International Journal of Innovative Research in Advanced Engineering (IJIRAE)
Volume 1 Issue 2 (April 2014)

Review Paper on Recovery of Data during Software Fault

Sunita [1], Shekher Dahiya [2]
1. Asst Professor, SBMEC, Rohtak
2. PG Scholar, SBMEC, Rohtak

Abstract-- In this paper here we will discuss different types of technique that are used for recovery of data during software fault. The major objective of this paper to specify recovery technique that are used during software and hardware fault.

Keyword: Fault detection and recovery, hardware redundancy, fault recovery, software fault

I. INTRODUCTION

As computers become an integral part of today's society, making them dependable becomes increasingly important. Field studies and everyday experience make it clear that the dominant cause of failures today is software faults, both in the application and system layers. Reducing the number of failures caused by software faults is therefore an important challenge for the fault-tolerance community.

The best way to ensure fault-tolerance [1] is to avoid faults in the first place. But in reality faults occur due to a variety of reasons. So in the presence of faults, the two major components of fault-tolerance are the ability to detect that a fault or an error as a manifestation of the fault has occurred and the ability to eliminate the effects of the fault and continue the correct operation of the system.

Figure 1, shows the time-line for a generic fault-tolerance mechanism that detects a fault and tries to recover from the resulting error or failure. Most of these mechanisms save the state of the process regularly to maintain failure transparency. Failure transparency aims to mask the effects of the failure from the user. This means that the recovery mechanism has to deal with the issue of not displaying inconsistent visible events to the outside world. The figure shows one such point in time where the state was saved. The figure then shows the occurrence of a fault and an eventual detection of the fault. A failure of the process can be thought of as a very crude fault detection mechanism. The recovery mechanism then tries to restore the process to the state that was last saved and retries the computation from there on. There are two assumptions made implicitly by this fault-tolerance mechanism [2]. The first one is that the saved state does not contain the state of the fault itself or any state corrupted by the fault.

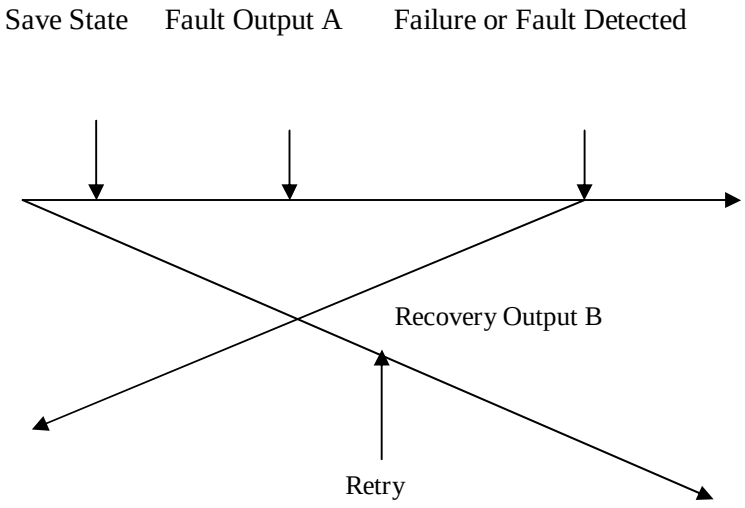

Figure 1 A generic time-line for fault detection and recovery

The second one is that after recovery the same fault does not happen again and cause the same failure. Only when these two assumptions hold good that the recovery process is successful. The time line also shows an instance where the recovery mechanism fails to uphold failure transparency. The event "output a" happens during the initial run. But during recovery the event "output b" happens. This exposes the user to the fact that there was a failure of the application and that a recovery mechanism recovered the application. Also there is a sequence of events present and that do not appear consistent to the outside world. Hence the failure was not transparent to the user.

II. LITERATURE REVIEW

Statistical methods, machine learning method, and mixed techniques are widely used in literature to predict software faults. Faults contain in software systems and it continue to work, is a major problem in future. A software bug is an error, flaw, mistake, failure, or fault in a computer program that prevents it from behaving as intended (e.g., producing an incorrect result). A software fault is a deficiency that causes software failure in an executable product. In software engineering, the non-conformance of software to its requirements is commonly called a bug. Most bugs arise from mistakes and errors made by people in either a program's source code or in its design, and a few are caused by compilers producing incorrect object code. Meaningful the causes of possible defects as well as identifying general software process areas that may need attention from the initialization of a project could save capital, time and effort. The possibility of early estimating the potential faultiness of software could help on planning, controlling and executing software development activities. Software is said to be faulty if we feed some input and it produce incorrect output. For each execution of the software program where the output is incorrect, a failure is observed. Software engineers distinguish software faults from software failures. In case of a failure, the software does not do what the user expects but on the other hand fault is a hidden programming error that may or may not actually evident as a failure. A fault can also be described as an error in the correctness of the semantic of a computer program. A fault will become a failure if the exact computation conditions are met, one of them being that the faulty portion of computer software executes on the CPU. A fault can also turn into a failure when the software is ported to a different hardware platform or a different compiler, or when the software gets extended. Software faults are all due to human errors in creating the software. The following depicts the types of failure with its description. The increasing demand for higher preparation competency and security in engineering processes has resulted in huge interest in fault-detection techniques. Engineering researchers and practitioners remain concerned with accurate prediction on qualitative and quantitative factors. So it is in this environment Software quality prediction models seek to predict quality factors such as whether a component is fault prone or not, despite earlier attempts to estimate the fault prediction, the less stress has been paid on the reducing of the software defect and predicting it before occurred. Thus the study is a significant attempt that will be helpful for the prediction of faults.

III. FAULT DETECTION

Fault detection is the process of recognizing that a fault has occurred in the system [3]. A lot of times the fault detection process does not detect the fault itself but detects the erroneous state caused by the fault. This is required before any fault recovery schemes can be implemented. Fault detection schemes also try to determine the location of the fault in the system to help in choosing the correct recovery method. An important property of fault detection is to detect the fault as soon as possible to isolate its effects and aid easy recovery. The property that a malfunctioning component halts before it saves the state of the error to permanent storage or transfers faulty state to other components is known as the halt-on-failure property and forms the core of the fail-stop model.

All fault detection methods depend upon some form of redundancy to succeed. Redundancy allows us to check a result and reduce the probability of using an incorrect result. Without redundancy and checking of results, we have no way of deciding if a result is correct or not. Redundancy can be both physical and temporal. Physical redundancy is having multiple copies of the hardware, software or information exists simultaneously and comparing results with each other and therefore checking each other. Temporal redundancy is having the same computation being run at different points in time and comparing the results. Redundancy can also be either complete or partial. In partial redundancy only a part of the computation is redone and heuristics are used to give an answer for correctness. A few commonly used fault detection techniques and the ways in which they used redundancy are described below. The following three techniques are based upon partial redundancy. Error detecting codes are implemented both in software and hardware. Error detecting codes: They are based on the principle of information redundancy and are implemented both in software and hardware. They are formed by the addition of redundant information to data units or by mapping data unit words into new representations with redundant information. They are also designed to detect faults not only in the logic being monitored but also faults in the checking logic itself. Software consistency checks: Consistency checks use knowledge about the characteristics of data to verify its correctness. An example is that the amount of cash withdrawn at an ATM should never exceed a certain amount. These types of checks are usually per- formed using assertion statements in software written in high level languages Self-checking data structures are another example of consistency checks where operations on the data structure are automatically checked to ensure that they do not leave the data structure in an unstable state. This is especially important in dynamically allocated data structures [4,5].

The following two techniques are based upon complete redundancy.

Hardware redundancy: There are three forms of hardware redundancy - passive, active and hybrid. Passive hardware redundancy uses voting mechanisms to mask the occurrence of faults. This method does not truly detect the fault and requires no action from the system or an operator. The most common form of passive hardware redundancy is triple modular redundancy. The basic concept of TMR is to have three copies of the hardware and perform a

majority vote to determine the output of the system. The other two fault-free modules can mask the failure of one copy of the hardware. Active hard- ware redundancy techniques try to locate and remove a faulty component after an error has been detected and enable a spare component to continue operation normally or at a degraded level. One form of active redundancy is the standby replacement technique. In standby sparing, one module is operational while one or more modules are used as spares or standbys. When a fault occurs in the operational module, one of the spares is made the operational module. Hybrid redundancy tries to combine the best features of the passive and active techniques. Fault masking is used to prevent erroneous results and fault detection and recovery are used to reconfigure the system.

N-version programming: N-version programming was designed to tolerate design and coding flaws in software. The basic concept is to design and code software module N times and to vote the N results produced by the modules. It is hoped that by performing the N designs independently, the same mistakes will not be made in all the modules The voting module will be able to detect a fault because the same fault is not expected to occur in all the modules. In theory, this technique should be able to detect and recovery from most software faults. However it is very expensive to produce N independent designs and implementations of a software module. Moreover it is not clear if the n-modules will not have correlated faults.

Fault Recovery

Fault recovery is the process of getting the system back to an operational state after a fault has been detected or after a failure has occurred. The main goal to be achieved during recovery is failure transparency. There are two actions that a fault recovery mechanism has to perform to achieve failure transparency. The first one is that no visible events are output after recovery that is inconsistent with the events output before the failure. In other words the system should try to recover to a state as recent in time as possible to minimize the number of visible events it has to redo. The second one is correctness where the goal of the system is to recover to a state, which is free of any errors caused by the fault or failure, and also to recover to a state that will not lead to the same fault or failure to occur again.

All fault recovery mechanisms depend upon the state of the computation being saved in some form. Some recovery mechanisms actively save state to use it later for recovery and other mechanisms rely upon redundant state present in the system to recover from in the case of a failure. Systems can also use temporal redundancy to recomputed state in the case of a failure. The state has to be saved on some form of storage that will survive the fault or a failure caused by the fault. Some common examples of such storage are hard disk drives and memory of backup or redundant machines that are not affected by the failure. The following are examples of commonly used fault recovery mechanisms that explicitly save state during normal operation for use in the case of a failure. Checkpoint and recover schemes: This scheme is based on temporal redundancy. The state of the system, checkpoint, is saved periodically during the failure-free operation of the system. After an error or a failure has occurred, the system rebuilds its state from a checkpoint and tries to continue execution. This scheme works on the optimism that the fault has been fixed or that the fault activation is transient or intermittent.

Recovery blocks: Recovery blocks, used in software fault-tolerance, are a special case of checkpoint and recover schemes. N versions of a software block are provided and a set of acceptance tests is used. One version of the block is designated as the primary and the rest are used as spares or standbys'. The primary version is used until it fails the acceptance tests. Then the system rebuilds its state to a checkpoint taken before the beginning of the module. One of the spares is designated as the primary and the computation is redone using the new module.

Process pairs: Process pairs are a special case of recovery blocks. Process pairs comprise of a pair of processes, one designated as the primary and the other as standby. The primary periodically transfers its state to the backup. When the primary fails the backup continues execution from the last state it received from the primary. Process pairs are different from recovery blocks in the sense that the same code is executed but by a different process. This method works well with Heisenberg's. A slight variation of this technique is used in the TARGON/32 system, which executes the backup process on any one of the other processors in the system. The following are examples of recovery mechanisms, which use redundancy built into them to reconstruct the state of the computation after a failure.

Error correcting codes: They are similar to error correcting codes with enough information redundancy in them to both detect and correct the error. Some examples of error correcting codes are overlapping parity codes and Hamming error correcting codes. Hardware redundancy: As described earlier in detection techniques, hardware redundancy is used to recover from component failures. The failed component is bypassed or masked and the system continues execution.

Software Faults

Faults may be classified into two categories: operational and design. Operational faults are caused by conditions such as wear-out and changes in the environment can be handled with simple replication. Faults caused by design bugs are much more difficult to handle, because simply replicating a buggy design often results in dependent failures in which all the replicas fail. Software faults are all design bugs. They are caused due to programming errors, algorithmic errors, specification errors etc. So these faults are present in the software throughout its lifetime. In saying that a software fault

occurred, we mean that the piece of code, which is faulty, was executed. We can also describe this in other words as a software fault have been activated. Software faults, which are activated every time we run the software, are usually detected during the testing phase and are corrected before the piece of software is released. Software faults, which are rarely activated escape the test and debug process and end up in the production release of the piece of software. If the faults are not being activated every time we run the software, it is because that piece of code is executed when a certain external event occurs. This event can be termed as the "fault trigger". The fault trigger or the external event may happen very infrequently or can also be non-deterministic in its occurrence. An example of a rare event is the occurrence of a leap day (Feb. 29) in a year. An example of a non-deterministic event is a particular scheduling order, which leads to a deadlock. During a different execution of the program the scheduling order may change and not cause the deadlock again. Faults caused by such rare or non-deterministic triggers escape the testing phase because we cannot simulate all the possible inputs to any substantial piece of software on a reasonable time. A subset of all the possible input space is chosen to test software and the subset is chosen is such a way so as to maximize the amount of code tested. In the next few sections of this chapter we will discuss the two properties of soft- ware faults that affect recovery and see how various fault recovery methods make assumptions about these properties.

Fail-Stop Property

A faulty program may perform arbitrary tasks that make recovery complicated or impossible. For example, a malfunctioning program may overwrite critical state or send incorrect information to other processors. To ease the task of building fault-tolerant systems, most designers try to ensure that a malfunctioning program halts before it writes erroneous data to stable storage or sends incorrect information to other processes. This property is known as halt-on-failure and forms the core of the fail-stop model. Many fault-tolerant systems assume that faulty application programs follow this model. There are a number of ways to achieve fail-stop failures in the presence of software faults. The key principle behind these methods is to stop executing the program before it commits any faulty state. By “committing faulty state”, we mean that the state is saved in such a manner that it survives the application crash and is part of the application state after recovery. There are two major factors which decide whether this principle is being upheld or not during a failure. The first one is the quality of error detection present in the program and the second one is has to do with the recovery method being used. Different recovery methods have different requirements about how frequently state needs to be saved and what part of the state of the process needs to be saved. Figure 2, on shows examples of failures that uphold the fail-stop property and examples of failures that potentially violate the fail-stop model. If the last state commit before the failure happened at the point in time marked "State Commit 1" then the failure upholds the fail-stop model. This is because state was saved before the fault trigger occurred and so the committed state cannot contain any state corrupted by the fault or the state associated with the fault itself.

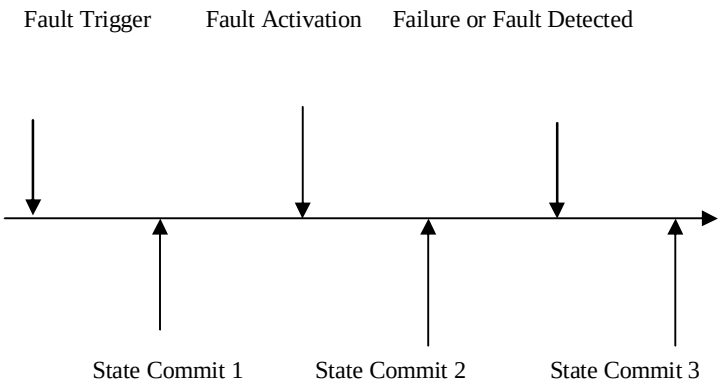

Figure 2 A time-line showing events related to the fail-stop property

On the other hand, if the last state commit before the failure, happened at the point in time marked "State Commit 2" then whether the failure violated the fail-stop property depends upon what was saved in the state commit. If the state commit saved state related to the trigger or recorded the occurrence of the trigger then the failure violates the fail-stop model. This is because when the recovery mechanism recovers the process, the state of the trigger will have survived the failure and will lead to the same fault being activated after recovery.

If the committed state does not contain any information related to the trigger then the failure will have upheld the fail-stop property. Similarly, if the last state commit happened at the point in time marked "State Commit 3" then whether the failure violated the fail-stop property depends upon what was saved in the state commit. In addition to the state of the trigger, if the state commit saved any state changed or corrupted as a result of the fault, the failure will have violated the fail-stop model. If the state commit saved neither any state related to the trigger nor any state changed by the fault the failure upholds the fail-stop model.

Non-Determinism

A lot of recovery mechanisms rely on time redundancy to be able to successfully recover from a fault. For example, process pairs are similar to recovery blocks, but instead of retrying the operation on a different implementation of the code block, process pairs retry the operation on the same code (possibly on a different computer). Process pairs, and rollback-recovery protocols in general, survive a specific class of software faults known as "Heisenberg's". Heisenberg's are transient, non-deterministic faults that disappear when the operation is retried, even if the same code is used. The transient nature of the fault arises because some factor external to the program has changed; for example, a different interleaving order of threads may occur during retry and so avoid a race condition. It has been hypothesized that most faults that occur in released applications are transient. The intuition for this hypothesis is that transient faults such as race conditions are more difficult to reproduce and hence debug than non-transient faults (so- called Bourbons), so transient faults are more likely to remain in released software. This is an important hypothesis for guiding research in how to recover from software faults.

Application-Specific and Generic Recovery Methods

We categorize techniques for recovering from software faults as either application- specific recovery or application-generic recovery. In application-specific recovery, a non- fault-tolerant design is made fault-tolerant by adding code that is specific to the application. This includes techniques where the programmer makes calls to fault-tolerance libraries or reconstructs part of the program state during recovery. An extreme example of application-specific recovery is N-version programming, which uses N independent implementations of the same program. Recovery blocks also use multiple implementations of a block of code, but use a passive-replication style with error checking and rollback to avoid running multiple versions at the same time. Application-specific recovery can be very effective, but is often prohibitively expensive to implement. Because of the cost of implementing application-specific recovery for each application, researchers have suggested various application-generic ways to survive software faults. These techniques do not add redundant, specific code for each application (we do not consider error checking code as redundant, though technically most error checks are application-specific and redundant). As a result, application-generic techniques need no information about the application, nor do they require any assistance from the application programmer in the form of extra code. For example, process pairs periodically transfer the state of the primary process to a backup process in a generic manner. They use operating system services to make a copy of the complete state of the application without any help from the application itself. When the primary process suffers a failure, the backup process takes over and executes the application. Again operating system services are used to periodically update the state of the backup process in an application independent manner and also to control the execution of the backup process. Note that a truly generic recovery mechanism must preserve all application state (e.g. by check pointing or logging), because there is no application-specific code to reconstruct missing state. Hence only a change external to the application can allow the application to succeed on retry. Rollback- recovery protocols in general, save state periodically and rollback to the last saved state after a failure and try to recover from there.

IV.OVERVIEW OF RECOVERY TECHNIQUES

Different kinds of recovery are possible for a database. The "kinds of recovery" that we consider are in fact "qualities of recovery," which can be useful in comparing and evaluating different recovery techniques.

The kinds of recovery considered are:

1) Recovery to the correct state.

2) Recovery to a correct state which existed at some moment in the past (i.e., a checkpoint).

3) Recovery to a possible previous state; this would allow, for example, restoration of a set of previously existing states of files that may not have existed simultaneously before.

4) Recovery to a valid state.

5) Recovery to a consistent state.

6) Crash resistance (explained below).

Crash resistance is provided if the normal algorithms of the system operate on the data in such a manner that after certain failures the system will always be in a correct state, i.e., the state the system was in before the last operation on the data was started (or possibly the last series of operations). Thus, crash resistance obviates the need for recovery techniques to cope with a certain class of failures.

REFERNCES

1. B. W. Johnson, "Designing and analysis of fault tolerant digital systems," Addison-Wesley, 1989.

2. Garg, "Soft error fault tolerant systems: cs456 survey," URL: www.csc.ncsu.edu/faculty/xie/softerrors.html

3. D. K. Pradhan, "Fault-tolerant computer system design," Prentice Hall PTR, 1996.

4. G.K.Saha, "Software implemented fault tolerance through data error recovery," ACM Ubiquity, 6(35), 2005,

5. G. K. Saha, "Software based fault tolerant array," IEEE Potentials, 25(1), pp.41-45, Jan 2006, IEEE Press.