Copyright

By

Nandita Raman

2012

The Thesis Committee for Nandita Raman
Certifies that this is the approved version of the following thesis:

Benchmarking Tests on Recovery oriented Computing

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Dewayne. E. Perry

Herb, Krasner

Benchmarking Tests on Recovery oriented Computing

By

Nandita Raman, BTech

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2012

Dedication

First, I thank god for his grace. I would like to dedicate this thesis to my parents, Raman Rajagopal and Sathyabama Raman, and my sister Pavithra Raman. They have always been confident in me and truly supportive in all my endeavors.

Acknowledgements

I express my sincere gratitude to my advisor, Professor Dewayne E. Perry for his tremendous support and guidance. His continuous encouragement was a great help in doing my thesis. I would like to thank Professor Herb Krasner, for consenting to be in the supervising committee, reviewing my work and providing valuable suggestions.

Abstract

Benchmarking Tests on Recovery Oriented Computing

Nandita Raman, MSE

The University of Texas at Austin, 2012

Supervisor: Dewayne. E. Perry

Benchmarks have played a very important role in guiding the progress of computer science systems in various ways. Specifically, in Autonomous environments it has a major role to play. System crashes and software failures are a basic part of a software system's life-cycle and to overcome or rather make it as less vulnerable as possible is the main purpose of *recovery oriented computing*. This is usually done by trying to reduce the downtime by automatically and efficiently recovering from a broad class of transient software failures without having to modify applications. There have been various types of benchmarks for recovering from a failure, but in this paper we intend to create a benchmark framework called the warning benchmarks to measure and evaluate the recovery oriented systems. It consists of the known and the unknown failures and few benchmark techniques which the warning benchmarks handle with the help of various other techniques in software fault analysis.

Table of Contents

List of Figures

CHAPTER 1: INTRODUCTION

There are many reasons for having to restart software, and many ways to do it. It is shown from a lot of studies that the main source of downtime in large scale software systems is caused by intermittent or transient bugs. Some systems have a variety of ways to stop - an operating system can shut down cleanly, panic, hang, crash, lose power, etc. When shutting down programs cleanly, there is often long wait before starting to use the applications again because of the time it takes to shut down and the time to come back up; when crash-rebooting, wait time is less and unavailability consists only of the time to recover. Ironically, shutting down and reinitializing can sometimes take longer than recovering from a crash.

But by doing so, it might affect the system's performance. There is always ideal goal of making the system perform very well and not crashing at all. It is not always possible but we can only try to make them better by using various techniques to make the system as recoverable as possible.

The software development of a system also has effects on the system performance. There are some risks with them that would be a cause for the system performance to degrade. No software is perfect before delivering (or even after it is in the market). When that is the case, any software that has not been tested for errors and faults, is going to make the system less recoverable. A failure is a departure of system behavior in an execution and a fault is the defect that causes the failure when executed.

When such failure occurs, the system or the application must be able to handle the failure and not bail out on it. To do that, there are many recovery processes needed and

benchmarks to evaluate the recovery processes. Recovery oriented computing has helped in understanding the nuances of what is expected from a system when it fails and how it can be dealt with without affecting the performance.

The main challenge for recovery–oriented techniques involves knowing when and how frequently to try them. Although techniques such as machine learning and control theory can play useful roles in the future, there are ways to empower application writers and service operators to minimize recovery time in the face of failures. Once it is understood the best that can be done by these expert users and what can be learnt from their experiences, automating the solutions or deploying them in other systems can be developed. While trying to automate, it is better to avoid any human interaction because tools for human error recovery are not going to be possible in an automated system.

Benchmarking [16][18][20], is another concept of comparing the current system to a standard and trying to equalize or outperform the standard. This is considered to be one of the best practices for fault tolerance and error recovery. We have emphasized more on the benchmarks and different kinds of benchmarks that are used and can be used for recovering from a failure to make the system more robust. Availability and dependability benchmarks are some of the benchmarks that have been widely used for checking the system before delivering.

The contribution of this thesis is a new benchmarking approach called the *warning benchmark framework*. This framework enables us to evaluate failure recovery. The warning benchmark framework consists of methods to evaluate the failure and once it is detected, recovery process begins. Most of the recovery is done using the backward recovery process that has been described in chapter 2. But the main focus of the research

is about handling the errors that are unexpected. Expected errors can be handled using some existing benchmarks and metrics that are very effective but there is always a trouble when it comes to unexpected faults. Exceptional handling is one main method for unexpected errors.

An exception handling mechanism is a control structure that allows programmers to describe the replacement of normal program execution by exceptional execution when the occurrence of an exception is detected. Unknown errors are handled using the warning benchmarking framework.

Chapter 2 has the details of software metrics and fault tolerant mechanisms. Recovery oriented computing is explained more in detail in Chapter 3 and what a benchmark is and what are the types of benchmarks used is explained in chapter 4. Chapter 5 talks about the method we intend would work for recovery using benchmarks, warning benchmark framework.

CHAPTER 2: SOFTWARE METRICS AND FAULT TOLERANCE

2.1. Software metrics

Software Engineering needs some kind of measurement at every level to make sure the quality is not compromised. To measure that we use metrics and in general it usually covers a wide range of diverse activities including Cost and effort estimation, Productivity measure, Quality control and assurance, Data collection, Reliability models, performance evaluation, Algorithmic/Computational complexity, and Structural and Complexity metrics. All of these are huge topic by itself.

Of these metrics, one of the most important that would be taken into consideration for all Software Engineering measurement is the Cost and Effort estimation. To predict the project costs at the early phases in software life cycle was the aim and to do so, there were lot of software cost and effort estimation have been proposed and used. Some of them are the well known Boehm's COCOMO model [7], Putnam's SLIM model [8]. Effort in these models is measured by making the effort a predefined function of one or more variables (e.g. size) of the product which is defined as Lines of Code (LOC) in some models (COCOMO) and as Function Points (FP) in some other models.

Data Collection [9] is considered as one of the important metric only because to predict and measure cost and its productivity to monitor the software process is going to completely depend on the Data Collection. The collection of data requires human observation and reporting. But this requires system analysts, programmers, testers to record the raw data and is not guaranteed to be correct. Automatic data capture is therefore desirable and sometimes very essential to record the execution time.

Function point analysis is another measure in software metrics. It is used instead of the source lines of code (SLOC) that is usually used in any software metric application. FPA is used to measure software effort, costs, and size. FPA can be used for estimating test cases, estimating overall project costs, schedule and effort, understanding the appropriate set of metrics.

2.2. Fault Tolerance

At every stage of developing software we encounter faults and errors. One way to reduce the risk of software design faults and there by enhance software dependability is to use Fault Tolerance techniques. This is used either in the procurement or the Development phase of software. The term 'Fault Tolerance' means it has to tolerate the errors or the mistakes that could happen in the system or the application during the development. To be able to tolerate the software faults in the system, techniques have been implemented to prevent system failure from occurring. Depending on the type of environment, fault tolerant techniques are used. The types of environment and the techniques [15] associated with them are –

Single Version Software Environment [15] - This is used for partially tolerating the software design faults and the techniques associated are monitoring techniques, atomicity of actions, decision verification and exceptional Handling.

Multiple Version Software Environment [15] – Design diverse techniques are used in this and they usually utilize functionally equivalent yet independently developed software versions to provide tolerance of software design faults. Recovery Blocks and N-Version Programming are some of the techniques used in this environment.

Multiple Data Representation Environment [15] – In this, it utilizes different representations of input data to provide tolerance to software design faults. Retry Blocks and N-Copy Programming are some of the few techniques used

These are the general types of fault tolerant techniques that are widely used in software systems and applications. Different techniques are used for different situations according to their needs. Some of the fault tolerant techniques that are of utmost importance for benchmarking will be explained in the later chapters.

2.2.1. INTRODUCTION TO RECOVERY

Error Recovery is one of the important processes in fault tolerance that has fault detection, error diagnosis, error containment / isolation and error recovery. Since recovery is the main focus, and the rest of the processes are to certain extent self explanatory, we will be looking more into error recovery process.

Error recovery [23] can be done in two methods; one is forward recovery and the other, backward recovery [22]. This is a very broad and general way to look at any type of error recovery.

2.2.1.1. Backward Recovery

When the program encounters an error, it generally enters the erroneous state. The recovery techniques then try to return the system to an error-free state. Backward recovery does this by rolling back or restoring it to a previously saved state. It is assumed

that the previous state is error-free. If it does have an error, the same steps are repeated until we reach to an error free state. At every fault detection and rolling back to previous state, it is saved and recorded. This is called Check Pointing, and we store it in such a way that it is not affected by any failure. Hence upon error detection, the system is restored back to its previously saved error-free state [10] and then operations continue from there. Below is the pictorial representation of backward recovery.

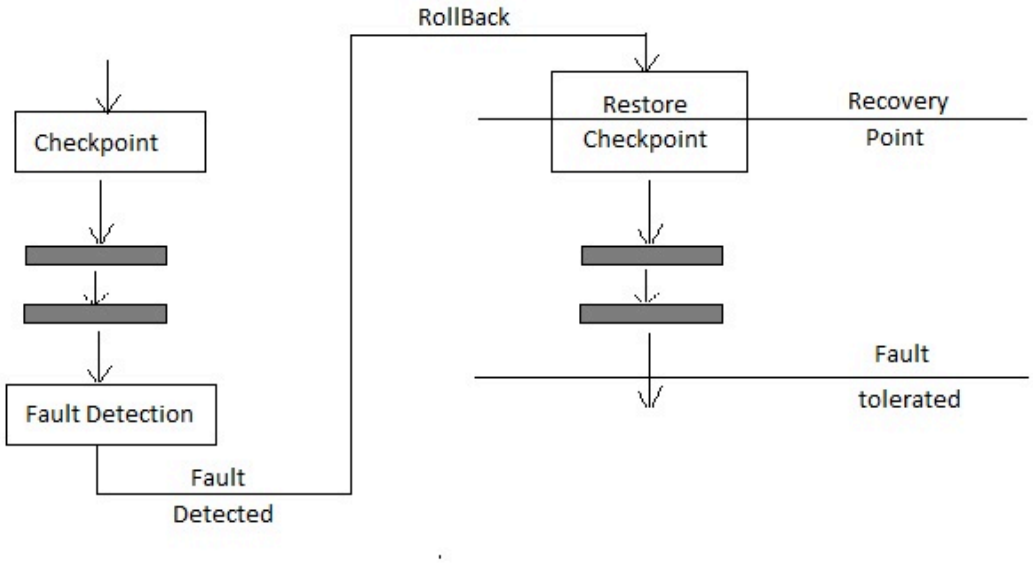

Figure 1: Backward Recovery

There are advantages to backward recovery and especially to the benchmarks that will be covered in future chapters, this is very important. The main advantage of backward recovery is that it can handle unpredictable errors [11] (if the error does not affect the recovery system). This feature of backward recovery seems to be very important because it is possible to make any system not fail completely and abruptly.

Backward recovery can also be used regardless of damage in the system. It not only requires zero knowledge of the errors in the system, but also has a uniform pattern of error detection and recovery and is application independent. Backward recovery is particularly suited for transient errors. Once the recovery is done, the error might have gone for good when restarting with the checkpoint. Only considered disadvantage is that it requires lot of resources for the recovery and check pointing. Implementation-wise, the system might have to be halted temporarily.

Given all these points, backward recovery clearly is a fault tolerant technique that would be used in almost all of the software systems/applications.

2.2.1.2. Forward Recovery

In forward recovery [23], it basically finds an error free state based on an algorithm that uses redundancy to select the correct state. All the redundant processes are executed in parallel and they together provide a group of acceptable state. From the potential results, fault detection and handling unit performs error compensation and derives an answer deemed acceptable. Forward recovery is fairly efficient in terms of overhead and if the characteristic of faults are well understood, forward recovery can provide more efficient solution. Faults involving missed deadlines may be better recovered from using Forward recovery than by introducing delay in rolling back and recovering. The main drawback of forward recovery is that it is very application specific and it requires the knowledge of the error. There are lots of limitations to the recovery that it can do if it is out of the specifications.

Forward recovery can be done when there is need for recovery quickly and does not have enough time for recovering.

2.2.2. RECOVERY BLOCK

Recovery block is one of the two diverse software fault tolerance techniques. It is also considered as a dynamic technique and the implementations were developed by Randell [16] and Hecht.

Recovery block uses Acceptance Test (AT) and backward recovery to get through fault tolerance. Most of the program functions can be performed in more than one way using different algorithms and designs. All of the functions have varying degrees of efficiency in terms of memory management, reliability, time, complexity, etc. Recovery block chooses these variants such that the most efficient module is located first and is titled the primary try block. The less efficient variants are placed serially after the primary try block and are referred to as secondary or alternate try blocks. This would result in the graceful degradation in the performance of the variants.

2.2.2.1. Recovery Block Technique

Recovery block implementation consists of an executive, Acceptance Test (AT), primary and secondary try blocks. The general Syntax is-

```
Ensure          Acceptance Test
By              Primary Alternate
Else by         Alternate 2
Else by         Alternate 3
….
Else by         Alternate n
Else failure exception
```

Figure .2. Recovery Block Syntax

In the Recovery block syntax, it will first try to ensure the AT by using the primary try block. If it does not pass the test, then n-1 alternates will be attempted until an alternate's results pass the AT. If no alternates are successful, an error occurs.

2.2.3. PROGRAMMING TECHNIQUES

Many software tolerance techniques use programming techniques. The programming techniques that are generally used are checkpointing [21], assertions and atomic actions.

2.2.3.1. Assertions

Assertions are the common means of program validation and error detection. Randell [5] presented executable assertions as the central for the design of fault tolerant programs. An executable assertion is a statement that checks whether a certain condition holds good among the other variables and if it does not hold good, it tries to do something else. In simple word, it basically check the current program state to determine if it is corrupt by testing for out of range values , relationship between variables and inputs and known corrupted states.

Assertions may be used to only produce a warning upon detection of corrupt state . For example, when a corrupt state is detected, the assertion may have to halt the program execution or try and recover from the corrupt state. This is completely application-dependant task. Traditional assertions generate warnings when the condition is not checked but do not try and recover.

Recovery assertions specifically, use forward recovery mechanisms that attempt to replace the current corrupt state with a correct stare. In check-pointing, the entire state

or only specific variables can be replaced, depending on the system constraints and overhead involved in saving and restoring the variables or its values.

Executable assertions are usually Boolean in nature and the functions are also Boolean, true when the condition holds well, false otherwise.

If not *assertion* then *action*

Where *assertion* is a Boolean expression and *action* is a method or procedure.

2.2.3.2. CHECKPOINTING:

It is used in error recovery when we try to restore a previously saved state of the system when a failure is detected. Recovery points, points in time during the process execution at which the system is saved, is established. The recovery point is discarded when the process accepts the result and is restored when a failure is detected.

There are mechanisms other than checkpointing for establishing the recovery points. Audit trails [7] and recovery cache [8] [9] are the other mechanism that can be used for recovery points. In checkpointing, a complete copy of the state is saved when it needs to establish the recovery point. In recovery cache, it only saves the original state of the objects whose values change after the latest recovery point. Audit trails record all the changes made to the process state. There are advantages and drawbacks to each of the method in its own way, but checkpointing will mean combination of all the mechanism and will do the things that are of top priority to the system.

The information should be stored on a stable storage so that even if node fails, the saved information will be safe. For a single process system, checkpointing and recovery is simpler than those with multiple processed and multiple nodes.

There are two approaches to multi-process backward recovery – asynchronous and synchronous checkpointing. In asynchronous checkpointing, like the name itself, checkpointing by different nodes in the system are not coordinated. But, sufficient information is maintained in the system so that when rollback and recovery is required, the system can be rolled back. There is the risk of unbounded rollback although the cost of asynchronous checkpointing is lower. Almost all of the checkpoints need to be saved because when there is a rollback, any remote state can be restored. The drawback of asynchronous checkpointing is that it can be useful only when there is a rare case of an expected failure and has very limited communication between the processes.

In a synchronous checkpointing, the checkpoints are all coordinated and comprised of a consistent system state. Since there is a need for coordination, cost is more but the rollback is reduced. Unlike in the asynchronous method, only a few checkpoints are needed to be saved.

An example of checkpointing and backward recovery technique implemented [6] in C++.

```
Try
{
        T oldobject =object;
        Alternate (object);
        If (accept (object))
        {
                Return;
        }
}
Catch (…)
```

```
{
        Object = oldobject;
        Continue;
}
```

Figure.3. Backward recovery technique as a part of generic recovery block

Another way of implementing backward recovery is by making the alternate return a new object than modifying the old object. But the previous method is preferred because it only involves initialization but this has to be initialized and assigned. It is preferable to save and restore the state that has changed.

2.2.3.3. ATOMIC ACTIONS

This is generally used for error recovery in concurrent systems. These systems must be structured in such a way that the complex asynchronous activities related to fault tolerance can be achieved. Quality and reusability has increased by using the atomic actions and also reduced the code complexity [9]. An atomic action is an action that is, indivisible, Serializable and recoverable. Indivisibility is the property where all the steps are either complete or none of them are.

The property of atomicity is that if an action is successfully executed, its results and changes it made on shared data become visible. However, if it is a failure, it will detect the failure and return without any changes in the shared data. This enables easy damage containment and error handling because the fault, error propagation and error recovery are all in one single action. Thus, it will not cause any disruption to the other system activities while this is done.

CHAPTER 3: RECOVERY ORIENTED COMPUTING

From the statistics [1] that were obtained, even with marketing campaigns promising 99.999% availability, well-managed servers today achieve 99.9% to 99%, or 8 to 80 hours of downtime per year. Each hour can be costly, from $200,000 per hour for an Internet service like Amazon to $6,000,000 per hour for a stock brokerage firm. Total cost of ownership (TCO) [1] ranges from 3 to 18 times the purchase cost of many cluster-based systems, and a third to a half of that money is spent recovering from or preparing for failures. We know that certain failures are inevitable and there is truly nothing much that can be done to eradicate them completely. But, we can definitely try to reduce the number of failures and bugs and not make them affect the system performance completely.

It is said that in Recovery Oriented Computing (ROC) the hardware faults, software bugs, and operator errors are facts to be coped with, not problems to be solved. This is Peres's law. In software metrics, Mean Time to Repair (MTTR) is better to look at rather than Mean Time to Failure (MTTF) because ROC reduces recovery time and hence higher availability is obtained.

Recovery–oriented approaches [13][14] are usually challenging, because they require us to deal with cleaning up after something has already gone wrong, but it's unclear and not obvious what to clean up. For example, if there is a crash because of a corrupted system data structures, these should be discarded and rebuilt before continuing, but the application data should remain intact or worse, should be reconstructed if it has been damaged. Moreover, we have to be very quick in identifying what to keep, throw

away, or rebuild. If a decision is too difficult to automate, we must also determine what kind of visibility into the system and what manual-repair facilities the human operators charged with recovery would find most useful.

This kind of computing is derived from autonomic computing environments, where everything is automated and has its own properties. Recovery oriented computing is autonomous in nature and possesses the same properties that autonomic computing [25] has; self-healing, self-configuration, self-optimization, and self-protection. On the whole, these features put together can be called as self-managing and systems that thrive on this are self managing systems. An autonomic system makes decisions on its own, using high-level policies; it will constantly check and optimize its status and automatically adapt itself to changing conditions.

There is a lot to ROC world and to begin with, when there were problems with the system performance, the concept of ACME [26] was introduced. ACME stands for Availability, Changeability, Maintainability and Evolutionary growth. Scalability, cost and performance are also incorporated into this. Since there is a problem with the system performance and to make sure it is not compromised, we use the properties to make the system more robust.

Previously in ACME, availability needed to be delivered to the users 24X7. Changeability had to support rapid deployment of new software, version control upgrades, new apps and user interfaces. Maintainability has to reduce burden on system administrators and provide helpful, forgiving system Admin environments. Evolutionary Growth has to allow easy system expansion over time without sacrificing availability or

maintainability. These were the expectations for a system to work efficiently and reduce the risk of failure.

But now-a-days in ACME, failures have become a part of life and traditional fault-tolerance doesn't solve all the problems. Changeability in back-end system tiers, software upgrades are difficult, failure-prone or ignored. System maintenance environments are unforgiving and human operator error is single largest failure source.
In evolutionary growth, evolves well in the front end but back-end scalability is difficult because it is operator intensive. There are statistics for such availability failures and causes for them.

Failures have become a major concern in system availability and maintenance, thus leading to an unreliable system. Reliability is also affected in the process. One leading to the other problem does not sound very good in a system.

The quote "*If a problem has no solution, it may not be a problem but a fact, not to be solved but to be coped with over time*" by Peres is considered as a guiding proverb of Recovery Oriented Computing (ROC) [13]. We take errors by people, software, and hardware into consideration to be facts, not problems that we must solve, and fast recovery is how we cope with these inevitable errors.

It is proposed and proved previously to cope with the fact and to have a graceful recovery when there is a failure. A widely accepted equation for system availability is A=MTTF/ (MTTF+MTTR), where MTTR mean time to recover after a failure and MTTF is mean time to Failure. Since unavailability is approximately MTTR/MTTF, shrinking time to recover by a factor of ten is just as valuable as stretching time to fail by a factor of ten.

The focus is on recovery due to lot of reasons. First, we all know Human errors are inevitable. From the surveys that were conducted using Telephone Switched Networks [3] and Internet Cluster Services [2], we deduce human error as one of the prime reasons for focusing on recovery. Secondly, MTTR can be measured directly. Either number of days or number of hours, it is done directly.

The goal of Recovery-Oriented Computing is to improve system dependability. To evaluate the progress in developing ROC systems and to compare the results with existing systems, benchmarks are used that provide a reproducible measure of system dependability.

There are some standard dependability benchmarks [27] that are developed and use the injection of system-level faults and perturbations to evaluate the impact of realistic failures on delivered quality of service. This will be explained in detail in chapter 4.

CHAPTER 4: BENCHMARKING

Software engineering as such has lot of problems such as Traceability, Reverse Engineering, Concept Location, Reuse, Impact Analysis and so on. For these problems to arrive at a solution, the concept of benchmarking arose. Benchmarking [19] [28] has been used in computer science to compare the performance of computer systems like in Information retrieval algorithms, databases and many other technologies. Because the primary goal of benchmarks has always been to allow comparison of competing systems, benchmark design has emphasized on some features like comprehensiveness, repeatability, representativeness of real life situations, fairness, relevance over time, and independence from the specifics of the system under test. Dimensions typically measured are quality, time and cost. In the process of benchmarking, identifying the best algorithm and comparing them with the one we have is the way we learn how well the targets perform.

There are different Types of benchmarks used for various applications. Depending on the application and its need, the benchmarks vary accordingly. Some of the benchmarks are productivity benchmarks, quality benchmarks, performance benchmarks [30], dependability benchmarks [31], function point analysis [29], etc.

Though these are benchmarks used in various domains and are different from one another, the basic similarity or the basis of benchmarking concept is comparison.

There are types of benchmarks that we use for recovery process.

1. Availability and Performance Benchmark.
2. Maintenance Benchmark.
3. Dependability Benchmark.

4.1. Availability and Performance Benchmark

The term availability has lot of interpretations. Usually, availability has been defined as a binary metric that describes whether a system is *up* or *down* at a one point of time. Further, to add to this definition it also computes the percentage of time, on average, that a system is available (*up*)—this is how availability is defined when a system is described as having 99.999% availability, for example.

Once the availability metric for a given type of system is selected, our challenge is to accurately measure them in a controlled benchmarking environment. But, it might be complicated to do this, because typical benchmark environments are explicitly designed to prevent the kinds of exceptional behavior that would cause availability to be affected in real-world systems. Hence, in order to perform availability benchmarks [32], it is necessary to have a benchmark environment that provides a means of generating fault-provoking stimuli and maintenance events (human maintenance) and then test them. The primary technique that enables such an environment to be constructed is direct fault injection into the system under test.

To build an availability benchmark, we also need a way to generate a realistic workload and to measure the appropriate quality of service metrics. But we can save some time and effort to certain extent by trusting the performance benchmarking

community. While they exist, we simply can use the existing performance benchmarks to generate a representative workload for the type of system under test, and to measure the desired metrics at that point of time. If the time for measurement is too small, these workload-generating performance benchmarks may need to be adapted to run continuously, repeatedly measuring the desired metric over small time periods.

Performance benchmarks [30] have been an integral part of computer systems research and have addressed these issues in various ways. History has shown that performance benchmarks are commonly used for two purposes. They are used for competitive head-to head comparisons and evaluation of complete systems on full workloads in scenarios designed to emulate a production environment. But they are also used by developers and researchers who are attempting to evaluate design tradeoffs and measure incremental design and implementation improvements.

Metrics like accuracy and completeness need to be measured and to do so, the system under test may also need to be modified that are generally neglected by performance benchmarks. Lastly, if workload induced faults such as erroneous inputs are to be used, the workload generator may need further modification to inject these faults at the appropriate time.

Now that we have a benchmark environment that supports fault injection and a performance benchmark that does both work, a continuous workload generator and a quality of service data collector, running an availability benchmark consists of two steps. First, the workload generator is run without injecting faults and the desired metrics are recorded with the traces of the values obtained. At this, it establishes a baseline

measurement for a non-faulty system. Second, the workload generator is run while simultaneously injecting a fault workload, and again desired metrics are recorded with the trace of the values obtained. This step is important, since it produces a trace of the behavior of the system's quality of service over time in response to various faults, which is the time-dependent availability metric that is we were looking for.

There is still a challenging problem in developing benchmarks based on multi-fault workloads that is to be human free administration for maintaining the system. How to realistically and reproducibly simulate the behavior of a human administrator in maintaining the system and in responding to failures originating from fault injection is the problem. Such maintenance events cannot be ignored, as very few modern systems are self maintaining and others will require human intervention to complete the tasks. The simplest solution is to run the benchmark with simulated maintenance events that represent correct and appropriate human maintenance actions.

4.2. Maintainability Benchmarking:

Quantifying the maintainability of a computer system is very challenging. There are two challenges that are particularly important and difficult. The first is that a system's maintainability depends directly on the way that the system is used. Every system is different with respect to workload, availability requirements, and administrative policies between installations and all this can lead to vastly different maintainability challenges for the system.

The second challenge arises because maintainability is by default a human metric, which would reflect in the interaction of a human administrator's behavior with a system design. Any measure of maintainability needs to capture a wide range of human factors, including everything from the amount of time it takes to perform maintenance, to the usability of the system's administrative interface, to the psychological effects of complexity in a maintenance task, and to the user's probability for causing errors.

Furthermore, the importance and behavior of these factors can differ between users themselves, we know if there is a system, two different administrators will run into different types of problems while maintaining the system, and will draw different judgments about the system's ease of maintenance. This point reflects the inherent variability that arises when human factors are considered. This is not an issue with performance benchmarks since it often attempts to strip out any potential for human variability. The human factors are so important in measuring maintainability that their accompanying variability must be accepted and incorporated into the benchmark.

The first challenge in maintainability of workload can be addressed by applying the ideas of *application-specific benchmarking* [6]. The basic concept behind application-specific benchmarking is to characterize the system of interest along as many different axes as possible, producing a large set of results that capture all of the fundamental behaviors of the system and any important interactions between the behaviors. Once this system characterization has been obtained, the system's workload is characterized in a similar way, by identifying and ordering the particular characterization of system behaviors that are relevant to the application or workload at that time. When

appropriately merged and distilled, these two independent characterizations produce an overall benchmark score that is relevant to both the system and the workload of interest. Though application-specific benchmarking till now has only been applied to the performance domain, we still think it is a good approach to use for maintainability since it can provide overall maintainability scores for a variety of different workloads and environments, while using a base system characterization that can be derived independently of the workload and environment.

The second challenge in benchmarking maintainability [32] would be about how to incorporate human factors into the benchmark metrics, and techniques. One way to approach this challenge would be to build a model of how human administrators respond to system-generated stimuli while carrying out maintenance tasks, and to then apply this model to the various systems under test. But, this is an extremely difficult task, obviously because it essentially requires building an artificial intelligence that is able to handle the myriad unforeseen situations that arise during system maintenance. If this model is built, it would probably be a huge step forward in the state of the art of system maintenance, as the model could be used as a replacement for human administrators, instantly turning existing systems into futuristic self-maintaining entities.

A much more practical approach to addressing the human challenge is to simply incorporate human beings into the benchmarking process. But by doing so ,it exposes the human factors involved in the maintenance of a test system.

However, using human subjects has several implications on benchmark design. First, it makes experiments much more labor-intensive due to the need to train, evaluate,

and debrief human subjects. More importantly, it requires extra care in choosing metrics and measurement techniques that can capture the impact of human factors while not being confounded by the inherent variability in behavior both across and within different human subjects.

Though we know it is going to be a complex problem brining in human subjects to maintain, it still is going to be an option for maintaining benchmark, after all, it's all developed by human.

4.3. Dependability Benchmarking:

We characterize a system's dependability as its ability to provide the desired quality of service in the face of internal and external perturbations. There are some papers and research journals that have defined full-scale competitive benchmarks that attempt to fully address issues such as completeness, representativeness of real-life operational environments, fairness, and repeatability. Because they strive for comprehensiveness, these monolithic benchmarks are expensive to run. Further, certain attributes of such dependability benchmarks make them inherently more time consuming, and hence more expensive to run than that of performance benchmarks. For example, the need to inject and measure the effect of many different kinds of perturbations and to collect data to develop a realistic error model.

We believe that just like the case for performance benchmarks, it is important for developers and researchers to be able to run dependability benchmarks in-house and on a small scale. Since it would be used day-to-day for development and research, the

benchmarks must be much less expensive per run in terms of time and money than full-scale competitive benchmarks.

Just like how commercial and research interest in system dependability has grown a lot, interest in benchmarking not just performance, but also system dependability has also grown quite a lot. The dependability benchmark [31] that was created basically does two different things: one that measures a system's overall end-to-end dependability by quantifying system response to realistic injected errors, and one that informally assesses particular aspects of a system's dependability.

There are lots of features or attributes that are closely related to dependable software. A benchmark on any system or software would be considered a good one if there are no negative impacts of the technology on dependability of the system.

CHAPTER 5: PROPOSAL AND DISCUSSION

We have seen in the previous chapters that some failures and faults are inevitable and to overcome them, came the benchmarking strategy. Using the benchmarks, rigorous testing and using various metrics makes the system or application more efficient before it has been delivered to the customer. However, software failures still occur during production runs since some bugs inevitably slip through even the strictest testing and benchmarks. For such situations, we use the warning benchmark framework that would go through the known and the unknown failure/fault phases to try and recover from the failure. There are some of the techniques that were used previously for different purposes that we will be using for this framework.

The contribution of this thesis is the Warning Benchmark Framework that provides the following benchmarks to measure and evaluate the recoverability of recovery-oriented systems:

1. Known Fault
    a. Unit – Testing
    b. Availability and Dependability Benchmarks
    c. Maintainability Benchmark

2. Unknown Fault
    a. Exception Handling
        i. Dependability cases
    b. Rest- of – unknowns

5.1. Warning Benchmark Framework

There are functional and nonfunctional faults in any system or application. We have seen lot of benchmarks and metrics the developers use before delivering a system or while developing. The warning benchmark framework that we propose comprises of various benchmarks and metrics put together that handle any kind of fault and failures in a system.

Warning Benchmark can be broadly categorized into two sections; one is the anticipated or the known faults and the other being unanticipated or the unknown faults.

5.1.1. KNOWN FAULT BENCHMARKS

Known faults are those that the system would be prone to fail in. We have two most important known anticipated faults that would arise from. One is the faults or errors that come from Software/Hardware failures, the other being Human Error. It is a well known fact that these errors are inevitable and as a good developer, one must anticipate errors from these. As we have mentioned earlier about the different Benchmarks and metrics, some of them are going to be of use for this recovery oriented benchmark that we are going to discuss. A few metrics that we propose to be part of this Warning Benchmark are FPA, Complexity, and Unit Testing.

Anticipated errors are primarily treated by using benchmarks. Since there would be enough knowledge about the failure, using a benchmark that is appropriate would be more than sufficient for recovering. Some of the benchmarks that are useful in such cases are Availability and Performance benchmark, dependability benchmark and maintainability benchmark. Detailed explanations about these benchmarks are given in chapter 4.

5.1.2. UNKNOWN FAULT BENCHMARKS-

The second type of faults is the unanticipated faults. Exception handling is the only available unanticipated fault. Robust exception handling in software can improve software fault tolerance and fault avoidance, but no structured techniques are available for implementing dependable exception handling.

5.1.2.1. Exception Handling:

Exception handling is the method of building a system to detect and recover from exceptional conditions. Exceptional conditions are any unexpected occurrences that are not accounted for in a system's normal operation. It is difficult to protect a system from the effects of exceptional conditions because, by nature, all unusual occurrences cannot be anticipated when the system is designed. Some examples of exceptional conditions are incorrect inputs from the user, bit level memory or data corruption, software design defects that cause a system to enter an undefined state, and environmental anomalies. If these exceptional conditions are not properly caught and handled, they can cause an error or failure in the system. Failures due to exceptions are estimated to account for two thirds of system crashes and fifty percent of system security vulnerabilities.

More than two thirds of code written for systems is often devoted to properly detecting and handling exceptions. However, most software testing efforts focus on exercising the correct operation of code, and not determining how robust it is to exceptional conditions. Therefore, exception handling code is the least tested and most susceptible to bugs. [33]

We do know it is practically impossible to handle all the exceptional cases and is not possible to expect the system to tolerate all the unexpected errors. To get over this problem, graceful degradation or some other technique is adopted to keep the system safe without causing major hazards.

Programmed exception handling [34] modules are mechanisms built into software for specific exceptional cases that are known are likely to occur. Since these exceptional cases are relatively well understood, protection for them can be incorporated into the system. When a program is executing, if one of the exceptional conditions is detected, control is passed from the main process block to the special exception handling block. In our warning benchmark framework, it enters the unknown faults block. This code will deviate from normal execution to compensate for the exceptional condition and will attempt to mask it to prevent propagating an error condition to higher levels in the software hierarchy.

If the condition cannot be recovered, the exception handler may call checkpointing recovery code to return the system to a known state before the exception occurrence and retry the operation. And once it is done guessing the exception and fails, it goes to the ret-of-unknown fault block and simply does the backward recovery irrespective of what the error might have been.

5.1.2.2. Exception Handling Techniques:

There are many techniques to handle exceptions and some of the old and good software engineering techniques are code reviews, code walkthroughs, and complete testing, but are limited to the software of the system.

Code reviews and walkthroughs are human oriented metrics that is used and still considered a good practice. Code reviews are done by a set or group of technically sound people in the relevant field. Suggestions would be made by the elate group; styles of programming, looks of user interface are some of the example suggestions. They tend to see more of the results and focus on how to make the product better.

Walkthroughs are also human oriented; some of the good technical people are chosen to check the code and usually this is for finding faults with the code. It is a manual method of testing and finding errors through an expert in the relevant fields.

Testing is usually adopted to make sure some of the expected errors are eliminated and to make them bug free. Though there are lots of types of testing, unit testing and integration testing are widely used for checking if there is an error in the code. This is obviously not manual; it is automated and has been proved to be very effective to certain extent for unexpected errors.

Though there are these usual techniques for handling exceptions, there are no better methods for generating robust exception handlers or to ensure that all exceptions

have been accounted for. This has been put into the research arena and we have attempted to try to solve this problem.

Dependability cases [4], is one of the techniques for exceptional handling that we are using to incorporate into the warning benchmark framework. Dependability cases basically are a framework by itself that has the methodologies and lists all the possible ways an exceptional condition might occur. Once it is listed, the system designer can use them as a basis for developing the exceptional handlers. Robustness of the system is accentuated when this is done by following good software engineering processes.

Hazard analysis techniques [4] that include fault trees and fishbone diagrams are also a part of the framework that helps the designer in looking for exceptional conditions. With this framework, exceptional conditions can be developed into an taxonomy. But, it is definitely impossible to find all the exceptional conditions and work on all of them and hence we are not so sure as to how much efficient this might be on the system. However good the taxonomy of exceptional conditions we develop, there are high probabilities of excluding a class of exceptions that will lead to system vulnerability.

Though a class of exception might lead to system vulnerability, using the dependability cases have proved to be more efficient than before. It is proved to be 40 % more reliable, which is a huge difference from the previous systems.

Hence, to make it the system as less vulnerable as possible and more efficient, dependability cases framework along with rest- of – unknown faults benchmark and

benchmark for known faults are integrated into one complete master framework, Warning benchmarks.

5.2. WARNING BENCHMARK ARCHITECTURE:

The architecture of a warning benchmark is very simple. It consists of known and unknown blocks. Known block contains set of benchmarks that were previously specified. Unknown block contains dependability cases and rest-of- unknowns. Dependability cases are a framework by itself and all the three blocks return to the recovery process within the warning benchmark.

Hazard analysis helps in creative thinking about all the possible ways in which hazards or operating problems might arise. The technique gives new types of hazards in new designs, as well as hazards that have not been considered before in the checklist standards that were developed. Hence the basic property that hazard analysis has is the ability to explore all the unexpected conditions. Hazard analysis does not provide quantitative results but it provides us with a qualitative method of approach to the exception.

Fault trees are widely used, in conjunction with hazard analysis, helps in a logical way, to the events leading to an exception or all of the exceptions. Fault trees are a used for analyzing the causes of hazards and not for identifying what the hazard is. A fault tree is a graphical representation that helps organize information about faults and the causes that lead to them.

To certain extent, hazard analyses and fault trees together can be used to obtain a reasonably clear picture of a system's exception vulnerabilities. In the hazard analysis, it

generates a list of hazards to guard against, either by fault avoidance or by fault tolerance. A fault tree takes that list of hazards and helps to find the path to the given hazard with the help of the causal events or conditions.

Hazard analyses can be considered partially dependant on human, because the hazard list that is created depends on the analyst's imagination. It is better to have prior knowledge of what kinds or classes of vulnerabilities to look for. For these reasons, having a checklist or taxonomy of exception types would be useful.

The disadvantage of using this framework and checklist of hazards in a system design is that lots of long lists of hazards or events are apt to go unused. They are hard to remember, and are very difficult to immediately get them in printed form when needed. To add to this, due to its huge size of data, it is difficult to reproduce them from memory, even if they are organized taxonomically or hierarchically. We would need a starting point to start using this checklist.

Known faults block leads to all of the benchmarking techniques for the anticipated faults and the system acts according to the type of recovery process that benchmark specifies. Unknown faults lead to the exception handling block and tried to use the methods from the dependability cases to analyze and find what the error is. Once it is found, the recovery process for that error would be using the benchmarks specified for that particular error. If the error or the fault is still unknown and we cannot make the system wait for too long for the recovery, it goes into the newly added block in the warning benchmark, rest-of-unknowns, where it tries to see where the error is and it does

backward recovery process. This way, worst case, at least the system would not be damaged completely and a graceful degradation happens. It is expected to be 50% more reliable when using warning benchmark framework.

CHAPTER 6: CONCLUSIONS

We would like applications to be able to automatically recover from a variety of transient failures, freeing system operators to focus on higher-leverage tasks and requiring fewer operators to oversee larger- sized systems.

It is unrealistic to build a system that is completely bulletproof to exceptional conditions because we cannot anticipate all possible situations. Therefore it is necessary to build in default exception handlers that will attempt to recover from any of these unanticipated conditions. If the application is safety critical or has real-time deadlines, some form of graceful degradation must be put in place to reduce the harm or damage done by any system failures.

Benchmarks have been recognized for many years as an essential way to objectively evaluate design choices and full systems. Benchmarks were initially used primarily to measure and compare performance. And using such benchmarks, the system performance has improved tremendously.

In the recent times, the focus is mainly on the recovery process. To make the system more robust, warning benchmark framework is introduced. It has two blocks, known and unknown faults and some techniques and benchmarks are used to recover from the failure. In known block, availability, performance and maintainability benchmarks are used to recover from the failure and since they are known failures, it is

easy to use the appropriate benchmark. In unknown fault block, exception handling is the only fault tolerant technique used. It uses dependability cases to analyze the failure and find a way back to the cause of the failure for recovering from where it failed. If the dependability cases do not help in finding the fault, it traces back to a point where it is error free and performs backward recovery.

The contribution of this thesis is a new benchmarking approach called the *warning benchmark framework*. This framework enables us to evaluate failure recovery. The warning benchmark framework consists of methods to evaluate the failure detection and recovery process. .Using the warning benchmark framework can make the performance of the system better and try and avoid almost all of the error and is expected to be 50 % more reliable than already existing system, or worse have a graceful degradation. The warning benchmarks can be of very much importance for a system having lot of applications and big in size.

BIBLOGRAPHY:

[1] R. Kembel. Fibre Channel: A Comprehensive Introduction, p.8, 2000.

[2] David Oppenheimer and David A. Patterson. Studying and using failure data from large-scale Internet services. In Proc. SIGOPS European Workshop, Sept. 2002

[3] P. Enriquez, A. Brown, and D.A. Patterson. Lessons from the PSTN for dependable computing. In Workshop on Self-Healing, Adaptive, and Self- Managed Systems (SHAMAN), New York, June 2002.

[4] Maxion, Roy A.; Olszewski, Robert T., "Improving Software Robustness With Dependability Cases." Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing, June 1998, p. 346-355

[6] M. Seltzer, D. Krinsky, et al. The Case for Application-Specific Benchmarking. *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems (HotOS-VII)*, Rio Rico, AZ, March 1999.

[7] Boehm BW, Software Engineering Economics, Prentice-Hall, New York, 1981.

[8] Puntam LH, A General Empirical Solution to the macro Software sizing and estimating problem, *IEEE Trans Soft Eng* SE-4(4), 1978,345-361.

[9] Basili VR, Weiss DM, A Methodology for collecting valid software engineering data, *IEEE Trans Soft Eng* SE-10(6), 1984, 728-738.

[10] Jalote, P., Fault Tolerance in Distributed Systems, *Englewood Cliffs, NJPrentice-Hall,* 1994*.*

[11] Randell, B., and J. Xu, The evolution of the Recovery block Concept, in MR.Lyu(ed.), Software fault tolerance, New York: John Wiley and Sons, 1995, pp,1-21.

[12] Xie, Z., Sun, H. & Saluja, K. A survey of software fault tolerance techniques.*Engineering* (1988).

[13] David Patterson, Aaron Brown, et al, Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies*; Computer Science Technical Report UCB//CSD-02-1175, U.C. Berkeley* March 15, 2002

[14] Armando Fox. 2002. Toward recovery-oriented computing. In *Proceedings of the 28th international conference on Very Large Data Bases* (VLDB '02).

[15] Brian Randell. 1999. Fault Tolerance in Decentralized Systems. In *Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems* (ISADS '99). IEEE Computer Society, Washington, DC, USA, 174-.

[16] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. 2007. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering* (ASE '07).

[17] Barry W. Boehm. 1988. A Spiral Model of Software Development and Enhancement. *Computer*21, 5 (May 1988), 61-72.

[18] Aaron B. Brown and Joseph L. Hellerstein. 2004. An approach to benchmarking configuration complexity. In *Proceedings of the 11th workshop on ACM SIGOPS European workshop* (EW 11).

[19] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. 2008. Benchmarking Classification Models for Software Defect Prediction: A Proposed Framework and Novel Findings.*IEEE Trans. Softw. Eng.* 34, 4 (July 2008), 485-496.

[20] Stein, Roger M., "Be*nchmarking Default Prediction Models: Pitfalls and Remedies in Model Validation*", Journal of Risk Model Validation, Vol. 1, No. 1, (Spring 2007), pp. 77-113

[21] R. Koo, S. Toueg. Checkpointing and Recovery-Rollback for Distributed Systems. *IEEE Transactions on Software Engineering*; Vol. SE-13, No. 1; pp. 23-31; 1987.

[22] Bartek Kiepuszewski, Ralf Muhlberger, and Maria E. Orlowska. 1998. FlowBack: providing backward recovery for workflow management systems. *SIGMOD Rec.* 27, 2 (June 1998), 555-557.

[23] R. A. Maxion and R. T. Olszewski. 1998. Improving Software Robustness with Dependability Cases. In *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing* (FTCS '98). IEEE Computer Society, Washington, DC, USA, 346.

[24] Johann Eder and Walter Liebhart. 1996. Workflow Recovery. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems* (COOPIS '96).

[25] Jeffrey O. Kephart. 2005. Research challenges of autonomic computing. In *Proceedings of the 27th international conference on Software engineering* (ICSE '05). ACM, New York, NY, USA, 15-22.

[26] David Patterson, Recovery oriented computing *Presented at Princeton University, University of Illinois, and University of Michigan*, October 2002.

[27] Constantinescu, C. (2008) Dependability Benchmarking Using Environmental Test Tools, in Dependability Benchmarking for Computer Systems (eds K. Kanoun and L. Spainhower), John Wiley & Sons, Inc., Hoboken, NJ, USA.

[28] Siewiorek, D.P.; Hudak, J.J.; Suh, B.-H.; Segal, Z.; , "Development of a benchmark to measure system robustness," *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on* , vol., no., pp.88-97, 22-24 June 1993

[29] Yen Cheung, Rob Willis, Barrie Milne, (1999),"Software benchmarks using function point analysis", Benchmarking: An International Journal, Vol. 6 Iss: 3 pp. 269 – 276

[30] Claes Wohlin, Aybuke Aurum, H\&\#229;kan Petersson, Forrest Shull, and Marcus Ciolkowski. 2002. Software Inspection Benchmarking - A Qualitative and Quantitative Comparative Opportunity. In *Proceedings of the 8th International Symposium on Software Metrics* (METRICS '02). IEEE Computer Society, Washington, DC, USA, 118.

[31] Jesus Friginal, David de Andres, Juan-Carlos Ruiz, and Regina Moraes. 2011. Using Dependability Benchmarks to Support ISO/IEC SQuaRE. In *Proceedings of the 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing* (PRDC '11). IEEE Computer Society, Washington, DC, USA, 28-37.

[32] Aaron B. Brown. 2001. *Towards Availability and Maintainability Benchmarks: a Case Study of*. Technical Report. University of California at Berkeley, Berkeley, CA, USA.

[33] Dewayne E. Perry and W. Michael Evangelist, An Empirical Study of Software Interface Faults --- An Update, *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, January 1987, Volume II, pages 113-126.

[34] Cristian, F.; , "Exception Handling and Software Fault Tolerance," *Computers, IEEE Transactions on* , vol.C-31, no.6, pp.531-540, June 1982

Vita

Nandita Raman was born in Chennai, India. After her schooling at Padma Seshadri Bala Bhavan School and DAV school, she received her bachelor of Engineering in Electrical and Electronics from Anna University, India in may 2009. She joined the University of Texas at Austin in Spring 2010, to pursue her Master of Science.

Permanent address: nanditaaraman@gmail.com

This thesis was typed by Nandita Raman