow signature tracks the expected progression through chunks.
- If a discrepancy is detected, the program restores the saved accumulator and recomputes the affected chunk. It then validates the recovered result.

The design contains five components:

1. **Computation redundancy:** Repeat selected calculations and compare their results.
2. **Data-integrity checking:** Verify a checksum relationship.
3. **Semantic assertions:** Check expected properties of the input and intermediate values.
4. **Program-flow monitoring:** Verify the expected sequence of chunk processing.
5. **Checkpoint recovery:** Restore the accumulated result and recompute a faulty chunk.

3.2 Formal model

Let $(P(x))$ be the output of the original baseline program for input (x) .

If a transient fault F occurs, the corrupted program yields an altered output:

$$y_F = P_F(x)$$

The fault-tolerant protected algorithm - denoted by SVFT – processes the program P and input x to produce three distinct metrics:

$$(y^*, d, r) = \text{SVFT}(P, x)$$

Where:

- y^* is the final, potentially recovered system output.
- d represents the fault detection status indicator.
- r indicates the success or status of the recovery mechanism.

The intended behaviour is:

- Accept the result if the required checks pass.
- Recover from a detected error when a valid checkpoint and recomputation are available.
- Report a controlled error if validation or recovery fails.

This model applies only to the defined fault model. It does not imply protection against every possible execution error.

3.3 Algorithm 1: Single-version fault tolerance

Input: Non-negative integer array (A), chunk size (k).

Output: Validated sum of squares and recovery count.

Algorithm Steps:

1. Initialize the accumulated result to zero.
2. Initialize the recovery counter and control-flow signature.
3. Divide the array into chunks.
4. Save the current accumulated result as a checkpoint.
5. Calculate the sum of squares for the current chunk.
6. Repeat the calculation to obtain a reference result.
7. Calculate the data checksum.
8. Verify the expected program-flow signature.
9. Evaluate the semantic assertions.
10. Compare the computed and reference results.
11. If a discrepancy occurs, restore the checkpoint.
12. Recompute the affected chunk.
13. Validate the recomputed result.
14. Continue if validation succeeds.
15. Otherwise, report an unrecoverable error.
16. Validate and return the final result.

3.4 Flowchart

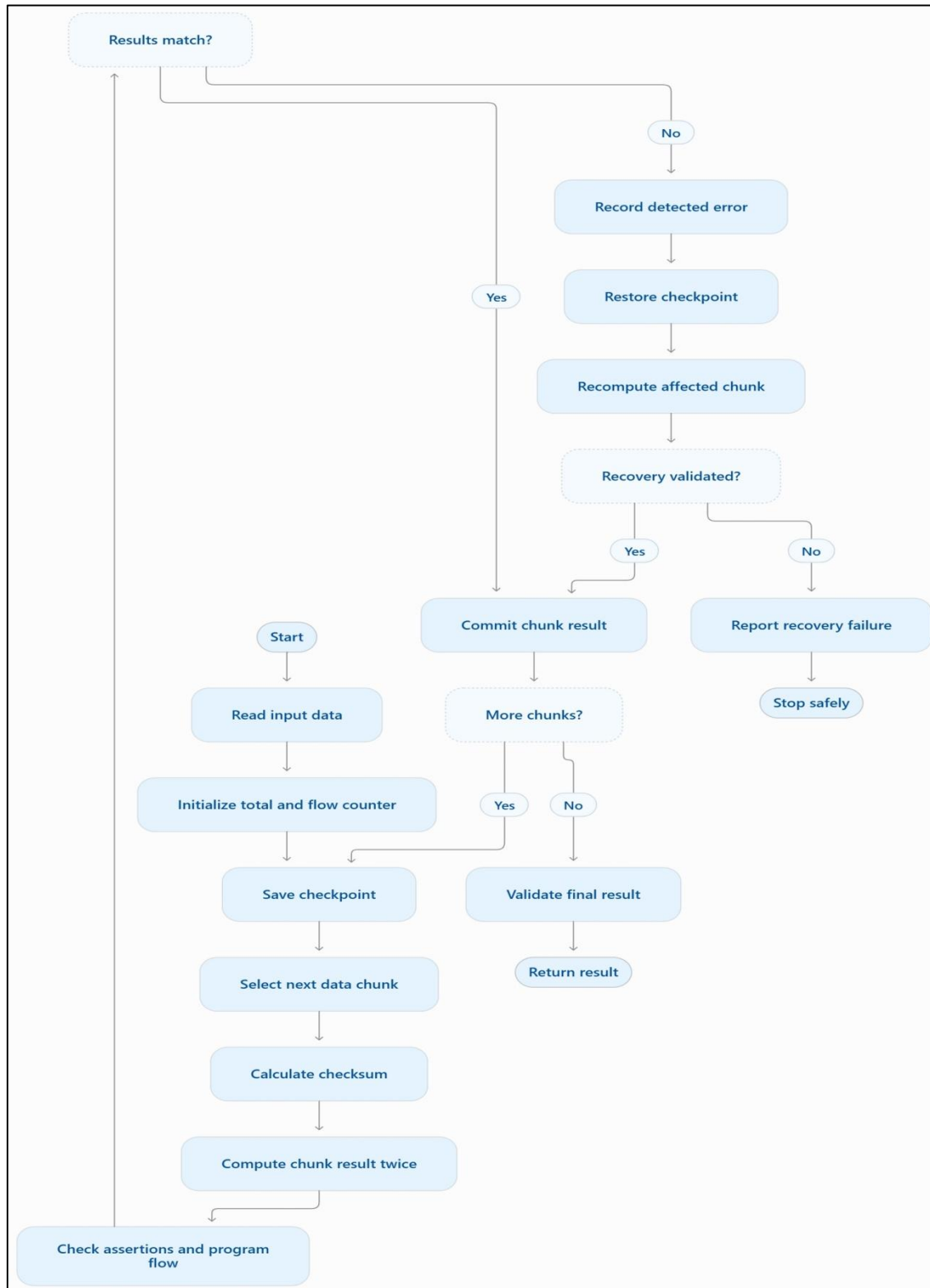


Figure 1. Flowchart of the proposed single-version fault-tolerance algorithm.

The flowchart shows a simplified execution path. A production implementation would also require bounds on recovery attempts, logging of failures, and protection of the checkpoint itself.

Code redundancy represents the additional software required for fault detection and recovery. The flowchart identifies the main sources of that overhead: checkpoints, duplicate computation, assertions, checksums, monitoring, and recovery.

4. Implementation

4.1 Application

A sum-of-squares application is used for the initial demonstration. Given an array (A), it calculates:

$$S = \sum_{i=1}^n a_i^2$$

This workload is simple to verify. It also supports controlled corruption of an intermediate result.

The baseline calculates the sum once. The protected version calculates each chunk result twice and compares the results. It adds integrity checks, assertions, and basic control-flow monitoring.

The demonstration injects a fault by altering the computed accumulator for the first chunk. The checker must detect the mismatch and recover the correct value.

4.3 Implementation limitations

This is a prototype, not a production fault-tolerance library. The duplicate computation is executed in the same process. A fault that corrupts both computations may remain undetected.

The checksum verifies the chunk against its preserved input in this experiment. It does not provide complete protection against corruption of the input before the checksum is calculated. The control-flow signature checks chunk progression but does not cover every instruction or branch.

The recovery procedure also assumes that the original input remains available. A real system may need protected input storage, checkpoint integrity checks, bounded retries, and an explicit unrecoverable-error path.

5. Experimental Methodology

5.1 Experimental setup

The preliminary experiment uses an array containing 10,000 non-negative integers, from 0 through 9,999. The chunk size is 256 elements.

Two implementations are compared:

- **Baseline:** One sum-of-squares calculation.
- **Protected:** Chunk processing, duplicate calculations, assertions, checksum validation, control-flow monitoring, and checkpoint recovery.

Three warm-up runs are performed for each implementation. Each timed configuration is then executed 50 times. The median runtime is reported.

The fault-injection test adds 12,345 to the first chunk's observed result. This simulates corruption of an intermediate accumulator. The expected value is independently recomputed from the preserved chunk.

5.2 Evaluation metrics

Code redundancy

Code redundancy can be measured in several ways. A useful static measure is:

$$R_C = \frac{L_P - L_B}{L_B} \times 100$$

where **L_P** is the number of executable source lines in the protected implementation and **L_B** is the number in the baseline.

The result depends on the counting rules. Blank lines, comments, assertions, and library calls must be handled consistently. Therefore, a reproducible static code-redundancy percentage should be calculated from the final source files rather than estimated from a description.

A separate measure of **computational redundancy** is more direct for this prototype. The sum-of-squares calculation is executed twice per chunk during fault-free protected execution. Thus, the selected arithmetic computation has 100% additional recomputation relative to one execution of that computation. This does not mean the entire program contains exactly 100% more instructions.

Execution-time overhead

Execution-time overhead is calculated as:

$$O_T = \frac{T_P - T_B}{T_B} \times 100$$

where **T_P** is the protected median runtime and **T_B** is the baseline median runtime.

The runtime ratio is:

$$Q_T = \frac{T_P}{T_B}$$

A ratio of 1.0 indicates equal runtime. A ratio of 2.0 indicates that the protected implementation takes twice as long.

Recovery success

Recovery success is evaluated by comparing the fault-injected output with the baseline result. The recovery count records detected and recovered injected errors.

A broader experiment should also measure detection coverage, silent data corruption, false alarms, recovery latency, and memory overhead.

6. Results

6.1 Functional correctness

The baseline produces the expected result:

The **baseline program** yields the expected reference result:

S=333,283,335,000

When evaluating the system under a transient fault, this reference value serves as the ground truth. The protected algorithm's detection status indicator $\mathbf{d}$ evaluates the computed output $\mathbf{y}^*$ against this baseline value:

- $\mathbf{d} = \mathbf{0}$ (No Fault Detected): The protected program successfully matches the reference result, where $\mathbf{y}^* = \mathbf{S}$.
- $\mathbf{d} = \mathbf{1}$ (Fault Detected): The algorithm identifies a mismatch caused by a transient fault, triggering the recovery mechanism $(\mathbf{r})$ to rectify the corrupted state.

The protected implementation produces the same result in fault-free execution.

When the selected accumulator error is injected, the comparison detects the discrepancy. The program restores the saved accumulator and recomputes the affected chunk. The recovered result matches the baseline.

This demonstrates recovery from the specific injected error. It does not establish coverage for other fault types.

6.2 Execution-time results

The following measurements were obtained from a preliminary run of the Python prototype.

Measure	Baseline	Protected
Input size	10,000 integers	10,000 integers
Median execution time	358,894 ns	1,450,974 ns
Median execution time	0.359 ms	1.451 ms
Output in fault-free execution	333,283,335,000	333,283,335,000
Additional computation	None	One repeated sum-of-squares calculation per chunk

The protected program exhibits a **runtime ratio** ($\mathbf{Q_T}$) approximately **4.04 times** that of the baseline program. This corresponds to an **execution-time overhead** ($\mathbf{O_T}$) of **304.3%**, calculated as:

$$O_T = \frac{1,450,974 - 358,894}{358,894} \times 100$$

$O_T \approx 304.3\%$

The measured execution-time overhead is therefore approximately **304.3%** in this particular prototype and execution environment.

This value includes more than duplicate arithmetic. It also includes chunk slicing, assertions, checksum calculation, control-flow checks, and function-level overhead. It must not be interpreted as a universal overhead for single-version fault tolerance.

6.3 Code and computational redundancy

The prototype introduces one repeated sum-of-squares calculation for every chunk. With 10,000 elements and a chunk size of 256, there are 40 chunks.

In fault-free execution, the primary and reference calculations therefore perform 80 chunk-level sum-of-squares operations instead of 40 in the baseline.

The arithmetic recomputation overhead is 100% for this selected operation. However, total program overhead is different because the protected implementation performs additional validation and recovery-related bookkeeping.

The static source-code redundancy percentage remains to be calculated using a documented line-counting method. It should not be inferred from the arithmetic recomputation percentage.

6.4 Fault-injection result

Test	Expected behaviour	Observed behaviour
Fault-free baseline	Correct output	Correct output
Fault-free protected execution	Correct output	Correct output
Injected accumulator error	Detect and recover	One recovery recorded
Final output after recovery	Match baseline	Match baseline

The injected error is detected because the primary and reference results differ. The protected implementation then recomputes the affected chunk.

The test does not examine faults that simultaneously corrupt both computations, alter the preserved input, damage the checkpoint, or bypass the comparison. Those cases require separate experiments.

6.5 Interpretation

The preliminary measurements reveal a clear trade-off. The protected implementation detects the selected error, but it runs more slowly.

This is expected for a design that repeats computations and checks intermediate state. The overhead may be acceptable for some high-integrity applications and excessive for performance-sensitive workloads.

The next optimization should investigate selective redundancy. For example, the program could protect only critical chunks or operations identified through a documented vulnerability assessment. Any reduction in overhead must be evaluated alongside the possibility of reduced detection coverage.

7. Present Limitations

Limited fault model

Only one deterministic accumulator error is injected. The experiment does not cover arbitrary bit flips, branch corruption, memory corruption, or simultaneous faults.

Limited workload

A sum-of-squares program is easy to validate but does not represent all computational workloads. Matrix operations, branching algorithms, array transformations, and stateful applications should be examined in future experiments.

Runtime variability

The reported measurements come from one preliminary execution environment. The environment's processor model, operating system, Python version, and system load were not recorded in the benchmark summary. Consequently, the measurements are illustrative.

Incomplete coverage of protection mechanisms

The prototype includes basic checksum validation, assertions, control-flow progression checks, and checkpoint restoration. It does not implement complete instruction-level monitoring, protected checkpoint

storage, or comprehensive branch-signature validation. Its results should not be generalized to those stronger mechanisms.

8. Discussion

The experiment demonstrates that a single-version program can detect and recover from a selected transient execution error without requiring a second independently developed program version. The method combines redundancy with application-level checks. It also demonstrates the cost of that protection. However, successful recovery from one injected error is not sufficient evidence of comprehensive fault tolerance. Stronger claims require a larger fault-injection campaign and independent validation of outputs. The most important next step is to compare several protection configurations. These should include redundant computation alone, checksum and assertion checking, checkpoint recovery, and the integrated design. Their fault coverage and runtime costs should be reported separately. Such comparisons would reveal whether the combination provides additional protection beyond its individual components.

9. Conclusion

This paper presents a single-version software fault-tolerance algorithm based on computation redundancy, checksum validation, assertions, program-flow monitoring, and checkpoint recovery. The design builds on the research direction established by related publications on transient software fault tolerance, data-error recovery, and static redundancy. A small Python application demonstrates the algorithm. The preliminary benchmark reports a baseline median runtime of approximately 0.359 milliseconds and a protected runtime of approximately 1.451 milliseconds. The corresponding execution-time overhead is 304.3%. The prototype also detects and recovers from a deliberately injected accumulator error. These findings establish the feasibility of the demonstration, not general fault coverage. In future, further experiments will be done to measure static code redundancy, memory overhead, detection coverage, silent data corruption, recovery success, and performance across multiple workloads and fault types. The main research opportunity is to optimize the balance between protection and overhead. Selective redundancy and validated recovery provide a practical starting point for that investigation.

References

- Avizienis, A. (1985). The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12), 1491–1501.
- Elnozahy, E. N., Alvisi, L., Wang, Y.-M., & Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3), 375–408.
- Huang, K.-H., & Abraham, J. A. (1984). Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6), 518–528.
- Montezanti, D., Rucci, E., De Giusti, A., Naiouf, M., Rexachs, D., & Luque, E. (2020). Soft errors detection and automatic recovery based on replication combined with different levels of checkpointing. *Future Generation Computer Systems*, 113, 240–254.
- Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), 220–232.
- Saha, G. K. (2003). Transient software fault tolerance through recovery. *Ubiquity*, 2003(September), Article 2. Association for Computing Machinery.
- Saha, G. K. (2005a). Transient software fault tolerance using single-version algorithm. *Ubiquity*, 2005(August). Association for Computing Machinery.
- Saha, G. K. (2005b). Software implemented fault tolerance through data error recovery. *Ubiquity*, 2005(September). Association for Computing Machinery.
- Saha, G. K. (2005c). Software-based fault tolerant computing. *Ubiquity*, 2005(November). Association for Computing Machinery.
- Saha, G. K. (2006a). A single-version scheme of fault tolerant computing. *Journal of Computer Science and Technology*, 6(1), 22–27.
- Saha, G. K. (2006b). A single-version algorithmic approach to fault tolerant computing using static redundancy. *CLEI Electronic Journal*, 9(2).
- Young, J. W. (1974). A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9), 530–531.