

Single Version Fault Tolerance Algorithms - Insights

Goutam Kumar Saha

Dr. Brij Mohan Pareek

SE&T, CSE/CSA, NUJ

Abstract

Insights into Single version fault tolerance (SVFT) algorithms have been explored in this paper. This research study work aims to evaluate the evolution and present state of single-version transient fault tolerance algorithms. Multi-version approaches require diverse software implementations. Single version techniques leverage temporal, data, and resource redundancy within a solitary software instance. This review analyzes Algorithm-based Fault Tolerance (ABFT), checkpointing mechanisms, control-flow checking, checksum, application-aware fault tolerance (FT) and Assertion schemes. It discusses future research directions for heterogeneous and extreme-scale systems.

Keywords: Algorithm-based Fault Tolerance, Single Version Fault Tolerance Algorithms, Assertions, Application-Aware FT, Checkpoints, Control flow Checks, soft errors.

Introduction

Fault tolerance is the ability of a computing system to continue providing an acceptable service in the presence of faults. It has traditionally been investigated at several levels. These include hardware, operating systems, compilers, runtime systems, and application software. Software-based fault tolerance is particularly important because protection mechanisms can be introduced without necessarily redesigning the underlying hardware.

Recent research confirms the continuing importance of software techniques for system resilience. Jia et al. (2023) reviewed software approaches for high-performance computing and identified checkpointing, replication, soft-error resilience, algorithm-based fault tolerance, fault detection, and fault prediction as major research areas. Ko (2022) similarly reviewed software-implemented protection against soft errors. These studies show that software can play an important role in detecting, containing, and recovering from transient faults.

A major distinction exists between **single-version** and **multi-version** software fault tolerance. Multi-version techniques use independently developed versions of software. N-version programming is a well-known example (Avizienis, 1985). Recovery blocks are another

established approach (Randell, 1975). Such methods can address certain design faults through diversity.

Single-version fault tolerance follows a different philosophy. It uses one software version and introduces redundancy, checking, monitoring, or recovery mechanisms within that version. Torres-Pomales (2000) identifies single-version techniques as an important category of software fault tolerance. These techniques include error detection, exception handling, checkpointing, process pairs, and data diversity.

Fault tolerance is an important property of dependable computing systems. A fault may occur in hardware, software, memory, storage, or the execution environment. The resulting error may be detected immediately. It may also remain hidden and produce an incorrect result. Such an error is commonly referred to as a silent data corruption (SDC).

The increasing complexity of computing systems has made software-level resilience increasingly important. Large-scale systems contain many processors and memory components. They also execute long-running applications. As a result, the probability of encountering faults during execution increases. Jia et al. (2023) identified checkpointing, replication, soft-error resilience, algorithm-based fault tolerance (ABFT), and fault detection as major software approaches to resilience.

Software-based protection has several advantages. It can often be implemented without modifying the processor. It can also be adapted to different applications. Ko (2022) reviewed software-implemented soft-error protection and emphasized its flexibility compared with hardware-only approaches.

The literature can broadly be divided into **multi-version** and **single-version** approaches. N-version programming uses multiple independently developed implementations (Avizienis, 1985). Recovery blocks use alternative implementations together with acceptance tests (Randell, 1975). These approaches can address certain software design faults.

Single-version fault tolerance follows a different strategy. It uses one program version and introduces additional mechanisms for detection, recovery, or error masking. These mechanisms can include redundancy, checksums, assertions, control-flow monitoring, checkpointing, and recomputation.

Insights into Single Version Fault Tolerance (SVFT) Algorithms

This research work focuses on **single-version algorithmic fault tolerance** with the objective to improve the fault tolerance of an application without requiring independently developed software versions. The approach is therefore different from N-version programming and other design-diversity techniques.

Single-version software fault tolerance applies redundancy and error-detection mechanisms to a single implementation of a program. The redundancy may be introduced in time, space, data, instructions, control flow, or program state.

The basic idea is simple. The application performs its normal computation. Additional operations are then used to verify the correctness of the computation. When an inconsistency is detected, the application can recover, restart, or enter a fail-safe state.

Torres-Pomales (2000) describes single-version techniques as a distinct class of software fault tolerance. They differ from multi-version techniques because they do not depend on independently developed implementations. This distinction is important for systems with limited memory, processing power, or development resources.

Saha (2003, 2005a, 2005b, 2005c) examined software-based fault tolerance using a single-version application. The approach introduced self-checking code redundancy to tolerate memory errors during application execution. Saha (2006a) later described a single-version scheme that combined runtime signature-based control-flow checking with application-level fault detection.

A further development was presented by Saha (2006b). The study proposed a single-version algorithmic approach based on static redundancy. It used time redundancy to detect and mask transient bit errors affecting processor registers and memory. The approach did not attempt to tolerate software design faults. Instead, it concentrated on operational and transient faults occurring during program execution.

These studies are important for the present research. They show that a single program version can be strengthened by carefully designed redundancy and checking mechanisms. They also establish that fault tolerance does not necessarily require multiple independently developed software versions.

Jia et al. (2023) reviewed software approaches for high-performance computing and identified **checkpointing, replication, soft-error resilience, algorithm-based fault tolerance**, fault detection, and fault prediction as major research areas.

Ko (2022) similarly reviewed **software-implemented protection against soft errors**. These studies show that software can play an important role in detecting, containing, and recovering from transient faults.

Avizienis (1985) introduced multi-version techniques use independently developed versions of software. N-version programming is a well-known example.

Randell (1975) introduced recovery blocks are another established approach. Such methods can address certain design faults through diversity.

Torres-Pomales (2000) identifies single-version techniques as an important category of software fault tolerance. These techniques include **error detection, exception handling, checkpointing, process pairs, and data diversity**. Torres-Pomales (2000) describes single-version techniques as a distinct class of software fault tolerance. They differ from multi-version techniques because they do not depend on independently developed implementations. This distinction is important for systems with limited memory, processing power, or development resources.

Ahmad and Sedaghat (2024), for example, investigated **selective protection based on instruction-level vulnerability assessment**. Their work demonstrates the importance of identifying vulnerable computation rather than protecting every instruction equally. Ahmad and Sedaghat (2024) emphasized selective protection based on instruction vulnerability. Their approach is relevant because protecting every instruction can produce excessive overhead. Vulnerability-based selection provides a way to concentrate protection on computations that are more likely to affect program correctness.

Venkatesha and Parthasarathi (2024) reviewed **redundancy-based fault-tolerance** methods and highlighted the continuing importance of selective and configurable protection. Their survey covers processor and accelerator systems and shows the broad range of redundancy mechanisms used for fault tolerance.

Torres-Pomales (2000) provided a detailed tutorial on software fault tolerance and classified important mechanisms for fault detection and recovery. The work remains useful for understanding the relationship between error detection, error recovery, and fault masking.

Saha (2005a, 2005b) investigated transient software fault tolerance using a **single-version algorithm**. The work demonstrated that a software application could be enhanced with fault-detection mechanisms without introducing multiple independent versions.

Venkatesha and Parthasarathi (2024) reviewed **redundancy-based fault-tolerance mechanisms** for processors and hardware accelerators. Their survey shows that redundancy remains an important technique across heterogeneous computing systems.

Ko (2022) similarly showed that software redundancy can provide **flexible protection** against soft errors. However, the protection level must be balanced against runtime and resource overhead.

Saha (2006a,2006b) investigated static redundancy and repeated computation as mechanisms for software-based transient fault tolerance. Such approaches demonstrate that useful redundancy can be introduced at the algorithm level.

Instruction duplication has been investigated extensively for processor and embedded-system protection.

Ko et al. (2017) examined protection mechanisms for processor caches and highlighted the continuing problem of soft errors in computing systems.

So et al. (2018) proposed EXPERT, an approach based on **redundant multithreading**. The objective was to provide flexible error protection while controlling the associated performance cost.

So et al. (2019) subsequently investigated **software-level redundant multithreading** for soft- and hard-error detection and recovery.

Algorithm-based fault tolerance (ABFT) embeds fault-detection or recovery mechanisms within an algorithm. The concept has been studied extensively in numerical and parallel computing.

Huang and Abraham (1984) established an important foundation for ABFT by showing how algorithmic properties could be used to detect and correct errors in computations. Later research has expanded the approach to different numerical algorithms and modern computing systems. Zhao et al. (2021) demonstrated the continuing relevance of ABFT through FT-CNN, which applies algorithm-based fault tolerance to convolutional neural networks. Their work shows that algorithmic protection can be adapted to modern computational workloads.

Vankeirsbilck et al. (2017) investigated random additive signature monitoring for **control-flow error detection**. Their work demonstrates how signatures can be used to monitor program execution.

Vankeirsbilck et al. (2018) further examined random **additive control-flow error detection**.

Solouki et al. (2022) investigated the implementation of control-flow checking from a model-based software-design perspective.

Yan et al. (2023) proposed a configurable control-flow error-detection method based on basic-block repartitioning.

Assertions are useful because they can express application-specific knowledge. Generic hardware mechanisms may not know that a particular variable must remain within a particular range. The application developer does know this information.

Saha (2005c) emphasized the importance of application-specific checks in single-version transient fault tolerance. Such checks can be based on the semantics of the application and its processing logic.

Assertions also have limitations. A weak assertion may fail to detect an error. An incorrect assertion can generate false alarms. Excessive assertion checking can increase execution time.

The effectiveness of assertions therefore depends on their placement and quality. Strategic placement is preferable to indiscriminate checking.

Checksums are attractive because a relatively small amount of additional information can protect a larger set of data. They are therefore useful in algorithmic fault detection.

According to Huang & Abraham (1984) ABFT frequently uses checksum relationships for this reason.

Recent research by Zhao et al.,(2021) has continued to apply **checksum-based protection** to modern workloads, including machine-learning applications. However, a checksum does not guarantee detection of every possible error. Some error patterns may cancel one another. Different checksum functions also provide different detection capabilities. The checksum should therefore be selected according to the expected fault model. The present research is concerned with computational faults rather than malicious cryptographic attacks. The emphasis is consequently on error-detection capability, computational cost, and suitability for the target algorithm.

Checkpointing provides a mechanism for backward error recovery. A checkpoint stores a valid representation of program state. Checkpointing has been extensively studied in resilient computing.

Bautista-Gomez et al. (2024) reviewed checkpointing strategies and examined issues such as checkpoint intervals, silent errors, task structures, and combinations with replication. Checkpoint frequency is important. Frequent checkpoints reduce the amount of work that may be lost. They also increase storage and execution overhead. Infrequent checkpoints reduce checkpointing

overhead but may increase recovery time and lost computation. For single-version algorithms, checkpoint placement should therefore be related to program structure. Critical computational stages provide natural locations for checkpoints.

Replication is a widely used fault-tolerance technique. Two or more computations are performed and their results are compared. Replication can provide strong detection capability. However, traditional replication may require additional processors, processes, or software instances.

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.

Huang and Abraham (1984) established a major foundation for ABFT through their work on matrix operations. Their approach demonstrated that algorithmic relationships could be used to detect and correct computational errors.

Bosilca et al. (2009) extended ABFT concepts to high-performance computing. Their work demonstrated how algorithm-level checks can be integrated into large-scale numerical applications.

Zhai et al. (2021) proposed FT-BLAS, a fault-tolerant BLAS implementation. The approach combines instruction duplication for memory-bound routines with ABFT for computationally intensive Level-3 routines. Zhao et al. (2021) developed FT-CNN for convolutional neural networks. The study designed checksum-based ABFT mechanisms for CNN inference and reported relatively low runtime overhead.

Wu et al. (2024) incorporated two-sided checksum mechanisms into GPU-based FFT computation. Their design enables error detection and correction while limiting additional memory operations.

Peltekis and Dimitrakopoulos (2024) proposed GCN-ABFT, which combines checks across a complete graph-convolution operation. The approach reduces checksum operations while retaining error-detection capability.

Correct data does not necessarily imply correct program execution. A transient fault may alter a branch condition. It may cause an unintended jump. It may also cause a basic block to be skipped or executed twice. Control-flow checking attempts to detect such errors. A program can be represented by a control-flow graph. Nodes represent basic blocks. Edges represent possible transitions.

A **runtime signature** can be updated as execution proceeds. The signature represents information about the executed path. At selected points, the actual signature can be compared with the expected signature.

Vankeirsbilck et al. (2017) studied random additive signature monitoring for control-flow error detection. The approach demonstrates how lightweight signatures can be used for runtime monitoring.

Vankeirsbilck et al. (2018) further investigated random additive control-flow error detection.

Solouki et al. (2022) examined control-flow checking from a model-based software-design perspective. The work highlights the importance of systematic control-flow monitoring.

Yan et al. (2023) proposed a configurable method based on basic-block repartitioning. The method uses signatures to represent control-flow structure and allows the detection mechanism to be adjusted according to reliability requirements.

Control-flow checking is therefore a natural component of single-version fault tolerance. It complements data-level checks.

Silent data corruption is particularly challenging because the system may continue operating while producing an incorrect result.

Traditional fail-stop errors are easier to manage. The system stops or reports a failure. SDCs are different. The corrupted value may propagate through several computational stages before the error becomes visible.

Jia et al. (2023) identified SDC as an important challenge in large-scale computing systems. Their survey notes that ABFT, replication, and software-level checking can be used to address the problem.

Wang et al. (2024) investigated SDC in processors and examined the implications of silent corruption for reliability and fault tolerance.

Omana et al. (2024) examined SDC risks associated with high-performance processor caches. Their results demonstrate that even error-detection structures can themselves be affected by faults.

These findings reinforce an important principle. A fault-tolerance mechanism must itself be designed carefully. Protecting only the application data may not be sufficient.

Aponte-Moreno et al. (2023) reviewed fault-injection tools for reliability estimation of microprocessor-based embedded systems. Their work highlights the importance of selecting appropriate injection levels and methodologies.

Reitz and Fohry (2024), for example, investigated task-level checkpointing and localized recovery for nested fork-join programs.

Bautista-Gomez et al. (2024) reviewed checkpointing strategies and discussed the limitations of fixed checkpoint intervals. They examined the Young/Daly model and its extensions to task graphs, uncertain failure behavior, silent errors, and combinations with replication

Montezanti et al. (2020) proposed SEDAR, which combines process replication with different checkpointing levels. The system can detect errors and then recover from a valid checkpoint.

Samfass et al. (2021) proposed a task-based approach in which suspicious computational outcomes are recomputed. Their work shows that redundancy can be applied selectively rather than to every task.

Samfass et al. (2021) showed that application-specific error criteria can be used to identify suspicious task outcomes. However, they also found that domain-specific design is important.

Li et al. (2019) designed fault tolerance specifically for introsort. The approach illustrates how protection can be integrated into the structure of an algorithm rather than added as a completely independent layer.

Azimi et al. (2023) investigated the effect of soft errors on real-time embedded operating systems. Their results show that soft errors can affect both execution and system reliability.

De Sio et al. (2023) compared reliability in bare-metal and RTOS-based embedded systems. Their results showed differences in susceptibility to silent-data and control-flow errors between execution environments.

Cherezova et al. (2023) examined fault-tolerance vulnerabilities in advanced SoC FPGAs. Their work emphasizes the complexity of heterogeneous systems and the need for cross-layer reliability mechanisms.

ABFT continues to evolve beyond traditional matrix operations. FT-BLAS applies ABFT to numerical libraries (Zhai et al., 2021). FT-CNN extends it to convolutional neural networks (Zhao et al., 2021). Automatic ABFT has been investigated for stencil computations (2023). TurboFFT extends checksum-based protection to GPU FFT computation (Wu et al., 2024). GCN-ABFT applies similar principles to graph convolution (Peltekis & Dimitrakopoulos, 2024). More recent work has also considered transformer attention. Dai et al. (2025) proposed end-to-end fault-tolerant attention with architecture-aware ABFT and selective protection.

Replication differs from the single-version approach considered in this research. In a multi-version system, independently developed implementations may be executed and compared. Such methods attempt to exploit design diversity (Avizienis, 1985).

Single-version approaches do not depend on independent software development. Instead, they introduce redundancy within the same software design. This distinction is important. Design diversity can increase fault tolerance against some software design faults. However, it also increases development and maintenance costs.

Littlewood et al. (2001) noted that the reliability benefit of design diversity depends strongly on the independence of failures between versions.

The single-version approach is therefore attractive where low implementation cost and limited resource consumption are important.

A central theme in recent fault-tolerance research is **selective protection**. Protecting every operation can be unnecessarily expensive.

Ahmad and Sedaghat (2024) investigated selective tolerance of silent data corruption using instruction-level vulnerability assessment. Their work highlights the importance of identifying computations that have a greater effect on final program correctness.

Selective and Application-Aware Fault Tolerance

A central theme in recent fault-tolerance research is **selective protection**. Protecting every operation can be unnecessarily expensive. Ahmad and Sedaghat (2024) investigated selective tolerance of silent data corruption using instruction-level vulnerability assessment. Their work highlights the importance of identifying computations that have a greater effect on final program correctness.

A similar principle applies to single-version algorithmic fault tolerance. The algorithm can identify critical variables, critical operations, critical control-flow transitions, and important program states. Protection can then be concentrated on these elements. This approach may reduce runtime overhead. It can also reduce memory requirements. More importantly, it allows the fault-tolerance mechanism to exploit application semantics.

Such application awareness is one of the main strengths of algorithmic fault tolerance. The program itself contains information about what constitutes a valid state and a valid result.

Limitations

Despite their advantages, single-version approaches have **several limitations**.

First, they do not automatically tolerate software design faults. If the underlying algorithm contains a systematic design error, repeating the same incorrect computation may reproduce the same error.

Second, redundancy increases overhead. Time redundancy increases execution time. Data redundancy increases memory requirements. Checkpointing increases storage and recovery costs. Third, individual detection mechanisms have coverage limitations. A checksum may miss some error patterns. An assertion may be too weak. Control-flow checking may fail if the checking mechanism itself is corrupted.

Fourth, multiple faults can be difficult to handle. A method designed for a single transient error may not provide adequate protection when several faults occur within a short period.

Fifth, numerical applications introduce additional difficulties. Floating-point rounding and approximation can make exact comparisons inappropriate.

These limitations indicate that no single technique is sufficient for all applications. A combination of complementary mechanisms may provide stronger protection.

Gaps

The research literature demonstrates substantial progress in software fault tolerance. However, several **Gaps** remain.

Much of the recent resilience literature emphasizes high-performance computing, distributed systems, checkpointing, replication, and hardware-assisted protection (Jia et al., 2023; Bautista-Gomez et al., 2024). Less attention is given to a unified **single-version algorithmic framework** that combines multiple lightweight software mechanisms.

There is a need for adaptive protection. Not every instruction, variable, or program state has the same vulnerability. Recent selective-protection research supports this direction (Ahmad & Sedaghat, 2024).

Summary & Future Works

The research literature shows that software fault tolerance has evolved from multi-version techniques toward a broad range of software-based resilience mechanisms. Classical approaches such as recovery blocks and N-version programming established the importance of redundancy (Randell, 1975; Avizienis, 1985). Single-version research subsequently demonstrated that fault tolerance could be incorporated into one application using static redundancy, runtime checking, and data recovery (Saha, 2005a, 2005b, 2006a, 2006b). Recent research has expanded this foundation. ABFT has been applied to BLAS, CNNs, stencil computations, FFTs, and graph neural networks (Zhai et al., 2021; Zhao et al., 2021; Peltekis & Dimitrakopoulos, 2024; Wu et al., 2024). Control-flow checking continues to develop through configurable signature mechanisms (Yan et al., 2023). Checkpointing research is increasingly concerned with adaptive intervals and silent errors (Bautista-Gomez et al., 2024). Selective protection is also becoming important because uniform redundancy can produce excessive overhead (Ahmad & Sedaghat,

2024). The literature therefore suggests that effective single-version fault tolerance should be **selective, application-aware, and multi-layered**.

This conclusion provides the basis for future investigating a single-version algorithm that combines data checking, control-flow monitoring, semantic assertions, redundancy, and recovery mechanisms.

References

Ahmad, H. A., & Sedaghat, Y. (2024). BiGResi: Robust bit-level fault injection framework for assessing intrinsic software resilience against soft errors. *Computers & Electrical Engineering*, 116, 109222.

Aponte-Moreno, A., Isaza-González, J., Serrano-Cases, A., Martínez-Álvarez, A., Cuenca-Asensi, S., & Restrepo-Calle, F. (2023). Evaluation of fault injection tools for reliability estimation of microprocessor-based embedded systems. *Microprocessors and Microsystems*, 96, 104723.

Avizienis, A. (1985). The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, SE-11(12), 1491–1501.

Azimi, S., De Sio, C., Portaluri, A., Rizzieri, D., Vacca, E., Sterpone, L., & Merodio Codinachs, D. (2023). Exploring the impact of soft errors on the reliability of real-time embedded operating systems. *Electronics*, 12(1), 169.

Ahmad, H. A., & Sedaghat, Y. (2024). An automated framework for selectively tolerating SDC errors based on rigorous instruction-level vulnerability assessment. *Future Generation Computer Systems*, 157, 392–407.

Bautista-Gomez, L., Benoit, A., Di, S., Herault, T., Robert, Y., & Sun, H. (2024). A survey on checkpointing strategies: Should we always checkpoint à la Young/Daly? *Future Generation Computer Systems*, 161, 315–328.

Bai, J., Wang, X., Zhao, Z., Zhang, Z., Cai, C., & Chen, G. (2024). A flexible soft error mitigation framework leveraging dynamic partial reconfiguration technology. *Microelectronics Reliability*, 153, Article 115311.

Bezsmertnyi, V., Cioranescu, J.-M., & Eisenbarth, T. (2024). Duplication-based fault tolerance for RISC-V embedded software. In J. Garcia-Alfaro, R. Kozik, M. Choraś, & S. Katsikas (Eds.), *Computer Security – ESORICS 2024: 29th European Symposium on Research in Computer Security, Proceedings, Part IV* (pp. 86–104). Springer.

Bosilca, G., Delmas, R., Dongarra, J., & Langou, J. (2009). Algorithm-based fault tolerance applied to high performance computing. *Journal of Parallel and Distributed Computing*, 69(4), 410–416.

Cherezova, N., Shibin, K., Jenihhin, M., & Jutman, A. (2023). Understanding fault-tolerance vulnerabilities in advanced SoC FPGAs for critical applications. *Microelectronics Reliability*, 146, Article 115010.

Dai, H., Wu, S., Huang, J., Jian, Z., Zhu, Y., Hu, H., & Chen, Z. (2025). *FT-Transformer: Resilient and reliable transformer with end-to-end fault tolerant attention* (arXiv:2504.02211). arXiv [cs.AR].

De Sio, C., Azimi, S., & Sterpone, L. (2023). Evaluating reliability against SEE of embedded systems: A comparison of RTOS and bare-metal approaches. *Microelectronics Reliability*, 150, Article 115124.

Egwutuoha, I. P., Levy, D., Selic, B., & Chen, S. (2013). A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3), 1302–1326.

Huang, K.-H., & Abraham, J. A. (1984). Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6), 518–528.

Isukapalli, S., & Srirama, S. N. (2024). A systematic survey on fault-tolerant solutions for distributed data analytics: Taxonomy, comparison, and future directions. *Computer Science Review*, 53, Article 100660.

Jia, J., Liu, Y., Zhang, G., Gao, Y., & Qian, D. (2023). Software approaches for resilience of high performance computing systems: A survey. *Frontiers of Computer Science*, 17(4), Article 174105.

Jayasekara, S., Karunasekera, S., & Harwood, A. (2022). Optimizing checkpoint-based fault-tolerance in distributed stream processing systems: Theory to practice. *Software: Practice and Experience*, 52(1), 296–315.

Kirti, M., Maurya, A. K., & Yadav, R. S. (2024). Fault-tolerance approaches for distributed and cloud computing environments: A systematic review, taxonomy and future directions. *Concurrency and Computation: Practice and Experience*, 36(13), e8081.

Ko, Y. (2022). Survey of software-implemented soft error protection. *Electronics*, 11(3), Article 456.

Ko, Y.; Jeyapaul, R.; Kim, Y.; Lee, K.; Shrivastava, A. (2017). Protecting Caches from Soft Errors: A Microarchitect's Perspective. *ACM Transactions on Embedded Computing Systems*. 16(4), Article 93.

Li, S., Li, H., Liang, X., Chen, J., Giem, E., Ouyang, K., Zhao, K., Di, S., Cappello, F., & Chen, Z. (2019). FT-iSort: Efficient fault tolerance for introsort. In *SC '19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–17.

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.

Marinissen, E. J., Dixit, H. D., Blanton, S., Kuo, A., Li, W., Mitra, S., Nigh, C., Purdy, R., Kaczer, B., Sanganil, D., Weckx, P., Roussel, P., & Gielen, G. (2024). Silent data corruption: Test or reliability problem? In *2024 IEEE European Test Symposium (ETS)*. 1-7, IEEE.

Omana, M., Manfredi, A., Metra, C., Locatelli, R., Chiavacci, M., & Petrucci, S. (2024). Silent data corruption and reliability risks due to faults affecting high performance microprocessors' caches. In *2024 IEEE 30th International Symposium on On-Line Testing and Robust System Design*.

Peltekis, C., & Dimitrakopoulos, G. (2025). GCN-ABFT: Low-cost online error checking for graph convolutional networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(7), 2836–2840.

Randell, B. (1975). System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), 220–232.

Reitz, M., & Fohry, C. (2024). Task-level checkpointing and localized recovery to tolerate permanent node failures for nested fork–join programs in clusters. *SN Computer Science*, 5, Article 320.

Saha, G. K. (2003). Transient software fault tolerance through recovery. *Ubiquity*, 2003(September), Article 2, ACM.

Saha, G. K. (2005a). Transient software fault tolerance using single-version algorithm. *Ubiquity*, 2005(August), ACM.

Saha, G. K. (2005b). Software implemented fault tolerance through data error recovery. *Ubiquity*, 2005(September), ACM.

Saha, G. K. (2005c). Software-based fault tolerant computing. *Ubiquity*, 2005(November), ACM.

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).

- Samfass, P., Weinzierl, T., Reinartz, A., & Bader, M. (2021). Doubt and redundancy kill soft errors—Towards detection and correction of silent data corruption in task-based numerical software. In *Proceedings of the 2021 IEEE/ACM 11th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)* (pp. 1–10). IEEE.
- So, H., Didehban, M., Ko, Y., Shrivastava, A., & Lee, K. (2018, March 19–23). EXPERT: Effective and flexible error protection by redundant multithreading. In *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (pp. 533–538). IEEE.
- Amel Solouki, M., Sini, J., & Violante, M. (2022). Implementation of control flow checking—A new perspective adopting model-based software design. *Electronics*, 11(19), Article 3074.
- Strati, F., Friedman, M., & Klimovic, A. (2025). PCcheck: Persistent concurrent checkpointing for ML. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)* (pp. 811–827). Association for Computing Machinery.
- Torres-Pomales, W. (2000). *Software fault tolerance: A tutorial* (NASA/TM-2000-210616). NASA Langley Research Center.
- Vankeirsbilck, J., Penneman, N., Hallez, H., & Boydens, J. (2017). Random additive signature monitoring for control flow error detection. *IEEE Transactions on Reliability*, 66, 1178–1192.
- Vankeirsbilck, J., Penneman, N., Hallez, H., & Boydens, J. (2018). Random additive control flow error detection. In *Proceedings of the International Conference on Computer Safety, Reliability, and Security* (pp. 220–234).
- Venkatesha, S., & Parthasarathi, R. (2024). Survey on redundancy based-fault tolerance methods for processors and hardware accelerators—Trends in quantum computing, heterogeneous systems and reliability. *ACM Computing Surveys*, 56(11), Article 275, 1–76.
- Wang, S., Zhang, G., Wei, J., Wang, Y., Wu, J., & Luo, Q. (2024). Understanding silent data corruption in processors for mitigating its effects. *ACM Transactions on Architecture and Code Optimization*, 21(4), Article 84.
- Wu, S., Zhai, Y., Liu, J., Huang, J., Jian, Z., Dai, H., Di, S., Cappello, F., & Chen, Z. (2025). TurboFFT: Co-designed high-performance and fault-tolerant fast Fourier transform on GPUs. *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, 70–84.
- Yan, Z., Zhuang, Y., Zhang, Q., Gu, J., & Guo, L. (2023). A configurable control flow error detection method based on basic block repartition. *Microelectronics Reliability*, 147, Article 115070.

Yang, W., Li, Y., Zhang, W., Guo, Y., Zhao, H., Wei, J., Li, Y., He, C., Chen, K., & Guo, G. (2020). Electron inducing soft errors in 28 nm system-on-chip. *Radiation Effects and Defects in Solids*, 175(7–8), 745–754.

Zhai, Y., Gien, E., Fan, Q., Zhao, K., Liu, J., & Chen, Z. (2021). FT-BLAS: A high performance BLAS implementation with online fault tolerance. In *Proceedings of the 35th ACM International Conference on Supercomputing* (pp. 127–138). Association for Computing Machinery.

Zhang, Y., Itsuji, H., Uezono, T., Toba, T., & Hashimoto, M. (2023). Vulnerability estimation of DNN model parameters with few fault injections. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E106.A(3), 523–531.

Zhao, K., Di, S., Li, S., Liang, X., Zhai, Y., Chen, J., Ouyang, K., Cappello, F., & Chen, Z. (2021). FT-CNN: Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, 32(7), 1735–1748.