Analysis of Different Software Fault Tolerance Techniques

Golam Moktader Nayeem, Lecturer, Department of ECE, Southern University Bangladesh
&
Mohammad Jahangir Alam, Lecturer, Department of CSIT, Southern University Bangladesh

Abstract
Without doubt, fault tolerance is one of the major issues in computing system design because of our present inability to produce error-free computing system for its inherited complexity. Fault tolerance is and will continue to be an important consideration in designing systems. Fault tolerance system required for developing highly reliable computer systems that can function under adverse conditions, which also provide indispensable in safety critical applications. In this paper, we discuss recent development of software fault tolerance techniques and find out some limitations of these techniques.

1. Introduction

Fault tolerance is one of the major research areas in computer system design nowadays. Because in modern society, computer are used in every aspect and it required operating correctly without any interruption for a long period of time. Fault tolerance makes a system capable to operate correctly in a faulty condition and protect against accidental or malicious destruction of information, protect against generating erroneous output and also guaranteed that confidential information cannot be divulged. On the other hand software is the key part of many critical applications flight control systems, medical systems, etc) as well as in real time system. So, the researchers have tried to develop fault tolerant software system.

Despite its widespread use, it is extremely difficult to develop flawless software. In practice, at the end of software testing phase, project managers always want assessment of the software reliability (or quality) and wish to know when to reach to desired target. According to the ANSI definition, Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment.

2. Concepts and Terminologies

Some important terminologies related to fault tolerance are as follows:

- **Fault:** Fault can be defined as incorrect state of hardware or software resulting from physical defects, design flaw or operator error.
- **Fault Models:** Depending on the system behavior after a fault has occurred, faults have been characterized into different groups or classes.
- **Error:** An error is a part of a system state that may lead to a failure or we can say error is the manifestation of a fault.
- **Failure:** When a system or module is designed, its behavior is specified. When in service, we can observe its behavior. When observed behavior differs from the specified behavior we call it failure. Also we can say failure is the system level effect of an error.
- **Crash failure.** A process undergoes crash failure, when it permanently ceases to execute its actions. This is an irreversible change, excluded are *napping* failures, where a process may play dead for a finite period of time, and then resume operation. In *fail-stop* models, its neighbors can detect the faulty process, which crashes.
- **Omission failure.** Consider a transmitter process sending a sequence of messages to a receiver process. If the receiver does not receive some of the messages sent by the transmitter then an omission failure occurs.
- **Transient failure.** The agent inducing this failure may be temporarily active, but it can make a lasting effect on the global state. This failure can affect the global state in an arbitrary way.
- **Byzantine failure.** When a process behaves arbitrarily, we say that Byzantine failure has occurred. It represents the weakest of all the failure models that allows every conceivable form of erroneous behavior.
- **Software failure.** Many reasons lead to software failures- (1) In some cases, the execution of program suffers from the degeneration of the run-time system due to ‘memory leaks’, leading to a system crash. (2) There may be problem with the inadequacy of the specification, as in the Y2K problem. Many of the failures like crash, omission, transient, or Byzantine can be caused by software bugs.
- **Temporal failure.** Real time systems require actions to be completed within a specific amount of time. When the deadline is not met, a temporal failure occurs.

3. Software Fault Tolerance Techniques

In this section, we present fault tolerance techniques applicable to software based on the paper [1]. Basically, software fault tolerance is divided into two groups: single version and multi-version software technique. Single version techniques are concern with the single software by adding several types of mechanism during the design targeting the detection, containment, and handling of errors. Multi-version fault tolerance techniques use multiple version of same software in a structured way to ensure that design faults in one version do not cause system failures.

3.1 Single Version Software Fault Tolerance Techniques

Single-version fault tolerance is based on the use of redundancy applied to a single version of a piece of software to

detect and recover from faults. Among others, single-version software fault tolerance techniques include considerations on program structure and actions, error detection, exception handling, checkpoint and restart, process pairs, and data diversity.

3.1.1 Software Structure and Actions

The software architecture provides the basis for implementation of fault tolerance. Different types of software structure and actions are available. Among them the most used techniques are: Modularizing, Partitioning, System closure, and Temporal Structuring. The use of modularizing techniques to decompose a problem into manageable components is as important to the efficient application of fault tolerance as it is to the design of a system. Partitioning is a technique for providing isolation between functionally independent modules. System closure is a fault tolerance principle stating that no action is permissible unless explicitly authorized. Temporal structuring of the activity between interacting structural modules is also important for fault tolerance.

3.1.2 Checkpoint and Restart

For single-version software there are few recovery mechanisms. The most often mentioned is the checkpoint and restart mechanism. A restart, or backward error recovery (*Figure 1*) has the advantages of being independent of the damage caused by a fault, applicable to unanticipated faults, general enough that it can be used at multiple levels in a system, and conceptually simple. There exist two kinds of restart recovery: static and dynamic. A static restart is based on returning the module to a predetermined state. This can be a direct return to the initial reset state, or to one of a set of possible states, with the selection being made based on the operational situation at the moment the error detection occurred. Dynamic restart uses dynamically created checkpoints that are snapshots of the state at various points during the execution. Checkpoints can be created at fixed intervals or at particular points during the computation determined by some optimizing rule.

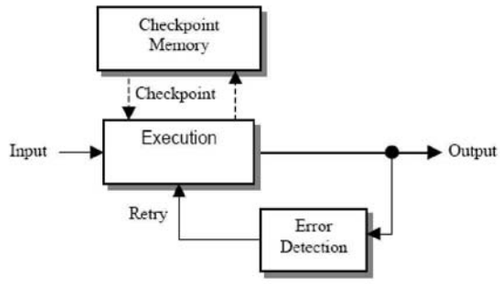

Figure 1: Logical Representation of Checkpoint and Restart

3.1.3 Process Pairs

A process pair uses two identical versions of the software that run on separate processors (*Figure 2*). The recovery mechanism is checkpoint and restart. Here the processors are labeled as primary and secondary. At first the primary processor is actively processing the input and creating the output while generating checkpoint information that is sent to the backup or secondary processor.

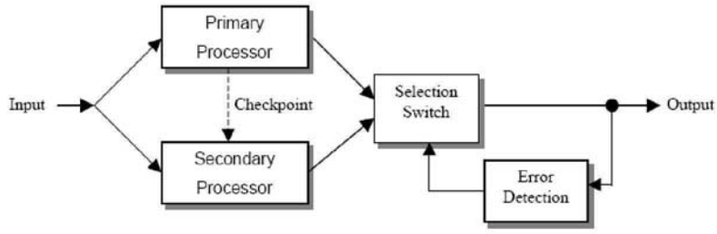

Figure 2: Logical Representation of Process Pairs

3.2 Multi-Version Software Fault Tolerance Techniques

Multi-version fault tolerance is based on the use of two or more versions (or "variants") of a piece of software, executed either in sequence or in parallel. The versions are used as alternatives (with a separate means of error detection), in pairs (to implement detection by replication checks) or in larger groups (to enable masking through voting).

3.21 Recovery Blocks

The Recovery Blocks technique combines the basics of the checkpoint and restart approach with multiple versions of a software component such that a different version is tried after an error is detected Figure 3). Checkpoints are created before a version executes. Checkpoints are needed to recover the state after a version fails to provide a valid

operational starting point for the next version if an error is detected.

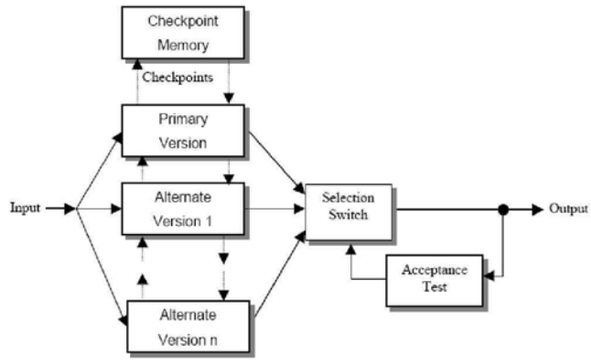

Figure 3: Recovery Block Model

3.2.2 N-Version Programming

N-Version programming [6] is a multi-version technique in which all the versions are designed to satisfy the same basic requirements and the decision of output correctness is based on the comparison of all the outputs (Figure 4). The use of a generic decision algorithm (usually a voter) to select the correct output is the fundamental difference of this approach from the Recovery Blocks approach, which requires an application dependent acceptance test.

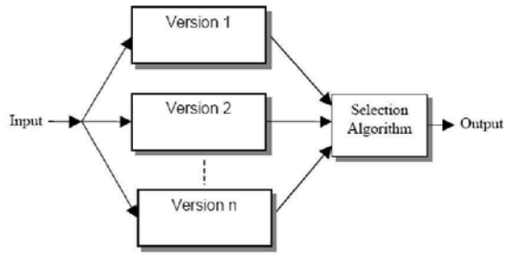

Figure 4: N-Version Programming Model

3.2.3 N Self-Checking Programming

N Self-Checking programming is the use of multiple software versions combined with structural variations of the Recovery Blocks and N-Version Programming. N Self-Checking programming using acceptance tests is shown on Figure 5. Here the versions and the acceptance tests are developed independently from common requirements. This use of separate acceptance tests for each version is the main difference of this N Self-Checking model from the Recovery Blocks approach.

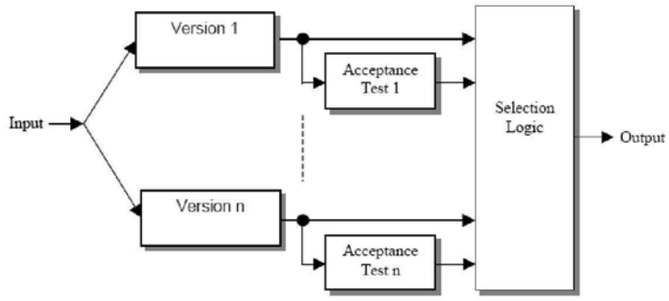

Figure 5: N Self-Checking Programming using Acceptance Tests

N self-checking programming using comparison for error detection is shown in Figure 6. Similar to N-Version Programming, this model has the advantage of using an application independent decision algorithm to select a correct output. This variation of self-checking programming has the theoretical vulnerability of encountering situations where multiple pairs pass their comparisons each with different outputs.

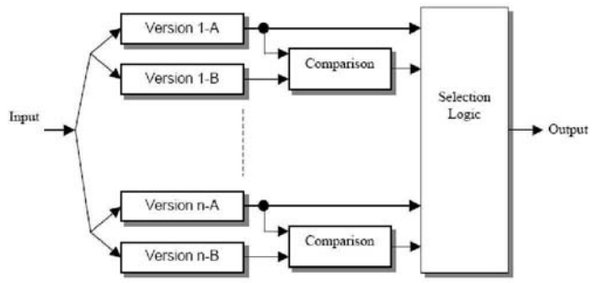

Figure 6: N Self-Checking Programming using Comparison

3.2.4 Consensus Recovery Blocks

The Consensus Recovery Blocks (Figure 7) approach combines N-Version Programming and Recovery Blocks to improve the reliability over that achievable by using just one of the approaches. The acceptance tests in the Recovery Blocks suffer from lack of guidelines for their development and a general proneness to design faults due to the inherent difficulty in creating effective tests. The use of voters as in N-Version Programming may not be appropriate in all situations, especially when multiple correct outputs are possible. In that case a voter, for example, would declare a failure in selecting an appropriate output. Consensus Recovery Blocks uses a decision algorithm similar to N-Version Programming as a first layer of decision. If this first layer declares a failure, a second layer using acceptance tests similar to those used in the Recovery Blocks approach is invoked.

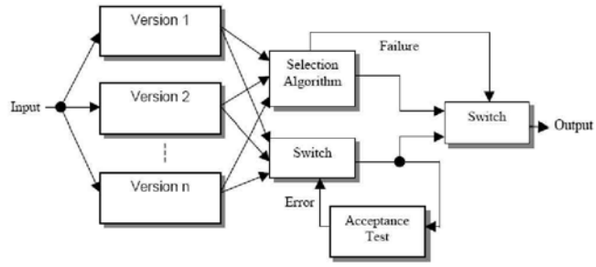

Figure 7: Consensus Recovery Blocks

4. Software Faults Detection

George A. Reis [3] proposed some techniques to detect and recover from transient faults. Transient faults (also known as soft errors), unlike manufacturing or design faults, do not occur consistently. To counter these faults, designers typically introduce redundant hardware such as RAID architecture, N-modular system, and error correcting code (ECC) to detect and recover from faults. But using these hardware fault tolerant mechanism is too expensive for many processor markets. On the other hand software only approaches to redundancy are attractive because they essentially come free of cost.

4.1 Error Detection by Duplication Instruction (EDDI)

EDDI [2] is a software-only fault detection system that operates by duplicating program instructions and using this redundant execution to achieve fault tolerance. The program instructions are duplicated by the complier and are intertwined with the original program instructions. Each copy of program however uses different registers and different memory locations so as to not interfere with one another. At certain synchronization points in the combined program code, the compiler to make sure that the original instructions inserts check instructions and their redundant copies agree on the computed values. Since program correctness
is defined by the output of a program, if we assume memory-mapped I/O, then a program have executed correctly. Consequently, it is natural to use store instructions as synchronization points for comparison. Unfortunately, it is insufficient to use the store instruction s as only synchronization points since misdirected branches can cause store to be skipped, incorrect stores to be executed, or incorrect values to ultimately feed a store. Therefore, branch instructions must also be synchronization points at which redundant values are compared.

4.2 Software Implemented Fault Tolerance (SWIFT)

SWIFT [3] is an efficient software-only, transient-fault detection technique. SWIFT efficient manages redundancy by reclaiming unused instruction-level resources present during the execution of most programs. SWIFT makes several key refinements to EDDI and incorporates software only signature based control flow-checking scheme to achieve exceptional fault coverage. The major difference between EDDI and SWIFT is, while EDDI's SoR includes the memory subsystems, SWIFT moves memory out of the S0R, since memory structures are already well-protected by hardware schemes like parity and ECC, with or without scrubbing. SWIFT's performance greatly benefits from having only half the memory subsystems.

5. Analysis and finding limitations of the existing software fault tolerance techniques

The methods discussed in the preceding sections are mostly used in many critical and highly available systems. Fault tolerant techniques provide high reliability and availability. Most of the software fault tolerance techniques have some advantages and also have some disadvantages. For example in single version fault tolerant technique the reliability is achieved by sacrificing the process time [4]. On the other hand, in multi-version fault tolerant technique, the availability and reliability is achieved by using redundant component leads to extra cost. Moreover, developing the multi-version software is more complex than the normal software [5]. We noted how
software faults tend to be stated dependent and activated by particular input sequences. Although component reliability is an important quality measure for system level analysis, software reliability is hard to estimate and the use of post-verification reliability estimates remains a controversial issue. For some applications software safety is more important than reliability, and fault tolerance techniques used in those applications are aimed at preventing catastrophes. Single version software fault tolerance techniques discussed include system structuring and closure, atomic actions, inline fault detection, exception handling, and checkpoint and restart. Process pairs exploit the state dependence characteristic of most software faults to allow uninterrupted delivery of services despite the activation of faults. Similarly, data diversity aims at preventing the activation of design faults by trying multiple alternate input sequences. Multi-version techniques are based on the assumption that software built differently should fail differently and thus, if one of the redundant versions fails, at least one of the others should provide an acceptable output. Recovery blocks, N-version programming [7, 8], N self-checking programming, consensus recovery blocks, and *n* / (*n*-1)-variant techniques were presented. Special consideration was given to multi-version software development and output selection algorithms. Operating systems must be given special treatment when designing a fault tolerant software system because of the cost and complexity associated with their development, as well as their criticality for correct system functionality.

6. Conclusion

In this paper, we first reviewed the existing software fault tolerance techniques. We also investigated the different aspects of these techniques and tried to identify the different characteristics of several software fault tolerance techniques. Finally, we find and analyze the limitations of these techniques.

References

1. Wilfredo Torres-Pomales, Software Fault Tolerance: A Tutorial*, Langley Research Center, Hampton, Virginia,* October 2000.
2. N. Oh. P.P. Shirvani, and E.J. McCluskey, Error detection by duplicated instructions in super-scalar processors, *IEEE Transactions on Reliability*, 51(1):63-75, March 2002.
3. George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan and David I. August, *Proceedings of the International Symposium on Code Generation and Optimization (CGO'05), IEEE.*
4. Chin-Yu Huang, Chu-Ti Lin, and Chuan-Ching Sue, *Software Reliability Prediction and Analysis durng Operationa use*, 0-7803-8932, IEEE, 2005.
5. Marek Reformat, Efe Igbide*, Isolation of Software Defects: Extracting Knowledge with Confidence*, 0-7803-9093, IEEE, 2005.
6. Sergiy A. Vilkomir, David L. Parnas, Veena B. Mendiratta,and Eamonn Murphy, *Availability evaluation of hardware/software systems with several recovery procedures*, Proceedings of the 29th Annual International Computer Software and Applications Conference, IEEE, 2005.
7. Xia Cai and Michael R. Lyu, and Mladen A. Vouk, *An Experimental Evaluation on Reliability Features of N-Version Programming*, Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering, IEEE, 2005.
8. PETER J. DENNING, *Fault Tolerant Operating Systems, Computing Surveys*, Vol. 8, No. 4, December 1976.
9. Algirdas Avizienis, *Toward Systematic Design of Fault- Tolerant Systems*, 0018-9162, IEEE, 1997.