NASA-CR-193401

June 1993

UILU-ENG-93-2227
CRHC-93-15

Center for Reliable and High-Performance Computing

IN-60-CR

118

[illegible]

EXPERIMENTAL ANALYSIS OF COMPUTER SYSTEM DEPENDABILITY

Ravishankar K. Iyer
Dong Tang

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Experimental Analysis of Computer System Dependability

Ravishankar K. Iyer and **Dong Tang**

Technical Report CRHC-93-15
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign

Urbana, Illinois, USA
July 1993

ABSTRACT

This paper reviews an area which has evolved over the past 15 years: experimental analysis of computer system dependability. Methodologies and advances are discussed for three basic approaches used in the area: simulated fault injection, physical fault injection, and measurement-based analysis. The three approaches are suited, respectively, to dependability evaluation in the three phases of a system's life: design phase, prototype phase, and operational phase. Before the discussion of these phases, several statistical techniques used in the area are introduced. For each phase, a classification of research methods or study topics is outlined, followed by the discussion of these methods or topics as well as representative studies.

The statistical techniques introduced include the estimation of parameters and confidence intervals, probability distribution characterization, and several multivariate analysis methods. Importance sampling, a statistical technique used to accelerate Monte Carlo simulation, is also introduced. The discussion of simulated fault injection covers electrical-level, logic-level, and function-level fault injection methods as well as representative simulation environments such as FOCUS and DEPEND. The discussion of physical fault injection covers hardware, software, and radiation fault injection methods as well as several software and hybrid tools including FIAT, FERRARI, HYBRID, and FINE. The discussion of measurement-based analysis covers measurement and data processing techniques, basic error characterization, dependency analysis, Markov reward modeling, software dependability, and fault diagnosis. The discussion involves several important issues studied in the area, including fault models, fast simulation techniques, workload/failure dependency, correlated failures, and software fault tolerance.

ACKNOWLEDGMENTS

The authors thank Kumar Goswami, Timothy Tsai, Gwan Choi, and Weilun Kao for their contributions to this manuscript. Special thanks go to Fran Wagner for proofreading the whole manuscript. Thanks are also extended to Inhwan Lee and Darren Sawyer for their comments on the manuscript.

We highly appreciate D. P. Siewiorek, J. Arlat, E. W. Czeck, J. Karlsson, and G. B. Finelli for permission to use figures (Figures 3.6, 4.2, 4.3, 4.4, 4.8, and 5.11) and algorithms (Sections 3.2.2 and 5.7.1) in their publications. Figure 3.9 is generated from "NEST: A Network Simulation and Prototyping Testbed," authored by A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, *Communications of the ACM*, Vol. 33, No. 10, copyright ACM 1990, by permission of the ACM.

TABLE OF CONTENTS

I. INTRODUCTION

In computer science more than in physical sciences, the experimenter must decide what to consider and what to ignore in data gathering and analysis, sometimes without the benefit of prior information or easily available intuition. How to obtain general models from experiments or measurements made in a particular environment is by no means clear. This paper discusses the current research in the area of experimental analysis of computer system dependability. The discussion centers around methodologies, major developments, and major directions of the research in the area.

Experimental evaluation of the dependability of a system can be performed at different phases of the system's life. In the early design phase, CAD (Computer-Aided Design) environments are used to evaluate a design via simulations, including simulated fault injections. Fault injection simulations can be used to investigate the effectiveness of fault tolerant mechanisms, to evaluate system dependability, and to provide timely feedback to system designers. However, simulations need accurate input parameters and the validation of output results. These should be estimated based on previous measurement-based analysis. In the prototype phase, a system runs under controlled workload conditions, and controlled fault injections are used to evaluate the system behavior under faults. Fault injections on real systems can provide information about the process from fault occurrence to system recovery, including error latency, propagation, detection, and recovery (reconfiguration may be involved). But fault injection can only study artificial faults, and it cannot provide some dependability measures such as MTBF (Mean Time Between Failures) and availability. In the operational phase, a direct measurement-based approach can be used to measure systems in the field under real workloads. The collected data contain a large amount of information about naturally occurring errors/failures. Analysis of such data can provide understanding of actual error/failure characteristics and insight into analytical models. Although measurement-based analysis is useful for evaluating real systems, it is limited to detected errors. Further, conditions in the field can vary widely from one system to another, casting doubt on the statistical validity of the results. Thus, all three approaches are complementary and essential for accurate dependability analysis.

In the design phase, fault injection simulations can be conducted at different levels: the electrical level, the logic level, and the function level. The objectives of simulated fault injection are to determine dependability bottlenecks, the coverage of error detection/recovery mechanisms, the effectiveness of reconfiguration schemes, the system TTF (Time To Failure) distributions, reliability, availability, performance loss, and other dependability measures. The resulting feedback of simulations can be extremely useful in cost-effective redesign of a system. In this paper, we discuss different techniques used in fault injection simulations. We also introduce different levels of simulation tools.

In the prototype phase, while the objectives of physical fault injections are similar to those of simulated fault injections in the design phase, the methods are radically different because real fault injection and monitoring facilities are involved. Physical fault injections can be conducted at the hardware level (logic or electrical) or at the software level (code or data corruption). Further, heavy-ion radiation techniques can also be used to inject faults and stress a system. Instrumentations used in fault injection experiments are illustrated using real examples, including several fault injection environments.

In the operational phase, the measurement-based approach needs to address issues such as how to monitor computer errors and failures and how to analyze measured data to quantify system dependability characteristics. Although there is extensive research on methods for the design and evaluation of fault tolerant systems, little is known about how well these strategies work in the field. A study of production systems is valuable not only for accurate evaluation but also for identifying reliability bottlenecks in system design. Several issues in measurement-based analysis, including workload/failure dependency, modeling and evaluation based on data, software dependability in the operational phase, and fault diagnosis are addressed.

Results of measurement-based analysis discussed in this paper are based on over 100 machine-years of data gathered from IBM, DEC, and Tandem systems. The evaluation methodology discussed includes: the use of the measured hardware and software error data to jointly characterize the interdependence between performance and dependability, correlation analysis to quantify correlated failures and their impact on dependability, Markov reward modeling of measured data to evaluate the loss of system service due to errors and failures, and algorithms that use on-line error logs to perform automatic fault diagnosis and failure prediction.

Before discussing methodologies and developments for each of the three phases discussed above, we present an overview of the relevant statistical techniques used in this area. The techniques cover the estimation of parameters and confidence intervals, distribution characterization including function fitting, and multivariate analysis methods including clustering analysis, correlation analysis, and factor analysis. Importance sampling, a statistical technique to accelerate Monte Carlo simulation, is also introduced. These techniques are later used in the discussion of analysis of data obtained from fault injections or measured from operational systems.

In discussing the experimental analysis approaches used in the three phases, a wide range of dependability issues, including error latency, error propagation, error detection, error recovery, error correlation, workload/error dependency, availability, reliability, performability, and reward rate, are addressed. In addition to presenting methodologies and major developments in each of these phases, we also critique the relative merits and research issues for different approaches. Most evaluation techniques introduced are illustrated via case studies of their uses on actual systems.

II. STATISTICAL TECHNIQUES USED IN THE AREA

In this section, we will introduce several statistical techniques commonly used in the analysis of data collected from fault injections and operational systems and used in simulation. The techniques discussed are not intended to be comprehensive. For a comprehensive study of statistical methods, the reader is referred to the advanced texts of statistics [Kendall77], [Dillon84]. In particular, we will discuss parameter estimation, distribution characterization, and multivariate analysis techniques. Most of these techniques are widely used in every phase of the experimental evaluation of dependability.

2.1. Parameter Estimation

The most important characteristics of a random variable are its distribution, mean, and variance. In practice, means and variances are usually unknown parameters. Thus, how to estimate these unknown parameters from data needs to be addressed.

2.1.1. Point Estimation

Point estimation is often used in experimental analysis, such as the estimation of the detection coverage from fault injections and the estimation of MTBF (mean time between failures) from field data. Each fault injection and each failure occurrence can be treated as an experiment. The following theory is based on the assumption that all experiments are independent and have the same underlying distribution.

Given a collection of n experimental outcomes $x_1, x_2, \cdots x_n$, of a random variable X, each x_i can be considered as a value of a random variable X_i. These X_i's are independent of each other and identical to X in distribution. The set $\{X_1, X_2, ..., X_n\}$ is called a random sample of X. Our purpose is to estimate the value of some parameter θ (θ could be E[X] or Var[X]) of X using a function of $X_1, X_2, ..., X_n$. The function used to estimate θ, $\tilde{\theta} = \tilde{\theta}(X_1, X_2, \ldots, X_n)$, is called an *estimator* of θ, and $\tilde{\theta}(x_1, x_2, \ldots, x_n)$ is said to be a *point estimate* of θ.

An estimator $\tilde{\theta}$ is called an *unbiased estimator* of θ, if E[$\tilde{\theta}$] = θ. The unbiased estimator that has the minimum variance, i.e., it minimizes Var($\tilde{\theta}$) = E[($\tilde{\theta} - \theta)^2$] among all $\tilde{\theta}$'s, is said to be the *unbiased minimum variance estimator*. It can be shown that the sample mean

$$\bar{X} = \frac{1}{n}\sum_{i=1}^{n} X_i \tag{2.1}$$

is the unbiased minimum variance linear estimator of the population mean μ, and the sample variance

$$S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \bar{X})^2 \tag{2.2}$$

is, under some mild conditions, an unbiased minimum variance quadratic estimator of the population variance σ^2. If an estimator $\tilde{\theta}$ converges in probability to θ, i.e.,

$$\lim_{n\to\infty} P(\ |\tilde{\theta}(X_1, X_2, \ldots, X_n) - \theta\ | \geq \varepsilon) = 0\ , \tag{2.3}$$

where ε is any small positive number, it is said to be *consistent*.

A. *Method of Maximum-Likelihood*

If the functional form of the p.d.f. of the variable is known, the maximum likelihood is a good approach to parameter estimation. In many cases, approximate functional forms of empirical distributions can be obtained (to be discussed in Section 2.2). For example, the software TTR (Time To Error) in two measured distributed operating systems was shown to have an hyperexponential distribution (see Section 5.3). In such cases, the maximum likelihood method can be used to determine distribution parameters.

The idea of the maximum likelihood method is to choose an estimator based on the assumption that the observed sample is the most likely to occur among all possible samples. The method usually produces estimators that have minimum variance and consistence properties. But if the sample size is small, the estimator may be biased.

Assuming X has a p.d.f. (probability distribution function) $f(x|\theta)$, where θ is an unknown parameter, the joint p.d.f. of the sample $\{X_1, X_2, ..., X_n\}$,

$$L(\theta) = \prod_{i=1}^{n} f(x_i|\theta) \tag{2.4}$$

is called the *likelihood function* of θ. If $\tilde{\theta}(x_1, x_2, \ldots, x_n)$ is the point estimate of θ that maximizes $L(\theta)$, then $\tilde{\theta}(X_1, X_2, \ldots, X_n)$ is said to be the *maximum likelihood estimator* of θ.

Now we use an example to illustrate the method. Let X denote the random variable "time between failures" in a computer system. Assuming X is exponentially distributed with an arrival rate λ, we wish to estimate λ from a random sample $\{X_1, X_2, \ldots, X_n\}$. By Equation (2.4),

$$L(\lambda) = \prod_{i=1}^{n} \lambda e^{-\lambda x_i} = \lambda^n e^{-\lambda \sum_{i=1}^{n} x_i} .$$

How do we choose an estimator such that the estimated λ maximizes $L(\lambda)$? An easier way is to find the λ value that maximizes $\ln L(\lambda)$, instead of $L(\lambda)$. This is because the λ that maximizes $L(\lambda)$ also maximizes $\ln L(\lambda)$, and $\ln L(\lambda)$ is easier to handle. In this case we have

$$\ln L(\lambda) = n\ ln(\lambda) - \lambda \sum_{i=1}^{n} x_i .$$

To find the maximum, consider the first derivative

$$\frac{d[lnL(\lambda)]}{d\lambda} = \frac{n}{\lambda} - \sum_{i=1}^{n} x_i .$$

The solution of this equation at zero,

$$\tilde{\lambda} = \frac{n}{\sum_{i=1}^{n} x_i} ,$$

is the maximum likelihood estimator for λ.

B. *Method of Moments*

Sometimes it is impossible to find maximum likelihood estimators in closed form. For instance, it is difficult to maximize the following p.d.f. of the gamma distribution $G(\alpha\theta)$

$$g(x) = \frac{1}{\Gamma(\alpha)\theta^{\alpha}} x^{\alpha-1} e^{-x/\theta}$$

in estimating α and θ, because of the existence of the gamma function $\Gamma(\alpha)$. The gamma distribution is often found useful for characterizing interval times in the real world. It will be shown in Section 5.3 that the software TTE in a measured single-machine operating system fits a multi-stage gamma distribution. In such cases, the method of moments can be used if an analytical relationship between the *moments* of the variable and the parameters to estimate can be found.

To introduce the method of moments, We first bring out the concepts of *sample moment* and *population moment*. The k-th (k=1,2,...) sample moment of the random variable X is defined as

$$m_k = \frac{1}{n} \sum_{i=1}^{n} X_i^k \ , \tag{2.5}$$

where $X_1, X_2, \ldots, X_n$ are a sample of X. The k-th population moment of X is just E[X^k].

Suppose there are k parameters to be estimated. The idea of the method of moments is to set the first k sample moments equal to the first k population moments which are expressed as the unknown parameters, and then to solve these k equations for the unknown parameters. The method usually gives simple and consistent estimators. However, some estimators may not have unbiased and minimum variance properties. The following example shows details of the method.

Consider the above gamma distribution example. We wish to estimate α and θ, based on a sample $\{X_1, X_2, \ldots, X_n\}$ from a gamma distribution. Since $X \sim G(\alpha, \theta)$, we know

$$E[x] = \alpha\theta \ , \qquad E[x^2] = \alpha\theta^2 + \alpha\theta \ .$$

The first two sample moments, by definition, are given by

$$m_1 = \frac{1}{n} \sum_{i=1}^{n} x_i = \overline{X} \ , \qquad m_2 = \frac{1}{n} \sum_{i=1}^{n} x_i^2 \approx S^2 + \overline{X}^2 \ .$$

Setting $m_1 = E[X]$ and $m_2 = E[X^2]$ and solving for α and θ, we obtain

$$\tilde{\alpha} = \frac{\overline{X}^2}{S^2} \ , \qquad \tilde{\theta} = \frac{S^2}{\overline{X}} \ .$$

These are the estimators for α and θ from the method of moments.

2.1.2. Interval Estimation

So far our discussion is limited to the point estimation of unknown parameters. The estimate may deviate from the actual parameter value. To obtain an estimate with a high confidence, it is necessary to construct an interval estimate such that the interval includes the actual parameter value with a high probability. Given an estimator $\tilde{\theta}$, if

$$P(\tilde{\theta} - e_1 < \theta < \tilde{\theta} + e_2) = \beta \ , \tag{2.6}$$

the random interval $(\tilde{\theta} - e_1, \tilde{\theta} + e_2)$ is said to be 100×β% *confidence interval* for θ, and β is called the *confidence coefficient* (the probability that the confidence interval contains θ).

A. Confidence Intervals for Means

In the following discussion, the sample mean $\bar{X}$ will be used as the estimator for the population mean. As mentioned before, it is the unbiased minimum variance linear estimator for μ. We first consider the case in which the sample size is large. By the central limit theorem, $\bar{X}$ is asymptotically normally distributed, no matter what the population distribution is. Thus, when the sample size n is reasonably large (usually 30 or above, sometimes at least 50 if the population distribution is badly skewed with occasional outliers), $Z = (\bar{X} - \mu)/(S/\sqrt{n})$ can be approximately treated as a standard normal variable. To obtain a 100β% confidence interval for μ, we can find a number $z_{\alpha/2}$ from the $N(0, 1)$ distribution table such that $P(Z > z_{\alpha/2}) = \alpha/2$, where $\alpha = 1 - \beta$. Then we have

$$P(-z_{\alpha/2} < \frac{\bar{X} - \mu}{S/\sqrt{n}} < z_{\alpha/2}) = 1 - \alpha \ .$$

Thus, the 100$(1 - \alpha)$% confidence interval for μ is approximately

$$\bar{X} - z_{\alpha/2} \frac{S}{\sqrt{n}} \le \mu \le \bar{X} + z_{\alpha/2} \frac{S}{\sqrt{n}} \ . \tag{2.7}$$

If the sample size is small (considerably smaller than 30), the above approximation can be poor. In this case, we consider two commonly used distributions: normal and exponential. If the population distribution is normal, the random variable $T = (\bar{X} - \mu)/(S/\sqrt{n})$ has a Student t distribution with $n - 1$ degrees of freedom. By repeating the same approach performed above with a t distribution table, the following 100$(1 - \alpha)$% confidence interval for μ can be obtained:

$$\bar{X} - t_{n-1;\alpha/2} \frac{S}{\sqrt{n}} < \mu < \bar{X} + t_{n-1;\alpha/2} \frac{S}{\sqrt{n}} \ , \tag{2.8}$$

where $t_{n-1;\alpha/2}$ is a number such that $P(T > t_{n-1;\alpha/2}) = \alpha/2$. Theoretically, Equation (3.8) requires that X have a normal distribution. However, we will show later that the estimator is not very sensitive to the distribution of X when the sample size is reasonably large (15 or more).

If the population distribution is exponential, it can be shown that $\chi^2 = 2n\bar{X}/\mu$ has a chi-square distribution with $2n$ degrees of freedom. Thus, the chi-square distribution table should be used. Because the chi-square distribution is not symmetrical about the origin, we need to find two numbers, $x^2_{2n;1-\alpha/2}$ and $x^2_{2n;\alpha/2}$, such that $P(\chi^2 < x^2_{2n;1-\alpha/2}) = \alpha/2$ and $P(\chi^2 > x^2_{2n;\alpha/2}) = \alpha/2$. The obtained 100$(1 - \alpha)$% confidence interval for μ is

$$\frac{2n\bar{X}}{x^2_{2n;\alpha/2}} < \mu < \frac{2n\bar{X}}{x^2_{2n;1-\alpha/2}} . \tag{2.9}$$

B. Confidence Intervals for Variances

The estimation of confidence interval for variances is more complicated than that for means, because the sample variance cannot be simply approximated by a unique distribution (such as normal distribution) regardless of the population distribution. However, irrespective of the population distribution, $\lim_{n\to\infty} \mathrm{Var}[S^2] = 0$. Thus, a good confidence interval can be expected as long as the sample size n is large. Next, our discussion will be focused on the two commonly used distributions: normal and exponential.

If X is normally distributed, the sample variance S^2 can be used to construct the confidence interval. It is known that the random variable $(n-1)S^2/\sigma^2$ has a chi-square distribution with $n-1$ degrees of freedom. To determine a $100(1-\alpha)\%$ confidence interval for σ^2, we follow the procedure for constructing Equation (3.9) to find the numbers $x^2_{n-1;1-\alpha/2}$ and $x^2_{n-1;\alpha/2}$ from the chi-square distribution table. The confidence interval is then given by

$$\frac{(n-1)S^2}{x^2_{n-1;\alpha/2}} < \sigma^2 < \frac{(n-1)S^2}{x^2_{n-1;1-\alpha/2}} . \tag{2.10}$$

Similar to Equation (2.8), our experience shows that this equation is not restricted to the normal distribution when the sample size is reasonably large (15 or more).

If X is exponentially distributed, Equation (2.9) can be used to estimate the confidence interval for σ^2, because for the exponential random variable, σ^2 equals μ^2. Since all terms in Equation (2.9) are positive, we can take square for them. The result gives a $100(1-\alpha)\%$ confidence interval for σ^2:

$$\left(\frac{2n\bar{X}}{x^2_{2n;\alpha/2}}\right)^2 < \sigma^2 < \left(\frac{2n\bar{X}}{x^2_{2n;1-\alpha/2}}\right)^2 . \tag{2.11}$$

C. Confidence Intervals for Proportions

Often, we need to estimate the confidence interval for a proportion or percentage whose underlying distribution is unknown. For example, we may want to estimate the confidence interval for the detection coverage after fault injection experiments. In general, given n Bernoulli trials with the probability of success on each trial being p and the number of successes being Y, how do we find a confidence interval for p? If n is large (particularly when $np \geq 5$ and $n(1-p) \geq 5$ [Hogg83]), Y/n has an approximately normal distribution, $N(\mu, \sigma^2)$, with $\mu = p$ and $\sigma^2 = p(1-p)/n$. Note that Y/n is the sample mean which is an estimate of μ or p. By Eq. (2.7), the 100(1-α)% confidence interval for p is

$$\frac{Y}{n} \pm z_{\alpha/2}\sqrt{p(1-p)/n}\,. \tag{2.12}$$

This equation can be used to determine the number of injections required to achieve a given confidence interval for an estimated fault detection coverage. Let n represent the number of fault injections and Y the number of faults detected in the n injections. Assume that all faults have the same detection coverage, which is approximately p. Now we wish to estimate p with the 100(1-α)% confidence interval being e. By Eq. (2.12), we have

$$e = z_{\alpha/2}\sqrt{p(1-p)/n}\,. \tag{2.13}$$

Solving the equation for n:

$$n = \frac{z_{\alpha/2}{}^2\, p(1-p)}{e^2}, \tag{2.14}$$

where n is the number of injections required to achieve the desired confidence interval in estimating p.

For example, assume detection coverage $p = 0.6$, confidence interval $e = 0.05$, and confidence coefficient $1 - \alpha = 90\%$. Then the required number of injections is

$$n = \frac{1.645^2 \times 0.6 \times 0.4}{0.05^2} = 260$$

2.2. Distribution Characterization

Mean and variance are important parameters that summarize data by single numbers. Probability distribution provides more information about data. Analysis of distributions can help one understand data in detail as well as the underlying models. For example, if the waiting times in all states of a transition model are exponential, then the model is a Markov model. Otherwise, it is a semi-Markov model. We will discuss empirical distribution functions and function fitting in this subsection.

2.2.1. Empirical Distribution

Given a sample of X, the simplest way to obtain an empirical distribution of X is to plot a histogram of the observations. The range of the sample space is divided into a number of subranges called *buckets*. The lengths of the buckets do not have to be the same. Assume that we have k buckets, separated by x_0, x_1, ..., x_k, for the given sample with the size of n. In each bucket, there are y_i instances. Obviously, the sample size n is $\sum_{i=1}^{k} y_i$. Then, y_i/n is an estimation of the probability that X takes a value in bucket i. We will call the histogram an *empirical probability distribution function* (p.d.f.) of X. It is easy to construct the following *empirical cumulative distribution function* (c.d.f.) from the histogram.

$$F_k(x) = \begin{cases} 0, & x < x_0 \\ \sum_{l=1}^{i} \frac{y_l}{n}, & x_{i-1} \le x < x_i \\ 1, & x_k \le x \end{cases} \tag{2.15}$$

The key problem in plotting histograms is determining the bucket size. A small size may lead to a large variation among buckets so that the characterization of the distribution cannot be identified. A large size may lose details of the distribution. Given a data set, it is possible to obtain very different distribution shapes by using different bucket sizes. One guideline is that if any bucket has less than five instances, the bucket size should be increased or a variable bucket size should be used. By our experience, 10 to 50 buckets are appropriate in most cases, depending on the sample size. We will call the histogram constructed from data the *empirical distribution.*

2.2.2. Function Fitting

Analytical distribution functions are useful in analytical modeling and simulations. Thus, it is often desirable to fit an analytical function to a given empirical distribution. Function fitting is not a trivial task and relies on certain knowledge of statistical distribution functions. The procedure given in the following is based on our experience. Given an empirical distribution, the first step is to make a good guess of the closest distribution function(s) by observing the shape of the empirical distribution. The second step is to use a statistical package such as SAS to obtain the parameters for a guessed function by trying to fit it to the empirical distribution. The third step is to perform a significance test of the goodness-of-fit to see if the fitted function is acceptable. If the function is not acceptable, we have to go to step 2 to try a different function.

Now we discuss step 3 — significance test. Assume that the given empirical c.d.f. is F_k, defined in Eq (2.15), and the hypothesized c.d.f. is $F(x)$ (obtained from step 2 in the above). Our task is to test the hypothesis

$$H_0\text{: } F_k(x) = F(x).$$

There are two commonly used goodness-of-fit test methods: the *chi-square test* and the *Kolmogorov-Smirnov test*. We now briefly introduce the two methods.

A. *Chi-Square Test*

The chi-square test assumes the distribution under consideration can be approximated by a multinomial distribution, which usually stands. Let

$$p_i = F(x_i) - F(x_{i-1}) \,, \quad i = 1, \ldots, k$$

where p_i is the probability that an instance falls into bucket i. If we define

$$P[x_{i-1} \le X_i < x_i] = p_i \,, \quad i = 1, \ldots, k \,,$$

then X_1, X_2, ..., X_k have a multinomial distribution which is equivalent to the original distribution $F(x)$. Thus, for a sample size of n, the expected instances falling into bucket i is np_i, by the above distribution. The sum of error squares divided by the expected numbers

$$q_{k-1} = \sum_{i=1}^{k} \frac{(y_i - np_i)^2}{np_i} \tag{2.16}$$

is a measure of the "closeness" of the observed number of instances, y_i, to the expected number of instances, np_i, in bucket i. If q_{k-1} is small, we tend to accept H_0. The "smallness" can be measured in terms of statistical significance if we treat q_{k-1} as a particular value of the random variable Q_{k-1}. It can be shown that if n is large ($np_i \ge 1$), Q_{k-1} has an approximate chi-square distribution with $k - 1$ degrees of freedom, $\chi^2(k-1)$. If H_0 is true, we expect that q_{k-1} falls into an acceptable range of Q_{k-1} so that the event is likely to occur. The boundary value, or critical value, of the acceptable range, $\chi^2_\alpha(k-1)$ is chosen such that

$$P[Q_{k-1} > \chi^2_\alpha(k-1)] = \alpha$$

where α is called the *significance level* of the test. Thus, we should reject H_0 if $q_{k-1} > \chi^2_\alpha(k-1)$. Usually, α is chosen to be 0.05 or 0.1.

B. *Kolmogorov-Smirnov Test*

The Kolmogorov-Smirnov test is a non-parametric method in that it assumes no particular distribution for the variable in consideration. The method uses the empirical c.d.f., instead of the empirical p.d.f., to perform the test, which is more stringent than the chi-square test. The Kolmogorov-Smirnov statistic is defined by

$$D_k = sup_x[|F_k(x) - F(x)|] , \tag{2.17}$$

where sup_x represents the least upper bound of all pointwise differences $|F_k(x) - F(x)|$. In calculation, we can choose the midpoint between x_{i-1} and x_i, for $i = 1, \ldots, k$, to obtain the maximum value of $|F_k(x_i) - F(x_i)|$. It is seen that D_k is a measure of the closeness of the empirical and hypothesized distribution functions. It can be derived that D_k submits to a distribution whose c.d.f. values are given by the table of Kolmogorov-Smirnov Acceptance Limits [Hogg83]. Thus, given a significance level α, we can find the critical value d_k from the table such that

$$P[D_k > d_k] = \alpha \ .$$

The hypothesis H_0 is rejected if the calculated value of D_k is greater than the critical value d_k. Otherwise, we accept H_0.

2.3. Multivariate Analysis

In reality, measurements are usually made on more than one variable. For example, a computer workload measurement may include usages on CPU, memory, disk, and network. A computer failure measurement may collect data on multiple components. Multivariate analysis is the application of methods that deal with multiple variables. These methods, including clustering analysis, correlation analysis, and factor analysis to be discussed, identify and quantify simultaneous relationships among multiple variables.

2.3.1. Clustering Analysis

Clustering analysis is useful for characterizing workload states in computer systems by clustering similar points in resource usage. Assume we have a sample of p variables with a size of n. We call each instance in the sample a *point*, which consists of p values. Clustering analysis identifies similar points and clusters them into groups (clusters). Let $x_i = (x_{i1}, x_{i2}, \ldots, x_{ip})$ denote the ith point of the sample. The Euclidean distance between points i and j,

$$d_{ij} = \mid x_i - x_j \mid \; = (\sum_{l=1}^{p}(x_{il} - x_{jl})^2)^{1/2}$$

is usually used as a similarity measure between points i and j.

There are several different clustering algorithms. The goal of these algorithms is to achieve small *within-cluster* variation relative to the *between-cluster* variation. A commonly used algorithm is the *k-means* clustering algorithm. The algorithm partitions a sample with p dimensions and n points into k clusters, C_1, C_2, ..., C_k. Denote the mean, or centroid of the C_j by $\bar{x}_j$. The error component of the partion is defined as

$$E = \sum_{j=1}^{k} \sum_{i \in C_j} \mid x_i - \bar{x}_j \mid ^2 . \qquad (2.18)$$

The goal of the k-means algorithm is to find a partition that minimizes E.

The clustering procedure is as follows: Start with k groups each of which consists of a single point. Each new object is added to the group with the closest centriod. After a point is added to a group, the mean of that group is adjusted to take the new point into account. After a partition is formed, search for another partition with smaller E by moving points from one cluster to another cluster until no transfer of a point results in a reduction in E.

2.3.2. Correlation Analysis

Correlation analysis can be used to quantify error or workload dependency between two components in a system. The correlation coefficient, $Cor(X_1, X_2)$, between the random variables X_1 and X_2 is defined as

$$Cor(X_1, X_2) = \frac{E[(X_1 - \mu_1)(X_2 - \mu_2)]}{\sigma_1 \sigma_2} \qquad (2.19)$$

where μ_1 and μ_2 are the means of X_1 and X_2, and σ_1 and σ_2 the standard deviations of X_1 and X_2, respectively. If we use ρ to denote the correlation coefficient, then ρ satisfies $-1 \le \rho \le 1$. The correlation coefficient is a measure of the linear relationship between two variables. When $|\rho| = 1$, we have $X_1 = aX_2 + b$, where b>0 if $\rho = 1$, or b<0 if $\rho = -1$. In this extreme case, there is an exact linear relationship between X_1 and X_2. When $|\rho| \neq 1$, there is no exact linear relationship between X_1 and X_2. In this case, ρ measures the goodness of the linear relationship $X_1 = aX_2 + b$ between X_1 and X_2. Usually, a ρ value of 0.5 or above is considered reasonably high.

Given random variables, X_1, X_2, and X_3, and correlation coefficients between each pair, ρ_{12}, ρ_{23}, and ρ_{13}, we know these variables are related each other by ρ_{12}, ρ_{23}, and ρ_{13}. Since X_1 is related to X_2 and X_2 is related to X_3, a partial dependence between X_1 and X_3 may be due to X_2. The partial correlation coefficient defined below quantifies this partial dependence.

$$\rho_{13.2} = \frac{\rho_{13} - \rho_{12}\rho_{23}}{\sqrt{(1 - \rho^2{}_{12})(1 - \rho^2{}_{23})}} \tag{2.20}$$

Partial correlation coefficient can be considered as a measure of the common relationship among the three variables.

If a random variable, X, is defined on time series, the correlation coefficient can be used to quantify the time serial dependence in the sample data of X. Given a time window $\Delta t > 0$, the *autocorrelation coefficient* of X on the time series t is defined as

$$Autocor(X, \Delta t) = Cor(X(t),\ X(t + \Delta t))\ , \tag{2.21}$$

where t is defined on the discrete values (Δt, $2\Delta t$, $3\Delta t$, ...). In this case, we treat $X(t)$ and $X(t + \Delta t)$ as two different random variables and the autocorrelation coefficient is actually the correlation coefficient between the two variables. That is, $Autocor(X, \Delta t)$ measures the time serial correlation of X with a window Δt.

2.3.3. Factor Analysis

The limitation of correlation analysis is that the correlation coefficient can only quantify dependency between two variables. However, dependency may exist within a group of more than two variables or even among all variables. The correlation coefficient cannot provide information about this multiple dependency. *Factor analysis* is one of statistical techniques to quantify multi-way dependency among variables. The method attempts to find a set of unobserved common factors which link together the observed variables. Consequently, it provides insight into the underlying structure of the data. For example, in a distributed system, a disk crash can account for failures on those machines whose operations depend on a set of critical data on the disk. The disk state can be considered to be a common factor for failures on these machines.

Let $X = (x_1, \ldots, x_p)$ be a normalized random vector. We say that the k-factor model holds for X if X can be written in the form

$$X = \Lambda F + E \tag{2.22}$$

where $\Lambda = (\lambda_{ij})$ $(i = 1, \ldots, p;\ j = 1, \ldots, k)$ is a matrix of constants called *factor loadings*, and $F = (f_1, \ldots, f_k)$ and $E = (e_1, \ldots, e_k)$ are random vectors. The elements of F are called *common factors*, and the elements of E are called *unique factors* (error terms). These factors are unobservable variables. It is assumed that all factors (both common and unique factors) are independent of each other and that the common factors are normalized.

Each variable x_i $(i = 1, \ldots, p)$, can then be expressed as

$$x_i = \sum_{j=1}^{k} \lambda_{ij} f_j + e_i$$

and its variance can be written as

$$\sigma_i^2 = \sum_{j=1}^{k} \lambda_{ij}^2 + \psi_i$$

where ψ_i is the variance of e_i. Thus, the variance of x_i can be split into two parts. The first part

$$h_i^2 = \sum_{j=1}^{k} \lambda_{ij}^2$$

is called the *communality* and represents the variance of x_i which is shared with the other variables via the common factors. In particular $\lambda_{ij} = Cor(x_i, f_j)$ represents the extent to which x_i depends on the jth common factors. The second part, ψ_i, is called the *unique variance* and is due to the unique factor e_i; it explains the variability in x_i not shared with the other variables.

2.4. Importance Sampling

Importance sampling is a statistical method to reduce sampling size while keeping estimates obtained from the sample at a high level of confidence [Kahn53]. The method has been recently used to reduce the number of runs in Monte Carlo simulations for evaluating computer dependability [Goyal92] [Choi92]. In the following, we first give an overview of the method and then discuss its applications in the Monte Carlo simulation of discrete-time Markov chains (DTMC's).

2.4.1. Overview of the Method

Assume that a random variable X has p.d.f. $f(x)$ and that $Y = h(X)$ is a function of X. Our goal is to estimate the expected value of Y,

$$\theta = E[Y] = E[h(X)] = \int_{-\infty}^{+\infty} h(x)f(x)dx \,, \tag{2.23}$$

through sampling. That is, we generate a sample $\{x_1, x_2, \ldots, x_n\}$ according to $f(x)$, therefore generating $\{y_1, y_2, \ldots, y_n\}$, and then calculate

$$\tilde{\theta} = \overline{Y} = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{1}{n}\sum_{i=1}^{n} h(x_i) \,.$$

It may be very expensive to generate a statistically significant sample of X. For example, if $y_i = h(x_i) = 0$ for most generated x_i, we may need an extremely large size of sample to estimate θ with a high level of confidence. However, if we can make the rare x_i's which are "important" for estimating θ be much more frequently selected in sampling while keeping the estimate unbiased, the sample size will be greatly reduced. This is the basic idea of the importance sampling method.

To do importance sampling, we change the p.d.f. of X from $f(x)$ to $g(x)$ such that those x's which are of importance in our parameter estimation have higher occurrence probabilities in $g(x)$. We use X' to represent the variable which has p.d.f. $g(x')$. By Eq. (2.23), we have

$$\theta = \int_{-\infty}^{+\infty} h(x)f(x)dx = \int_{-\infty}^{+\infty} h(x)\frac{f(x)}{g(x)}g(x)dx = \int_{-\infty}^{+\infty} h(x)\Lambda(x)g(x)dx \,, \tag{2.24}$$

where

$$\Lambda(x) = \frac{f(x)}{g(x)} \tag{2.25}$$

is called *likelihood ratio*. Let $Y' = h(X)\Lambda(X)$, then Eq. (2.24) becomes

$$\theta = \int_{-\infty}^{+\infty} y'g(x)dx = E[Y'] \,. \tag{2.26}$$

Thus, instead of sampling from $f(x)$ to estimate the expected value of Y, the experiment is changed to sampling from $g(x')$ to estimate the expected value of Y'. That is, we generate a sample $\{x_1', x_2', \ldots, x_n'\}$ according to $g(x')$, therefore generating $\{y_1', y_2', \ldots, y_n'\}$, and then calculate

$$\tilde{\theta} = \overline{Y'} = \frac{1}{n}\sum_{i=1}^{n} y'_i = \frac{1}{n}\sum_{i=1}^{n} h(x'_i)\Lambda(x'_i) \ .$$

The variance of the above estimator is

$$Var(Y') = E[(Y' - \theta)^2] = \int_{-\infty}^{+\infty} \frac{h(x)f(x)^2}{g(x)} \ g(x)dx - \theta^2 \ .$$

To achieve the minimum variance, we should have

$$g(x) = \frac{h(x)f(x)}{\theta} \ .$$

But θ is the unknown parameter to estimate. A heuristic is that the shape of $g(x)$ should follow the shape of $h(x)f(x)$ as closely as possible.

2.4.2. Applications in DTMC Simulation

In many cases, the operation of a computer system can be modeled by a DTMC (Discrete Time Markov Chain) [Trivedi82]. If the built DTMC is very large (such that it exceeds the available storage) or the functional simulation (simulation of the execution of machine instructions, algorithms, etc.) is used above a DTMC, the Monte Carlo simulation method is perhaps the only feasible way to solve the model. In dependability models, system failures are usually rare events with extremely small probabilities. In order to obtain statistically significant results, large simulation runs are required, which can be very time consuming. In such a case, importance sampling can be used to reduce simulation runs, usually by orders of magnitude.

Assume we have a DTMC $\{Y_s, s \geq 0\}$ with a set of states $\{S_1, S_2, \ldots, S_m\}$ and a transition matrix $[p_{ij}]$. For each simulation run, we have a path $x_i = S_{i_0}, S_{i_1}, \ldots, S_{i_k}$. The occurrence probability of path x_i is [Goyal92]

$$P(x_i) = p_{i_0} p_{i_0 i_1} \cdots p_{i_{k-1} i_k} \ ,$$

where each p_{ij} is an element in $[p_{ij}]$. All possible paths constitute the probability space of a random variable: $X = \{x_1, x_2, x_3, \cdots\}$.

To reduce simulation runs, we change the transition probability matrix from $[p_{ij}]$ to $[p'_{ij}]$ such that those paths which are of importance in our dependability evaluation are more likely to be sampled. After the change, the occurrence probability of path x_i is

$$P'(x_i) = p'_{i_0} p'_{i_0 i_1} \cdots p'_{i_{k-1} i_k} .$$

Assume the dependability measure to evaluate is $\theta = E[h(X)]$. Then θ can be estimated using a sample, $\{x_1, x_2, \ldots, x_n\}$, obtained from simulations by

$$\tilde{\theta} = \frac{1}{n} \sum_{i=1}^{n} h(x_i) \Lambda(x_i) , \tag{2.27}$$

where

$$\Lambda(x_i) = \frac{P(x_i)}{P'(x_i)} = \frac{p_{i_0} p_{i_0 i_1} \cdots p_{i_{k-1} i_k}}{p'_{i_0} p'_{i_0 i_1} \cdots p'_{i_{k-1} i_k}} . \tag{2.28}$$

The remaining question is how to determine $[p'_{ij}]$. Several heuristics called *failure biasing* have been proposed in the literature [Lewis84] [Goyal92]. Here we introduce one of the commonly used heuristics. Assume that in state S_i, transitions out of the state go to either a set of failure states, F (e.g., the system suffers one more component failure), or a set of recovery states, R (e.g., the system recovers from a component failure). (S_i itself can be treated as either in F or in R.) It is obvious that we have

$$\sum_{j \in F} p_{ij} + \sum_{j \in R} p_{ij} = 1 .$$

Define a parameter b such that p'_{ij}'s satisfy

$$\sum_{j \in F} p'_{ij} = b , \quad \sum_{j \in R} p'_{ij} = 1 - b . \tag{2.29}$$

Then we determine each p'_{ij} in state S_i by

$$p'_{ij} = \begin{cases} b \dfrac{p_{ij}}{\sum_{k \in F} p_{ik}} & j \in F \\ (1-b) \dfrac{p_{ij}}{\sum_{k \in R} p_{ik}} & j \in R \end{cases} \tag{2.30}$$

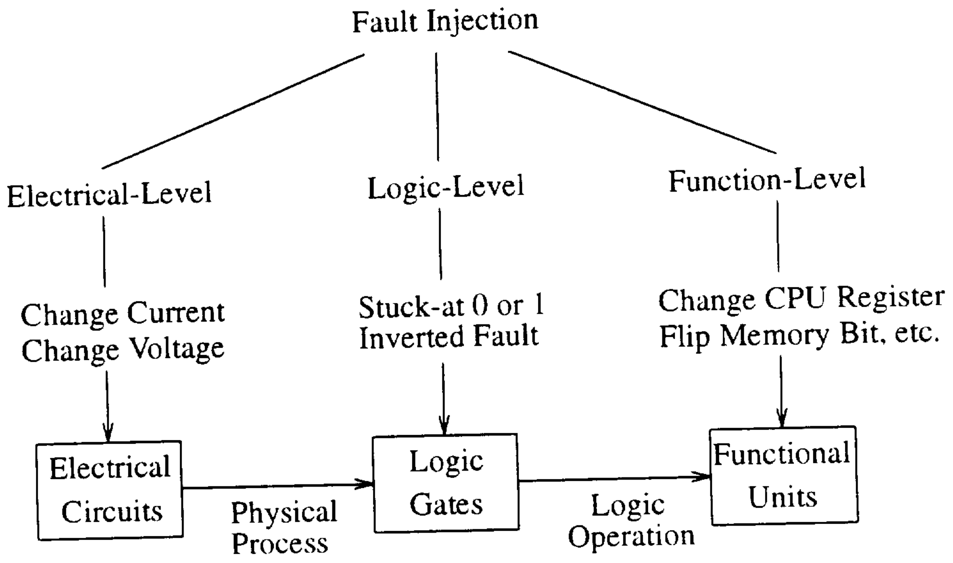
The parameter b is usually chosen to be 0.5 [Goyal92]. Since the sum of the original probabilities to failure states is often very small, by Eq. (2.29), the selection of b can significantly increase these probabilities, thus making these transitions much more likely to occur in simulations.

III. DESIGN PHASE

In the early design phase of highly reliable systems, simulation is an important experimental means for performance and dependability analysis. Compared to analytical modeling, simulation has the capability to model complex systems in detail without being restricted to assumptions made in analytical modeling to keep the model mathematically tractable. Thus, simulation is able to provide more accurate dependability evaluation than analytical models. Simulations for dependability analysis can be performed by injecting faults in the system under study at the electrical level, the logic level, and the function level. Dependability issues studied usually include but are not limited to: 1) fault propagation, 2) fault latency, and 3) fault impact such as coverage, reliability, availability, and performance loss. Figure 3.1 shows fault injections at the different levels.

Electrical-level fault injection simulation is usually used to emulate transient faults by changing the electric current and voltage inside the circuits. The faulty current and voltage may cause errors in logic values at the gate level. The gate-level errors may then propagate to other functional units and output pins of the chip. It has been reported that transient faults account for more than 80% of the failures in computer systems [Siewiorek78], [Iyer86]. These faults result from physical causes such as power transients, capacitive or inductive crosstalk, or cosmic particle interventions [Yang92]. Electrical-level simulation can be used to study the impact of transient faults from the physical

Figure 3.1. Simulated Fault Injections at Different Levels



level, but since the simulation has to track the propagation of faults from circuits to gates, to functional units, and eventually out to the pins, it can be very time consuming and memory bound.

For this reason, logic-level fault injection simulation applies abstractions of physical fault models to logic gates to study large VLSI, even computer systems. Commonly used fault models include *stuck-at-0*, *stuck-at-1*, and *inverted fault*. These models are considered to be representative of faults at the gate level. Although simulation at the logic level ignores the physical processes underlying gate faults, it still needs to trace the impact of gate-level faults to higher levels. For the same reason that electrical-level simulation cannot be effectively used to study large VLSI systems, logic-level simulation cannot effectively study large computer systems.

Function-level fault injection simulation is usually used to study dependability features of large computer or network systems. Faults are injected into various components of the system under study. Functional fault models are used in the simulation, while detailed processes of fault occurrence at lower levels are ignored. Functional models represent the manifestation of faults at the lower levels and are extracted from results obtained from electrical-level or logic-level fault injections or from field measurements. For example, "flipped memory bit" and "CPU register error" are two typical fault models. Analytical dependability models of computer systems are usually built at this level. Compared to analytical models, simulation is capable of representing detailed architectural features, real fault conditions, and inter-component dependencies, thereby providing more accurate and believable results.

There are several common issues for fault injections at all levels. The first issue is that given fault models (e.g., one bit flip in memory) and types (e.g., transient fault), where do we inject faults? A simple way is to randomly choose a location from the injection space (e.g., all gates in a VLSI chip or all memory bits). This scheme is easy to implement, but many faults may have similar impact (e.g., all faulty bits in an ALU may have the same effect) and many faulty locations may not be exercised. Another way is to inject faults only to representative locations which have different impact, or only to representative workload areas. This approach can be used to study fault impact in terms of locations or workloads.

The second issue involves workloads. The impact of faults on system dependability is workload-dependent. Hence it is important to analyze a system while it is executing representative workloads. These workloads can be real applications, selected benchmarks, or synthetic programs. If the goal of study is to investigate fault impact on a mission task, the real applications running in the mission may be used in the simulation. If the research goal is to study fault impact on general workloads, several representative benchmarks may be selected for the simulation. If we want to exercise every functional unit and location in the simulation, both real applications and benchmarks may not be appropriate. In this case, synthetic workloads can be designed for achieving the goal. The workload issue further complicates simulation models and increases simulation time. It is necessary to develop ways to represent realistic workloads while still maintaining reasonable simulation times.

The third issue is simulation time explosion which occurs when: 1) too much detail is simulated such as modeling physical processes in fault injections at the electrical level, and 2) extremely small failure probabilities require large simulation runs to obtain statistically significant results (the theory is discussed in Section 2.1). Several techniques, including mix-mode simulation [Saleh90] [Choi92], importance sampling [Goyal92] [Choi92], hybrid simulation [Bavuso87], [Goswami93a], and hierarchical simulation [Goswami92] have been used to tackle the time explosion problem.

Table 3.1 summarizes features and representative studies in simulated fault injections at different levels. We will discuss these studies in the following three sections.

Table 3.1. Summary of Simulated Fault Injections

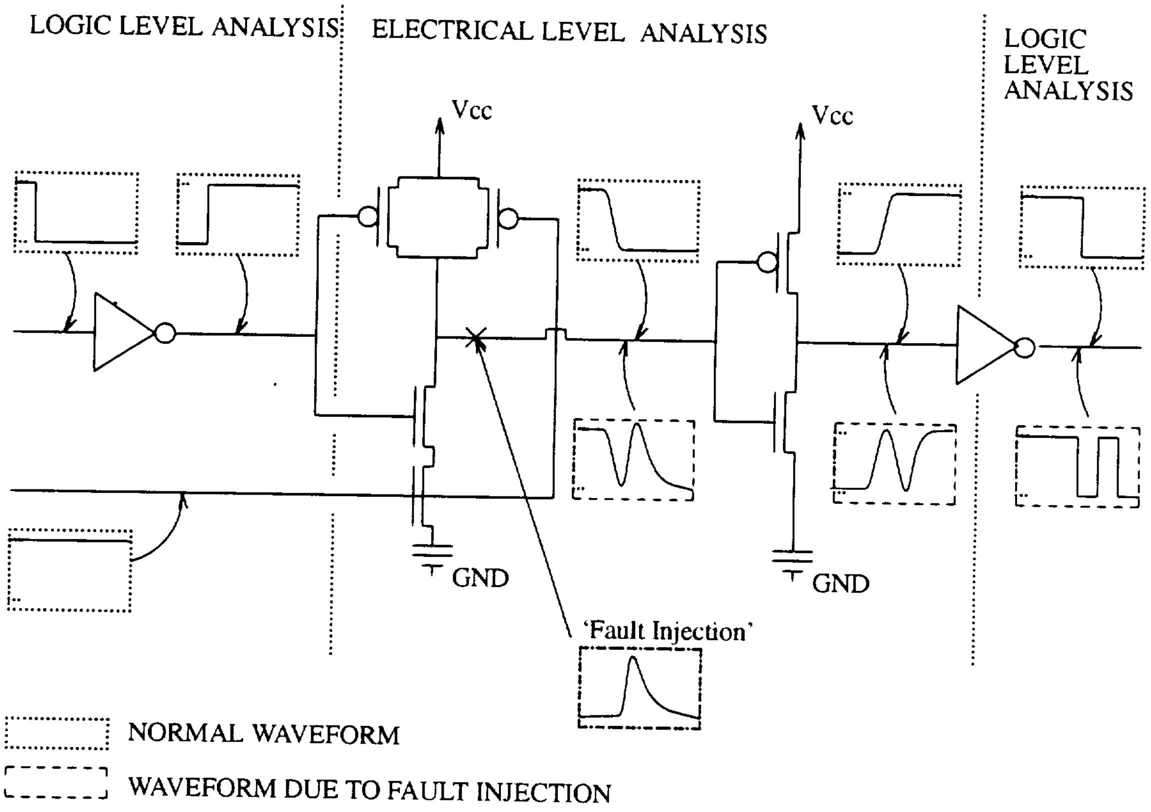
Category	Electrical Level	Logic Level	Function Level
Approach	Alter electrical current and voltage in circuits	Inject stuck-at or inverted faults to logic gates	Inject faults to CPU, memory, I/O devices, etc.
Target Under Study	VLSI chip Software running on the chip	VLSI chip Computer system Software	Computer system Network system Software
Studies	Fault simulation [Yang92] HA1602 [Duba88] FOCUS [Choi92]	BDX930 [McGough81] BDX930 [Lomelino86] IBM RT PC [Czeck91]	Trace-driven [Chillarege87] NEST [Dupuy90] DEPEND [Goswami92] REACT [Clark93]

3.1. Simulated Fault Injection at the Electrical Level

There are several reasons for performing fault injections at the electrical level. First, the fault injection at this level can be used to study the impact of physical causes which lead to faults and errors. Secondly, it has been pointed out by previous studies [Banerjee82], [Beh82] that simple stuck-at fault models do not represent some types of faults. Thirdly, some circuits are of a mixed analog/digital nature which cannot be fully characterized by logic-level fault models. Thus, there is a growing need for fault simulators which can handle electrical transient faults and permanent physical failures for the purposes of both circuit testing and dependability evaluation.

The basic simulation methodology used in fault injections at the electrical level is the *mixed-mode* method in which the fault-free portions of the circuit are simulated at the logic level while the faulty portions of the circuit are simulated at the electrical level [Saleh90]. Figure 3.2 illustrates the method. A simple CMOS AND gate with buffered output is drawn in the figure. The dotted boxes indicates normal voltage waveforms for the circuit and the dashed boxes contain waveforms resulting from a transient injection at the location marked by X. Notice that waveforms within the electrical-level analysis behave in an analog fashion, but are discrete in the logic-level analysis.

Figure 3.2. Illustration of Fault Injection at Electrical Level



A representative mixed-mode fault simulator is SPLICE1 [Saleh84]. The electrical analysis in SPLICE1 is based on the method of iterated timing analysis (ITA) which incorporates a nonlinear relaxation method with event-driven selective tracing. ITA has been shown to be accurate and fast (can provide a speed-up of up to two orders of magnitude). The logic analysis in SPLICE1 is performed using a relaxation-based method including MOS-oriented models. Recently, more advanced techniques, such as the concurrent mixed-mode simulation of permanent faults and the dynamic mixed-mode simulation of transient faults have been developed [Yang92].

We now discuss two studies in the electrical-level fault injection. Both use SPLICE1 as the fault simulation engine. The first is a case study of the impact of different levels of current transients on a microprocessor-based chip. The second is a fault injection tool which integrates fault injection engine, tracing facility, and graphical and statistical analysis packages into a user environment.

3.1.1. Simulation of a Microprocessor-Based Chip

One of the studies in this field was an experimental analysis of susceptibility of a microprocessor-based jet engine controller to upsets caused by current and voltage transients through simulated fault injections [Duba88]. The target system for the study was an HA1602, a microprocessor-based digital jet-engine controller designed by Hamilton Standard for commercial aircraft and made available to NASA Langley AIRLAB. SPLICE1 was chosen for the fault simulation in the study. A number of enhancements to SPLICE1 were made to facilitate the fault injection simulations.

The parameters used in the simulations were extracted from those used in the HA1602 design and circuit layout. The application code running on the simulated processor was chosen such that all the functional units at which transient fault injections were made were exercised. Fault injections were made at seven randomly chosen nodes in six functional units. For each node, current transients were injected at five different charge levels: 0.5, 1.0, 2.0, 3.0, and 4.0 pico Coulombs. Each charge level was injected at five different time-points during the execution of the application code sequence. This amounted to over 1000 fault injections/simulations.

The error data was generated by comparing each faulted simulation with a fault-free simulation. An error event was defined as either a logic state change or a voltage level change large enough to cause a node to be faulted at a

Table 3.2. Severity of Injected Transient Faults

Error Category	Occurrences	Percentage	Charge Threshold
Injected Transients	1050	100.0	3.0 pC
Logic Upsets	437	41.6	3.0 pC
Latched Error	60	5.7	3.0 pC
Pin Errors	59	5.6	3.0 pC

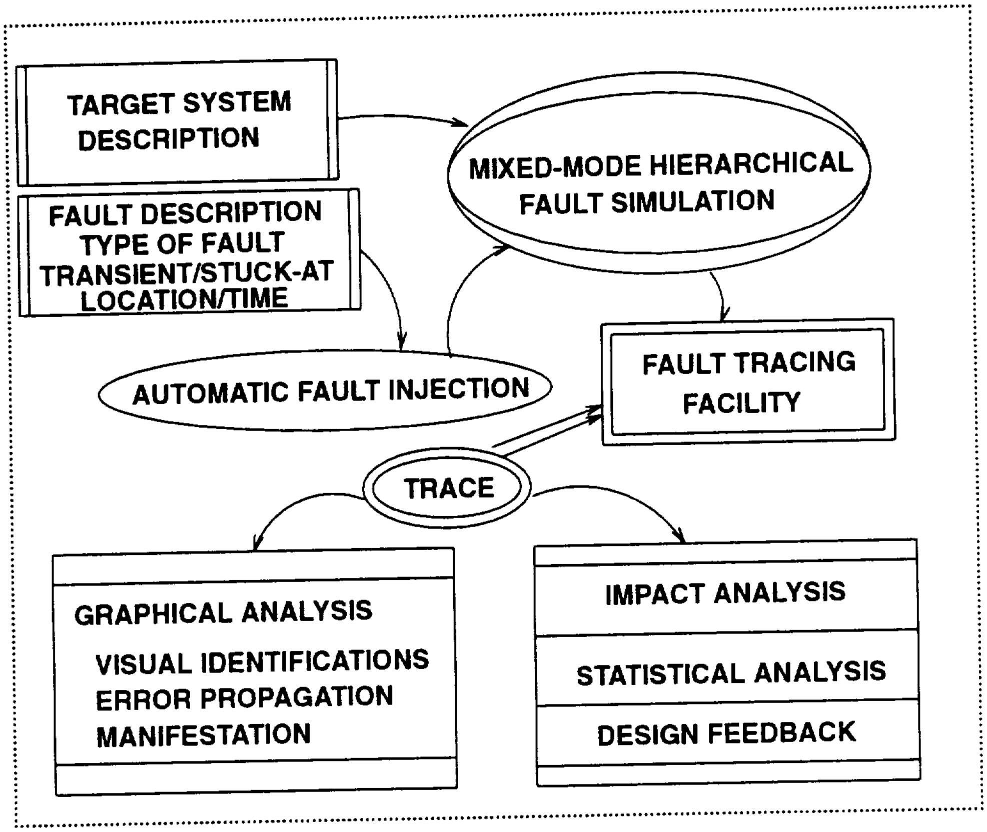
future time. Error events were classified as three categories: 1) logic upsets — voltage transients large enough to constitute logic level errors, 2) latched errors — errors in the first-level latches, and 3) pin errors — errors at the chip I/O pins. The overall results from the experiments are shown in Table 3.2. It can be seen that the injected transients have a 41.6% chance of causing a logic upset (no errors), a 5.7% chance of resulting in a latched error (a latent error in the circuit), and 5.6% chance of error propagating to pins. The other 47% of the injected transients have no impact on the processor. Thus, only 11% of all injected transients cause either a permanent change in circuit behavior or affect the external environment. The table shows that transients below 3.0 pico-Coulombs have no significant impact on the circuit.

The study also investigated the impact of current and voltage transients occurring in the different functional units of the processor. An ALU transient was found to most likely result in logic upsets and pin errors. Further, the analysis of variance (ANOVA) technique was used to quantify the sensitivity of pin-level errors to error activity in the different functional units. The results of ANOVA are shown in Table 3.2, which indicate that the output pin errors are most sensitive to error activity in ALU.

3.1.2. FOCUS — A Chip-Level Simulation Environment

FOCUS is a simulation environment, developed at University of Illinois, for fault sensitivity analysis of IC chips [Choi92]. In the environment, a range of user-specified faults are automatically injected at the circuit level, and fault propagation is measured at the gate and higher levels. Figure 3.3 depicts the overall experimental environment. The environment takes as input a net-list of the hardware description of the system and converts it into a simulation model. SPLICE1 is used as the fault simulation engine. The importance sampling technique, which has been introduced in Section 2.4, is used in FOCUS to accelerate simulations.

Figure 3.3. FOCUS Experimental Environment



The *fault injection* process is implemented as a run-time modification of the circuit, whereby a current source is added to a target node,[1] thus altering the voltage level of the node over the time interval of the injected current waveform. The experimental environment allows both transient and permanent (single or multiple) fault injections. Since the injected current source is specified as a mathematical function, the resulting transients can be of varying shapes and duration. For example, electrical power surge, in-chip alpha particle intervention, lightning, and bridging faults can be modeled. The user can control the location of a fault, the time and duration of a fault, and the shape of the current source.

The *tracing facility* monitors all switching activities in the target system, including fault propagation through each gate or transistor, for all processed events. The trace data for each event consists of the time of the event, the hierarchical node name, and the new and previous voltage levels (for electrical nodes) or logic levels (for logic nodes).

[1] A node is defined as a point in a conductive interconnection between electrical and/or logical elements.

The *graphical analysis facility* is used to visualize the error activity in different functional units of the processor and the fault propagation on the major interconnects and at the external pins. The *statistical analysis tools* provide impact analysis of the target system and generate necessary models to depict the fault behavior in the system (e.g., I/O pin error distribution, latch error distribution, and internal fault propagation model).

The application of FOCUS is illustrated by studying a target system. The target system is a microprocessor used in commercial aircraft for real-time control of jet-engine functions. The 16-bit microprocessor consists of six major function units: ALU, control, decoder, multiplexer, countdown, and watchdog, as shown in Figure 3.4. The system incorporates a variety of fault tolerant design features at different levels, including software checks, parity checks, memory test, and error counting. The objective of the study is to investigate the impact of charge-level transients on latch, pin, and functional levels.

Nearly 80 instruction cycles (90300 nanoseconds) of the application code were executed on the target system during each simulation run. The application code was carefully selected to ensure that all of the functional units were

Figure 3.4. The Target Microprocessor System

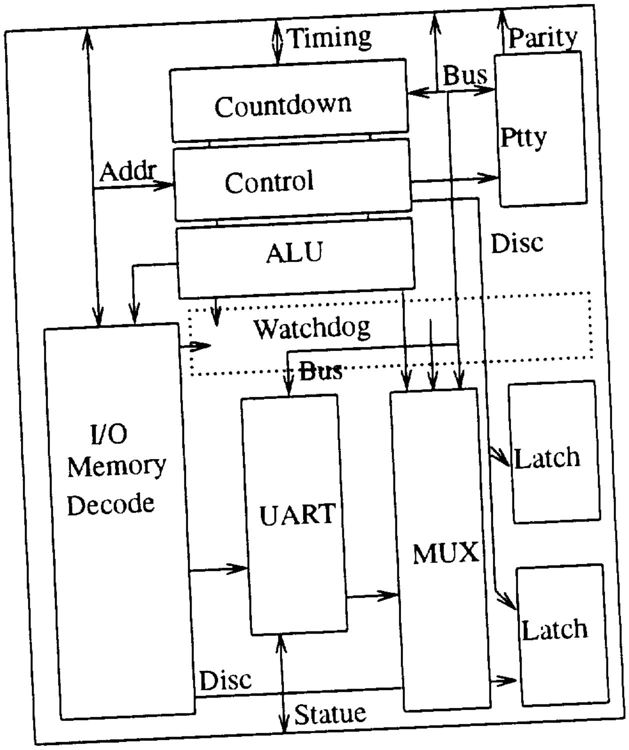


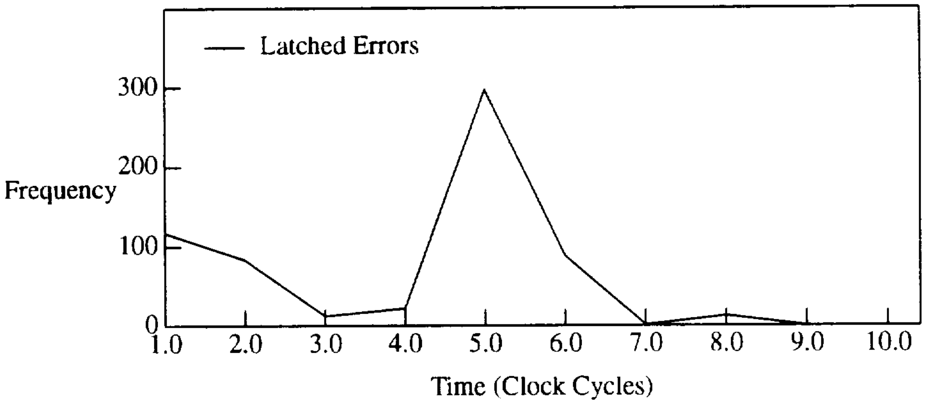
Table 3.3. Impact of Transients Injected to the Target System

Type	Injections Involved	Percent of Total Injections	Resultant Errors
First-Order Latch Errors	470	22.4	2149
Second and Higher-Order Latch Errors	120	5.7	1829
First-Order Pin Errors	255	12.1	1168
Second and Higher-Order Pin Errors	90	4.3	839
Functional Errors	193	9.2	747

executed. A total of 2100 simulations were performed for obtaining stable results. During the simulation, all nodes (including all latches and external I/O pins) in the circuit were monitored and processed. Table 3.3 summarizes the overall impact of transients in the range 0.5 to 9.0 picoCoulombs. In the table, a first-order error is defined as one which occurs during the first clock cycle following a transient fault injection; second and higher-order errors are those occurring during the second and subsequent clock cycles.

Figure 3.5 shows the propagation of the latch errors in time. In the figure, the x-axis represents the clock cycles from the fault injection time, and the y-axis represents the total latch error count for each clock cycle. It can be seen that, given a certain number of latch errors in the first clock cycle, the number of latch errors degenerates significantly until the fourth clock cycle. At approximately the fifth clock cycle, the number of errors rapidly multiplies. This is because at this time, the error signal enters a unit with a large number of latches and high fan-out, e.g., the ALU registers. After the sixth cycle, the number of errors degenerates significantly until finally disappearing after the eighth cycle. Thus, the impact of latch errors lasts at most up to 8 clock cycles from the time of fault injection.

Figure 3.5. Latch Error Occurrence in Time



3.2. Simulated Fault Injection at the Logic Level

Simulated fault injection at the logic level is similar to that at the electrical level in that they are both circuit-level simulations. The difference is in the fault models used. In the electrical-level injection, physical fault models are used, while in the logic-level injection, logic fault models are used. Logic-level fault simulation uses abstract logical models for both faults and circuit functions to evaluate the behavior of a system. In contrast to the evaluation of the physical models used in the electrical-level simulation, logic-level simulations perform binary operations that represent the behavior of a given device. They take binary input vectors and to evaluate the output of the device for a given input pattern. Each signal in the circuit is represented by a member in a set of boolean values depicting the steady state conditions of the physical circuit. For example, set {1,0,X} is often used to describe high, low, and unknown voltage values for logic gates. Fault injection at this level simply forces a node to either stuck-at-1 or stuck-at-0, or it inverts a logic value. Fault injection location and time can be set arbitrarily. Hence, with logic simulation, one obtains outputs with discrete values and possibly with some approximate timing information. Typically, outputs of the faulty and non-faulty systems are compared to determine whether a fault has been detected.

For MOS technology, a gate-level logic simulator is inadequate to handle circuits containing pass transistors, ratioed logic, buses, and other features that exhibit bidirectional signal flow and/or charge-sharing effects. To handle such transistor networks without resorting to expensive electrical-level analysis, switch-level simulation is proposed in [Bryant84]. Switch-level analysis allows bidirectional signal flow and different levels of signal strengths. For example, a discrete set {0,1,..,9} can be used to model different signal strengths or voltage levels. At this level, transistor-level fault modeling can also be incorporated easily.

Fault simulation has been widely used for evaluating the coverage of a given set of test vectors for testing manufacturing defects in a chip. Typically a set of test vectors generated either by an automatic test pattern generator (ATPG) randomly, or manually is submitted to the fault simulator in order to decide how many faults can be detected by the test vectors. In this case, the generated test vectors become workloads or inputs to the system. In the beginning of such a simulation, a stuck-at fault is injected, and the faulty circuit is simulated while a given test is applied at the primary inputs of the circuit. A similar run is performed again without any fault injection. The logic values at the primary outputs of the faulty circuit are then compared to the outputs of the fault-free run to determine if there is any difference in the outputs. If the injected fault altered logic values at the outputs compared to the clean run, then the fault is assumed to be detected. If a fault is detected, there is no reason to continue the simulation for that specified fault. The process of test generation and fault simulation is iterated until satisfactory fault coverage (the percentage of faults detected of all theoretically detectable faults) is achieved. This application has been widely used in industry to evaluate and assist test generation [Ruehli83] [Rogers85].

The use of fault simulation to perform dependability analysis at the design phase, and thus avoid the high cost of an additional redesign/modification iteration after the finalized design is submitted for fabrication is an ongoing research effort. New techniques are being introduced to perform fault sensitivity analysis of very large circuits. The simulation approach permits determination of a chip's fault sensitivity during the design stage. Through simulated fault injection and subsequent fault propagation at the logic level, it is possible to identify critical bottlenecks in reliability. To characterize a highly dependable VLSI systems, we need to evaluate, simultaneously, the fault behaviors of all components and their combined behavior.

For the dependability analysis of a system either stuck-at or transient faults can be simulated. Stuck-at faults can be simulated using conventional logic-level fault simulators that are widely available. A stuck-at-fault injection is performed by forcing the state of a node to a specified value for the entire simulation duration. By selectively tracing/detecting a set of internal and external nodes, fault propagation can be monitored. Fault behavior in a system can be modeled and analyzed through studying the fault propagation trace.

Transient fault injection is more complicated than stuck-at fault injection. Transient faults are injected by altering the logic values of the target node momentarily during the simulation. For example, the output of a gate is set to 1 while it should normally be 0. This faulty logic is forced on the output for a specified time period. Logic-level transient injection can be performed in two different ways. The above "bit-flip" effect can be performed on the combinational part of the circuit using a timing simulator. The created "pulse" can then propagate and become latched. Another way is to change the state of a machine by flipping a bit in a register or a memory element in the system. These approaches, however, may not reflect the actual device-level transient behavior at the logic level, because a transient can propagate in multiple paths and result in more than one latch error.

To evaluate system dependability based on realistic fault models, a fault-dictionary approach can be taken [Choi93]. A fault-behavior dictionary generated from electrical-level fault analysis can be used as a fast look-up table for a logic-level concurrent or parallel fault-injection simulation. First, an electrical-level fault-behavior dictionary for a given chip can be generated by extensive fault simulations. In this step, gates around the fault-injection location are extracted, and a subcircuit consisting of these gates is formed. This subcircuit is exercised by exhaustively applying all input combinations while fault injection is performed. Faulty behavior at each of the subcircuit outputs is analyzed and recorded in a dictionary. The resulting entry in the dictionary consists of the input vector, fault-injection time, and fault location. Second, concurrent run-time fault injections of the generated logical error at the subcircuit level using the fault dictionary can be performed. The concurrent simulator is used to propagate, in a single simulation pass, the effects of the injected errors.

Both transient and permanent faults can be injected using switch-level or gate-level logic simulation. The overall simulation approach for fault injections at the logic level consists of the following steps:

(1) Obtain the net-list of a design and devise appropriate simulation models to emulate the given design.

(2) Simulate the model using a logic-level simulator.

(3) During the simulation, run a given workload depicting the application or test software (by applying test vectors to the primary inputs).

(4) Save the behavior of the system under fault-free conditions by tracing all the changes in the evaluated logic events of monitored nodes for comparison with the subsequent fault-injection runs.

(5) Run the same workload again and inject a fault to a selected node during the simulation period and trace.

For a stuck-at fault: Force the state of the selected node to either 1 (for stuck-at-1 fault) or 0 (for stuck-at-0 fault) and evaluate the circuit. Hold the state to stuck-at fault value throughout the simulation.

For a transient fault: Force the state of the selected node to a logic value that is the reverse of the normal state (i.e., force a 0 if the normal state is a 1, and vice versa). Hold the state to the reverse value on the node for one or more clock cycles. Let the fault effect propagate by evaluating the circuit with the corrupted logic state. Release the forced state when a new signal/event arrives at that node.

(6) Monitor the behavior of the system under fault conditions.

(7) Compare the traces from the faulty and fault-free runs and identify the differences to determine where and when the fault has propagated.

(8) Use collected statistical measurements to determine dependability parameters (e.g., detection coverage) and the fault impact (e.g., minor program error or system failure).

The above fault injection steps should be repeated a large number of times for a given workload. If the experiment is intended to estimate single measures (e.g., detection coverage), the number of injections required for achieving a given confidence interval can be determined using Eq. (2.14). If the experiment purpose is to obtain distributions (e.g., error latency distribution), the fault injections should not be stopped until the constructed distribution is stable, i.e., the two consecutive distributions constructed are not different statistically. Importance sampling can be used to significantly reduce simulation runs.

Two early studies in fault injections at this level took a digital avionic miniprocessor, BDX-930, as the target system. The first study investigated the impact of faults at gates and pins on the output results of programs, with emphasis on the fault latency and fault coverage issues [McGough81]. The second study investigated error propagation from the gate level to the pin level [Lomelino86]. A recent study explored the behavior of transient faults which occur during the normal execution of a processor [Czeck91]. The study quantified faults that can be emulated by software-implemented fault injections (to be discussed in Section 4.2). We discuss these studies in the following two subsections.

3.2.1. Study of Bendix BDX-930

An early study in this field was the simulated fault injection to the Bendix BDX-930, a digital avionic miniprocessor, to investigate fault latency and coverage [McGough81]. The BDX-930 was composed of bit-slice processors (AMD2901) and was used in a number of flight control avionic systems. Fault tolerance was achieved by replication of the processing and voting in software. A gate-level emulator of the BDX-930 was developed for this study. The run speed of the emulator was 25,000 times slower than the BDX-930 when hosted on a PDP-10.

The methodology used in the study was: Given a software program running on the processor, inject a single stuck-at or inverted fault at a gate or pin selected randomly and observe the time to detection, assuming that a detection occurs whenever there is a difference between the outputs of the faulty and fault-free processors executing the same program. The experiment was repeated 600 to 1,000 times to construct an empirical latency distribution. Six different programs were selected to repeat the above experimental procedure. In addition, a typical avionic flight control system self-test program was written for this study and executed to determine fault detection coverage.

Results showed that most detected faults were detected in the first repetition of the program. Subsequent repetitions did not significantly increase the propagation of detected faults. A large percentage of faults (about 60% for the gate-level faults and 30% for the pin-level faults) remained undetected after as many as eight repetitions of the program. The fault coverage of the self-test program was found to be 87% for the gate-level faults and 98% for the pin-level faults.

The above study emphasized the impact of faults at gates and pins on the output results of programs. Another study on the Bendix BDX-930 computer investigated error propagation from the gate level to the pin level [Lomelino86]. In this study, a single AMD2901 processor chip in the BDX-930 was selected for fault injection and error data collection. The processor was simulated using an event-driven, gate-level logic simulator developed at NASA Langley [Migneault85]. The simulator was driven by a self-test program, developed for the BDX-930, which provides a high probability of detecting error activity.

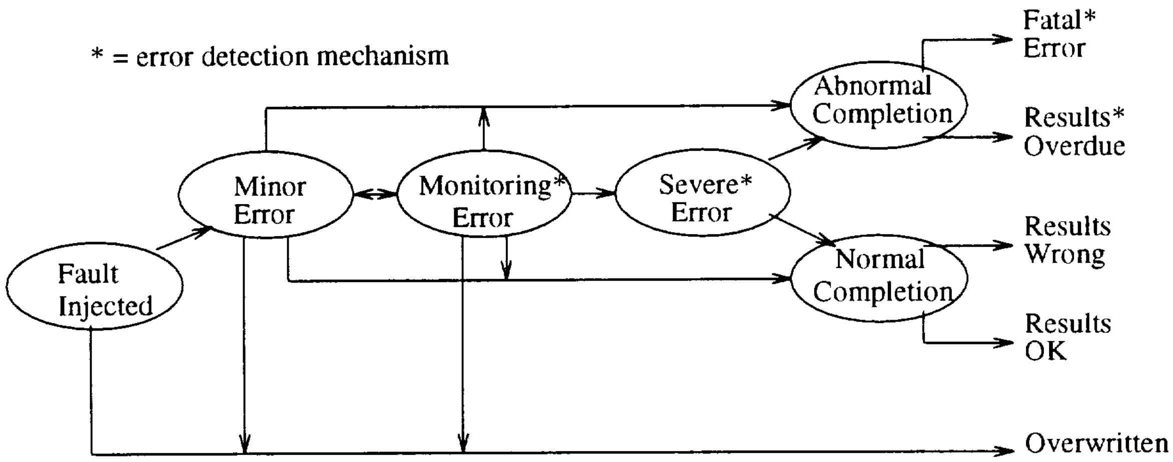
In the simulation, the single stuck-at fault model was applied to 150 selected gates for fault injection. The gates were distributed throughout the nine function units of the AMD2901. Error data was collected by first simulating a fault-free circuit, then simulating the circuit with a single injected fault, and finally comparing the two simulation output for obtaining differences. Four sets of simulation experiments, consisting of 150 simulations per set, were conducted. Results showed that 78.7% of faults produced error propagation detected within the chip and 66.7% of faults produced errors that propagate to the output pins, within the first 100 clock cycles. The error distribution at the output pins was found to be sensitive to the locations of faults. The results also showed that the error activity increases with the increase of concurrent microinstruction activity.

3.2.2. Study of IBM PC RT

In [Czeck91], a simulation model of the IBM RT PC was used to inject transient, gate-level faults for exploring the behavior of transient faults which occur during the normal execution of a processor. The emulated hardware functional units in the processor included: instruction prefetch buffer (IPB), microinstruction fetch (MIF), data fetch and storage (DFS), ALU and shifter (ALU), and ROMP storage channel interface (RSCI). Both original error detection mechanisms (EDM) which reside in the IBM RT PC (such as illegal instruction traps and memory access violation) and additional error detection mechanisms which are provided in this study for evaluating their effectiveness (such as timeout and control flow monitoring) were included in the simulation model.

Figure 3.6 shows possible error manifestations after a fault injection. In the figure, *minor errors* are differences in the internal processor state between the faulty and fault-free simulation runs, which have not been detected by an EDM. *Monitoring errors* are those which are uncovered by the "duplicate and compare" EDM which monitors bus addresses and data. *Severe errors* are those resulting in the change of a microinstruction and the instruction address registers, which lead to a change in the control flow of the program. *Fatal errors* have triggered a system resident EDM and caused an abnormal termination of the application task. *Results overdue* occurs when the task executes longer than a predetermined time limit and the execution is halted. *Overwritten* means that the injected fault does not manifest to a minor error or a minor error is overwritten by correct data.

Figure 3.6. Error Manifestations in the IBM PC RT Simulation Model



Three workloads were selected for this study: an iterative matrix multiplication, a recursive Fibonacci program, and an iterative Fibonacci program. These workloads were considered to be representative of the characteristics of instruction set architectures and diversity in program structure. For each workload and each fault location, 1000 faults were injected. Following is the method of the experiments:

(1) For each workload, the fault-free behavior of the workload is extracted from the internal state of the processor and saved for comparison during the subsequent fault injection experiments.

(2) A fault location is selected such that the fault in the location has a high probability of producing an error and locations for different injections do not yield the same error behavior.

(3) The fault injection time is set to the start of the workload execution. The fault injection time will be advanced by one cycle for each successive fault injection experiment.

(4) The fault is injected for a duration of one clock cycle at the location and time selected.

(5) For each successive clock cycle, the internal processor state of the faulty processor is compared with that obtained in step 1. Differences are recorded for off-line analysis.

(6) The faulty behavior is monitored at each clock cycle until the program execution is completed or a severe error causes the monitor to cease.

(7) The simulation run is restarted at step 3 and the time of next fault injection will be advanced by one clock cycle.

Results of the study include: 40% to 55% of injected faults do not produce an error. Among the faults that manifest to errors, approximately 2/3 of them can be emulated by the *software-implemented* fault injection approach (to be discussed in Section 4). The other 1/3 of these faults manifest to errors in CPU components (e.g., microinstruction control registers) that are not accessible to software. At the system level, the fault behavior showed a strong dependency on the workload structure and instruction sequencing rather than the instruction mix. Error detection latency was found to follow a Weibull distribution with a decreasing detection rate. The distribution represents two error occurrence processes: a fast process in which fault manifestation and error propagation occur within a small time window and a slow process in which dormant faults and errors are activated gradually by the workload.

3.3. Simulated Fault Injection at the Function Level

Function-level fault injection simulation is used to study complete computer and network systems rather than the components of which they consist. These studies typically consider the hardware, the software, their interactions, and the inter-dependence between the various components of the system. There are at least five issues in developing functional simulation models at the system level.

The first is a lack of well-established system level fault models. This is partly due to the second issue: a large and varied component domain. At the gate level, the basic components are gates with single functions and well-defined interconnections. At this level, it is possible to establish a fault model, such as the single stuck-at fault model which can be consistently applied to all gates to model their fault behavior. At the system level, the basic components include CPUs, communication channels, disks, software systems and memory. The components have complex inputs, perform multiple functions, have varied physical attributes (e.g. hardware and software) and complex interconnections. In addition to the diversity of the components that make up a system, two similar components (such as two CPUs) can have different functions and behavior. This makes it difficult to establish a single fault model that can be applied consistently to all components.

For this reason, various types of fault models are required and will depend upon the type of component being injected. The fault models are functional fault models that simulate system-level manifestations of gate-level faults. For instance, a single bit-flip is typically used to simulate a memory or register fault. Various fault models can be used for communication channels. Messages traversing the channel can be corrupted or destroyed, or the channel can be made inoperative. A fault in a processing node can be modeled as a service interrupt caused by CPU, memory, disk, or software faults in the node. More detailed fault models for a processor or other system components can be derived from lower-level simulations using the fault-dictionary approach discussed in Section 3.2. For instance, a gate-level simulation of a processor can be injected with faults while executing a typical workload. The effect of the faults on the workload can be stored in a fault dictionary that contains, for each gate-level fault injected, the types of effects and the probability of these effects. This dictionary can then serve as a fault model for system level simulations.

The third issue, which is especially significant when simulating large, complex systems, is the effort and time required to develop a functional simulation model. For fault injection studies and dependability analysis, two factors contribute to this. One is the time and effort needed to describe the detailed functionality of the system components. The other is the time and effort required to inject faults, initiate repairs, abort, reschedule and synchronize events, and maintain a whole host of fault statistics. As the number of components in the system becomes large, a well-formulated, structured, and automated approach is needed to contend with the complexity. A solution is to have a tool which includes a library of software "objects" that provide the skeletal framework needed to conduct simulated fault injection studies and that can be easily customized to meet user specific needs.

The fourth issue is the impact of the software on system dependability. Dependability studies have tended to focus on a system's hardware components. But as the hardware becomes more reliable, the software component is becoming a more dominant factor [Gray90]. The effectiveness of functional detection and repair schemes depend upon several application-specific measures such as detection latency and error propagation times. In order to study the impact of the software on system dependability, methods are needed that allow the designer to incorporate the application into the overall dependability study. Thus, the simulation tool should permit the execution of actual user programs.

The fifth issue, and extremely important one, is simulation time explosion. This occurs when the system modeled has very small failure probabilities requiring large simulation runs to obtain statistically significant results. This is especially a problem with functional simulation because its primary benefit is detailed modeling, which further contributes to simulation time explosion. Different acceleration techniques are used at the system level to reduce simulation time explosion. Hierarchical and hybrid simulation methods have been shown to be very effective [Goswami92] [Goswami93a]. The basic approach of these techniques is to: 1) break down a large, complex model into simpler submodels, 2) analyze submodels individually, 3) combine the results from step 2 to build a simplified system model, and 4) analyze the system model to obtain the solution. If the models in step 1 and step 3 are both simulation models, the approach is called hierarchical simulation. If the models in step 1 are simulation models and the model in step 3 is an analytical model, the approach is called hybrid simulations. As long as the interactions among the subsystems are weak, this decomposition approach provides valid results. The approach is ideally suited for dependability analysis because dependability models can usually be broken into two submodels — a fault occurrence submodel and an error handling submodel — whose interactions are typically weak. Figure 3.7 illustrates the approach.

A good question is that since there are a large number of analytical modeling tools including petri-net based simulation tools, what is the need for functional simulation tools for system level dependability analysis? What additional information and capabilities can they provide over analytical tools? The answer is that analytical modeling tools only use probabilistic models to represent the behavior of a system. In essence, the effect of a fault on the system is pre-defined by a set of probabilities and distributions. Functional simulation tools not only use stochastic modeling, they also permit behavioral modeling, which does not require that the effect of the faults be pre-defined.

An example that demonstrates this capability is a study in which a distributed system using a centralized, prediction-based load balancing scheme is evaluated under faults with DEPEND [Goswami92], a function-level simulation tool to be discussed in Section 3.2.2. The study is demonstrated in Figure 3.8. The load-balancing software that makes task placement decisions and maintains the database is actually executed within the DEPEND environment on a simulated distributed system. DEPEND's fault injection facilities are used to inject communication faults which destroy and corrupt fields of the status messages sent to the CPU maintaining the database. Faults are also injected

Figure 3.7. Hierarchical Simulation

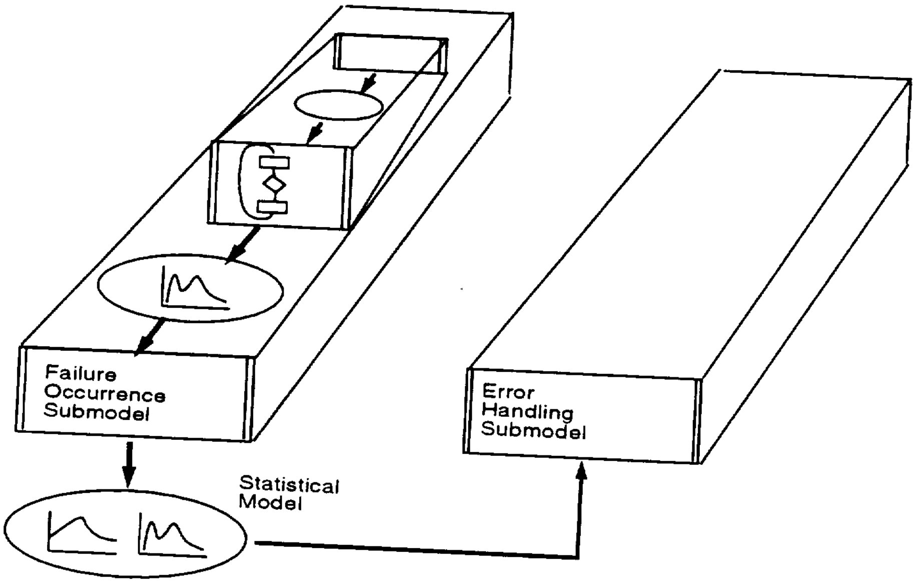
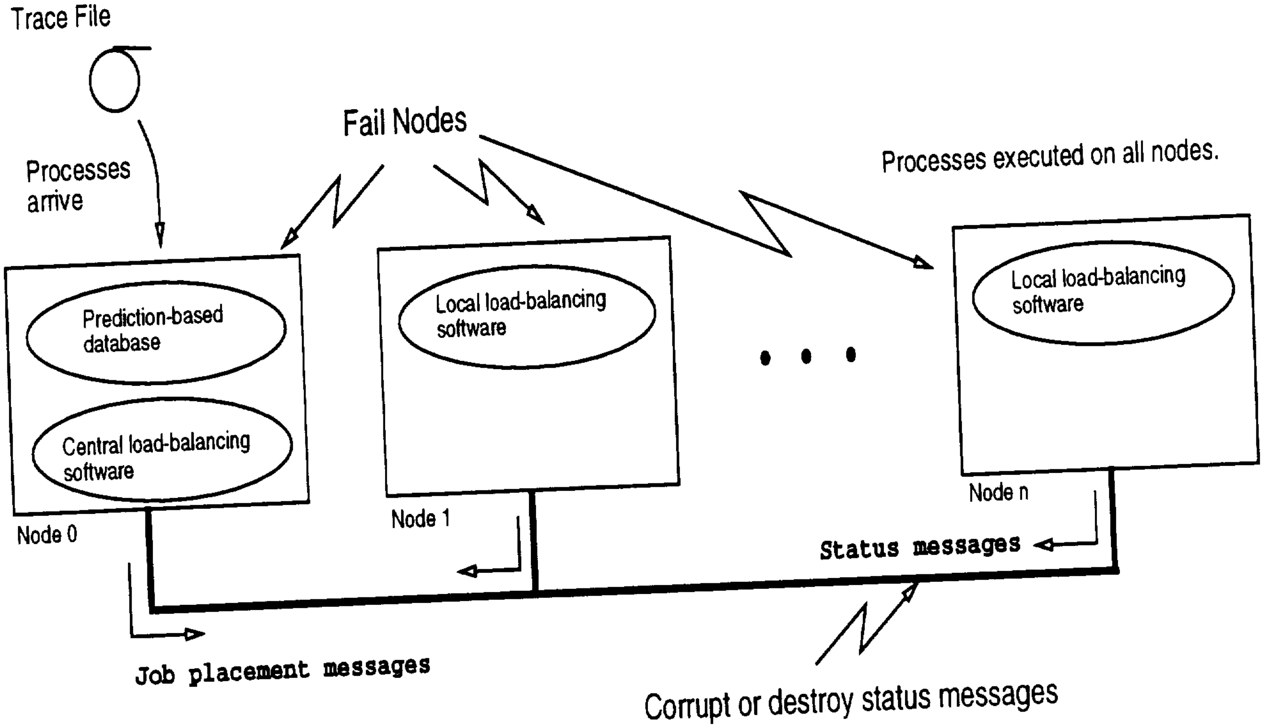


Figure 3.8. Distributed System Executing Load Balancing Software


Trace File
Processes arrive
Fail Nodes
Processes executed on all nodes.
Prediction-based database
Central load-balancing software
Local load-balancing software
Local load-balancing software
Node 0
Node 1
Node n
Status messages
Job placement messages
Corrupt or destroy status messages

into the CPU containing the load-balancing software, to erase its database. The effect of these faults is to corrupt the database and impair the placement decisions made by the load-balancing software. If a purely probabilistic modeling tool were used for this study, the user would have to pre-specify:

- the probability that a fault will corrupt the database,
- how each fault will corrupt the database,
- which portions of the database will be corrupted,
- the extent of corruption, and
- how each corruption will impair the placement decision made by the load-balancing software.

Needless to say, these factors are extremely difficult to obtain without executing actual software. Because DEPEND executes actual software, these parameters are the results of (and not inputs to) the fault injection experiment. Only the fault arrival rates and the types of faults injected need to be specified. Thus, DEPEND can identify the failure mechanisms, obtain failure probabilities, and quantify the effect of faults. It can be used to pick out the key features that must be modeled and help to determine and specify both the structure of, and the parameters to, analytical models.

A single distinguishing feature between probabilistic modeling and behavioral modeling is brought out by one of the results of this study (details of all the results can be found in [Goswami93b]). The study helped to uncover a design feature of the software that caused erratic increases in system response time only when status messages were destroyed. Once the software was modified, the erratic increase in response time ceased. Clearly, such results cannot be obtained with analytical modeling tools.

An additional advantage of functional simulation tools is that they allow the use of any type of TTF distributions. Unlike analytical modeling, in which only a few types of distributions are commonly used for the tractability of models, the simulation method can handle any form of distribution, empirical or analytical.

An early study used a trace-driven simulation approach to analyze error latency [Chillarege87]. The approach is based on sampled data of physical memory activity gathered, through hardware instrumentation, from a computer system running normal workloads. The data are then used for a trace-driven simulation in which faults are inserted into the trace to emulate fault occurrence and error discovery processes in the system. The approach provides a means to study error latency in memory systems under real workloads.

In recent years, several function-level simulation tools that can be used for fault injections have been or are being developed. NEST, DEPEND, and REACT are three representative tools. REACT, a software testbed that performs automated life testing of a variety of multiprocessor architectures through simulated fault injections, is being developed at the University of Massachusetts [Clark93]. Several system, workload, and fault/error models, which are representative of multiprocessor architectures and conditions, are embedded in the testbed. The tool can be used to evaluate system reliability and availability metrics. Preliminary versions of the software have been reported to be successfully employed in several studies of multiprocessor systems [Clark93].

NEST is a function-level testbed that specializes in modeling and evaluating distributed network systems [Dupuy90]. Although the tool is not designed for fault injections, users can make node or link failures by deleting or adding nodes and links or changing their features while the simulation is running. DEPEND, developed at the

University of Illinois, exploits the properties of the object-oriented paradigm to provide a general-purpose, system-level dependability analysis tool that can evaluate various types of fault tolerant architectures [Goswami92]. The object-oriented feature of DEPEND makes the tool capable of modeling multiple levels of functional units to meet a wide range of applications. The next two subsections discuss NEST and DEPEND, respectively.

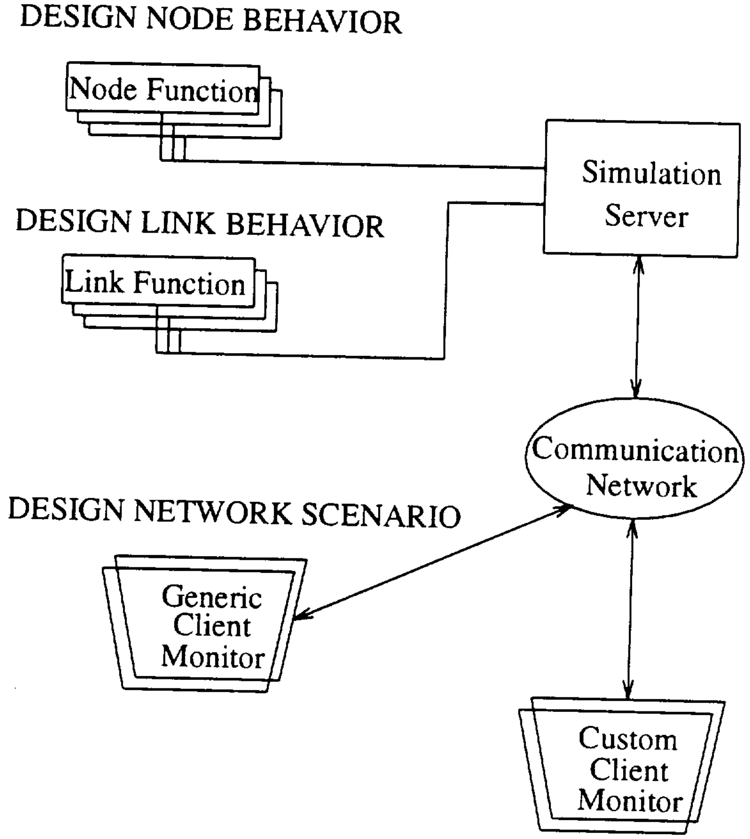
3.2.1. NEST — A Network Simulation Testbed

The NEtwork Simulation Testbed (NEST) is a graphical environment, running on the UNIX system, for modeling, executing, and monitoring distributed network systems and protocols [Dupuy90]. Using a set of graphical tools provided by NEST, the user can develop simulation models of communication networks. The model includes node functions (e.g., routing protocols) and communication link behaviors (e.g., packet loss or delay features), typically coded in C. These user procedures are linked with run-time routines embedded in NEST and executed by the NEST simulation server. The user can reconfigure modeled network system through graphical interaction or programming. Built-in graphical tools allow users to programming custom monitors to observe the simulation results on-line.

Figure 3.9 shows the overall architecture of NEST. NEST consists of a simulation server and several client monitors. The simulation server is responsible for running simulations. The generic client monitors are used to configure simulation models and control their executions. The custom client monitors are used to observe simulation behavior and display results. Clients can reside on separate machines so that the server is dedicated to time-consuming simulations.

Node functions are used to model distributed communicating processes running at network nodes (e.g., protocols and database transactions). NEST executes node processes and their communication calls using a set of embedded primitives for sending, broadcasting, and receiving packets. The motion of a packet over links is simulated by passing it through the link functions. Link functions are used to model the behavior of communication links (e.g., packet loss and link jamming). Link functions are also used to monitor and collect performance statistics of link traffic. The simulation server schedules the execution of the node and link processes to meet the delay and timing specified by the user.

Figure 3.9. Overall Architecture of NEST



The user can create and modify a network description (node and link functions and connections) using the NEST graphical tools. Once the user has defined a simulation scenario, it is sent to the simulation server to be executed. One of NEST's key features is its ability to reconfigure a scenario during the simulation run. The user may delete or add nodes and links (thus failures can be emulated) or change their features while the simulation is running. The impact of these changes may be instantly observed and interpreted. Such dynamically reconfigured simulations can be used to study the impact of node/link failure and recovery on the modeled network system.

3.2.2. DEPEND — A System Dependability Analysis Environment

DEPEND is an integrated design and fault injection environment [Goswami92]. It provides facilities to rapidly model fault tolerant architectures and conduct extensive fault injection studies. It is ideally suited for evaluating specific fault tolerant mechanisms, detailed fault scenarios such as latent errors, and software behavior due to hardware faults. It is a functional, process-based [Kobayashi78], [Schwetman86] simulation tool. The system behavior is described by a collection of processes that interact with one another. A process-based approach was selected for several reasons. It is an effective way to model system behavior, repair schemes, and system software in detail. It facilitates modeling of inter-component dependencies, especially when the system is large and the dependencies are complex, and it allows actual programs to be executed within the simulation environment. Both hierarchical and hybrid simulation techniques have been used in DEPEND.

DEPEND exploits the properties of the object-oriented paradigm, specifically, modular decomposition and modular composability [Meyer88], to model different levels of components and to implement a variety of fault models. Modular decomposition consists of breaking down a problem into small elements, whereas modular composition favors production of elements that can be freely combined with each other to provide new functionality. If, for instance, the fault injection process is divided into two elements or objects: an object that determines when to inject and interrupt the system, and an object that determines the response to a fault (the fault model), then the two criteria are met. The first object is common to all fault injection methods. It encapsulates the various mechanisms used to determine the arrival time of a fault and interrupt the system. The second object is the fault model and is specific to the component being injected and the type of fault injection study. The two are combined via function calls. Thus, by specifying different fault model objects, one injector object can be used for all types of fault injections. Key objects, such as the injector object, are designed to be parameterized. That is, the user can specify various fault arrival distributions or trace files. This same approach is used to model components that are similar but not identical; common aspects are encapsulated in an object which then invokes other objects to provide more specific functionality. Furthermore, because users can specify specific behaviors (e.g. their own fault model objects), the tool is not limited to any predefined set of fault models or component types.

A library of objects that provide the skeletal foundation necessary to model an architecture and conduct simulated fault-injection experiments is provided. This reduces the development time and effort needed to build simulation models. In addition to decomposition, composition, and parameterization, the concept of inheritance [Meyer88] makes it possible to provide a library with a minimum set of objects that can be readily specialized to model a wide gamut of different architectures and fault injection experiments. With inheritance, users can inherit the properties of

Table 3.4. Some Objects Provided in DEPEND

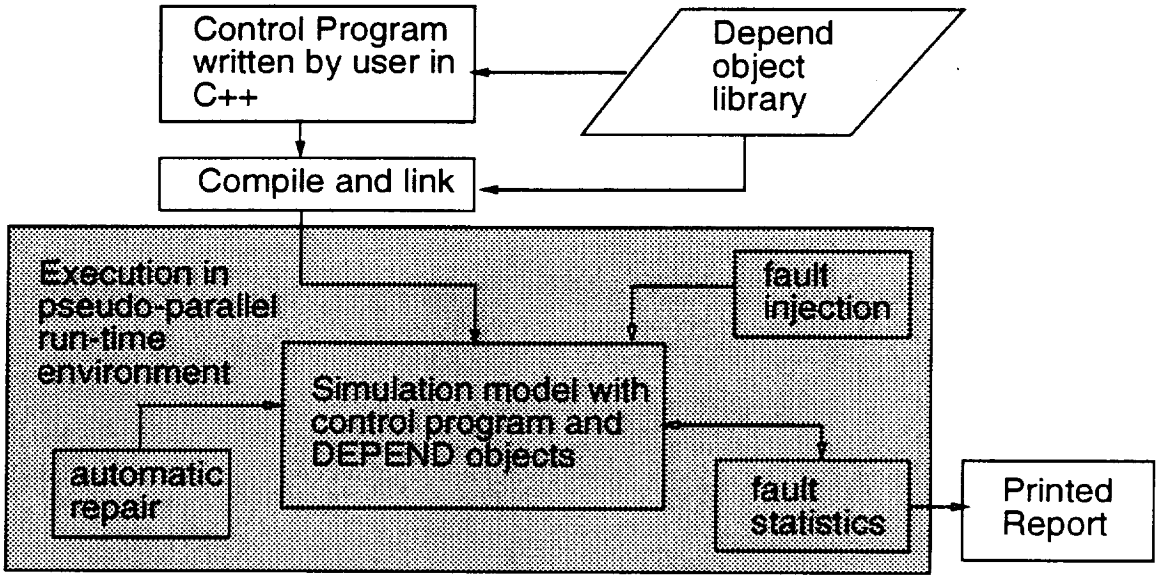
Name	Type	Description
Active_elem	Elementary	Simulates a basic server. Several disciplines: first come first serve, round robin, etc.
Injector	Elementary	Injects faults using distributions, trace files and workloads.
Checksum	Elementary	Computes checksums.
Fault Report	Elementary	Compiles and displays fault statistics.
Voter	Elementary	Simulates a basic voter with timeout.
Server	Complex	Simulates a server with spares. Three policies: no spare, graceful degradation, stand-by sparing. Automatic repair and reconfiguration
Link	Complex	Simulates communication channels. Several fault types: link dead packet corruption, packet loss, and user defined faults.
NMR	Complex	Simulates dual self-checking, triple-modular redundant and N-modular redundant components.
Fault Manager	Complex	Simulates software fault management schemes. Logs faults and shuts off components which exceed their fault threshold.

an existing object and develop more specialized objects with minimum effort. Table 3.4 briefly describes some of the major objects in the DEPEND library. Elementary objects provide basic functions, such as injecting faults and compiling statistics. Complex objects created from several elementary objects simulate fundamental components found in most fault tolerant architectures such as CPUs, self-checking processors, N-modular redundant processors, communication links, voters, and memory.

The steps required to develop and execute a model are shown in figure 3.10. The user writes a control program in C++ using the objects in the DEPEND library. The program is then compiled and linked with the DEPEND objects and the run-time environment. The model is executed in the simulated parallel run-time environment. Here, the assortment of objects including the fault injectors, CPUs, and communication links execute simultaneously to simulate the functional behavior of the architecture. Faults are injected and repairs are initiated according to the user's specifications, and a report containing the essential statistics of the simulation is produced.

DEPEND allows users to specify different fault models. In addition, DEPEND provides default fault routines for each object to minimize user design time. For instance, the default fault model for a communication medium simulates the effects of a noisy communication channel. Fields in the messages passed along the communication link are actually corrupted or the message is destroyed.

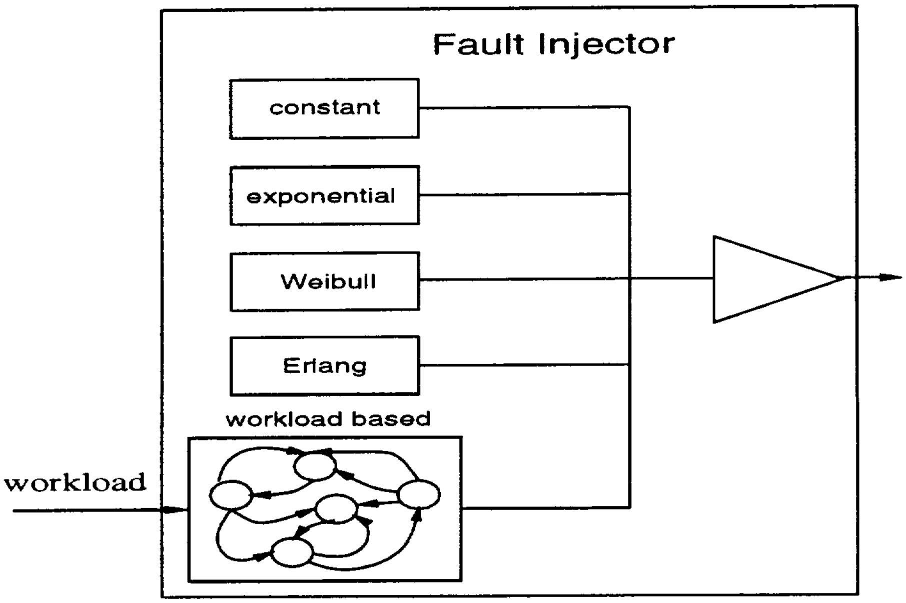
Figure 3.10. The Depend Environment


Control Program written by user in C++
Depend object library
Compile and link
Execution in pseudo-parallel run-time environment
fault injection
Simulation model with control program and DEPEND objects
automatic repair
fault statistics
Printed Report

The *fault injector* is a fundamental object of DEPEND (Figure 3.11). It encapsulates the mechanism for injecting faults. To use the injector, a user specifies the number of components, the TTF distribution for each component, and the fault subroutine that specifies the fault model. In addition to user-specified distributions, the injector provides constant time, exponential, hyperexponential, and the Weibull distributions. The injector also provides a workload-based injection scheme that varies the fault arrival rate based on a specified workload. The user provides a workload function, a set of workload states, and an exponential fault arrival rate for each state. For example, the workload function may be the utilization of a server. With this approach, the fault arrival rate will fluctuate with the utilization of the server. The fault injector will periodically poll the workload function to update a state transition diagram to maintain a history of the workload behavior. This history is used to inject a large number of faults during peak workload conditions and fewer faults when the workload is low. This technique models the workload/failure dependency observed in [Iyer80] and [Castillo81].

In addition to executing actual C++ and C programs, DEPEND provides an abstract software modeling environment to simulate program behavior during the early design stages when actual code does not exist. The environment represents application programs by decomposing them into graph models consisting of a set of nodes, a set of edges that probabilistically determine the flow from node to node, and a mapping of the nodes to memory. The graph

Figure 3.11. The Fault Injector Object



models are then mapped to virtual memory and executed while errors are injected into the program's memory space. The environment provides application-dependent parameters, such as detection and propagation times, and permits meaningful application-dependent evaluation of function- and system-level error detection and recovery schemes. This environment has been used to analyze memory-scrubbing schemes within the context of application programs [Goswami93c]. The application-dependent coverage values obtained were compared with those obtained by traditional schemes that assume uniform or random memory access patterns. The coverage values obtained using the traditional approach were found to be up to 100% larger than those obtained with the software graph model. The findings demonstrate the need for application-dependent evaluation — especially when evaluating the dependability of application-specific systems.

DEPEND has been applied to evaluate several computer systems. In [Goswami91] and [Goswami92], DEPEND was used to simulate the UNIX-based Tandem Integrity S2 fault tolerant system and evaluate how well it handles near-coincident errors caused by correlated and latent faults. Issues such as memory scrubbing, reintegration policies, and workload-dependent repair time were evaluated. The accuracy of the simulation model was validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a production Tandem Integrity S2 machine. DEPEND has also been used to study the CM5 connection machine, the Parsytec high-performance computer being developed by the European Esprit project, the Space Station Data Management System, and the computing element of the Hubble Telescope.

IV. PROTOTYPE PHASE

In the prototype phase of the development of fault tolerant systems, physical fault injection can be used to evaluate fault, error, failure, and fault tolerance characteristics of the developed systems. Normally, fault injection can be applied only to fault tolerant systems, because the injected faults, if activated, would almost always crash the target system without fault tolerant mechanisms. However, fault injection can also be used in non-fault-tolerant systems if the system control flow can be well traced and the system state information can be obtained when it crashes because of injected faults.

A fault injection environment typically contains the following components: target system, controller and monitor, fault injector, data collector, and data analyzer, as shown in Figure 4.1. The target can be a VLSI chip, a computer system, or a network system. When faults are injected into the target, either benchmarks or synthetic workloads should be running on the target to emulate real workloads. The controller is a special software program, sitting on the target or on another computer, which controls the overall fault injection experiments. The fault injector implements fault injections into the target. The monitor keeps track of normal and abnormal executions of the workload and initiates data collection whenever necessary. The data collector and analyzer perform on-line data collection and off-line data processing and analysis, respectively.

Figure 4.1. Components in a Fault Injection Environment

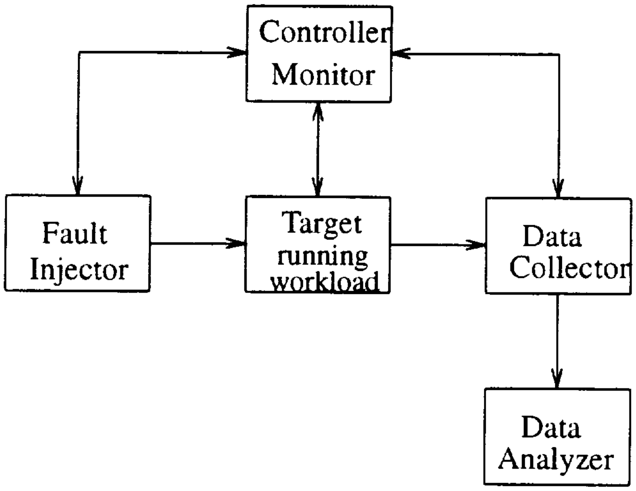


Table 4.1. Categories of Fault Injections

Category	Hardware-Implemented	Software-Implemented	Radiation-Induced
Approach	Inject faults to IC pins by hardware instrumentation	Inject faults to components by special software	Inject faults by applying radiation rays to target
Advantages	No disturbance to workload High time resolution	Flexibility Low cost	Can induce transient faults inside IC evenly
Disadvantages	Limited access points High cost	Workload disturbance Low time resolution	Fault injection points are uncontrollable
Studies	FTMP [Lala83] FTMP [Shin84], [Shin86] FTMP [Finelli87] MESSALINE [Arlat90]	Accelerated Injection [Chillarege89] FIAT [Segall88], [Barton90] FERRARI [Kanawati92] HYBRID [Young92] FINE [Kao93]	Z-80 [Cusick85] MC6809E [Karlsson89] MC6809E [Gunneflo89]

The fault injector can be implemented by hardware, software, or radiation. Correspondingly, fault injection can generally be divided into three categories: hardware-implemented (or hardware) fault injection, software-implemented (or software) fault injection, and radiation-induced (or radiation) fault injection. Table 4.1 lists features and representative studies in these categories. The monitor can also be implemented by hardware, software, or both (hybrid). If the fault injector is implemented by software and the monitor is implemented by hardware or by both hardware and software, the system is called a *hybrid* fault injection environment. The following three sections discuss in detail each type of fault injection.

4.1. Hardware-Implemented Fault Injection

Hardware-implemented fault injection is a method of introducing faults in the hardware of a computer system with the aid of additional hardware instrumentation. The method is well suited for studying dependability characteristics which require high time resolution, such as fault latency in the CPU, which cannot be easily achieved by other fault injection methods. For example, the occurrence of software-implemented faults is restricted by the system clock (i.e, the injections must occur synchronously). Detections are similarly restricted by the system clock, unless an external hardware monitor is used. Two main techniques are used to accomplish hardware-implemented fault injections.

The first approach involves the use of active probes attached to the desired hardware injection points. The currents through these injection points can be altered, thereby influencing the corresponding logic values. The types of faults attainable with probes are usually limited to stuck-at faults. However, it is also possible to introduce bridging faults by placing the probes across multiple hardware points. Care must be taken with the use of active probes that force values onto injection points, because damage to the target hardware can result due to an inordinate amount of current.

The second technique involves the insertion of additional hardware into the target computer system. Whereas the first method uses active probes which are external to the target system, this method introduces additional hardware, which becomes part of the target system. The most common approach requires the interpolation of a socket between a chip and the circuit board. This socket has the capability to inject stuck-at faults or open faults, where the chip pin is essentially tri-stated. In addition, more complex logical faults can be forced onto these pins. For instance, the pin signals can be inverted, or ANDed or ORed with adjacent pin signals or even with previous signals on the same pin.

In theory, the domain of possible injection locations is limited only by the physical constraints of the target system that prevent the introduction of probes or other hardware. Since the target system is usually a complete prototype computer system, fault injection below the chip pin level is impractical. Thus, the focus of most injections are the pins of chips. In addition, active probes can be attached to certain circuit board locations, such as buses or other signal lines.

In addition to the range of possible injection locations, a major concern of any fault injection environment is the fault types or models that are available. We have already discussed some types of faults achievable with probes and sockets: stuck-at, open, bridging, or complex logical functions. Another important aspect of fault types is the duration of the fault, which can be either permanent, transient, or intermittent. Permanent faults are simply held on the injection point until an error detection occurs. In contrast, transient faults are placed on the injection point only for an active period, after which they are removed. Thus, the possibility exists that the transient fault may never even be latched into a chip (i.e., the fault never produces an error), especially if the active period is less than the system clock period on a synchronous machine. Intermittent faults are injected in the same manner as transient faults, but they are also repeated, either randomly or according to some function. Both injection methods discussed previously are capable of creating any of the three temporal fault types.

In the following, we will discuss two representative hardware-implemented fault injection environments: FTMP [Lala83] and MESSALINE [Arlat89].

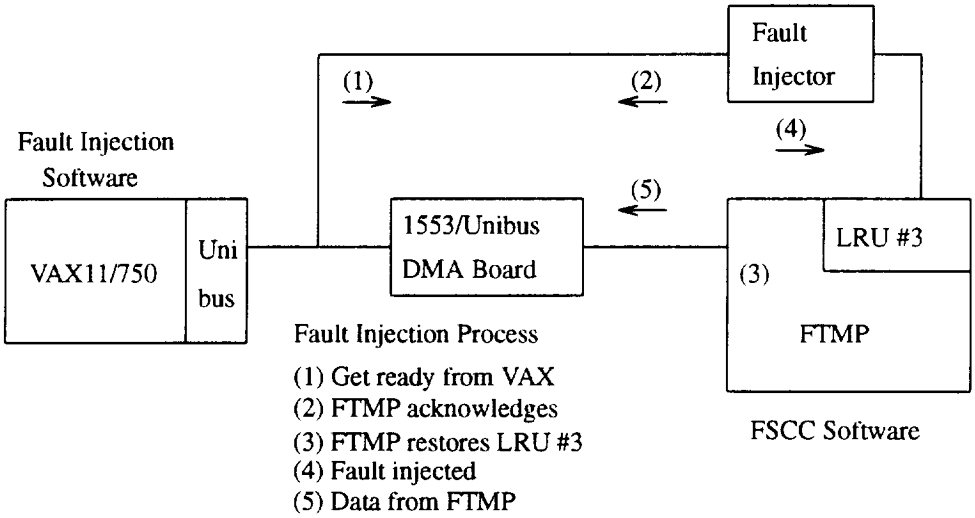
4.1.1. FTMP

Several studies in this area centered around the fault tolerant multiprocessor (FTMP) fault injection instrumentation [Lala83], [Shin86], [Finelli87]. FTMP is a computer architecture which evolved over a 10-year period in connection with several critical aerospace applications [Hopkins78]. The architecture was designed to have a failure rate of the order of 10^{-10} per hour. The basic blocks of the architecture are independent processor-cache modules and memory modules which communicate through redundant buses. The modules are dynamically grouped into several TMR triads or assigned to spare status. Jobs can be scheduled to any processor triad. All transactions between processor modules and memory modules in a triad are voted bit-by-bit. When a fault occurs, the faulty module is isolated and the faulty triad reconfigured. Fault detection, diagnosis, and recovery are handled in such a way that application programs are not involved.

Figure 4.2 shows the diagram of the FTMP fault injection instrumentation developed at the Charles Stark Draper Laboratory [Lala83], [Finelli87]. In an FTMP computer, there are several line replaceable units (LRUs), each containing a processor, clock generator, power subsystem, and bus interface circuits. LRU #3 is constructed for connection of the fault injector. All chips in LRU #3 are connected to sockets which allow them to be removed for insertion of the fault injection implant. Each fault injection implant contains circuitry which can interrupt and reconnect the pins in the sockets. Several different types of faults, such as stuck-at-0 and stuck-at-1, can be injected into the pins by the implants. These implants are controlled by a VAX 11/750 computer. A special version of the system configuration control (FSCC) program running in the FTMP communicates with the fault injection software (FIS) running in the VAX 11/750 through one of the FTMP I/O ports and a 1553/UNIBUS data link.

Faults are normally injected on one pin at a time. When an injection occurs, the FIS program chooses a fault and a pin, applies the fault to the pin, and records the injection time. Once the FTMP detects and identifies the fault and reconfigures the system, it sends this information along with the time of each event back to FIS. Upon receiving the information, FIS removes the fault by restoring the pin to its normal state and notifies the FTMP. The FTMP then puts

Figure 4.2 FTMP Fault Injection Environment



the victim module back into an active state and notifies FIS that it is ready for another fault injection. This process is repeated after a random delay.

In the experiments conducted at the Charles Stark Draper Laboratory [Lala83], a total of 21,055 faults were injected, and 17,418 (83%) were detected. All of the detected faults were identified correctly, and the system subsequently recovered successfully from each of these faults by replacing the faulty module. That is, the coverage in the FTMP was 100%, which validated the FTMP architecture and implementation.

Another study using the FTMP fault injection instrumentation was reported in [Shin84], with emphasis on the investigation of fault latency. Results showed that the hazard rate of fault latency is monotonically decreasing. Two distributions with monotonically decreasing hazard rates, Weibull and gamma distributions, were then used to fit the experimental results. The study also investigated the effect of fault latency on the probability of having multiple faults. It was shown that there exists an optimal fault latency in minimizing the multiple fault probability.

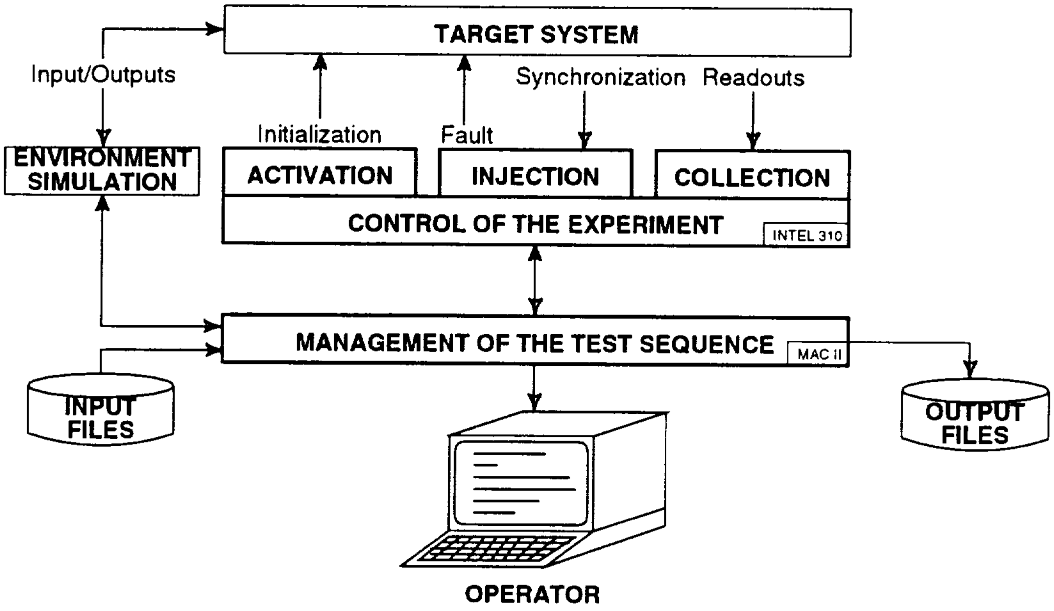
Later, fault injection experiments on the same instrumentation were conducted at the NASA Langley Research Center [Finelli87] to investigate two issues: fault sampling methods and fault recovery distributions. For each fault injection, two choices must be made: the fault location (pins) and the fault type (stuck-at-1, stuck-at-0, inverted signal, etc.). Thus, the possible fault set, or the collection of all different injected faults, can be very large. Exhaustive fault injection is costly and time consuming. It is necessary to find appropriate sampling methods to reduce the time and cost of testing. The study compared the effects (detection behavior) of different faults and grouped these faults into several subsets according to the similarity in their effects. The results showed that the effects are not homogeneous across the fault set. This indicates that stratified sampling methods, based on the fault subsets, should be developed for fault injection. The study also showed that the fault recovery time is not exponentially distributed.

4.1.2. MESSALINE

MESSALINE [Arlat90] is a flexible, pin-level fault injection tool that has been developed at LAAS-CNRS in Toulouse, France. The general architecture of MESSALINE and its environment is given in Figure 4.3. The injection, activation, and collection modules are implemented in hardware on an Intel 310 microcomputer. The software management module resides on a Macintosh II computer, which provides a flexible user interface.

The fault injection mechanism for MESSALINE uses active probes and socket insertion. Thus, fault types such as stuck-at, open, bridging, and complex logical functions can be injected. Because the duration and frequency of faults can be controlled, the fault injector can introduce permanent, transient, and intermittent faults. Signals collected from the target system can provide feedback to the injector. Also, a device is associated with each injection point to

Figure 4.3. General Architecture of MESSALINE



sense when and if each fault is activated and produces an error. MESSALINE has facilities to inject up to 32 different injection points simultaneously.

The application of MESSALINE has been shown in two experiments involving (1) a subsystem of a centralized, computerized interlocking system (called PAI) for railway control applications and (2) a distributed system corresponding to an implementation of the dependable communication system of the ESPRIT Delta-4 Project.

In the case of the PAI system, permanent stuck-at-0, stuck-at-1, and open circuit faults were injected to various memory and CPU chips. The results indicated that CPU errors were more difficult to detect than memory errors. The error detection mechanisms were analyzed individually, and it was discovered that the diagnosis software accounted for most of the error coverage. The elimination of hardware detection would have decreased the overall coverage by less than 3%.

The distributed communication system was injected with intermittent stuck-at-0 and stuck-at-1 faults. The actual faults were injected into the network attachment controllers (NAC), which provide the connection for each node to the local area network. Results showed that over 67% of all errors cause the injected NAC to be correctly identified and extracted. Also, 24% of the errors did not cause a detectable error. Thus, in over 91% of the injections, the distributed system was able to correctly handle the error. These experiments demonstrate the utility and flexibility of the MESSALINE fault injection tool.

4.2. Software-Implemented Fault Injection

While hardware-implemented fault injections require special hardware instrumentation and interface to circuits of the target systems, software-implemented fault injection provides a cheap and easy-to-control methodology. In software-implemented fault injections, no extra hardware instrumentation is needed, and users can choose fault locations in both hardware and software components accessible to machine instructions. In addition, software defects can only be emulated using the software approach by changing code. Several techniques have been proposed to emulate different types of hardware and software faults through software-implemented fault injections.

Software-implemented fault injection is done by changing the contents of memory or registers, based on some fault models, to emulate the occurrence of hardware or software faults. Hardware faults can lead to software errors and affect software executions (hardware-induced software errors). These faults can occur in CPU, memory, bus, and networks. They may cause the system to execute incorrect instructions, access incorrect data, and produce incorrect results. By *software faults*, we mean software design/implementation defects (e.g., incorrect initialization of a variable or failure to check a boundary condition), and they may change software states to unexpected states. If software data is corrupted by either hardware or software faults, we call them *software errors*.

At least two issues need to be addressed for software fault injections. The first is that when a fault is injected to a memory location or a register, who owns the memory location or which process is running on the processor. In other words, what is the target of the fault injection? The second issue is what fault models should be used to simulate hardware and software faults. We have discussed hardware fault models at function level in Section 3.3. Like hardware models, software models should be built based on engineer experience and field measurements.

Several fault models and implementation techniques are listed in Table 4.2. All these techniques are similar in that they change program or memory words. To inject software faults, the text segment needs to be modified. Some typical software faults are: a variable is used before it is initialized; a module's interface is defined or used incorrectly; statements are in the wrong order or omitted [Sullivan91]. As a result of executing faulty software code, the data segment may be corrupted, causing software errors. Software errors can also be directly injected by changing the data segment.

When the software approach is used to emulate hardware faults, the faults are normally of transient nature. For example, the faulty bits in memory or CPU registers can be overwritten by subsequent instructions. However, the

Table 4.2. Techniques Used for Software Fault Injection

Type	Method
Software Fault	Modify the text segment of the program.
Software Error	Modify the data segment of the program.
Memory Fault	Flip memory bits of the program.
CPU Fault	Use a trap to modify the memory area of the saved CPU registers.
Bus Fault	Use traps before and after an instruction to change the instruction or data used by the instruction and then restore them after the instruction is executed.
Network Faults	Modify or delete transmission messages.

software approach can be used to emulate permanent faults by repeatedly injecting the same fault to a location whenever there is an access to the location. For example, to emulate a permanent stuck-at-0 fault at a particular bit in a memory word, the bit is changed to 0 after every write operation to the word. To emulate a permanent stuck-at-1 fault at a bus address line, the corresponding bit in the effective address (in the program counter or in a CPU register) is set to one before any access to the bus. It is obvious that the emulation is expensive, involving the monitoring and execution of many extra instructions.

Unlike hardware-implemented fault injections which are difficult to gear toward specific workload areas, software fault injections can be targetted toward user applications, the operating system, or both. If the target is user applications, the fault injector can be inserted into user applications or can be an extra layer between the user applications and the operating system. If the target is the operating system, the fault injector has to be embedded in the operating system, because it is very difficult to add an extra layer between the machine and the operating system.

Although the software-based approach is very flexible, it has some restrictions. First, the approach cannot inject faults into locations not accessible to software. We have mentioned in Section 3.2 that approximately 1/3 of errors produced in logic-level fault injections cannot be emulated through the software approach [Czeck91]. Secondly, the software instrumentation may disturb the workload running in the target system and even change the structure of original software. A careful design can alleviate the perturbation to the workload. Another disadvantage is the low time resolution of the approach, which may cause fidelity problems. For the long latency faults, such as memory faults, the low time resolution may not be a problem. For the short latency faults, such as bus and CPU faults, the approach may fail to capture the error behavior (e.g., propagation). This problem can be solved by using a hardware monitor, i.e., the hybrid approach [Young92]. The hybrid approach combines the versatility of software-implemented injection and the accuracy of hardware monitoring. It is well suited for measuring extremely short latencies.

There have been several studies using the software-based approach. In [Chillarege89], a *failure acceleration* method is used to inject the *overlay* software faults into an IBM commercial transaction processing system. In the failure acceleration method, fault injections are designed such that the fault/error latency is decreased and the probability of a fault causing a failure is increased. An overlay occurs when a program writes into an incorrect area. It is estimated that about 1/3 of software errors can be mapped into the overlay model [Chillarege89]. The study quantified the

Table 4.3. Comparison of Software-Implemented Fault Injections

Tool	FIAT [Segall88]	FERRARI [Kanawati92]	HYBRID [Young92]	FINE [Kao93]
Hardware	PC RT	SPARC	Tandem S2	Sun
Injection Target	O.S. User	User	O.S. User	O.S. User
Monitor	Software	Software	Hybrid	Software
Fault types	Memory CPU Communication	Memory CPU Bus Control flow	Memory CPU Cache	Memory CPU Bus Software
To evaluate	Detection Latency Recovery	Detection Latency	Detection Latency Recovery	Detection Propagation

immediate impact and potential hazards (which may cause a catastrophic failure in the future) of the injected faults.

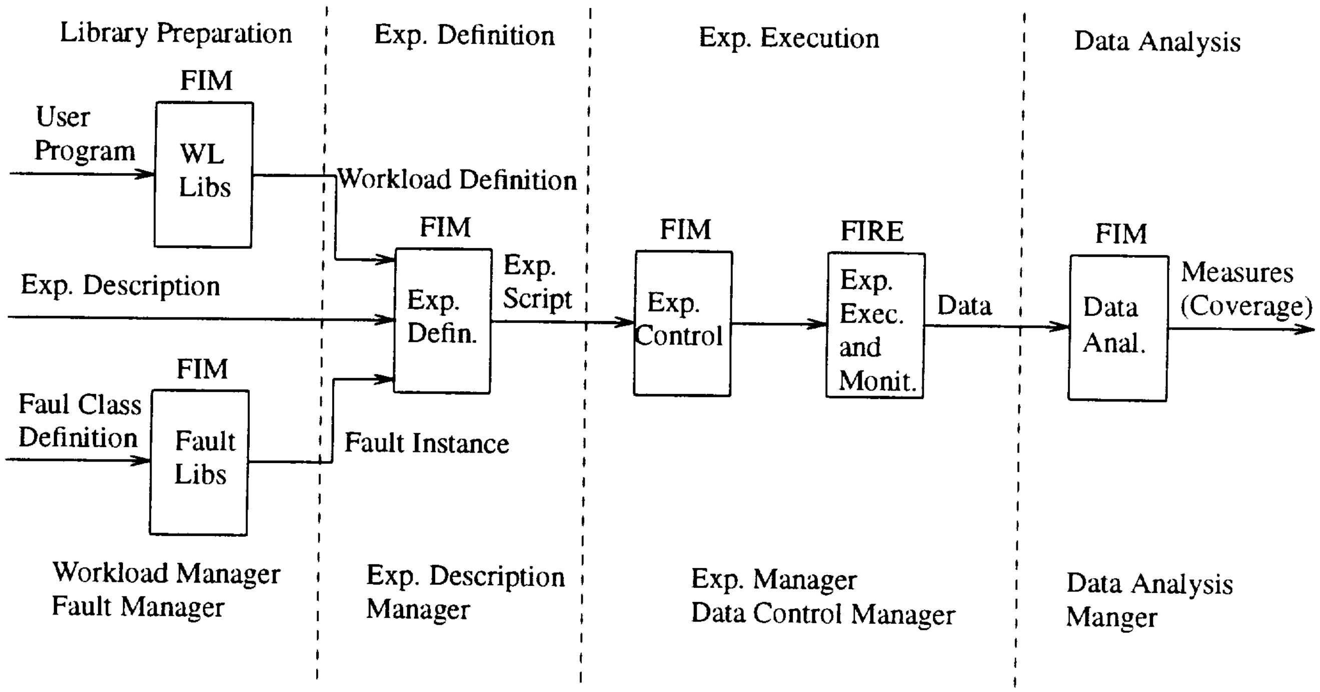
In recent years, interest in developing software-implemented fault injection tools has increased. Several environments have been published in literature: FIAT [Segall88], FERRARI [Kanawati92], HYBRID [Young92], and FINE [Kao93]. Table 4.3 lists features of these tools, which will be discussed in the following subsections.

4.2.1. FIAT

A number of fault injection studies at Canegie Mellon University centered around FIAT (Fault Injection Automated Testing), a software-implemented fault injection environment [Segall88], [Barton90], [Czeck91]. The FIAT hardware implementation consists of IBM RT PCs connected by a token ring network. The FIAT software structure is divided into two parts: Fault Injection Manager (FIM) and Fault Injection REceptor (FIRE). FIM is a global control program responsible for all phases of the experiment. FIRE, under the control of FIM, collects the experimental results and sends appropriate information to FIM for off-line analysis. Figure 4.4 shows the process of a typical fault injection experiment.

FIAT has been used to study the impact of faults on the application workload level [Barton90]. Two representative programs, a matrix multiplication task and a selection sort task, were chosen as application workloads. To achieve fault tolerance, each task is executed on two different processors and results are compared. Three fault types

Figure 4.4. Typical Fault Injection Experiment in FIAT



were injected in the experiment: zero-a-byte, set-a-byte, and two-bit compensating. The zero-a-byte or set-a-byte sets a consecutive 8 bits anywhere within a 32-bit word to zero or one. The 2-bit compensating complements 2 bits in a word such that the parity code would not detect it as an error. Faults were injected into all locations within a workload, with a total of over 130,000 faults injected.

Results showed that there are a limited number of system-level fault manifestations. The mean error detection coverage for different workloads and fault types is around 50% to 60%. Error detection latency was found to follow a normal distribution. This result conflicts with those presented in [Shin86], [Finelli87], where the latency was shown to follow either gamma, Weibull, or log-normal distributions. This difference may be explained by the differences in the experimental environment and detection mechanisms. In [Shin86], [Finelli87], the hardware-implemented fault injection technique is used, and the resolution of detection time is on the order of milliseconds, while the time resolution of the software-implemented FAIT is on the order of seconds, which may skew the results.

4.2.2. FERRARI

FERRARI (Fault and ERRor Automatic Real-time Injector), another software-implemented fault injection environment, was recently developed at the University of Texas at Austin [Kanawati92]. The purpose of the development of FERRARI was to evaluate complex systems by emulating most hardware faults in software. It was implemented on SPARC workstations in an X-window environment. FERRARI consists of four software modules: 1) the *initializer and activator*, 2) the *user information*, 3) the *fault and error injector*, and 4) the *collector and analyzer*. These four modules are controlled by the *manager* module which coordinates the operation of the four modules.

The initialization and activation module prepares the target program for fault injection by extracting its information, such as the starting address, the program size, and the execution time. The user information module receives experiment parameters provided by the user, such as experiment mode, fault and error types, and dependability measures to obtain. The fault and error injection module is responsible for injecting different types of transient or permanent faults, such as address line fault, data line fault, and fault in condition code flags. The data collection and analysis module records experiment results, such as information about error detection, error latency, and failures, and it determines statistics of these measures at the end of the experiment.

To demonstrate the capabilities of FERRARI and to study the behavior of the target system under faulty conditions, over 600,000 fault injection runs were conducted on SUN4 SPARC workstations under different applications. Results showed that the error coverage is highly dependent on the fault type. The highest coverage was obtained when errors were injected in the task memory image. This is because the injected errors are likely to be exercised repeatedly if the corrupted instructions are in a loop. An important finding is that a considerable number of undetected errors are those that corrupted input/output routines and system libraries. These routines may tend to be ignored when error detection techniques are embedded in the user code.

4.2.3. HYBRID

A major drawback of the above purely software-implemented fault injection environments is the low resolution of detection time. If the error detection mechanism is implemented with hardware, the time resolution is greatly enhanced. This approach is used in the hybrid fault injection environment developed at the University of Illinois at

Urbana-Champaign [Young92]. The hybrid environment combines the versatility of software injection and the accuracy of hardware monitoring. It is well suited for measuring extremely short error latencies, and the introduced overhead is minimal so that error propagation and control flow are not significantly affected by the presence of instrumentation.

In the hybrid environment, faults are injected via software, and the impact is measured by both software and hardware. Figure 4.5 illustrates the subsystems that make up the environment. It consists of a fault injection system, a hybrid monitor system to measure the effects of injected faults, and a supervisory system to automate the measurements. The hybrid monitor system is further divided into a hardware monitor and a software monitor. Figure 4.6 illustrates how these systems are physically situated. The *fault injector* and *software monitor* execute on the *test system*, while the *supervisor program* executes on the *control host*. Probes attach the hardware monitor to the address/data backplane of the test system so that the monitor can analyze and record the signals generated. Communication between the supervisor and the hardware monitor takes place over an RS-232 or GPIB connection.

Figure 4.5. Hybrid Fault Injection Environment

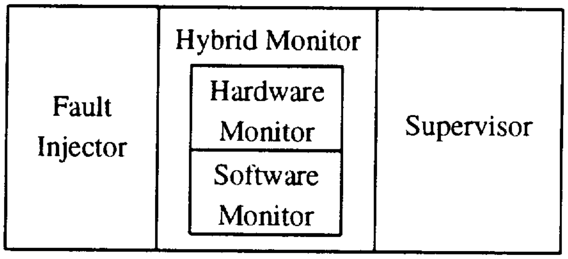
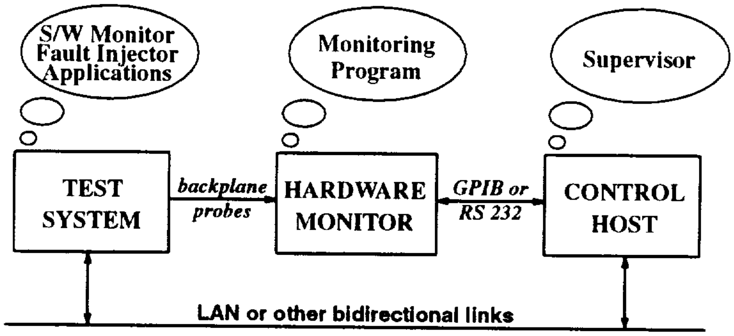


Figure 4.6. Physical Layout of Hybrid Fault Injection System



The function of the environment is to perform experiments that repeatedly inject faults and record observations. The environment introduces faults into the test system during the execution of a *target program*, measures the effects of that fault, and returns the test system to conditions present prior to fault injection. These operations form a single *observation loop*. Faults can be injected into any location that has a physical address, e.g., CPU registers, cache, local memory, mass storage, and network controllers. Faults can also be injected into locations allocated to a single, executing user program or even into the kernel, and propagation can be characterized down to the instruction level.

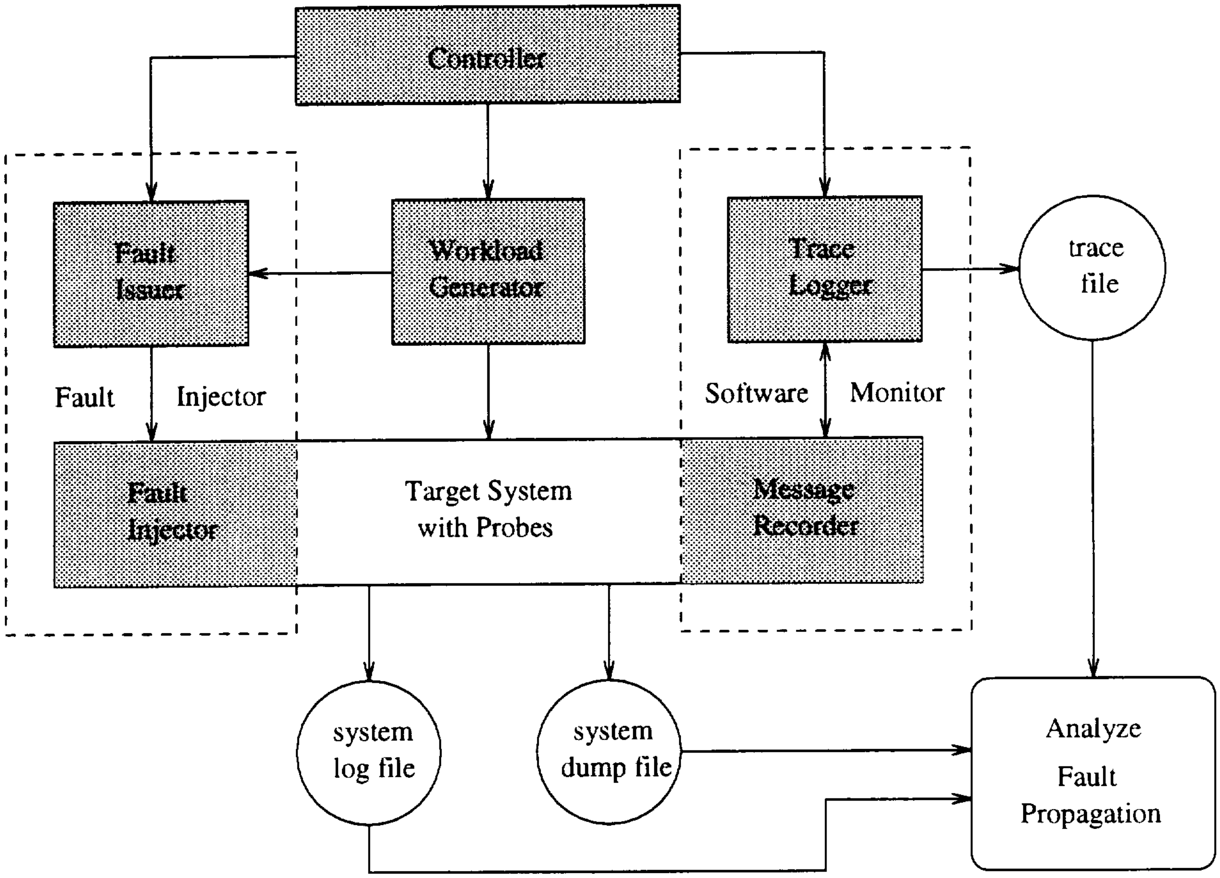
The fault injection environment was used to study dependability characteristics of a Tandem Integrity S2 fault tolerant computer system [Jewett91]. High degrees of accuracy in measuring latency (within 20ns) were obtained. Measurements of the sensitivity of different instructions to faults indicated a 5% chance that a faulted MIPS RISC instruction will not fail when executed. Modeling of multi-level error propagation showed that error detections were due to multiple corruptions of state in as many as 57% of reads from wrong addresses and 37% of writes to wrong addresses. The median latency associated with error detection by an individual CPU was on the order of 10 μs, and the median delay between detection and the start of CPU shutdown was on the order of 100ms. Kernel fault injection studies show that a fault in the kernel is 2.6 times as likely to bring down a CPU as a fault elsewhere.

4.2.4. FINE

FINE is a UNIX-based fault injection environment developed at the University of Illinois at Urbana-Champaign [Kao93]. The significance of FINE is twofold. First, it is the first tool that can inject software faults as well as hardware errors. Second, it is the first tool built for tracing fault propagation among software modules. The software faults that can be injected by FINE include initialization (missing or incorrect), assignment (missing or incorrect), condition check (missing or incorrect), and function (incorrect) faults. FINE can also inject hardware errors such as CPU (ALU, shifter, opcode decoder, or registers), memory (text segment or data segment), and bus errors (address lines or data lines).

Figure 4.7 shows the FINE environment. FINE consists of a *fault injector*, a *software monitor*, a *workload generator*, a *controller*, and several *analysis utilities*. The fault injector and software monitor are embedded in the kernel so that faults can be injected there and their propagation can be monitored. Fault injection is implemented by

Figure 4.7. The FINE Environment



modifying the system trap handling routines, so the fault injector can be considered an extra layer between the operating system and the machine. The software monitor traces the execution flow and key variables of the kernel. Software probes are inserted into functions in the kernel to record the execution flow and the values of arguments and key variables. The synthetic workload generator issues various system calls to activate injected faults. The distribution of generated system calls can be specified by users to emulate real workloads or to deliberately accelerate the activation of injected faults. The controller assigns experiment specifications to the fault injector and the monitor, and it initiates experiments. The analysis utilities provide assistance in analyzing fault propagation. The target of the study is the UNIX kernel, a non-stopped, highly parameterized, complex service program with high impact and a broad spectrum of workloads.

Experiments on SunOS 4.1.2 (on a SPARCstation IPC) were conducted by applying FINE to investigate fault propagation and to evaluate the impact of various types of faults. Results showed that memory faults and software faults usually have a very long latency, while bus faults and CPU faults tend to crash the system immediately. Nearly 90% of detected errors are detected by hardware. About half (47%) of the detected errors are data errors. these data errors are detected when the system tries to access an area it has no privilege to access. In the software fault propagation, incorrect control flow is the major impact for the first level of propagation, while data corruption is the major impact for the subsequent propagation. Analysis of fault propagation among the UNIX subsystems revealed that only about 8% of faults propagate to other UNIX subsystems.

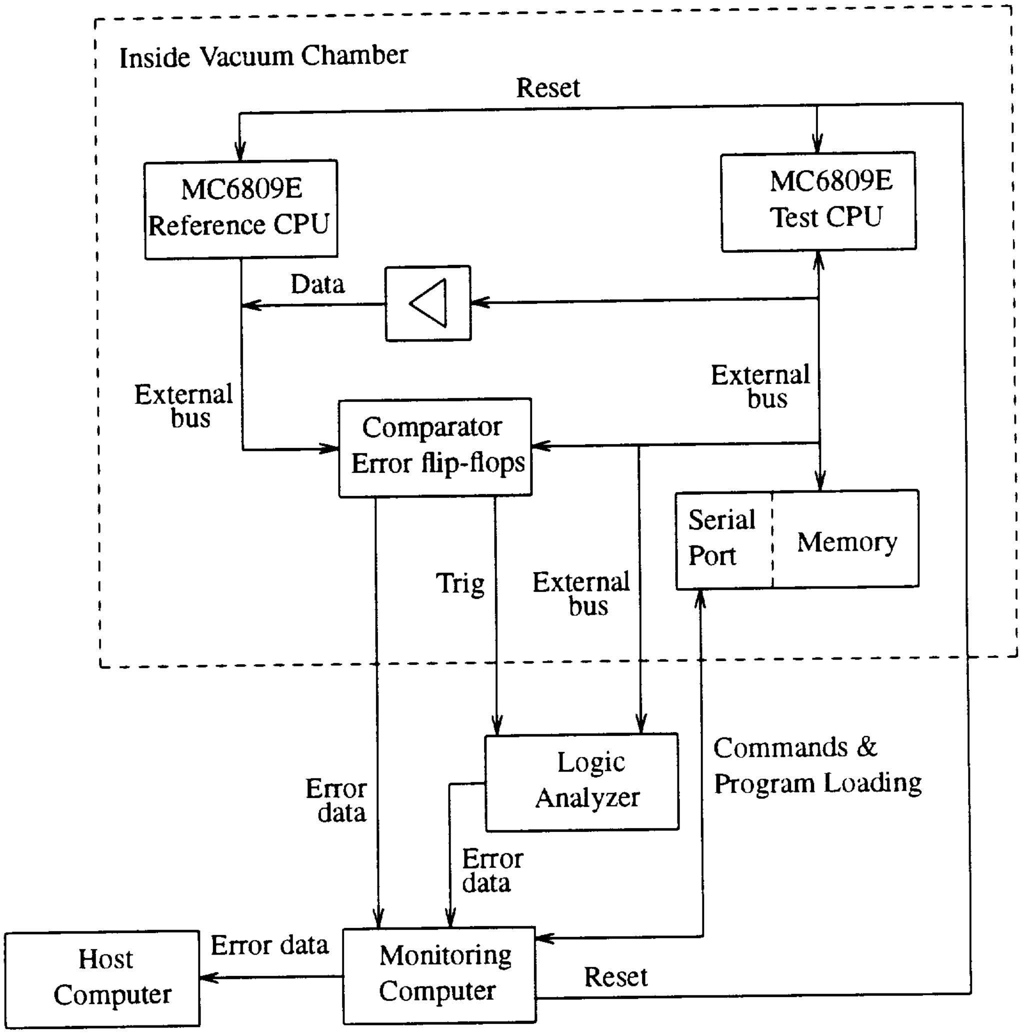
4.3. Radiation-Induced Fault Injection

Neither hardware-implemented nor software-implemented fault injections have a way to produce transient faults at random locations inside ICs. Radiation-induced fault injections provide such a capability. One way to do this is to expose the chip to the heavy-ion radiation from a *Californium*^{252} (Cf^{252}) source [Gunneflo89], [Karlsson89]. The heavy ions emitted from the source are capable of creating transient faults when they pass through a depletion region in the IC. One advantage of this method is that it can produce transient faults at random locations evenly and can cause either a single bit flip or multiple bit flips. This leads to large variation in the errors seen on the output pins of the IC.

In the fault injection experiments reported in [Gunneflo89], [Karlsson89], the Cf^{252} method was used to investigate error coverage and detection latency for error detection schemes for the MC6809E 8-bit microprocessor. The intention of the experiments was to characterize the effects of transient faults that originate inside a CPU. The MC6809E is fabricated in NMOS, a technology sensitive to heavy ion radiation. The error detection schemes under study are suitable for implementation with a watchdog processor that checks the behavior of the main processor on the external bus. The developed experimental system is called FIST (Fault Injection system for Study of Transient fault effects). Figure 4.8 shows the FIST diagram.

The heavy-ion radiation is implemented by using a commercially available 37×10^3 Becquerel (1 μC_i) Cf^{252} source. The Cf^{252} source is mounted inside a vacuum chamber together with a small computer system. One of the system boards is placed on a mechanical fixture movable in three dimensions for accurate positioning of the CPU beneath the Cf^{252} source. The system has two MC6809E CPUs which operate synchronously using the same clock. One CPU

Figure 4.8. FIST Diagram



is exposed to heavy-ion radiation. The other is used as a reference to detect errors via comparison on the output from the two CPUs. When errors are detected by the comparison logic, the logic analyzer is triggered to record the external bus signals. The monitoring computer is responsible for data acquisition and control of experiments.

A fault injection experiment is conducted in the following way. Before the experiment starts, the monitoring computer fetches from the host computer a load file which contains the test program to be executed. The test program is then loaded from the monitoring computer to the MC6809E system. After the loading, the test program is started with a "go" command from the monitoring computer. When a mismatch is detected, the monitoring computer fetches the recorded error data from the logic analyzer and the error flip-flops in the MC6809E system and transfers them to the host computer. Finally, the MC6809E system is reset, and the test program is reloaded for the next experiment.

It was found from fault injection experiments that 78% of all errors affected control flow (i.e., caused the processor to diverge from the correct sequence) and 17% caused errors in data. Results also showed that 30% of all errors were multiple bit errors on the output pins, although the origin of each of these errors was only one single heavy ion. The error recordings obtained from the experiments were also used as input to simulation models of different error detection mechanisms to evaluate these error detection mechanisms without implementing them. The coverage of several detection mechanisms was investigated. It was found that the best mechanism was the one that detects access to the memory outside permitted areas and that the combination of two mechanisms gave a better coverage than any one mechanism alone. It was also found that the type of the test program had a considerable influence on the results of error detection mechanisms.

V. OPERATIONAL PHASE

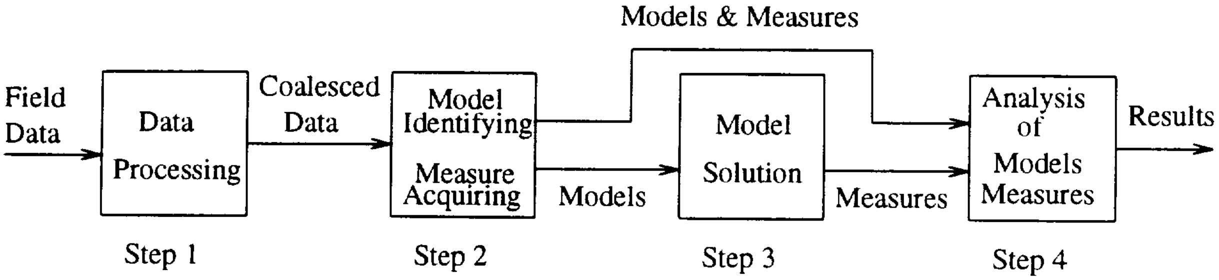
When a computer system is in normal operation, various errors occur in both hardware and software. There are many possible sources of errors, including untested manufacturing faults and software defects, wearing out of devices, transient errors induced by radiation, power surges, or other physical processes, operator errors, and environmental factors. The occurrence of errors is also highly dependent on workloads running in the system. A distribution of operational outages from various error sources for several major commercial systems are reported in [Siewiorek92].

To understand dependability characteristics of a complex computer system, there is no better way than measuring real systems and analyzing the measured data. Here, measuring real systems means monitoring and recording naturely occurring errors and failures in the system while it is running under user workloads. Analysis on such measurements can provide valuable information on actual error/failure behavior, identify system bottlenecks, obtain dependability measures, and verify assumptions made in analytical models.

Given field error data collected from a real system, a measurement-based study consists of four steps, shown in Figure 5.1: 1) data processing, 2) model identification and measure acquisition, 3) model solution if necessary, and 4) analysis of models and measures. Step 1 consists of extracting necessary information from field data (the result can be a form of compressed data, or flat data), classifying errors and failures, and coalescing repeated error reports. In a computer system, a single problem commonly results in many repeated error observations occurring in rapid succession. To ensure that the analysis is not biased by repeated observations of the same problem, all error entries which have the same error type and occur within a short time interval (e.g., 5 minutes) of each other should be coalesced in data processing. Thus, the output of this step is a form of coalesced data in which errors and failures are identified. This step is highly dependent on the measured system. Coalescing algorithms have been proposed in [Tsao83], [Iyer86], [Hansen92].

Step 2 includes identifying appropriate models (such as Markov models) and acquiring measures (such as MTBFs and TBF distributions) from the coalesced data. Several models have been proposed and validated using real data, such as the workload-dependent cyclostationary model in [Castillo81], the workload hazard model in [Iyer82a], and the correlation models in [Tang92a]. Statistical analysis packages such as SAS [SAS85] or measurement-based dependability analysis tools such as MEASURE+ [Tang93b] can be used to perform this analysis. Step 3 solves these

Figure 5.1. Measurement-Based Analysis



models to obtain some other measures (such as reliability and transient reward rates). Dependability and performance modeling and evaluation tools such as SHARPE [Sahner87] can be used in this step. The most creative part is step 4, the human analysis of models and measures obtained from data. New results are produced in this phase. For example, reliability bottlenecks can be identified from analysis of error/failure statistics, and workload/failure dependency can be concluded by analysis of models. However, analysis methods may vary significantly from one study to another, depending on research goals.

Measurement-based dependability analysis of operational systems has evolved significantly over the past 15 years. These studies addressed one or more of the following issues: basic error characteristics, dependency analysis, modeling and evaluation, software dependability, and fault diagnosis. The following paragraphs give a brief overview of these studies, which are listed in Table 5.1.

Early studies in this field investigated transient errors in DEC computer systems and found that more than 95% of all detected errors are intermittent or transient errors [Siewiorek78], [McConnel79]. The studies also showed that the inter-arrival time of transient errors follows a Weibull distribution with a decreasing error rate. This distribution was later shown to fit the software failure data collected from an IBM operating system [Iyer85b]. A recent study of failure data from three different operating systems showed that TTE (time to error) can be represented by a multi-stage gamma distribution for the measured single-machine operating system and by hyperexponential distributions for the measured distributed operating systems [Lee93a].

Studies of dependency between workload and failure in early 1980s, based on measurements from IBM [Butner80] and DEC [Castillo81] machines, revealed that the average system failure rate is strongly correlated with the

Table 5.1. Measurement-Based Studies of Computer System Dependability

Category	Issues	Studies
Data Coalescing	Analysis of time-based tuples Clustering based on type and time	[Tsao83], [Hansen92] [Iyer86], [Lee91], [Tang93a]
Basic Error Characteristics	Transient faults/errors Error/failure bursts TTE/TTF distributions	[Siewiorek78], [McConnel79], [Iyer86] [Iyer86], [Hsueh87], [Tang93a] [McConnel79], [Iyer85b], [Lee93a]
Dependency Analysis	Hardware failure/workload dependency Software failure/workload dependency Correlated failures and impact Two-way and multi-way failure dependency	[Butner80], [Castillo81], [Iyer82a] [Castillo82], [Iyer85b], [Mourad87] [Tang90], [Wein90], [Tang92a] [Dugan91], [Lee91], [Tang91]
Modeling and Evaluation	Performability model for single machine Markov reward model for distributed system Two-level models for operating systems	[Hsueh88] [Tang93a] [Lee93a]
Software Dependability	Error recovery Hardware-related & correlated software errors Software fault tolerance Software defect classification	[Velardi84], [Hsueh87] [Iyer85a], [Tang92b], [Lee93a] [Gray90], [Lee92], [Lee93b] [Sullivan91], [Sullivan92]
Fault Diagnosis	Heuristic trend analysis Statistical analysis of symptoms Network fault signature	[Tsao83], [Lin90] [Iyer90] [Maxion90a], [Maxion90b]

average workload on the system, The effect of workload-imposed stress on software was investigated in [Castillo82] and [Iyer85b]. Recent analyses of DEC [Tang90], [Wein90] and Tandem [Lee91] multicomputer systems showed that correlated failures across processors are not negligible, and their impact on availability and reliability are significant [Dugan91], [Tang91], [Tang92a].

In [Hsueh88], analytical modeling and measurements were combined to develop measurement-based reliability/performability models using data collected from an IBM mainframe. The results showed that a semi-Markov process is better than a Markov process for modeling system behavior. Markov reward modeling techniques were further applied to distributed systems [Tang93a] and fault tolerant systems [Lee92], to quantify performance loss due to errors/failures for both hardware and software. A census of Tandem system availability indicated that software faults are the major source of system outages in the measured fault tolerant systems [Gray90]. Analyses of field data from different software systems investigated several dependability issues including the effectiveness of error recovery [Velardi84], hardware-related software errors [Iyer85a], correlated software errors in distributed systems [Tang92b], software fault tolerance [Lee92], [Lee93b], and software defect classification [Sullivan91], [Sullivan92]. Measurement-based fault diagnosis and failure prediction issues were investigated in [Tsao83], [Iyer90], [Lin90], [Maxion90a], [Maxion90b].

In the following subsections, we discuss issues and representative studies involved in measurements, data processing, preliminary analysis of data, dependency analysis, modeling and evaluation, software dependability, and fault diagnosis.

5.1. Measurements

There are numerous theoretical and practical difficulties associated with making measurements. The question of what and how to measure is a difficult one. A combination of installed and custom instrumentation has been used in most studies. From a statistical point of view, sound evaluations require a considerable amount of data. In modern computer systems, especially in fault tolerant systems, failures are rare. To obtaining meaningful data for such systems, measurements must be made for a considerably long period of time, or sometimes the measured system must be exposed to high-stress conditions.

In an operational system, only detected errors can be measured, because an error is known only if it is detected. There are basically two ways to make measurements: on-line automatic logging and human manual logging. Many large computer systems such as IBM and DEC mainframes provide error-logging software in the operating system. The software records error reports from different subsystems, such as memory or disk subsystems, and other system events, such as reboots and shutdowns. The reports usually include information about the location, time, and type of the error, the system state at the error time, and sometimes error recovery (e.g., retry) information. The recorded reports are stored in a permanent system file chronologically. The main advantage of the on-line automatic logging is its ability to record a large amount of information about transient errors and to provide details of automatic error recovery processes, which cannot be done manually. Disadvantages are that information can be lost when a system fails too quickly for error messages to be recorded, and that an on-line log does not include information about the cause and propagation of the error or about off-line diagnosis.

Table 5.2 shows a sample of extracted error logs from a VAXcluster multicomputer system. Often, meanings of a record in the logs can differ between versions of the operating system and between machine models. Error detection and recording routines may be written and modified over time by different people. For example, a careful study of VAX error logs and discussion with the field engineers indicate that the operating system on different VAX machine models may report the same type of error into different categories. Thus, it is necessary to distinguish these errors in the subsequent error classification (to be discussed in Section 5.2).

Table 5.2. A Sample of Extracted Error Logs from a VAXcluster†

Entry	System ID	Logging Time	Subsystem & Unit	Interpretation
5815	Earth	20-DEC-1987 20:23:13.22	I/O, HO$DUA51:	Disk drive error
7005	Earth	4-JAN-1988 11:45:07.12	I/O, H3$MUA1:	Tape drive error
12979	Europa	8-JAN-1988 14:14:28.63	CI, EUR$PAA0:	Path #0 went from good to bad
13005	Europa	8-JAN-1988 16:23:17.41	CI, EUR$PAA0:	Error logging datagram received
13734	Europa	19-JAN-1988 17:31:30.74	CI, EUR$PAA0:	Virtual circuit timeout
3260	Mercury	24-DEC-1987 04:54:52.06	Memory, TR #2	Corrected memory error
10939	Jupiter	1-APR-1988 09:57:39.40	Unknown Device	
14209	Jupiter	16-MAY-1988 13:37:04.97	CPU, SBI	Unexpected read data fault
13941	Mars	25-FEB-1988 02:13:20.25	CPU, IBOX	Machine check
20937	Mars	18-APR-1988 16:46:39.75	BugCheck	Bad memory deallocation request size or address
27958	Mars	14-MAY-1988 20:57:46.48	BugCheck	Insufficient nonpaged pool to remaster locks
37790	Saturn	20-JUL-1988 18:51:49.15	BugCheck	Unexpected system service exception

† The sample is intended to illustrate the different types of errors logged. Therefore, the entry numbers are not consecutive.

Since the information provided by on-line error logs may not be complete, it is valuable to have operator logs compensate the missing information in on-line logs. Whenever possible, measurements should include both on-line and operator logs. A good operator log should include information about failure diagnosis, component replacement, hardware and software update, etc. It is not easy to maintain an accurate and complete operator log. Unremitting efforts must be made for a substantial period in obtaining measurements.

5.2. Data Processing

Usually, on-line logs contain a large amount of redundant and irrelevant information in various formats. Thus, data processing must be performed to obtain useful, classified information and put it into a flat format that will facilitate the subsequent analyses. The first step of data processing is *error classification*, which classifies errors in the measured system into a number of types based on the subsystems and components in which they occur. There is no uniform error classification, because different systems have different hardware and software architectures. But some

Table 5.3. Major Error Types in VAXcluster

System	Type	Description
Hardware	CPU	CPU or bus controller errors
	Memory	Memory ECC errors
	Disk	Disk, drive, and controller errors
	Network	Local network and controller errors
Software	Control	Problems involving program flow control or synchronization
	Memory	Problems referring to memory management or usage
	I/O	Inconsistent conditions detected by I/O management routines

error types, such as CPU, memory, and disk errors, are seen in most systems. Table 5.3 lists an error classification (major error types) for VAXcluster systems [Tang92b], [Tang93a].

After error classification, the following data processing can be broadly divided into two steps: *data extraction* and *data coalescing*. Data extraction selects useful entries such as error and reboot reports (throwing away useless entries such as disk volume change reports) from the log file and transforms them into a flat format. The design of the flat format depends on the necessity of the subsequent analyses. The following is a possible format:

entry number	logging time	error type	device id.	other fields

In on-line error logs, a single fault in the system can result in many repeated error reports in a short period of time. To ensure that the subsequent analyses will not be biased by these repeated reports, entries which correspond to the same problem should be coalesced into a single event. A commonly used coalescing algorithm [Iyer86] is merging all error entries which have the same error type and occur within a ΔT interval of each other into a *tuple*. The algorithm is as follows:

```
IF <error type> = <type of previous error>
AND <time away from previous error> $\leq \Delta T$
    THEN <put error into the tuple being built>
    ELSE <start a new tuple>
```

A tuple reflects the occurrence of one or more errors of the same type in rapid succession and can be represented by a record containing at least the following fields [Tsao83], [Tang93b]:

(1) tuple_id — identification of the tuple

(2) no_entry — number of error entries in the tuple

(3) start_time — logging time of the first entry in the tuple

(4) end_time — logging time of the last entry in the tuple

(5) err_type — error type of the tuple

Different systems may need different time intervals in data coalescing. A recent study on this issue [Hansen92] defined two types of mistakes that can be made in data coalescing: *collision* and *truncation*. A collision occurs when the detection times of two faults are close enough (within ΔT) such that they are combined into a tuple. A truncation occurs when the time between two reports caused by a single fault is greater than ΔT such that the two reports are split into different tuples. If ΔT is large, collisions are likely to occur. If ΔT is small, truncations are likely to occur. The study found that there is a threshold of time intervals beyond which collisions are rapidly increased. Based on this observation, the study proposed a statistical models which can be used to select an appropriate time interval to reduce collisions. According to our experience, collision is not a big problem if the error type and device information is used in data coalescing as shown in the above coalescing algorithm. Truncation is usually not considered to be a problem [Hansen92]. There are techniques [Iyer90], [Lin90] which deal with this problem and which are used for fault diagnosis and failure prediction (to be discussed in Section 5.7).

5.3. Preliminary Analysis

Once coalesced data is obtained, basic dependability characteristics of the measured system can be identified by a preliminary statistical analysis. Commonly used measures in the analysis include error/failure frequency, TTE or TTF distribution, and error/failure hazard rate function. In the following discussion, data from a VAXcluster system [Tang93a] is used to illustrate analysis methods.

5.3.1. Basic Statistics

Although it is not difficult, it is important to first obtain basic statistics such as frequency, percentage, and probability from the measured data. These statistics provide a basic picture of the measured system. Often, dependability

Table 5.4. Error/Failure Statistics for the VAXcluster

Category	Error		Failure		Recovery
	Frequency	Percentage	Frequency	Percentage	Probability
I/O	25807	92.87±0.30	105	42.86±6.20	0.996±0.001
Machine	1721	6.19±0.28	5	2.04±1.77	0.970±0.002
Software	69	0.25±0.06	62	25.31±5.44	0.101±0.071
Unknown	191	0.69±0.10	73	29.80±5.73	0.618±0.069
All	27788	100.0	245	100.0	0.991±0.001

bottlenecks can be identified by analysis on the statistics. Table 5.4 shows the error/failure statistics for the measured VAXcluster. In the table, I/O errors include disk, tape, and network errors. Machine errors include CPU and memory errors. Software errors are software-related errors. The 95% confidence intervals for the percentage and probability estimates shown in the table are calculated using the method discussed in Section 2.1 for estimating confidence intervals for proportions. Two bottlenecks can be identified from the table.

First, the major error category is I/O errors (93%), i.e., errors from shared resources. This category of error has a very high recovery probability (0.996). However, these errors still result in nearly 43% of all failures. This result indicates that, although the system is generally robust to the impact of I/O errors, the shared resources still constitute a major reliability bottleneck due to the sheer number of errors. An improvement in such a system may require using an ultra-reliable network and a disk system to reduce the raw error rate, not just providing high recoverability.

Secondly, although software errors constitute only a small part of all errors (0.3%), they result in significant failures (25%). This is because software errors have a very low recovery probability (0.1). This software failure estimation is conservative because there are significant unknown failures (30%). Some of these unknown failures could be attributed to software problems. Thus, software-related problems are severe in the measured system.

5.3.2. Empirical TTE Distributions and Hazard Rates

TTE/TTF probability distributions and error/failure hazard rates are commonly used to investigate how errors and failures occur across time. It is relatively easy to obtain empirical TTE/TTF distributions from data. Figure 5.2 shows the empirical TTE distribution function, $f(t)$, for a measured VAXcluster system [Tang93a]. Notice that the logarithmic coordinate is used for $f(t)$ because of the big contrast between the largest and smallest values. It is seen that about 67% of the TBEs are less than one minute. Most of these instances are "time between errors of two different machines" because errors of the same type occurring within a five minute interval of each other on the same machine have been coalesced into a single error event. This fact implies that errors are likely to occur on the different machines in the measured system within a very short period of time.

The *hazard rate* characterizes error/failure intensity on time series. It can be considered to be the probability that an error (failure) will occur within the coming unit of time, given that no error (failure) has occurred since the start of the system or the last error (failure) occurrence. The mathematical definition of the hazard rate [Ross85] is as follows:

$$h(t) = \frac{Pr\{error\ in\ (t,\ t{+}dt)\}}{Pr\{no\ errors\ in\ (0,\ t)\}\ dt} = \frac{f(t)}{1-F(t)} \qquad (5.1)$$

Figure 5.3 shows the empirical failure hazard rates computed from the VAXcluster failure data. The high hazard rate near the origin, i.e., the high probability that the second failure will occur within a short time after a failure occurrence, indicates that failures in the VAXcluster tend to occur in bursts. The most likely for a second failure is the first two hours after a failure occurrence. Failure bursts have been observed by many studies [Iyer86], [Hsueh87], [Bishop88]. Actually, in an early study of transient errors [McConnel79], the Weibull distribution with a decreasing failure rate identified for the interarrival time of failures caused by transient errors implicated the existence of failure bursts.

Figure 5.2. VAXcluster Empirical TTE Distribution

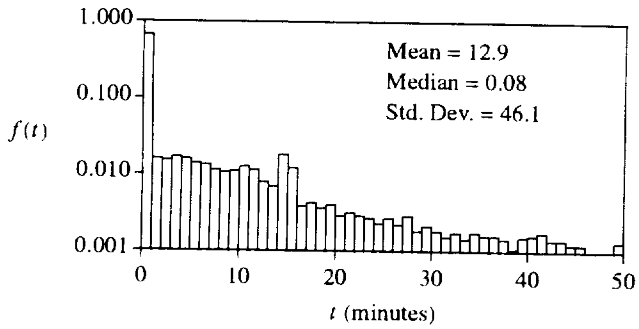
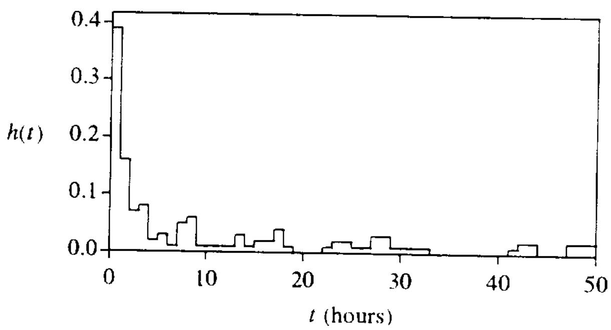


Figure 5.3. VAXcluster Failure Hazard



5.3.3. Analytical TTE Distributions

A realistic, analytical form of TTE distributions is essential in modeling and evaluating computer system dependability. Often, for simplicity or due to lack of information, TTEs are assumed to be exponentially distributed [Arlat90b], [Laprie84]. Early measurement-based studies found that the Weibull distribution with decreasing failure rate is representative of the time between failures (TBF) in a measured DEC computer system [McConnel79] and a measured IBM operating system [Iyer85b]. A recent comparative study of the dependability of the Tandem GUARDIAN, VAX VMS, and IBM MVS operating systems showed that the software TTE in a single machine can be represented by a multi-stage gamma distribution and the software TTE in multicomputers can be represented by a hyperexponential distribution [Lee93a]. In this section, we discuss these two types of distributions.

Before presenting the analytical TTE distributions, we first explain how a TTE distribution is obtained from a multicomputer system, because both measured GUARDIAN and VMS were running on multicomputer systems. In the measured multicomputer systems, all machine members are working in a similar environment and running the same version of the operating system. If the whole system is treated as a single entity in which multiple instances of an operating system are running concurrently, then every software error on all machines can be sequentially ordered and a distribution can be constructed. The constructed TTE distribution reflects the software error characteristics for the whole system. We will call this distribution the *multicomputer software TTE distribution.*

Figure 5.4. IBM MVS Software TTE Distribution

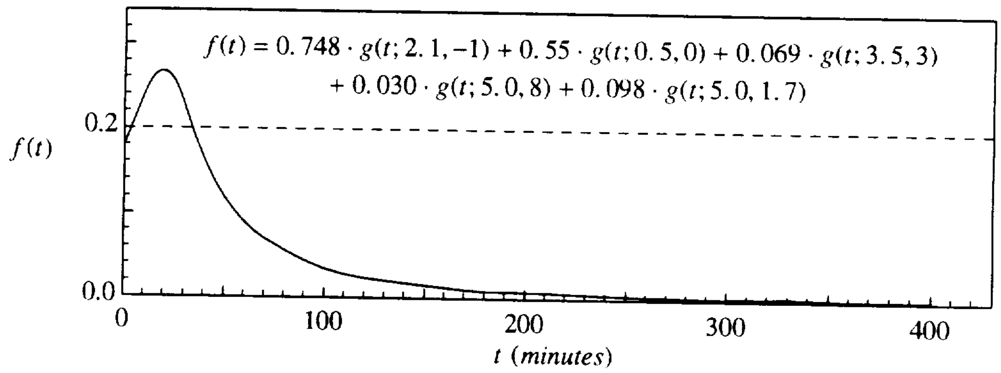


Figure 5.5. VAXcluster Software TTE Distribution

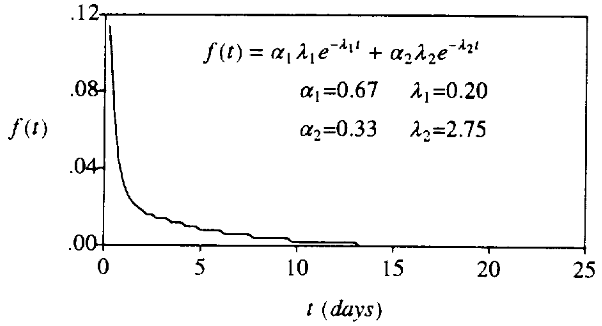
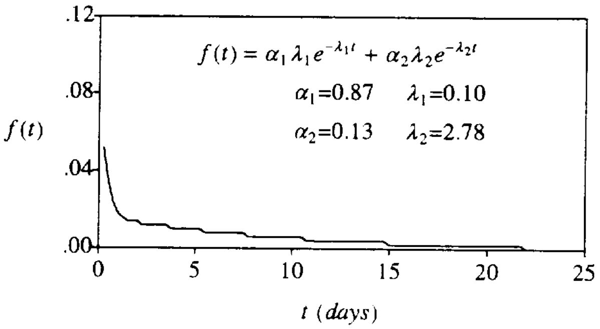


Figure 5.6. Tandem Software TTH Distribution



Figures 5.4 to 5.6 show the analytical TTE or TTH (Time To Halt) distributions fitted using SAS for the three measured systems. All the three empirical distributions failed to fit simple exponential functions. The fitting was tested using the Kolmogorov-Smirnov or Chi-square test (see Section 2.2) at a 0.05 significance level. The two-phase hyperexponential distribution provided satisfactory fits for the VAXcluster and Tandem multicomputer software TTE distributions. An attempt to fit the MVS TTE distribution to a phase-type exponential distribution led to a large number of stages. As a result, the following multi-stage gamma distribution was used:

$$f(t) = \sum_{i=1}^{n} a_i g(t;\, \alpha_i,\, s_i)$$

where $a_i \geq 0$, $\sum_{i=1}^{n} a_i = 1$, and

$$g(t;\, \alpha,\, s) = \begin{cases} 0 & t < s, \\ \dfrac{1}{\Gamma(\alpha)} (t - s)^{\alpha - 1} e^{-(t-s)} & t \geq s\,. \end{cases}$$

It was found that a 5-stage gamma distribution provided a satisfactory fit.

Figures 5.5 and 5.6 show that the multicomputer software TTE distribution can be modeled as a probabilistic combination of two exponential random variables, indicating that there are two dominant error modes. The higher error rate, λ_2, with occurrence probability α_2, captures both the error bursts (multiple errors occurring on the same operating system within a short period of time) and concurrent errors (multiple errors on different instances of an operating system within a short period of time) on these systems. The lower error rate, λ_1, with occurrence probability α_1, captures regular errors and provides an inter-burst error rate.

Error bursts can be explained as repeated occurrences of the same software problem or as multiple effects of an intermittent hardware fault on the software. Actually, software error bursts have been observed in laboratory experiments reported in [Bishop88]. The study showed that, if the input sequences of the software under investigation are correlated (rather than independent), one can expect more "bunching" of failures than those predicted using a constant failure rate assumption. In an operating system, input sequences (user requests) are highly likely to be correlated. Hence, a defect area can be triggered repeatedly.

5.4. Dependency Analysis

Many underlying dependencies exist among measured parameters and components, such as the dependency between workload and failure rate and the dependency among failures on different components. Understanding such dependency is important for improving system dependability and developing realistic models. In this regard, the workload/failure dependency issue was studied in the early 1980s and the correlated failure issue was investigated recently.

Dependency between workload and failure was addressed in two approaches: *statistical quantification* of the dependence between workload and failure rate [Butner80], [Iyer85b] and *stochastic modeling* of failures as functions of workload [Castillo81]. Both demonstrated the strong correlation between workload and failure rate. This result indicated that dependability models cannot be considered representative unless the system workload is taken into account. Based on this result, several workload-dependent analytical models have been proposed [MeyerJ88], [Aupperle89], [Dunkel90].

Recent measurements on VAXclusters [Tang90], [Wein90] and Tandem machines [Lee91] found that correlated failures are not negligible in distributed systems. Further studies showed that even a small correlation can have big impact on system dependability [Dugan91], [Tang91], [Tang92a]. It was also shown that neither traditional models assuming failure independence nor those few models believed to take correlation into account are representative of the actual occurrence process of correlated failures observed in the measured systems [Tang93b].

In the following three subsections, dependency analysis is illustrated through three examples: 1) using a workload hazard model to analyze the dependency between workload and software failures in an IBM 3081 system, 2) using the correlation analysis method to analyze the two-way dependency between errors on two different machines in a VAXcluster system, and 3) using the factor analysis method to analyze the multi-way dependency among failures on multiple processors in a Tandem fault tolerant system.

5.4.1. Workload/Failure Dependency

An early study [Castillo81] introduced a workload-dependent cyclostationary model to characterize system failure processes. The basic assumption is that the instantaneous failure rate of a system resource can be approximated by a function of the usage of the resource considered. The model was applied to a PDP-10 machine running a modified version of the standard TOPS-10 operating system. It was shown that the TTF distribution predicted by the model and the one observed from the real system have an extremely good fit.

In [Iyer82a], a *load hazard* model was introduced to measure the risk of a failure as the system activity increases. The proposed model is similar to the hazard rate defined in Eq (5.1). Given a workload variable X, the load hazard is defined as

$$z(x) = \frac{Pr[\mathit{failure\ in\ load\ interval}\ (x,\ x+\Delta x)]}{Pr[\mathit{no\ \ failure\ in\ load\ interval}\ (0,\ x)]\ \Delta x} = \frac{g(x)}{1-G(x)} \tag{5.2}$$

where $g(x)$ is the p.d.f. of the variable "a failure occurs at a given workload value x" and $G(x)$ is the corresponding c.d.f. That is,

$$g(x) = Pr[\mathit{failure\ occurs} \,|\, X = x] = \frac{f(x)}{l(x)} \tag{5.3}$$

where $l(x)$ is simply the p.d.f. of the workload in consideration:

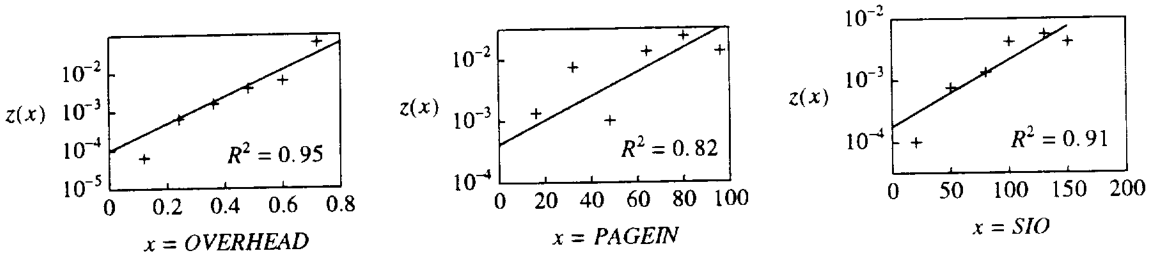
$$l(x) = Pr[X = x]\ , \tag{5.4}$$

and $f(x)$ is the joint p.d.f. of the system state (failure state or non-failure state) and the workload:

$$f(x) = Pr[\mathit{failure\ occurs}\ \&\ X = x]. \tag{5.5}$$

A constant hazard rate implies that failures are occurring randomly with respect to the workload. An increasing hazard rate on the increase of X implies that there is an increasing failure rate with increasing workload.

Figure 5.7. Workload Hazard Plots for the IBM 3081 System



The load hazard model was applied to the software failure and workload data collected from an IBM 3081 system running the VM operating system. Based on the collected data, $l(x)$, $f(x)$, $g(x)$, and $z(x)$ were computed for each workload variable. Figure 5.7 shows the $z(x)$ plots for three selected workload variables:

(1) OVERHEAD — fraction of CPU time spent on the operating system;

(2) PAGEIN — number of page reads per second by all users;

(3) SIO (Start I/O) — number of input/output operations per second.

The regression coefficient, R^2, which is an effective measure of the goodness of fit, is also provided in the figure.

The hazard plots show that the workload parameters appear to be acting as stress factors, i.e., the failure rate increases as the workload increases. The effect is particularly strong in the case of the interactive workload measures OVERHEAD and SIO. The correlation coefficients of 0.95 and 0.91 show that the failure closely fit an increasing load hazard model. The risk of a failure also increases with increased PAGEIN, although at a somewhat lower correlation (0.82). Note that the vertical scale on these plots is logarithmic, indicating that the relationship between the load hazard $z(x)$ and the workload variable is exponential, i.e., the risk of a software failure increases exponentially with increasing workload.

5.4.2. Two-Way Dependency

It was mentioned in Section 2.3 that the correlation coefficient can be used to quantify the linear dependence between two variables. When errors/failures on two components are related, the correlation coefficient between the two components is a good measure of such dependence. The question is how to obtain it from measured data.

The first step in correlation analysis is building a data matrix based on the measured data. Assume that there are n components in the measured system and the measured period is divided into m equal intervals of Δt (e.g., 5 minutes). An $m \times n$ data matrix can then be constructed in the following way. The n columns of the matrix represent the n components in the measured system. The m rows of the matrix represent the m time intervals. Element (i, j) of the matrix is set to the number of errors occurring within interval i on component j. Column j can be regarded as a sample of the random variable, X_j, which represents the state of component j in the system.

The second step is calculating correlation coefficients using Eq. (2.19) based on the data matrix. Each time, we pick up two columns (X_i and X_j) to calculate $Cor(X_i, X_j)$. This step can be automated by using a statistical package such as SAS. Table 5.5 lists the average correlation coefficients of the 21 pairs of machines in a VAXcluster for different types of errors and failures [Tang93a]. Generally, the error correlation is high (0.62) and the failure correlation is low (0.06). Disk and network errors are strongly correlated, because the processors in the system heavily use and share the disks and the network concurrently.

Table 5.5. Average Correlation Coefficients for VAXcluster Errors

Error						Failure
All	CPU	Memory	Disk	Network	Software	All
0.62	0.03	0.01	0.78	0.70	0.02	0.06

5.4.3. Multi-Way Dependency

If errors/failures on more than two components are related, the correlation coefficient is not enough to quantify the dependence among these components, i.e., multi-way correlation. In such a case, the factor analysis method introduced in Section 2.3 can be used to uncover the underlying multi-way correlation. In this subsection, the application of factor analysis is illustrated using the processor failure data collected from a Tandem fault tolerant system [Lee91].

Similar to the correlation analysis discussed above, the first step is building an $m \times n$ data matrix based on measurements, where n is the number of components in the system. The measured Tandem system is an 8-processor multicomputer, i.e., n is 8. The Δt used is 30 minutes. The element (i, j) of this matrix has a value of 1, if processor j halts during the i-th time interval; otherwise, it has a value of 0. The j-th column of the matrix represents the sample

Table 5.6. Factor Pattern of the Tandem Processor Halts

Processor	Factor 1	Factor 2	Factor 3	Factor 4	Communality
1	0.997	-0.004	-0.069	0.023	1.00
2	0.000	0.000	0.000	0.000	0.00
3	0.061	0.012	0.853	-0.133	0.75
4	0.001	0.999	-0.011	0.021	1.00
5	0.982	-0.000	0.188	-0.018	1.00
6	-0.001	0.447	-0.005	0.009	0.20
7	0.047	-0.002	0.862	0.506	1.00
8	-0.007	0.762	0.090	0.641	1.00
Var.	1.965	1.781	1.519	0.685	
Var. %	24.6	22.3	19.0	8.6	

halt history of processor j, while the i-th row of the matrix represents the state of the eight processors in the i-th time interval. The matrix is thus called a *processor halt matrix.*

The second step is performing factor analysis by applying the SAS procedure FACTOR to the processor halt matrix. The results are shown in Table 5.6. The numbers in the middle of the table are factor loadings, and the last column shows communality. The bottom two rows show the amount of variances explained by the common factors and their percentages to the total variance.

According to [Dillon84], factor loadings greater than 0.5 are considered to be significant. However, in reliability analysis, factor loadings lower than 0.5 can be significant. The results show that there are four common factors. Factor 1 captures the dependence between processor 1 and processor 5 and accounts for 24.6% of the total variance. Factor 2 captures the multi-way dependence among processors 4, 6, and 8, although the contribution of processor 6 is small (0.447^2, i.e., 20% of its variance is explained by this factor). Factor 2 explains 22.3% of the total variance. Factor 3 captures the dependence between processor 3 and processor 7, and contributes 19% to the total variance. Factor 4 captures the dependence, although it is lower (with factor loadings 0.506 and 0.641), between processor 7 and processor 8, and accounts for 8.6% of the total variance.

5.5. Markov Reward Modeling

Many natural and social phenomena can be modeled by Markov or semi-Markov stochastic processes [Trivedi82]. In computer area, Markov process is one of the most frequently used models in performance and dependability evaluation. Compared to combinatorial models, Markov models have several advantages, such as the ability to handle time-dependent failure rate, performance degradation, and interactions among components. In the area of analytical modeling of computer systems, performability models [MeyerJ80], [MeyerJ92], availability models [Goyal87], and Markov reward models [Reibman89], [Trivedi92] have all been addressed during the past 15 years. However, how to apply these techniques to measured data is still not clear. Assumptions made in building analytical models also need to be validated by measurement-based analysis.

In analytical analysis, Markov models are built based on some assumptions (such as independent failures on different components) using individual component parameters (such as failure and recovery rates). The evaluated results are highly dependent on input parameters and model assumptions. In measurement-based modeling, Markov models are identified from data and therefore called *measured models* [Tang93b]. No additional assumptions (more than the Markov property) are made in the construction of models. The measured models provide the best evaluation for real systems as well as insight into the development of representative analytical models. Thus, it is valuable to identify appropriate models from measured data. Measurement-based Markov reward modeling techniques are illustrated through a system model generated for a VAXcluster and a software model generated for an IBM operating system.

5.5.1. Modeling of a Distributed System

The data used for the modeling was collected from a DEC VAXcluster system, consisting of seven machines, for 250 days [Tang93a]. For this system, an *error* was defined as an abnormality in any component of the system. If an error led to a termination of service on a machine, it was defined as a *failure*. A failure was identified by a reboot following one or multiple error reports.

A. Model Construction

Since the measured VAXcluster has seven machines, an 8-state Markov error model is constructed. The eight states, E_0, E_1, ..., and E_7, are defined such that E_i represents the state wherein i machines observe errors at the same time (the time granularity is chosen to be 1 second). For example, state E_0 represents that none of the machines experiences errors, i.e., the VAXcluster is in the normal (error-free) state; state E_7 represents that all the machines experience errors. At any measured time, the VAXcluster is in one of these states.

The transition probabilities for the 8-state model is estimated from the error event data. Given that the system is in state i, the probability that it will go to state j, p_{ij}, is calculated as follows:

$$p_{ij} = \frac{observed\ number\ of\ transitions\ from\ E_i\ to\ E_j}{observed\ number\ of\ transitions\ out\ of\ E_i} . \tag{5.6}$$

Table 5.7 shows the transition probabilities calculated from the VAXcluster error data. Based on the table, an error propagation model can be obtained by calculating the probability that the system goes from state E_i ($i = 1, ..., 6$) to any of the lower states ($E_{i-1}, ..., E_0$) and the probability that it goes from E_i to any of the higher states ($E_{i+1}, ..., E_7$). These probabilities are easily determined by summing all the row elements to the left of element (i, i), and all row elements to the right of element (i, i) in the tables. The error propagation model is shown in Figures 5.8. An interesting error propagation characteristic is uncovered with this model. Notice that the transition probabilities to higher states (numbers in the upper line) tend to increase as the state increases. That is, once an error domain encompasses more than one machine, the probability of the domain involving more machines increases. In such situations, error containment can become increasingly difficult.

Table 5.8 shows the mean holding time, the total holding time in the measured period, and the occupancy probability in each state for the model. It is seen from the table that E_7 has the longest mean holding time (2.31 minutes)

Table 5.7. Transition Probability for the VAXcluster Error Model

State	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7
E_0	.000	.891	.084	.014	.004	.002	.002	.003
E_1	.824	.000	.145	.023	.004	.003	.001	.000
E_2	.239	.594	.000	.118	.035	.009	.004	.001
E_3	.126	.211	.401	.000	.227	.024	.009	.003
E_4	.079	.147	.102	.422	.000	.205	.034	.011
E_5	.058	.115	.054	.073	.367	.000	.315	.018
E_6	.070	.081	.024	.016	.073	.406	.000	.331
E_7	.125	.104	.000	.021	.036	.161	.552	.000

Figure 5.8. An Error Propagation Model for the VAXcluster

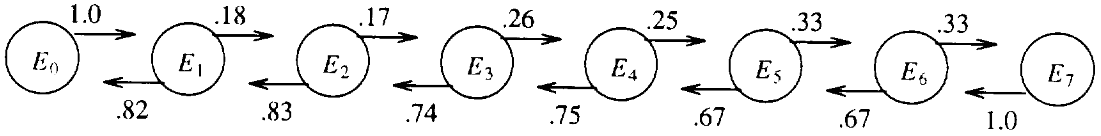


Table 5.8. Holding Time (HT) & Occupancy Probability for the VAXcluster Error Model

State	Mean HT (min.)	Total HT (hr.)	Occ. Prob.
E_0	22.39	5578.89	0.9298
E_1	1.27	347.42	0.0579
E_2	0.40	29.24	0.0049
E_3	0.56	14.07	0.0023
E_4	1.07	15.13	0.0025
E_5	0.40	3.37	0.0006
E_6	0.73	4.50	0.0007
E_7	2.31	7.38	0.0012

among all error states. Clearly, when all seven machines are affected by errors, the system takes the longest time to recover. The occupancy probabilities provide evidence that errors on different machines (i.e., errors in the higher states) are related. It is found that the measured occupancy probabilities for the higher states (E_3 to E_7) are quite different from the occupancy probabilities analytically determined assuming error independence. For example, we consider the occupancy probability for E_7. By Table 5.8, the measured occupancy probability for E_7 is 0.0012. Assuming that errors on different machines are independent, we can easily determine the occupancy probability for this state to be at most 0.02^7, where 0.02 is the highest error occurrence probability among the seven machines. That is, the measured value is higher than the calculated value by at least eight orders of magnitude.

B. Reward Analysis

Markov models can be used to conduct reward analysis [Trivedi92] to quantify the loss of service due to errors and failures. The key step is to define a reward function which characterizes the performance loss in each degraded state. For a multicomputer system, a generic reward function can be defined for both a single machine and the whole system. Given a time interval ΔT (random variable), a *reward rate* for the system in ΔT is determined by

$$r(\Delta T) = W(\Delta T)\ /\ \Delta T\ , \tag{5.7}$$

where $W(\Delta T)$ denotes the useful work done by the system in ΔT and is calculated by

$$W(\Delta T) = \begin{cases} \Delta T & \textit{in normal state} \\ \Delta T - n\tau & \textit{in error state} \\ 0 & \textit{in failure state}\ , \end{cases} \tag{5.8}$$

where n is the number of raw errors (error entries in the log, see Section 5.2) in ΔT and τ is the mean recovery time for a single error. Thus, one unit of reward is given for each unit of time when the system is in the normal state. In an error state, the penalty paid depends on the recovery time the system spends in that state, which is determined by the linear function $\Delta T - n\tau$ (normally, $\Delta T > n\tau$; if $\Delta T < n\tau$, $W(\Delta T)$ is set to 0). In a failure state, $W(\Delta T)$ is by definition zero.

Applying Eq. (5.8) to the VAXcluster, the reward rate formula has the following form:

$$r(\Delta T) = \sum_{k=1}^{7} W_k(\Delta T) \,/\, (7 \times \Delta T)\ , \tag{5.9}$$

where $W_k(\Delta T)$ denotes the useful work done by machine k in time ΔT. Here all machines are assumed to contribute an equal amount of reward to the system. For example, if three machines fail when the system is in E_3, the reward rate is 4/7.

The expected steady-state reward rate, Y, can be estimated by [Trivedi92]

$$Y = \frac{1}{T} \sum_{\Delta t_j \in T} r(\Delta t_j)\Delta t_j\ , \tag{5.10}$$

where T is the summation of all Δt_j's (particular values of ΔT) in consideration. If we substitute r from Eq. (5.9) and let ΔT represent the holding time of each state in the error model, Y becomes the steady-state reward rate of the VAXcluster, which is also an estimate of system availability (performance-related availability). If we substitute r from Eq. (5.9) and let ΔT represent the time span of the error event for a particular type of error, Y becomes the steady-state reward rate of the system during the event intervals of the specified error. Thus, $(1 - Y)$ measures the loss in performance during the specified error event. Note that it is possible that there are failed machines when the system is in an

Table 5.9. Steady-State Reward Rate for the VAXcluster

τ	0.1 ms	1ms	10 ms	100 ms
Y	0.995078	0.995077	0.995067	0.994971

Table 5.10. Steady-State Reward Rate for Each Error Type in the VAXcluster

CPU	Memory	Disk	Tape	Network	Software
0.14950	0.99994	0.61314	0.89845	0.56841	0.00008

error state. Since the model is an empirical model based on the error event data (of which the failure event data is a subset), the information about errors and failures of all machines for each particular Δt_j can be obtained from the data.

The steady-state reward rate for the VAXcluster was computed with τ being 0.1, 1, 10, and 100ms. The results are given in Table 5.9. The table shows that the reward rate is not sensitive to τ. This is because the overall recovery time is dominated by the failure recovery time, i.e., the major contributors to the performance loss are failures, not non-failure errors. In the range of these τ values, the VAXcluster availability is estimated to be 0.995. Table 5.10 shows the steady-state reward rate for each error type ($\tau = 1$ ms) for the VAXcluster. These numbers quantify the loss of performance due to the recovery from each type of error. For example, during the recovery from CPU errors, the system can be expected to deliver approximately 15% of its full performance. During the disk error recovery, the average system performance degrades to nearly 61% of its capacity. Since software errors have the lowest reward rate (0.00008), the loss of work during the recovery from software errors is the most significant.

5.5.2. Modeling of an Operating System

The modeled operating system is the IBM MVS system running on an IBM 3081 mainframe [Hsueh87]. The measurement period is one year. A Markov model is developed using data collected from the system to describe error detection and recovery inside an operating system. The MVS is a widely used IBM operating system. Primary features of the system are reported to be efficient storage management and automatic software error recovery. The MVS system attempts to correct software errors using recovery routines. The philosophy in the MVS is that for major system functions, the programmer envisages possible failure scenarios and writes a recovery routine for each. It is, however, the responsibility of the installation (or the user) to write recovery routines for applications.

Recovery routines in the MVS operating system provide a means by which the operating system prevents a total loss on the occurrence of software errors. When a program is abnormally interrupted due to an error, the supervisor routine gets control. If the problem is such that further processing can degrade the system or destroy data, the supervisor routine gives control to the recovery termination manager (RTM), an operating system module responsible for error and recovery management. If a recovery routine is available for the interrupted program, the RTM gives control to this routine before it terminates the program. The purpose of a recovery routine is to free the resources kept by the failing program, to locate the error, and to request either a retry or the termination of the program.

More than one recovery routine can be specified for the same program. If the current recovery routine is unable to restore a valid state, RTM can give control to another recovery routine, if available. This process is called *percolation*. The percolation process ends if either a routine issues a valid retry request or no more recovery routines are available. In the latter case, the program and its related subtasks are terminated. If a valid retry is requested, a retry is attempted to restore a valid state using the information supplied by the recovery routine and then give control to the program. For a retry to be valid, there should be no risk of error recurrence and the retry address should be properly specified. An error recovery can result in the following four situations:

(1) Resume op (resume operation) — The system successfully recovers from the error and returns control to the interrupted program.

(2) Task term (task termination) — The program and its related subtasks are terminated, but the system does not fail.

(3) Job term (job termination) — The job in control at the time of the error is aborted.

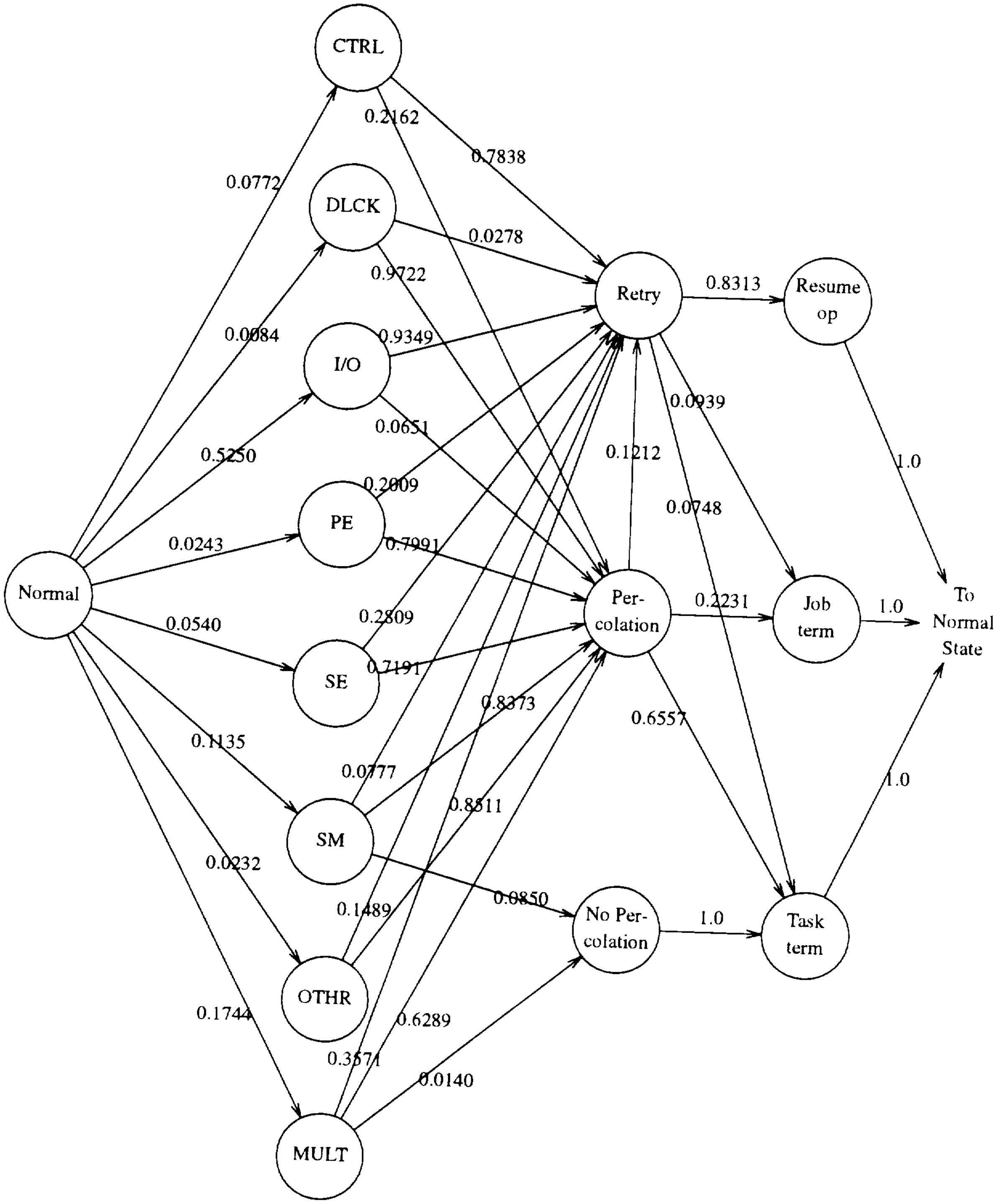
(4) System failure — The job or task, which was terminated, is critical for system operation. As a result of the termination, a system failure occurs.

A. Model Construction

The states of the model consists of eight types of error states (see Table 5.11) and four states resulting from error recoveries. Figure 5.9 shows the model. The *normal* state represents that the operating system is running error-free. The transition probabilities were estimated from the measured data using Eq. (5.6). Note that the system failure state is not shown in the figure. This is because the occurrence of system failure was rare, and the number of observed system failures was statistically insignificant.

Table 5.11 shows the mean waiting time characteristics of the normal and error states in the model. Note that the waiting time distribution of the normal state is the TTE distribution. It has been shown in Section 5.3.3 that this distribution is not simply exponential (a multi-stage gamma distribution), so the model is a semi-Markov model. In the table a multiple software error is defined as an error burst consisting of more than one type of software error. The

Figure 5.9. MVS Software Error/Recovery Model



average duration of a multiple error is at least four times longer than that of any type of single error which is typically in the range of 20 to 40 seconds, except for DLCK (deadlock) and OTHR (others). The average recovery time from a program exception is twice as long as that from a control error (21 seconds versus 42 seconds). This is probably due to the extensive software involvement in recovering from program exceptions.

An error recovery can be as simple as a retry or as complex as requiring several percolations before a successful retry. The problem can also be such that no retry or percolation is possible. Figure 5.9 shows that about 83.1% of all retries are successful. The figure also shows that the operating system is able to recover from 93.5% of I/O and data management errors and 78.4% of control related errors by retries. These observations indicate that most I/O and control related errors are relatively easy to recover from, compared to the other types of errors such as deadlock or storage errors. Also note that "no percolation" occurs only in recovering from storage management errors. This indicates that storage management errors are more complicated than the other types of errors.

Table 5.11. Mean Waiting Time

State	# Observations	Mean Waiting Time (Sec.)	Standard Deviation
Normal (Error-Free)	2757	10461.33	32735.04
CTRL (Control Error)	213	21.92	84.21
DLCK (Deadlock)	23	4.72	22.61
I/O (I/O & Data Management Error)	1448	25.05	77.62
PE (Program Exception)	65	42.23	92.98
SE (Storage or Address Exception)	149	36.82	79.59
SM (Storage Management Error)	313	33.40	95.01
OTHR (Other Type)	66	1.86	12.98
MULT (Multiple Software Error)	481	175.59	252.79

B. Model Evaluation

The steady-state measures evaluated from the model is listed in Table 5.12. The definitions of these measures are given in [Howard71].

(1) Transition probability (π_j) — probability that the transition is to state j, given a transition to occur

(2) Occupancy probability (Φ_j) — probability that the system occupies state j at any time point

(3) Mean recurrence time ($\bar{\Theta}_j$) — mean recurrence time of state j

The occupancy probability of the normal state can be viewed as the operating system availability without degradation. The state transition probability, on the other hand, characterizes error detection and recovery processes in the operating system. Table 5.12(a) lists the state transition probabilities and occupancy probabilities for the normal and error states. Table 5.12(b) lists the state transition probabilities and the mean recurrent times of the recovery and result states. A dashed line in the table indicates a negligible value (less than 0.00001). Table 5.12(a) shows that the occupancy probability of the normal state in the model is 0.995. This indicates that in 99.5% of the time the operating system is running error-free. In the other 0.5% of time the operating system is in the error or recovery states. In more than half of the error and recovery time (i.e., 0.29% out of 0.5%) the operating system is in the multiple error state. The average reward rate for all software error and recovery states is estimated from data to be 0.2736. Based on this reward rate and the occupancy probability for all error and recovery states shown in the table (0.005), the steady-state reward loss in the modeled MVS can be evaluated to be 0.00363.

By solving the model, it is found that the operating system makes a transition every 43.37 minutes. Table 5.12(a) shows that 24.74% of all transitions made in the model are to the normal state, 24.73% to error states (obtained by summing all the π's for all error states), 25.79% to recovery states, and 24.74% to result states. Since a transition occurs every 43 minutes, it can be estimated that, on the average, a software error is detected every 3 hours and a successful recovery (i.e., reaching the "resume op" state) occurs every 5 hours. Table 5.12(b) also shows that more than 40% of software errors lead to job or task terminations which cause the loss of service to users. However, a few of these terminations lead to system failures. This result indicates that recovery routines in MVS are effective in avoiding system failures but are not so effective in avoiding user job terminations.

Table 5.12. Error/Recovery Model Characteristics

Measure	Normal State	Error State							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR	MULT
π	0.2474	0.0191	0.0020	0.1299	0.0060	0.0134	0.0281	0.0057	0.0431
Φ	0.9950	0.00016	-	0.00125	0.000098	0.000189	0.00036	-	0.002913

(a)

Measure	Recovery State			Resultant State		
	Retry	Percolation	No-Percolation	Resume Op.	Task Term.	Job Term.
π	0.1704	0.0845	0.0030	0.1414	0.0712	0.0348
$\bar{\Theta}(hr.)$	4.25	8.55	241.43	5.11	10.16	20.74

(b)

5.6. Software Dependability

A great deal of research has been performed in the area of software reliability during the development phase. Different models have been proposed (reviewed in [Musa87]) to characterize the reliability growth of the candidate software through this phase. In general, these models can be divided into two classes. The first assumes that the failure rate is a function of the number of remaining defects in the software. Imperfect debugging and uncertainty in the projected number of initial defects have also been modeled [Goel85]. The second class of models does not depend on the knowledge of the number of the remaining defects [Littlewood80]. The failure rate is assumed to be a random variable and the software reliability model involves two stochastic processes. Although most models perform well within their own contexts, their performance varies significantly from one data set to another.

The operational phase of a mature software is much different from the development phase. In the operational phase, a typical situation involves frequent changes and updates installed either by system managers or by vendors. Often, without notification to the installation management, the vendor will install a change (patch) to fix a fault found at some other installation. In a sense, the system being measured represents an aggregate of all such systems being maintained by the vendor. In addition, software reliability in the operational phase is also attributed to workload effects, hardware problems, and environmental factors. Thus, software reliability in the operational phase cannot be characterized by simply applying analytical models proposed for the development phase.

Studies dealing with software dependability issues for the operational phase have also evolved over the past 15 years. Software TTE distributions (Section 5.3), dependency between software failure and workload (Section 5.4), and modeling of software error/recovery processes (Section 5.5) have been discussed in previous sections. In this section, several other issues, including error interactions (i.e., hardware-related and correlated software errors), software fault tolerance, and software defect classification are discussed.

5.6.1. Error Interactions

When software is running in a complex system, interactions between hardware and software, and interactions among multiple processors can cause software error scenarios that cannot been seen during testing. Investigation of such error scenarios is helpful for understanding characteristics of software errors in operational systems. In the following, two kinds of such error scenarios are discussed: *hardware-related software errors*, which are a result of interactions between hardware and software, and *correlated software errors*, which are a result of interactions among processors through software protocols.

A. *Hardware-Related Software Errors*

In [Iyer85a], software errors related to hardware errors were described as hardware-related software errors. More precisely, if a software error (failure) occurs in close proximity (within a minute) to a hardware error, it is called a hardware-related software (HW/SW) error (failure). There are several causes of hardware-related software errors. For instance, a hardware error, such as a flipped memory bit, may change the software condition, resulting in a software error. Therefore, even though it is reported as a software error, it is actually caused by faulty hardware. Another possibility is that the software may fail to handle an unexpected hardware problem such as an abnormal condition in the network communication. This can be attributed to a software design flaw. Sometimes, both the hardware error and the software error are symptoms of another, unidentified problem.

Table 5.13 shows the frequency and percentage of hardware-related software errors/failures (among all software errors/failures) measured from an IBM 3081 system [Iyer85b] and two VAXclusters [Tang92b]. In the IBM system, approximately 33% of all observed software failures are hardware-related. HW/SW errors are found to have large error-handling times (high recovery overhead). The system failure probability for the HW/SW errors is close to three times that for software errors in general. The VAXcluster data shows that most hardware errors involved in HW/SW errors are network errors (75%). This indicates that the major sources of hardware-related software problems in the measured VAXclusters are network-related hardware or software components. This is a unique feature in the multicomputer system, where processes highly rely on intercommunications through the network.

Table 5.13. Hardware-Related Software Errors/Failures

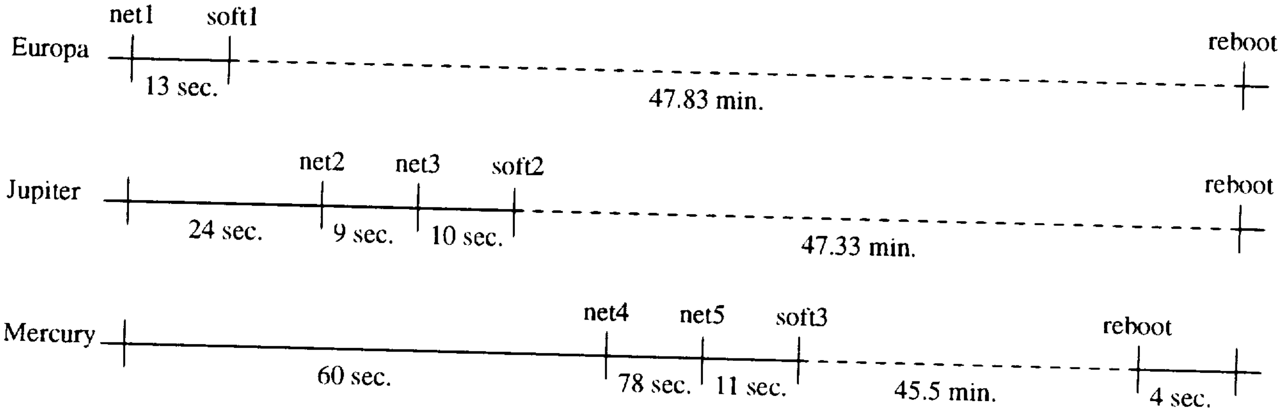
Category	HW/SW Errors		HW/SW Failures	
Measures	Frequency	Percent	Frequency	Percent
IBM/MVS	177	11.4	94	32.8
VAX/VMS	32	18.9	28	21.4

B. Correlated Software Errors

When multiple instances of a software system interact with each other in a multicomputer environment, the issue of correlated failures should be addressed. Several studies [Tang90], [Wein90], [Lee91] found that significant correlated processor failures exist in the measured multicomputer systems. Correlated software failures are also found in the VAX VMS and the Tandem GUARDIAN operating systems [Lee93a]. The data showed that about 10% of software failures in the measured VAXcluster and 20% of software halts in the measured Tandem system occurred on multiple machines concurrently. To understand how correlated software failures occur, it is instructive to examine a real case in detail.

Figure 5.10 shows a scenario of correlated software failures. In the figure, Europa, Jupiter, and Mercury are machine names in the VAXcluster. A dashed line represents that the corresponding machine is in a failure state. At one time, a network error (net1) was reported from the CI (Computer Interconnect) port on Europa. This resulted in a software failure (soft1) 13 seconds later. Twenty-four seconds after the first network error (net1), additional network errors (net2,net3) were reported on the second machine (Jupiter), which was followed by a software failure (soft2). The error sequence on Jupiter was repeated (net4,net5,soft3) on the third machine (Mercury). The three machines experienced software failures concurrently for 45.5 minutes. All three software failures occurred shortly after network errors occurred, so they were network error related. Further analysis of the data revealed that the network-related

Figure 5.10. A Scenario of Correlated Software Failures



Note: soft1, soft2, soft3 — Exception while above asynchronous system traps delivery or on interrupt stack. net1, net3, net5 — Port will be re-started. net2, net4 — Virtual circuit timeout.

software of the VAX/VMS is a potential software bottleneck in terms of correlated failures.

The higher percentage of correlated software failures in the Tandem system can be attributed to the architectural characteristics of the system. In the Tandem system, a single software fault can cause halts of two processors on which the primary and backup processes (see Section 5.6.2) of the faulty software are executing. If the two halted processors control a disk which includes files needed by other processors on the system, additional software halts can occur on these processors. (In the Tandem system, a disk can typically be accessed by two processors via dual-port disk controllers.) This explains why there is a higher percentage of correlated software failures in the Tandem system.

Note that the above scenario is a multiple component failure situation not expected in general system design, which assumes failure independence. Even the Tandem fault tolerant system is not designed explicitly to guard against this situation. Generally, correlated failures can stress recovery and break the protection provided by the fault tolerance.

5.6.2. Software Fault Tolerance

While hardware fault tolerance techniques have been used successfully, the issue of software fault tolerance is still not well addressed. Major approaches for software fault tolerance rely on design diversity [Avizienis84], [Randell75]. But these approaches are usually inapplicable to large operating systems because of immense cost in developing and maintaining the software. However, some fault tolerance techniques not explicitly designed for tolerating software faults can provide a certain amount of software fault tolerance. Understanding such techniques is important for designing good approaches to improving software dependability. The Tandem GURDIAN system, running on the single-failure tolerant multicomputer system, is a good target for such evaluations.

The Tandem GUARDIAN operating system is a message-based distributed system built for on-line transaction processing [Bartlett78]. High availability is achieved via single-failure tolerance techniques including the *process-pair* approach. For each user program, there are two processes — a *primary process* and a *backup process* — executing the same program on two processors. During normal operation, the primary process performs all operations for the user, while the backup process passively watches message flows. The primary process periodically sends checkpoint messages to its backup. When the primary process detects an inconsistency in its state, it fails fast, and the backup process takes over the responsibility of the primary process. This approach can tolerate transient software errors, which will usually not be repeated by reexecuting the process.

A study of operating system fault tolerance achieved by the single-failure tolerance techniques implemented in a Tandem multiprocessor system was reported in [Lee92]. The measured system had 16 processors and was working in a high-stress environment. The data source was the processor halt log maintained by the GUARDIAN system for a period of 23 months. The effect of the built-in fault tolerance mechanisms on software availability was evaluated by reward analysis. Two reward functions were defined in the analysis. In the definition, i represents the system state in which there are i failed processors, and n represents the total number of processors in the system. The first function (SFT) reflects the fault tolerance of the Tandem system. In this function, the first processor halt does not cause any degradation. For additional processor halts, the loss of service is proportional to the number of processors halted. The second function (NSFT) assumes no fault tolerance. The difference between the two functions allows evaluation of the improvement in service due to the built-in fault tolerance mechanisms.

SFT (Single-Failure Tolerance):

$$r_i = \begin{cases} 1 & \text{if } i = 0 \\ 1 - \dfrac{i-1}{n} & \text{if } 0 < i < n \\ 0 & \text{if } i = n \end{cases} \tag{5.13}$$

NSFT (No Single-Failure Tolerance):

$$r_i = 1 - \frac{i}{n} \qquad 0 \le i \le n \tag{5.14}$$

Based on the above reward functions, the expected steady-state reward rate, i.e., the Y in Eq. (5.10), was evaluated for software, non-software, and all halts. The results are given in Table 5.14. The bottom row shows the improvement in service time (i.e., reduction in reward loss) due to the fault tolerance. It is seen that the single-failure tolerance in the measured system reduces the service loss due to software halts by 89% and due to non-software halts by 92%. This clearly demonstrates the effectiveness of the implemented fault tolerance mechanisms against software failures as well as non-software failures. The table also shows that software problems account for 30% of the service loss in the measured system (with SFT). Although the system was working in a high-stress environment, the overall reward loss is small (10^{-4} with SFT). This reflects the high availability of the measured system.

Table 5.14. Loss of Service Caused by Halts in the Tandem System

Measure		Software	Non-Software	All
NSFT	$1-Y$	.00062	.00205	.00267
	Percent	23.2	76.8	100
SFT	$1-Y$	.00007	.00016	.00023
	Percent	30.4	69.6	100
Improvement		89%	92%	91%

5.6.3. Software Defect Classification

In recent studies of software defects reported from the IBM MVS operating system [Sullivan91] and two IBM large database management systems, DB2 and IMS [Sullivan92], a software defect classification scheme was proposed. The scheme uses three concepts — *error type*, *defect type*, and *error trigger* — to classify software faults and errors. The error type classifies the low-level programming mistakes that lead to software failures. The defect type is a higher-level classification that distinguishes design mistakes, coding mistakes, and administrative mistakes. The error trigger is related to the running environment; it distinguishes several ways that defective code which was not executed during testing could be executed at the customer site. Tables 5.15 to 5.17 list major categories generated

Table 5.15. Major Categories of Error Types

Error Type	Description
Allocation Management	A module uses a memory region after deallocating it.
Copying Overrun	The program copies data past the end of a buffer.
Data Error	The program produces or reads wrong data.
Interface Error	A module's interface is defined or used incorrectly.
Memory Leak	The program never deallocate memory it obtained from the system.
Pointer Management	A variable containing the address of data is corrupted.
Statement Logic	Statements are executed in the wrong order or are omitted.
Synchronization	An error occurs in locking or synchronization code.
Uninitialized Variable	A variable is used before it is initialized.
Undefined State	The system goes into a state the designers did not anticipate.
Wrong Algorithm	The program works but uses a wrong algorithm.

Table 5.16. Major Categories of Defect Types

Defect Type	Description
Function	A program's functionality is missing, incomplete, or incorrect.
Data Struct/Algorithm	A data structure or algorithm has a design flaw.
Assignment/Checking	A coding mistake involves variable assignment or validation.
Interface	Errors are discovered in the interaction between components.
Timing/Synchronization	Errors occur in the management of shared or real-time resources.
Build/Package/Merge	Errors occur in version control or roll-up of fixes.

Table 5.17. Major Categories of Error Triggers

Error Trigger	Description
Workload	Unusual workload conditions such as a user request with unexpected parameters.
Bug Fixes	A bug introduced when an earlier bug was fixed.
Client Code	Errors caused by propagation from application code running in protected mode.
Recovery/Exception	Problems in error recovery and exception handling.
Timing	Errors caused by an unanticipated sequence of events.

from the data under the three criteria.

The studies compared the error type, defect type, and error trigger distributions of the three products (DB2, IMS, and MVS) and found that the three product's distributions differ significantly. However, they have some common characteristics, such as the mode "undefined state." The studies also investigated the impact of software defects on system availability for the MVS operating system. A comparison between overlay defects (defects that corrupt a program's memory) and non-overlay defects demonstrated that the impact of an overlay defect is much higher. Boundary conditions and allocation management were found to be the major causes of overlay defects.

5.7. Failure Prediction

Fault diagnosis and failure prediction are of significance for maintaining highly reliable systems. Measurement-based studies have shown that it is possible to predict future failures based on the current and historical on-line error information. Several heuristic and statistical approaches have been proposed. The heuristic approach extracts characteristics of anomalous events, such as error reports [Lin90] or performance anomalies [Maxion90a], and relates them to failures or faults by heuristic rules or signatures. The statistical approach uses statistical techniques to quantify relationships among system error states defined on the basis of error rates and recognizes failure patterns using the quantified relationships [Iyer90]. In the following, we discuss two typical approaches: 1) failure prediction based on the heuristic trend analysis of error logs and 2) failure prediction based on the statistical analysis of error symptoms.

5.7.1. Prediction Based on Heuristic Trend Analysis

This approach is based on the observation that a system usually experiences a period of intermittent errors before a hard failure occurs. The symptoms of intermittent errors can be used to predict impending failures. The early study of this approach showed qualitatively that the frequency of error tuples was correlated to system failures, based on measurements from a DEC disk subsystem [Tsao83]. Later, a heuristic trend analysis method, the dispersion frame technique (DFT), was developed [Lin90]. DFT determines the relationship among errors by examining their closeness in time and space.

Two concepts are used in DFT: dispersion frame (DF) and error dispersion index (EDI). A DF is defined as the interval between two successive errors of the same type. The EDI is defined as the number of error occurrences following the previous DF during the interval of one half of the previous DF or the DF before the previous DF. Each DF is applied to the following two errors. A high EDI implicates that the errors following the DF used to measure the EDI are highly correlated. DFT consists of five heuristics rules developed from field experience:

(1) *3.3 rule*: The two consecutive EDIs obtained by applying the same frame are at least 3.

(2) *2.2 rule*: The two consecutive EDIs obtained by applying two successive frames are at least 2.

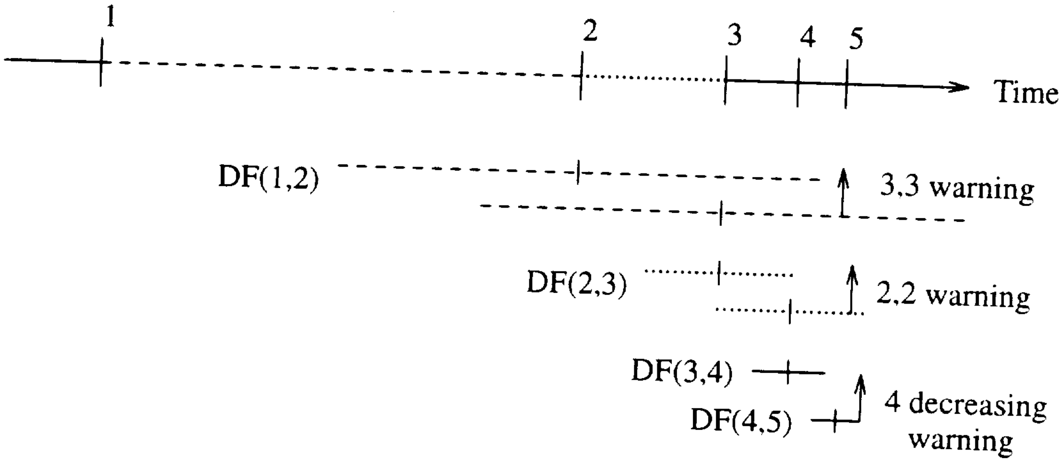
(3) *2 in 1 rule*: A frame is less than 1 hour.

(4) *4 in 1 rule*: Four errors occur within a 24-hour frame.

(5) *4 decreasing rule*: There are four monotonically decreasing frames, and at least one frame is half the size of its previous frame.

Figure 5.11 demonstrates an example, including some activated heuristics, of DFT. In the figure, the top line represents the time sequence of five error occurrences (1, ..., 5) in a particular device. DFT is activated when a frame size less than 168 hours (1 week) is encountered. Assume that all the frames in the figure fall into this threshold. Each

Figure 5.11. Dispersion Techniques



frame is applied to the following two errors by putting its center to the time points of the two error occurrences. For example, DF(1,2) is applied to errors 2 and 3, DF(2,3) is applied to errors 3 and 4, etc. An upward arrow represents a failure warning issued under the above heuristic rules.

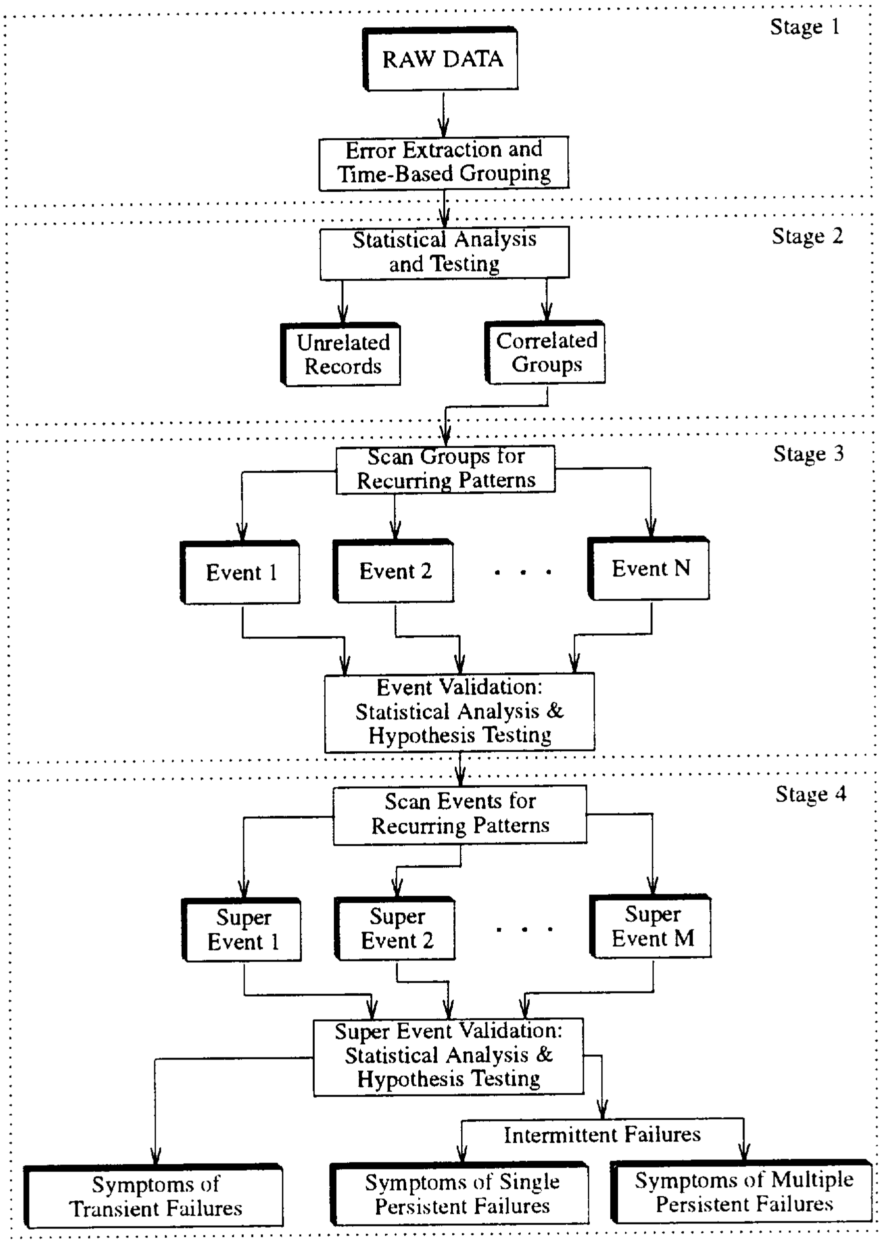
DFT was applied to the data collected from 13 public-domain file servers in Carnegie Mellon University over a 22-month period. Among 16 hard failures examined, DFT predicated 15, with 5 false alarms. That is, the successful prediction rate is 93.7%. This results shows that DFT is very effective when coupled with good system instrumentation. The disadvantage of this approach is that different systems may require different heuristics and parameters.

5.7.2. Prediction Based on Statistical Analysis

The objective of this approach is to recognize intermittent failures through statistical analysis and testing on recorded error data. The approach starts by identifying key error patterns potentially symptomatic of failure occurrences and then refines these patterns by scanning the rest of the data in stages for similar error patterns. At each stage, the similarity is statistically tested. The approach is illustrated by the flowchart in Figure 5.12.

In the first stage, data coalescing is performed on the raw data to eliminate redundant reports. The output of this stage is *error records* (tuples) characterized by *error states* (error type, machine condition, etc.). Next, all error records occurring within a small time interval (15 minutes) are identified as *error groups*. Error groups represent periods of high error activity (error bursts). Experience has shown that when system errors occur in bursts of a relatively high error rate, the errors are often related. In the second stage, statistical analysis and hypothesis testing are

Figure 5.12. Automatic Recognition of Persistent Failures


Stage 1
RAW DATA
Error Extraction and Time-Based Grouping
Stage 2
Statistical Analysis and Testing
Unrelated Records
Correlated Groups
Stage 3
Scan Groups for Recurring Patterns
Event 1
Event 2
. . .
Event N
Event Validation: Statistical Analysis & Hypothesis Testing
Stage 4
Scan Events for Recurring Patterns
Super Event 1
Super Event 2
. . .
Super Event M
Super Event Validation: Statistical Analysis & Hypothesis Testing
Intermittent Failures
Symptoms of Transient Failures
Symptoms of Single Persistent Failures
Symptoms of Multiple Persistent Failures

performed on each error group to determine whether a valid correlation exists among its members (error records). Randomly formed groups in which members are statistically independent are rejected. Thus, the original error groups consisting of records among which relationships can exist are refined to the validated error groups consisting of records among which relationships do exist.

Relationships can exist across error groups, i.e., a single cause can give rise to a persistent error and thus foster multiple error groups within a short time. In the third stage, the output groups from the second stage are examined to recognize related error groups and to eliminate stray error records. Several concepts are introduced for the analysis in this stage. An *error event* is defined as the collection of error groups occurring within a given period (e.g., 24 hours) and having at least two error states in common. A *symptom* is defined as a collection of statistically related error states that are common to at least half of the groups in an event. A *symptom set* is defined as the collection of all symptoms in an event. Figure 5.13 illustrates an event and its symptom set. The event is composed of three groups: G_1, G_2, and G_3. The error states in these groups are represented by $A_1, ..., A_7$. Two symptoms are extracted from these error states: S_1 which consists of A_2 and A_4, and S_2, which consists of A_5 and A_6. Thus, S_1 and S_2 constitute the symptom set for this group.

Further, three simple rules are used in the fourth stage to recognize related events and to group them into sets called *super events*. The rules ensure that the events so grouped will have sufficiently common structure to permit testing for correlation. Two events are grouped into a super event if they satisfy any one of the following criteria: 1) they have at least one symptom in common, 2) a symptom of one event is a proper subset of at least one symptom of another event, or 3) if they are single-group events, then they have at least two error states in common. Figure 5.14

Figure 5.13. Derivation of an Event's Symptom Set

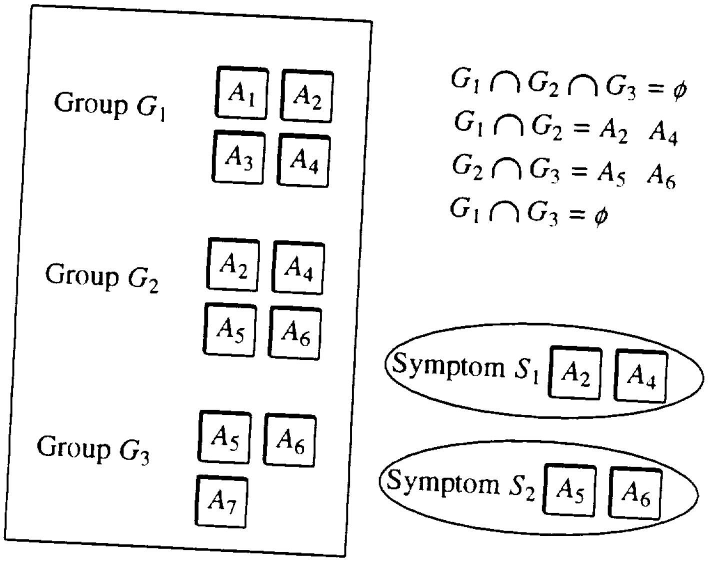
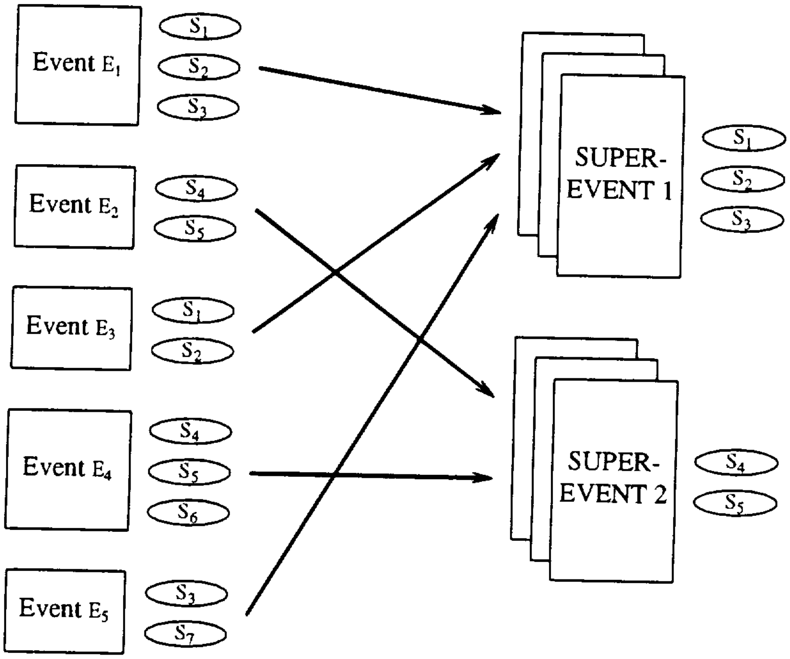

Group G_1
A_1
A_2
A_3
A_4
Group G_2
A_2
A_4
A_5
A_6
Group G_3
A_5
A_6
A_7
$G_1 \cap G_2 \cap G_3 = \phi$
$G_1 \cap G_2 = A_2 \;\; A_4$
$G_2 \cap G_3 = A_5 \;\; A_6$
$G_1 \cap G_3 = \phi$
Symptom S_1 A_2 A_4
Symptom S_2 A_5 A_6

Figure 5.14. Construction of Super Events



illustrates how super events are constructed. There is no time restriction when these rules are applied to the event data. When a super event is created, a corresponding *super symptom set* is also created. The super symptom set starts with just the symptoms of the first event of that super event. As another event is added, set intersection is performed between its symptom set and each of the symptom sets already in the super event. All intersections are then added to the super event set.

In each of the above stages, statistical analysis and hypothesis testing are performed to validate the correlations among members in the formed groups or sets. The super events derived in the final stage can be used by service engineers to judge potential failures. This methodology was applied to the on-line error log files from two CYBER systems, and the results were compared to the *log of failures and repair* maintained by the system staff. In nearly 85% of the cases, the engineers were directly able to confirm that the validated super events corresponded to real system problems. The evaluation was made both on the basis of their experience and from their field maintenance logs. For the remaining 15% of the cases, the engineers agreed that a problem had existed, but that its manifestation was not severe enough to be noticed by their analysis.

6. CONCLUSION

In this paper, we discussed methodologies and advances in the area of the experimental analysis of computer system dependability. The discussion covered three fields: simulated fault injection, physical fault injection, and measurement-based analysis of operational systems. The approaches used in the three fields are suited, respectively, to the dependability evaluation in the three phases of a system's life: design phase, prototype phase, and operational phase. Before discussing these fields, we introduced several statistical techniques used in all fields. For each field, we proposed a classification of research approaches or topics. Then we presented detailed methodologies and representative studies for each of these approaches or topics.

The statistical techniques introduced included the estimation of parameters and confidence intervals, probability distribution characterization, several multivariate analysis methods, and importance sampling. For simulated fault injections, we covered electrical-level, logic-level, and function-level simulation approaches as well as representative simulation environments, such as FOCUS and DEPEND. For physical fault injections, we discussed hardware, software, and radiation fault injection methods as well as several software and hybrid tools, including FIAT, FERRARI, HYBRID, and FINE. For measurement-based analysis of operational systems, after an introduction to measurement and data processing techniques, we presented methods used and representative studies in basic error characterization, dependency analysis, Markov reward modeling, software dependability, and fault diagnosis. The discussion covered several important issues previously studied, including workload/failure dependency, correlated failures, and software fault tolerance.

Fault injection simulations can be used to investigate the effectiveness of key design features of fault tolerant systems and to provide timely feedback to system designers. Generally, most dependability measures (except input parameters such as failure and recovery rates) can be obtained from simulations. However, simulations need accurate input parameters and the validation of output results, which come from physical fault injections and measurement-based analysis. Fault injection on real systems can produce information about error latency, error detection, error propagation, error recovery, and system reconfiguration, but it can only study artificial faults and cannot produce some dependability measures, such as MTBF and availability. Measurement-based analysis of operational systems under real workloads can provide valuable information on actual failure characteristics and insight into analytical models.

This type of analysis provides a means to study naturally occurring errors and all measurable dependability metrics, such as failure and recovery rates, reliability and availability. However, the analysis is limited to detected errors. Further, conditions in the field can vary widely from one system to another, casting doubt on the statistical validity of the results. Thus, all three approaches are complementary and essential for accurate dependability analysis.

Significant progress has been made in all the three fields over the past 15 years, especially in the recent 5 years during which several dependability analysis tools have been developed. Increasing attention is being paid to: 1) combining analytical modeling and experimental analysis and 2) combining system design and evaluation. In the first aspect, state-of-the-art analytical modeling techniques are being applied to real systems to evaluate various dependability and performance characteristics. Results from experimental analysis are being used to validate analytical models and to reveal practical issues that analytical modeling must address to develop more representative models. In the second aspect, dependability analysis tools are being combined with each other and with other CAD tools to provide an automatic design environment which incorporates multiple levels of joint evaluation of functionality, performance, dependability, and cost. Software failure data from testing and operational phases are also providing feedback to the software design, improving software reliability. Further interesting studies and advances in this area can be expected in the near future.

REFERENCES

[Arlat90a] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.C. Fabre, J.C. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. Software Engineering*, Vol. 16, No. 2, pp. 166-182, Feb. 1990.

[Arlat90b] L. Arlat, K. Kanoun, and J.C. Laprie, "Dependability Modeling and Evaluation of Software Fault-Tolerant Systems," *IEEE Trans. Computers*, Vol. 39, No. 4, pp. 504-513, April 1990.

[Aupperle89] B.E. Aupperle, J.F. Meyer and L. Wei, "Evaluation of Fault-Tolerant Systems with Nonhomogeneous Workloads," *Proc. 19th Int. Symp. Fault-Tolerant Computing*, pp. 159-166, June 1989.

[Avizienis84] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67-80, Aug. 1984.

[Banerjee82] P. Banerjee and J.A. Abraham, "Fault Characterization of MOS VLSI Circuits," *Proc. Int. Conf. Circuits and Computers*, pp.564-568, 1982.

[Bartlett78] J.F. Bartlett, "A 'Nonstop' Operating System," *Proc. Int. Hawaii Conf. System Science*, pp. 103-117, 1978.

[Barton90] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. Computers*, Vol. 39, No. 4, pp. 575-582, April 1990.

[Bavuso87] S.J. Bavuso, J.B. Dugan, K.S. Trivedi, E.M. Rothman, and W.E. Smith, "Analysis of Typical Fault-Tolerant Architectures using HARP," *IEEE Trans. Reliability*, Vol. 36, No. 2, pp. 176-185, June 1987.

[Beh82] C.C. Beh, K.H. Arya, C.E. Radke, and K.E. Torku, "Do Stuck Fault Models Reflect Manufacturing Defects?" *Proc. Int. Test Conf.*, pp. 35-42, 1982.

[Bishop88] P.G. Bishop and F.D. Pullen, "PODS Revisited — A Study of Software Failure Behavior," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp. 2-8, 1988.

[Bryant84] R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. Computers*, Vol. 33, No. 2, pp. 160-177, Feb. 1984.

[Butner80] S.E. Butner and R.K. Iyer, "A Statistical Study of Reliability and System Load at SLAC," *Proc. 10th Int. Symp. Fault-Tolerant Computing*, pp. 207-209, Oct. 1980.

[Castillo81] X. Castillo and D.P. Siewiorek, "Workload, Performance, and Reliability of Digital Computer Systems," *Proc. 11th Int. Symp. Fault-Tolerant Computing*, pp. 84-89, July 1981.

[Castillo82] X. Castillo and D.P. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," *Proc. 12th Int. Symp. Fault-Tolerant Computing*, pp. 279-286, June 1982.

[Chillarege87] R. Chillarege and R. K. Iyer, "Measurement-Based Analysis of Error Latency," *IEEE Trans. Computers*, Vol. C-36, No. 5, May 1987.

[Chillarege89] R. Chillarrege and N.S. Bowen, "Understanding Large System Failures — A Fault Injection Experiment," *Proc. 19th Int. Symp. Fault-Tolerant Computing*, pp. 356-363, June 1989.

[Choi89] G.S. Choi, R.K. Iyer and V. Carreno, "FOCUS: An Experimental Environment for Validation of Fault Tolerant Systems: A case study of a Jet Engine Controller," *Int. Conf. Computer Design (ICCD)*, pp. 561-564, Oct. 1989.

[Choi92] G.S. Choi and R.K. Iyer, "FOCUS: An Experimental Environment for Fault Sensitivity Analysis," *IEEE Trans. Computers*, Vol. 41, No. 12, pp. 1515-1526, Dec. 1992.

[Choi93] G. Choi, R. Iyer, and D. Saab, *Fault Behavior Dictionary for Simulation of Device-Level Transients*, Technical Report, CRHC, University of Illinois at Urbana-Champaign, 1993.

[Clark93] J.A. Clark and D.K. Pradhan, "REACT: A Synthesis and Evaluation Tool for Fault-Tolerant Multiprocessor Architectures," *Proc. Annual Reliability and Maintainability Symposium*, pp. 428-435, 1993.

[Courtois79] B. Courtois, "Some Results about the Efficiency of Simple Mechanisms for the Detection of Microcomputer Malfunctions," *Proc. 9th Int. Symp. Fault-Tolerant Computing* pp. 71-74, June 1979.

[Cusick85] J. Cusick, R. Koga, W. Kolasinski, and C. King, "SEU vulnerability of the Zilog Z-80 and NSC-800 microprocessors," *IEEE Trans. Nuclear Science*, vol. NS-32, pp. 4206-4211, Dec. 1985.

[Czeck92] E.W. Czeck and D.P. Siewiorek, "Observations on the Effects of Fault Manifestation as a Function of Workload," *IEEE Trans. Computers*, Vol. 41, No. 5, pp. 559-566, May 1992.

[Dillon84] Dillon, W. R. and Goldstein, M., *Multivariate Analysis*, John Wiley & Sons, 1984.

[Duba88] P. Duba and R.K. Iyer, "Transient Fault Behavior in a Microprocessor: A Case Study," *Proc. 1988 IEEE Int. Conf. Computer Design: VLSI in Computers & Processors*, pp. 272-276, Oct. 1988.

[Dugan91] J.B. Dugan, "Correlated Hardware Failures in Redundant Systems," *Proc. 2nd IFIP Working Conf. Dependable Computing for Critical Applications*, Tucson, Arizona, Feb. 1991.

[Dunkel90] J. Dunkel, "On the Modeling of Workload-Dependent Memory Faults," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, pp. 348-355, June 1990.

[Dupuy90] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A Network Simulation and Prototyping Testbed," *Communications of the ACM*, Vol. 33, No. 10, pp. 64-74, Oct. 1990.

[Finelli87] G.B. Finelli, "Characterization of Fault Recovery through Fault Injection on FTMP," *IEEE Trans. Reliability*, Vol. R-36, No. 2, pp. 164-170, June 1987.

[Goel85] A.L. Goel, "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE Trans. Software Engineering*, Vol SE-11, No. 12, pp. 1411-1423, Dec. 1985.

[Goswami90] K.K. Goswami and R.K. Iyer, "DEPEND: A Design Environment for Prediction and Evaluation of System Dependability," *Proc. 9th Digital Avionics Systems Conference*, Oct. 1990.

[Goswami91] K.K. Goswami and R.K. Iyer, "A Simulation-Based Study of a Triple Modular Redundant System using DEPEND," *Proc. 5th Int. Tests, Diagnosis, Fault Treament Conf.*, pp. 300-311, Sept. 1991.

[Goswami92] K.K. Goswami and R.K. Iyer, *DEPEND: A Simulation-Based Environment for System Level Dependability Analysis*, Technical Report, CRHC 92-11, University of Illinois at Urbana-Champaign, June 1992.

[Goswami93a] K. K. Goswami and R. K. Iyer, "Use of Hybrid and Hierarchical Simulation to Reduce Computation Costs," *Int. Workshop Modeling Analysis & Simulation of Computer & Telecomm. Sys.*, San Diego, CA, pp. 197-202, Jan. 1993.

[Goswami93b] K. K. Goswami, R. K. Iyer and M. Devarakonda, "Prediction-Based Dynamic Load-Sharing Heuristics," *IEEE Trans. Parallel and Distributed Computing*, May 1993, to be published.

[Goswami93c] K. K. Goswami and R. K. Iyer, "Simulation of Software Behavior Under Hardware Faults," *Proc. 23rd Int. Symp. Fault-tolerant Computing*, June 1993.

[Goyal87] A. Goyal, S.S. Lavenberg and K.S. Trivedi, "Probabilistic Modeling of Computer System Availability," *Annals of Operations Research*, No. 8, pp. 285-306, March 1987.

[Goyal92] A. Goyal, P. Shahabuddin, P. Heidelberger, V.F. Nicola, and P.W. Glynn, "A Unified Framework for Simulating Markovian Models of Highly Dependable Systems," *IEEE Trans. Computers*, Vol. 41, No. 1, pp. 36-51, Jan. 1992.

[Gray90] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Trans. Reliability*, Vol. 39, No. 4, pp. 409-418, Oct. 1990.

[Gray91] J. Gray and D.P. Siewiorek, "High-Availability Computer Systems," *IEEE Computers*, pp. 39-48, Sept. 1991.

[Gunneflo89] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation," *Proc. 19th Int. Symp. Fault-Tolerant Computing*, pp. 340-347, June 1989.

[Hansen92] J.P. Hansen and D.P. Siewiorek, "Models for Time Coalescence in Event Logs," *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pp. 221-227, July 1992.

[Heimann90] D.I. Heimann, N. Mittal and K.S. Trivedi, "Availability and Reliability Modeling for Computer Systems," *Advances in Computers*, Vol. 31, pp. 175-233, 1990.

[Hogg83] R.V. Hogg and E.A. Tanis, *Probability and Statistical Inference*, Second Edition, Macmillan Publishing Co., Inc., 1983.

[Howard71] R.A. Howard, *Dynamic Probabilistic Systems*, John Wiley & Sons, Inc., New York, 1971.

[Hsueh87] M.C. Hsueh and R.K. Iyer, "A Measurement-Based Model of Software Reliability in a Production Environment," *Proc. 11th Annual Int. Computer Software & Applications Conf.*, pp. 354-360, Oct. 1987.

[Hsueh88] M.C. Hsueh, R.K. Iyer, and K.S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Trans. Computers*, Vol. 37, No.4, pp. 478-484, April 1988.

[Iyer82a] R.K. Iyer and D.J. Rossetti, "A Statistical Load Dependency Model for CPU Errors at SLAC," *Proc. 12th Int. Symp. Fault-Tolerant Computing*, pp. 363-372, June 1982.

[Iyer82b] R.K. Iyer, S.E. Butner, and E.J. McCluskey, "A Statistical Failure/Load Relationship: Results of a Multicomputer Study," *IEEE Trans. Computers*, Vol. C-31, No. 7, pp. 697-705, July 1982.

[Iyer85a] R.K. Iyer and P. Velardi, "Hardware-Related Software Errors: Measurement and Analysis," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 2, pp. 223-231, Feb. 1985.

[Iyer85b] R.K. Iyer and D.J. Rossetti, "Effect of System Workload on Operating System Reliability: A Study on IBM 3081," *IEEE Trans. Software Engineering*, Vol. SE-11, No. 12, pp. 1438-1448, Dec. 1985.

[Iyer86] R.K. Iyer, D.J. Rossetti and M.C. Hsueh, "Measurement and Modeling of Computer Reliability as Affected by System Activity," *ACM Trans. Computer Systems*, Vol. 4, No. 3, pp. 214-237, Aug. 1986.

[Iyer90] R.K. Iyer, L.T. Young, and P.V.K. Iyer, "Automatic Recognition of Intermittent Failures: An Experimental Study of Field Data," *IEEE Trans. Computers*, Vol. 39, No. 4, pp. 525-537, April 1990.

[Jewett91] D. Jewett, "Integrity S2: A Fault-Tolerant Unix Platform," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, June 1991.

[Kahn53] H. Kahn and A. W. Warshall, "Methods of Reducing Sample in Monte Carlo Computations," *Journal of the Operations Research Society of America*, Vol. 1, No. 5, pp. 263-278, 1953.

[Kanawati92] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pp. 336-344, July 1992.

[Kao93] W. Kao, R.K. Iyer, and D. Tang, "FINE: A Fault Injection and Monitor Environment for Tracing the UNIX System Behavior under Faults," *IEEE Transactions on Software Engineering*, Dec. 1993, to be published.

[Karlsson89] J. Karlsson, U. Gunneflo, and J. Torin, "The Effects of Heavy-ion Induced Single Event Upsets in the MC6809E Microprocessor," *Proc. 4th Int. Conf. Fault-Tolerant Computing Systems*, GI/ITG/GMA, Baden, Germany, 1989.

[Katzman78] J.A. Katzman, "A Fault-Tolerant Computing System," *Proc. Int. Hawaii Conference on System Science*, pp. 85-102, 1978.

[Kendall77] M.G. Kendall, *The Advanced Theory of Statistics*, Oxford University Press, 1977.

[Kobayashi78] H. Kobayashi, *Modeling and Analysis: An Introduction to System Performance Evaluation Methodology*, Addison-Wesley Publishing Co., 1978.

[Kronenberg86] N.P. Kronenberg, H.M. Levy and W.D. Strecker, "VAXcluster: A Closely-Coupled Distributed System," *ACM Trans. Computer Systems*, Vol. 4, No. 2, pp. 130-146, May 1986.

[Lala83] J. Lala, "Fault Detection, Isolation and Reconfiguration in FTMP: Methods and Experimental Results," *Proc. 5th AIAA/IEEE Digital Avionics Systems Conference (DASC)*, pp. 21.3.1-21.3.9, 1983.

[Laprie84] J.C. Laprie, "Dependable Evaluation of Software Systems in Operation," *IEEE Trans. Software Engineering*, Vol. SE-10, No. 6, pp. 701-714, Nov. 1984.

[Laprie85] J.C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pp. 2-11, June 1985.

[Law82] A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw Hill Book Company, 1982.

[Lee91] I. Lee, R.K. Iyer and D. Tang, "Error/Failure Analysis Using Event Logs from Fault Tolerant Systems," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, pp. 10-17, June 1991.

[Lee92] I. Lee and R.K. Iyer, "Analysis of Software Halts in Tandem System," *Proc. 3rd Int. Symp. Software Reliability Engineering*, pp. 227-236, Oct. 1992.

[Lee93a] I. Lee, D. Tang, R.K. Iyer, and M.C. Hsueh, "Measurement-Based Evaluation of Operating System Fault Tolerance," *IEEE Transactions on Reliability*, June 1993, to be published.

[Lee93b] I. Lee and R.K. Iyer, "Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System," *Proc. 23rd Int. Symp. Fault-Tolerant Computing*, June 1993.

[Lewis84] E.E. Lewis and F. Bohm, "Monte Carlo Simulation of Markov Unreliability Models," *Nuclear Eng. and Design*, Vol. 77, pp. 49-62, 1984.

[Lin90] T.T. Lin and D.P. Siewiorek, "Error Log Analysis: Statistical Modeling and Heuristic Trend Analysis," *IEEE Trans. Reliability*, Vol. 39, No. 4, pp. 419-432, Oct. 1990.

[Littlewood80] B. Littlewood, "Theories of Software Reliability: How Good Are They and How Can They Be Improved?" *IEEE Trans. Software Engineering*, Vol. SE-6, No. 5, pp. 489-500, Sept. 1980.

[Lomelino86] D. Lomelino and R. Iyer, "Error Propagation in a Digital Avionic Processor: A Simulation-Based Study," *Proc. Real Time Systems Symposium*, pp. 218-225, Dec. 1986.

[Maxion90a] R.A. Maxion, "Anomaly Detection for Diagnosis," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, pp. 20-27, June 1990.

[Maxion90b] R.A. Maxion and F.E. Feather, "A Case Study of Ethernet Anomalies in a Distributed Computing Environment," *IEEE Trans. Reliability*, Vol. 39, No. 4, pp. 433-443, Oct. 1990.

[McConnel79] S.R. McConnel, D.P. Siewiorek, and M.M. Tsao, "The Measurement and Analysis of Transient Errors in Digital Compute Systems," *Proc. 9th Int. Symp. Fault-Tolerant Computing*, pp. 67-70, 1979.

[McGough81] J.G. McGough and F.L. Swern, *Measurement of Fault Latency in a Digital Avionic Mini Processor*, NASA Contract Report 3462, NASA, Washington, DC 1981.

[Meyer88] B. Meyer, *Object-oriented Software Construction*, Prentice Hall International Series in Computer Science, 1988.

[MeyerJ80] J.F. Meyer, "On Evaluating the Performability of Degradable Computing Systems," *IEEE Trans. Computers*, Vol. C-29, No. 8, pp. 720-731, Aug. 1980.

[MeyerJ88] J.F. Meyer and L. Wei, "Analysis of Workload Influence on Dependability," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp. 84-89, June 1988.

[MeyerJ92] J.F. Meyer, "Performability: A Retrospective and Some Pointers to the Future," *Performance Evaluation*, Vol. 14, pp. 139-156, Feb. 1992.

[Migneault85] ?.?. Migneault, "The Diagnostic Emulation Technique in the Airlab," Internal Report, NASA-Langley Research Center, 1985.

[Mourad87] S. Mourad and D. Andrews, "On the Reliability of the IBM MVS/XA Operating System," *IEEE Trans. Software Engineering*, Vol. SE-13, No. 10, pp. 1135-1139, Oct. 1987.

[Musa87] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Company, 1987.

[Randell75] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Engineering*, Vol. SE-1, No. 2, June 1975.

[Reibman89] A. Reibman, R. Smith, and K. Trivedi, "Markov and Markov Reward Model Transient Analysis: An Overview of Numerical Approaches," *European Journal of Operational Research*, Vol. 40, pp. 257-267, 1989.

[Rogers85] W. Rogers and J. Abraham, "CHIEFS: A Concurrent Hierarchical and Extensible Fault Simulator," *Proc. IEEE Int. Test Conference*, pp. 710-716, 1985.

[Ruehli83] A.W. Ruehli and G.S. Ditlow, "Circuit Analysis, Logic Simulation, and Design Verification for VLSI," *Proc. of the IEEE*, vol. 71, No. 1, pp34-48, Jan. 1983.

[Sahner87] R.A. Sahner and K.S. Trivedi, "Reliability Modeling Using SHARPE," *IEEE Trans. Reliability*, Vol. R-36, No. 2, pp. 186-193, June 1987.

[Saleh84] R.A. Saleh, "Integrated Timing Analysis and SPLICE1," Mem. UCB/ERL M84/2, Elec. Res. Lab., U.C. Berkeley, 1984.

[Saleh90] R.A. Saleh and A.R. Newton, *Mixed-Mode Simulation*, Kluwer Academic Publishers, June 1990.

[Segall88] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "FIAT - Fault Injection Based Automated Testing Environment," *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pp. 102-107, June 1988.

[Schwetman86] H. Schwetman, "CSIM: A C-Based Process-Oriented Simulation Language," *Proc. Winter Simulation Conf.*, 1986.

[Shin84] K. Shin and Y. Lee, "Error Detection Process - Model, Design, and Its Impact on Computer Performance," *IEEE Trans. Computers*, vol. C-33, No. 6, pp. 529-540, June 1984.

[Shin86] K.G. Shin and Y.H. Lee, "Measurement and Application of Fault Latency," *IEEE Trans. Computers*, Vol. C-35, No. 4, pp. 370-375, April 1986.

[Siewiorek78] D.P. Siewiorek, V. Kini, H. Mashburn, S.R. McConnel, and M. Tsao, "A Case Study of C.mmp, Cm*, and C.vmp: Part I — Experience with Fault Tolerance in Multiprocessor Systems," *Proc. of the IEEE*, Vol. 66, No. 10, pp. 1178-1199, Oct. 1978.

[Siewiorek92] D.P. Siewiorek and R.W. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, Bedford, Mass., 1992.

[Sullivan91] M.S. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability — A Study of Field Failures in Operating Systems," *Proc. 21st Int. Symp. Fault-Tolerant Computing*, pp. 2-9, June 1991.

[Sullivan92] M.S. Sullivan and R. Chillarege, "A Comparison of Software Defects in Database Management Systems and Operating Systems," *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pp. 475-484, July 1992.

[Tang90] D. Tang, R.K. Iyer and Sujatha Subramani, "Failure Analysis and Modeling of a VAXcluster System," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, pp. 244-251, June 1990.

[Tang91] D. Tang and R. K. Iyer, "Impact of Correlated Failures on Dependability in a VAXcluster System," *Proc. 2nd IFIP Working Conf. Dependable Computing for Critical Applications*, Tucson, Arizona, Feb. 1991.

[Tang92a] D. Tang and R.K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *IEEE Trans. Computers*, Vol. 41, No. 5, pp. 567-577, May 1992.

[Tang92b] D. Tang and R.K. Iyer, "Analysis of the VAX/VMS Error Logs in Multicomputer Environments — A Case Study of Software Dependability," *Proc. Third Int. Symp. Software Reliability Engineering*, Research Triangle Park, North Carolina, pp. 216-226, Oct. 1992.

[Tang93a] D. Tang and R.K. Iyer, "Dependability Measurement and Modeling of a Multicomputer Systems," *IEEE Trans. Computers*, Vol. 42, No. 1, pp. 62-75, Jan. 1993.

[Tang93b] D. Tang and R.K. Iyer, "MEASURE+ — A Measurement-Based Dependability Analysis Package," *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, Santa Clara, California, pp. 110-121, May 1993.

[Trivedi82] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Trivedi92] K.S. Trivedi, J.K. Muppala, S.P. Woolet, and B.R. Haverkort, "Composite Performance and Dependability Analysis," *Performance Evaluation*, Vol. 14, pp.197-215, Feb. 1992.

[Tsao83] M.M. Tsao and D.P. Siewiorek, "Trend Analysis on System Error files," *Proc. 13th Int. Symp. Fault-Tolerant Computing*, pp. 116-119, June 1983.