# DEPEND: A Simulation-Based Environment for System Level Dependability Analysis

Kumar K. Goswami Ravishankar K. Iyer Luke Young

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801

**Abstract**

**The paper presents the rationale for a functional simulation tool, called DEPEND, which provides an integrated design and fault injection environment for system level dependability analysis. The paper discusses the issues and problems of developing such a tool, and describes how DEPEND tackles them. Techniques developed to simulate realistic fault scenarios, reduce simulation time explosion, and handle the large fault model and component domain associated with system level analysis are presented. Examples are used to motivate and illustrate the benefits of this tool. To further illustrate its capabilities, DEPEND is used to simulate the Unix-based Tandem triple-modular-redundancy (TMR) based prototype fault-tolerant system and evaluate how well it handles near-coincident errors caused by correlated and latent faults. Issues such as memory scrubbing, re-integration policies and workload dependent repair times which affect how the system handles near-coincident errors are also evaluated. Unlike any other simulation-based dependability studies, the accuracy of the simulation model is validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a production Tandem machine.**

## Keywords:

**Simulation, fault injection, dependability analysis, correlated errors, latent errors, inter-component dependence, object-oriented design, Tandem TMR-based prototype analysis, validation.**

---

• Kumar K. Goswami was with the Center for Reliable and High-Performance Computing and is now with Tandem Computers, Cupertino, CA 95014-2599. E-mail: kumar@loc3.tandem.com.

• Ravishankar K. Iyer is with the Center for Reliable and High-Performance Computing at the University of Illinois, Urbana, IL 61801. E-mail: iyer@crhc.uiuc.edu.

• Luke Young was with the Center for Reliable and High-Performance Computing and is now with Tandem Computers, Austin, TX 78728. E-mail: luke@mpd.tandem.com.

*This work was supported by the National Aeronautics and Space Administration under grant NAG-1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), by a NASA Graduate Student Researchers Fellowship, and by the Advanced Research Projects Agency under grant DABT63-94-C-0045. The findings, opinions, and recommendations expressed herein are those of the authors and do not necessarily reflect the position or policy of the United States Government and no official endorsement should be inferred.*

# 1 Introduction

The growth in the demand for dependable systems and their increasing complexity has created a need for automated design and analysis tools. The design life cycle of a system can be viewed as consisting of three stages. In the first stage, the hardware and software architecture are not established and detailed information about them is not available. Continuous time Markov chains (CTMC) and tools that solve CTMC models are ideally suited for this early stage to conduct high-level dependability (availability, reliability and fault tolerance) trade-off analysis and establish the core system architecture. As the design progresses and more information is made available, detailed evaluation of the system under realistic workload and failure conditions is necessary. Functional simulation tools are better suited at this stage because they can accurately model the functional behavior of the system, the inter-component dependencies, workload patterns and specific repair and reconfiguration schemes. Once a prototype system exists, actual fault-injection can be performed to determine whether it meets its dependability specifications. There has been substantial research and development of analytical tools used during the first stages of a design. In [12], the authors describe and compare several tools that solve CTMC models. There is increasing work in software and hardware fault injection of prototype systems [1, 3, 18, 27, 28, 40, 45, 47]. The focus of this paper is on the second stage, and in particular on the use of functional simulation-based tools for system level dependability analysis.

One can ask, given the large number of analytical 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 and fault-injection environments? This paper motivates the need for functional simulation tools, and in particular integrated design and fault injection tools, for system level dependability analysis. It discusses the issues and problems of developing such tools and describes how they are tackled by a tool called DEPEND. Techniques developed to simulate realistic fault scenarios, reduce simulation time explosion, and handle the large fault model and component domain associated with system level analysis are presented. Examples are used to demonstrate the benefits and uses of the tool. Finally, some of the capabilities and the features of DEPEND are illustrated with a fault injection study of the Tandem system—a TMR-based, fault-tolerant computer. It is well established that this system is very effective against single faults [25, 47]. An important question is how such systems cope with near-coincident errors generally caused by correlated failures and latent faults. Architectural issues that have a bearing on how the system handles near-coincident faults include memory scrubbing, re-integration policies and workload dependent repair times. To study these issues, DEPEND was used to simulate the target system and evaluate the combined effect of all these factors. This comprehensive study demonstrates the capabilities of DEPEND in a realistic setting. The simulation of the Tandem system is validated by comparing the results of the simulations with measurements obtained from fault injection experiments conducted on a prototype Tandem TMR machine. To our knowledge, no other simulation-based dependability study has been validated in such a fashion.

## 2 Existing Simulation Tools

There is a lack of simulation tools for system level dependability analysis. Most simulation tools are designed to facilitate performance analysis (CSIM [39], ASPOL [31], SES Workbench [41], RESQ [38]). VHDL [22] is a powerful hardware specification language but it does not contain built-in facilities to support dependability analysis. NEST [11] is a functional simulation and proto-typing tool used explicitly to analyze distributed networks and system protocols. It is very specialized and has a limited set of facilities to fail links and nodes. Another functional simulation tool called REACT [8] is specifically designed for analyzing alternative TMR architectures. UltraSAN [37] and the Rainbow Net [26] are Petri-net tools. A extended petri-net structure is used to input a model and solve it via simulation. Though the Petri-net tools have greater applicability than analytical tools, they are not as versatile as functional simulation tools. They provide only a limited set of fault models and provide no mechanism to reduce simulation time explosion.

## 3 System Level Functional Simulation Tool

The main advantage of functional simulation is that it can model the behavior of hardware and software architectures with greater accuracy. However, there are at least four issues that impede the development of general-purpose, functional simulation tools for system level dependability analysis. The first is a lack of well established system fault models. This is partly due to a second issue which is 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 that can consistently be 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. This makes it difficult to establish a single fault model that can be consistently applied to all components. Limiting the types of components or fault models represented is one solution but it restricts the tool's applicability. In addition to the diversity of the components that comprise a system, two similar components (such as two CPUs) can have different functions and behavior. The third issue is simulation time explosion which occurs when extremely small failure probabilities require large simulation runs to obtain statistically significant results. Finally, the fourth issue relates to simulating large complex systems and the design time required to develop and debug a functional simulation model. A general-purpose simulation-based system level dependability analysis tool must effectively handle the large component and fault model domain, furnish ways to accelerate the simulation and provide an environment that facilitates the development of appropriate simulation models.

DEPEND exploits the properties of the object-oriented paradigm and provides acceleration techniques to tackle these issues. The first two are solved with the combined use of two criteria: modular decomposition and modular composability [33]. 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: an element that determines when to inject

and interrupt the system, and an element that determines the response to a fault (the fault model), the two criteria are met. The first object [1] 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 pre-defined 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 to reduce the development time and effort needed to build simulation models. In addition to decomposition, composition and parameterization, the concept of inheritance [33] 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 an existing object and develop more specialized objects with minimum effort.

The acceleration techniques to speed-up simulation are presented in the next section which describes the DEPEND environment in detail. The object-oriented paradigm also facilitates the implementation and automation of the acceleration techniques.

# 4 The DEPEND Environment

DEPEND is an integrated design and fault injection environment. It provides facilities to rapidly model fault-tolerant architetures and conduct extensive fault injection studies. It is a functional, process-based [29, 39] 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.

The steps required to develop and execute a model are shown in figure 1. 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.

[1] In object-oriented programming, the 'elements' are classes. An object is an instance of a class, but here, for simplicity, they will be referred to simply as objects.

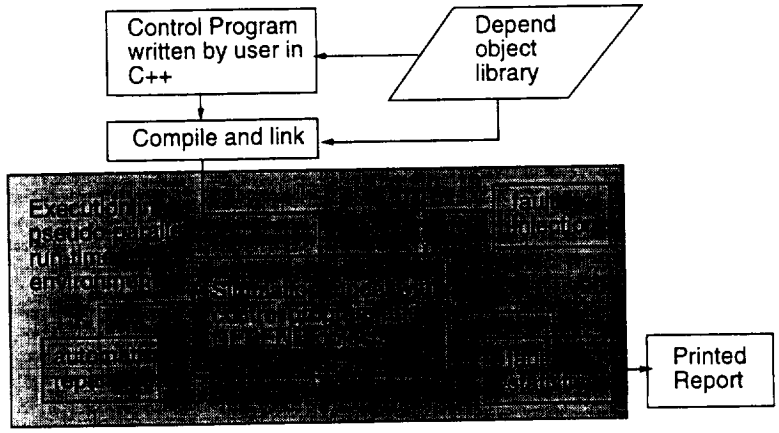


Figure 1: Steps in developing and simulating a model with DEPEND.

| Name | Type | Description |
|---|---|---|
| Active_elem | Elementary | Simulates basic server. Offers usage disciplines: first come first serve, round robin, etc. Allows manual fault injection & repair. |
| Injector | Elementary | Injects faults using distributions & trace files. Offers workload based injections. Injects correlated faults. |
| Checksum | Elementary | Compute checksums. |
| Fault Reporter | Elementary | Compiles fault statistics. Displays MTBF, MTBR, availability and coverage. Provides details of every fault injected and repair attempted. |
| Voter | Elementary | Simulates a basic voter with timeout. Default voting scheme: byte by byte comparison. Allows user defined voting algorithms. |
| Server | Complex | Inherits Active_elem. Simulates server with spares. Three sparing policies: no spare, graceful degradation, stand-by sparing. Automatic repair and reconfiguration. Automatic injection of faults. |
| Link | Complex | Simulates communication channels. Inherits Server. Several fault types: link dead, packet corruption, packet loss and user defined faults. Automatic retry. |
| 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. |

Table 1: Some objects in the DEPEND library.

## 4.1 The DEPEND Object Library

DEPEND is a library of elementary and complex objects. Elementary objects provide basic functions like 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. These few key objects can be combined and replicated to simulate a wide range of fault-tolerant architectures. All objects are designed with four criteria. The objects:

1. simulate the general behavior of a component (decomposability & composability),
2. connect with other objects via function calls (composability),
3. allow users to specify key parameters (parameterization), and
4. provide default functions to minimize design time.

Table 1 lists key objects in the library. A detailed description of all objects can be found in [16].

## 4.2 DEPEND Fault Models and Injection Facilities

### 4.2.1 Fault models

DEPEND uses functional fault models to simulate the system level manifestation of gate-level faults such as stuck-at faults. Functional fault models are used because they are best suited for system level fault injection where the focus is on the behavior of a component. Default fault routines are provided for each object to minimize user design time. The default CPU fault model assumes that the processor hangs when a fault is discovered. If the fault is transient, it disappears when the CPU is restarted. If the fault is permanent, it is corrected only when the CPU is replaced. This fault model represents the functional behavior of low-level, stuck-at faults and transient errors in key CPU registers and functional units. The default fault model for a communication medium simulates the effects of a noisy communication channel by corrupting bits in a message or destroying the message. Two default fault models are available for memory and I/O subsystems. Either a bit of a word is flipped or a flag is raised to represent the error. The error can be detected with a byte-by-byte comparison or by checksum comparison if the error is a flipped bit. Otherwise, it can be detected by checking the status of the flag.

### 4.2.2 Fault injector

The *fault injector* is a fundamental object of DEPEND. It encapsulates the mechanism for injecting faults. To use the injector, a user specifies the number of components, the time to fault distribution for each component, and the fault subroutine which specifies the fault model. The distributions supported are constant time (mostly used for debugging), exponential, hyperexponential and Weibull. The object also allows user specified distributions. When initialized, the injector samples from a random number generator to determine the earliest time to fault, sleeps until that time and calls the fault subroutine.

Initially, the injector used conditional failure distributions to determine the time to next fault, assuming a set of components have independent, identically distributed failure distributions. For instance, for a system with two components using a gracefully degrading sparing policy, the time to failure of the second component, $X$, given that the first component failed at time $t$ is given by $Pr[X < x|X > t]$. For a Weibull time to fault distribution with rate parameter $\lambda$ and shape parameter $\alpha$, the conditional distribution is:

$$Pr[X < x|X > t] = 1 - exp^{\lambda[(x+t)^{\alpha} - t^{\alpha}]} \quad (1)$$

This approach has several drawbacks. For non-exponential distributions, the conditional distributions become complex and cumbersome as repaired components are re-integrated into the system or cold spares are activated. The conditional distributions depend on the sparing policy. As a result, additional information regarding sparing policies has to be specified. The routines used to generate random samples are more compute bound. Generating a random sample using the inverse transform method for equation 1 requires and additional subtraction and an extra call to the math library's power() function than for a Weibull distribution. For large simulation runs, where thousands of faults are injected, the time spent on these additional calls becomes significant.

The current version of the injector uses a table-based approach. An entry is kept for each component, specifying its condition (OK, Failed), injection status (Injection off, Injection on), time to fault distribution, and time to next fault. The algorithm used to determine which component to inject is:

```
Initialize (performed one time)
    do for all components
        if (component is OK & On)
            compute and store time to fault
        else
            time to fault is ∞
        end if
    end do

Main body
    do forever
        find minimum_time_to_fault among components
        sleep (minimum_time_to_fault - current_time)
        if (sleep not aborted)
            call fault subroutine
            set time to fault of this component to ∞
        end if
    end do
```

Any time a component is repaired or turned on, its time to fault is computed and entered in the table. The injector is then awakened so that it takes the new component's fault time into account.

The table-based approach is versatile. The time to fault distributions of the components do not have to be identical and any user specified distribution can be supported. For example, the full "bathtub" reliability curve can be model with DEPEND. The table-based approach automatically takes the age of each component into account without using conditional probability distributions and averts the problem with modeling local and global times found in most analytical tools [44].

DEPEND provides a workload dependent injection facility to model the workload/failure dependency observed in [24, 4]. It can be used to test systems under stress conditions. To implement a workload dependent injection strategy, a statistical clustering algorithm is first used to identify high-density regions of the workload. These regions (or states) are used to specify a state transition diagram that characterizes the workload [20]. Associated with each state is a visit counter which counts the number of visits to that state and a fault rate, $\lambda$, which the system experiences in that state. The user provides a workload function which the injector polls periodically to identify the workload state and to update its visit counter. For example, the workload function may be the utilization of a processor or it may be any other function that provides a measure between 0 (low workload) and 1 (high workload). Based on an *injection_interval* specified by the user, the information from the state transition diagram is used to estimate a weighted average failure arrival rate ($Wgt\_lambda$) as follows:

$$Wgt\_lambda = \sum_{i=1}^{N} visit\_ratio_i \times \lambda_i$$

where:

$N$ is the number of states

$$visit\_ratio_i = \frac{counter\ for\ state_j}{total\ visits\ to\ all\ states}$$

Once $Wgt\_lambda$ is determined, it is used to compute the probability of a failure injection ($P\_inject(t)$) over the last interval $t$ (= *injection_interval*) as follows:

$$P\_inject(t) = 1 - e^{-Wgt\_lambda \times t}$$

The *fault injector* illustrates what we mean by "providing a simulation framework". The injector provides the basic algorithms and mechanisms needed to inject faults allowing the user to concentrate on the application specific aspects, the fault model and the simulation of a fault. Furthermore, the modular, object-oriented approach allows the user to easily experiment with different time to fault distributions, fault models and workload functions. Other DEPEND objects are similarly designed to provide the functionality that are commonly needed without restricting the applicability of the object.

## 4.3 Simulation Time Acceleration

A drawback with simulation is that it can be execution time bound. DEPEND provides three ways to reduce simulation time explosion. The main technique is the use of hierarchical simulation. The foundation of the approach is based on the notion of variable aggregation and decomposability [9]. With this technique, a large complex model is broken down into simpler submodels. The submodels are analyzed individually and their results are combined to derive the solution of the entire system. So long as the interactions among the subsystems are weak, this approach provides valid results. The approach is ideally suited for dependability studies because the models can be broken into two

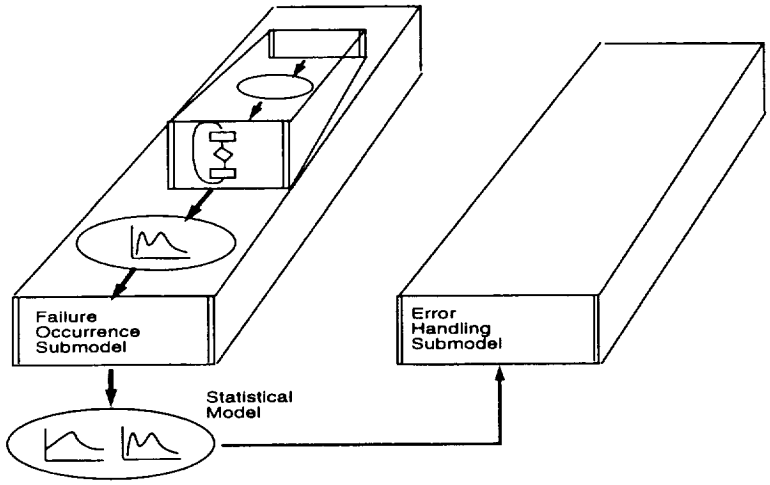


Figure 2: Hierarchical simulation within a submodel and between submodels.

submodels, a fault occurrence submodel and an error handling submodel, whose interactions are typically weak. Decomposition is used in HARP [2] to reduce state space explosion and solve stiff models. We adapt this approach for functional simulation and use it in two ways (figure 2). First, we take advantage of the cause and effect relationship between the failure occurrence submodel and the repair submodel to pre-simulate the failure occurrence submodel to extract a statistical model that captures its behavior and then use it to drive the repair submodel. An example of a statistical model is the failure distribution of the components of the system. Empirical tests show that the approach provides accurate results and has reduced execution times for the Tandem simulation, in cases, from 6 hours to just a few minutes [15]. Second, we use this approach to accelerate the simulation within each submodel. For instance, though DEPEND can execute actual software, the time to execute a naive simulation, where periods of several years must be simulated, can be prohibitive. Hierarchical simulation is used to first execute the software under faults to extract a statistical model such as the error detection latency of the software. The statistical model represents the behavior of the software under faults and is used with simulation models at a higher level of abstraction. This hierarchical process can be used repeatedly. The object-oriented paradigm facilitates the implementation of hierarchical simulation. Each submodel is an object with clearly defined inputs and outputs that connect to DEPEND objects that collect statistical models from one object and used them to drive others.

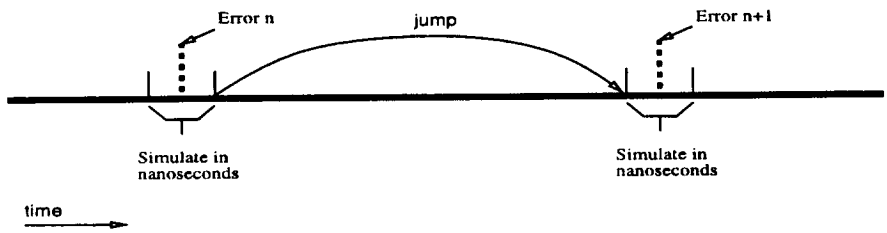


Figure 3: Time acceleration: "Error-driven" simulation.

Another technique, provided by DEPEND to speed-up simulation runs, is a general time acceleration mechanism which allows the simulation to leap forward in time (figure 3). DEPEND objects furnish the time of the next important event, such as the time when the next fault will arrive, or when the next latent fault will be activated. A list of these events that affect dependability measures are kept in a chronologically sorted list. The simulator leaps forward to the time of the event at the head of the list and resumes processing at the granularity of the system clock until the effect of the event has subsided. This is different from regular event-driven simulation because it allows user-specified "unimportant" events to be suspended during leaps while others continue. This acceleration technique is used with the Tandem simulation described in section 6.2.

Finally, DEPEND's C++, process-based environment facilitates the implementation of variance reduction techniques [30]. Unlike other simulation tools, it provides direct control of the simulation engine so that importance sampling techniques can be efficiently implemented. See [35] on the difficulties of implementing such schemes if such control is not provided. We chose not to incorporate any particular importance sampling technique in DEPEND because their use and the measures they can provide are application dependent.

# 5 Benefits of DEPEND

This section illustrates the uses of DEPEND and the need for its integrated design and fault injection environment during the second stage of the design process.

## 5.1 Behavioral Modeling

Analytical and Petri-net tools use stochastic 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. DEPEND uses stochastic modeling, but it also permits behavioral modeling which does not require that the effect of the faults be pre-defined. An example that demonstrates this capability of DEPEND is a study in which a distributed system using a centralized, prediction-based load balancing scheme is evaluated under faults [14, 13] (see figure 4). The load-balancing software that makes task placement decisions and maintains the database is actually executed within the DEPEND environment upon 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 into the CPU containing the load-balancing software, to erase its database. The effect of all 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 prespecify:

- the probability that a fault will corrupt the database,
- how each fault will corrupt the database, and
- which portions of the database will be corrupted

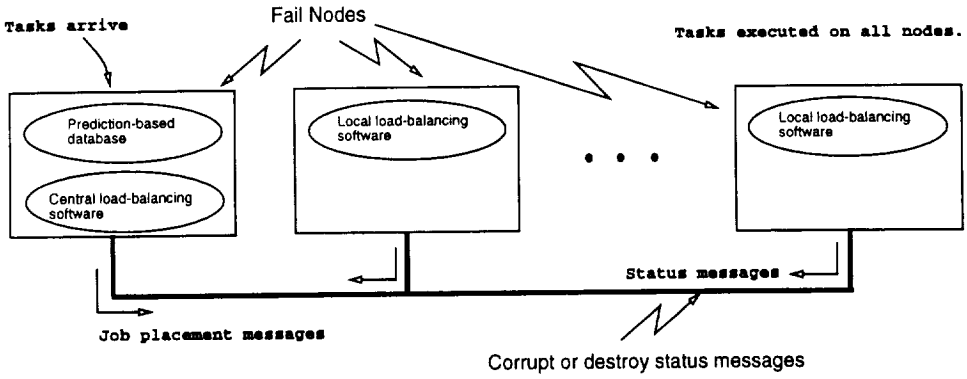


Figure 4: Distributed system executing load balancing software.

and would have to quantify:

- 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 a thorough *prior* fault injection study. Because DEPEND executes the 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 quantize 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 [13]). 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. Not only are such studies beyond the range of analytical tools, to the authors' knowledge, this experiment would be difficult to conduct with many current software fault injection tools.

## 5.2 Modeling Real Fault Scenarios

DEPEND uses a combination of behavioral and probabilistic modeling to simulate many realistic fault scenarios. In this subsection, a simple example is used to illustrate how DEPEND models latent errors that can substantially degrade system reliability. Latent errors can remain undetected in a system for long periods of time and are a potential hazard [6]. A measurement study of a VAX 11/780 [5] has shown the mean latency of an error can be in the order of minutes ($\mu = 44min.$, $\sigma = 29min.$) during peak hours and to several hours ($\mu = 8hrs.$, $\sigma = 4hrs.$) during off-times.

Modeling latent errors is difficult with Markov models for several reasons. First, the state space of the Markov model can be large, even for small systems, if each latent error and its location

within a component is represented. Second, simplifying assumptions such as independent repair processes must be eliminated in order to accurately evaluate the impact of latent errors. For

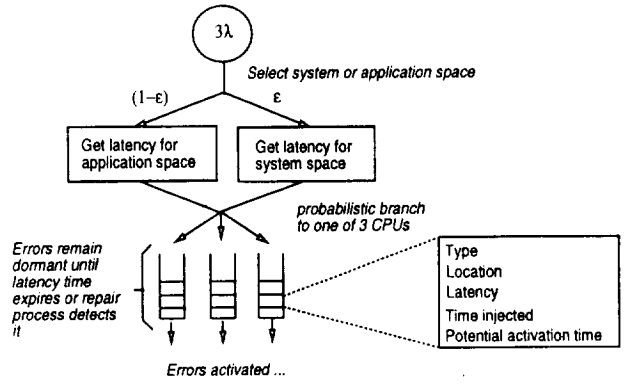


Figure 5: The error injection process that models error latency.

example, in a self-repairing system like the Tandem TMR-based prototype (detailed description can be found in section 6.1), the healthy processors reconfigure or repair a failed processor. If latent errors are detected in the healthy processors during a repair, the system fails. Modeling this *inter-component dependence* typically requires that the entire CTMC, its failure and repair process, be evaluated together, thus potentially leading to large, stiff models. In [10], the authors present a novel decomposition technique to avoid such large, stiff models. With their approach, the repair coverage is evaluated in isolation and then 'adjusted' to account for the probability of a second, independent error in another component. The example below extends this analysis to also evaluate the impact of near-coincident errors due to latent errors in a repairing CPU. This example also models the inherent dependencies that exist among the system components.

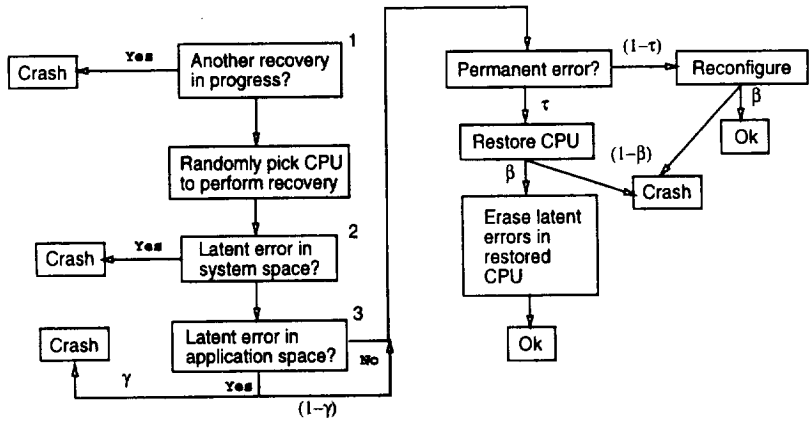


Figure 6: The repair process that considers the state of the other CPUs in the system.

Figure 5 illustrates how latent errors are injected into a system with three processors. DEPEND uses a chronologically sorted queue to maintain the latent errors injected into the system. The information associated with each latent error includes the time at which the error is injected, its location (the component and memory address), and its latency period. Typically, the errors are

| Description | Value assigned |
|---|---|
| Error Arrival Rate, $\lambda_1$ | 0.01388 |
| Repair Rate, $\mu$ | 30.0 |
| Percent Transient error, $\tau$ | 0.99 |
| Repair coverage, $\beta$ | 0.98 |
| Latent error in system space, $\epsilon$ | 0.05 |
| Latency for system space, exponential with mean | 15.0 |
| Latency for application space, exponential with means | 1, 2, 4, 6 & 8 |
| Prob. Failure due to latent errors in repairing CPU, $\gamma$ | 0.1 |

Table 2: The parameters used. All times are in hours.

detected when their latency period expires. However, the errors can be detected earlier by the repair process. Fault injection experiments on the Tandem TMR-based prototype have shown that latent errors in the repairing processor are detected with high probability during repair and reconfiguration because much of the system is exercised during a repair. Figure 6 is a flowchart of the repair process that models this phenomenon. It is invoked each time a latent error is activated. Note that the repair process models near-coincident errors caused by a second, independent error in another processor (box 1). The dynamic determination of the repair coverage which depend on whether there are latent errors in the repairing processor is shown in boxes 2 and 3. To reiterate, by storing latent errors in a queue, DEPEND can dynamically model the *actual* activation time of latent errors which is *dependent* on: 1) the component the error is located in, 2) the location within a component, 3) the failure rate of the components in the system, and 4) the system's repair scheme.

The example is evaluated under three different conditions. First, permanent and transient errors with no latency are injected (M1). Second, near-coincident errors due to a second, independent error in another processor is modeled but error latency is not considered (M2). The decomposition technique in [10] model this second condition. Third, the simulation model described above is evaluated. It includes the conditions in M2, and it also considers near-coincident errors caused by latent errors (M3). The specific parameters of the models are listed in table 2. Figure 7 shows a 33% decrease in MTTF when latent errors and their impact on system reliability is taken into account. The result emphasizes the importance of modeling real phenomenon such as latency, and it demonstrates that by so doing, more precise evaluation of fault tolerant mechanisms and repair schemes are possible. Later, in section 8.2, the "staggered machine failure" phenomenon found to be caused by correlated errors [46] is mimicked by injecting correlated errors with different latency times. This more faithful model is found to produce MTTF figures that are an order of magnitude larger than those produced with traditional models that assume correlated errors are detected simultaneously.

### 5.3 Discussion

These examples illustrate the benefits of the DEPEND environment. The combined fault injection and simulation facility allow accurate modeling of real fault scenarios, repair schemes and the impact of software on system dependability. For these reasons, DEPEND is ideally suited for the

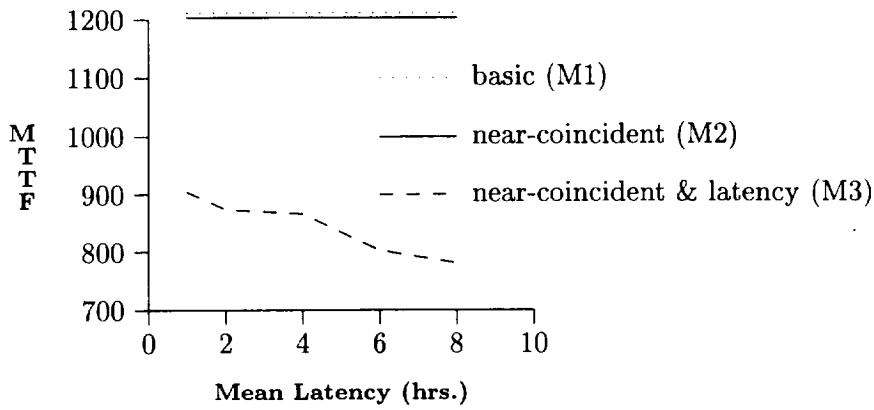


Figure 7: Comparison of system MTTF for the three models.

second stage of a design process when detailed evaluation of a system is necessary to ascertain that dependability specifications are being met. In addition, DEPEND can supplement existing analytical tools by determining fault mechanisms, identifying important fault conditions that must be considered, determining the appropriate analytical models to use, and providing parameter values for them.

# 6 A Case Study to Illustrate DEPEND

This section illustrates the capabilities of DEPEND with a simulation-based fault injection study of a Tandem TMR-based prototype fault tolerant computer system. This system has been shown to be very effective against single faults. An important question is how such a system copes with near-coincident faults generally caused by correlated failures and latent faults. Architectural issues that have a bearing on how the system handles near-coincident errors include memory scrubbing, re-integration policies and workload dependent repair times. To study these issues, DEPEND was used to simulate the TMR-based system and evaluate the combined effect of all these factors. The salient features of the target system are described below, followed by subsections that describe the simulation model.

## 6.1 The Tandem TMR-based Prototype Fault-Tolerant System

The Tandem TMR-based fault-tolerant system [25] is shown in figure 8. In the prototype, each CPU was a MIPS R3000 RISC processor with an on-chip virtual memory mechanism and a separate clock. The processors execute the same instruction stream simultaneously. The processors are synchronized and their requests are checked by the voter when global memory is accessed, I/O is performed, or 2047 cycles have elapsed. If there is a discrepancy during voting, the processor in disagreement is shut down. The faulty processor performs a power-on self-test (POST), and if successful, the system is halted and the contents of the good processors are copied to it. The POST takes approximately 70 seconds and the re-integration takes approximately 2.0 seconds for a system with 8 Mbytes of local memory.

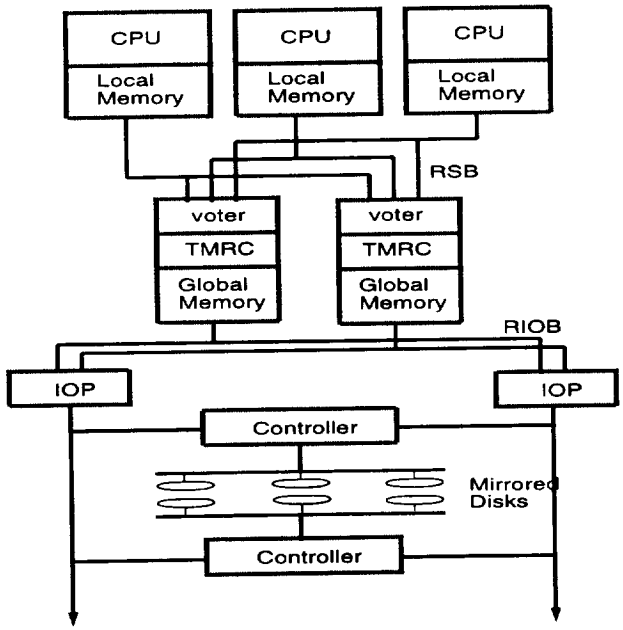


Figure 8: The Tandem TMR-based prototype processing subsystem.

The local memories in the target system did not have parity or ECC circuitry. The system relies on memory scrubbing to correct transient memory errors. The TMRC contains the voter and up to 128MB of global memory. The primary function of the TMRC is to vote upon the transactions sent by the CPUs. The global memories are protected by parity. When a parity error is detected by the TMRC, the backup memory takes over. A global memory is re-integrated in the background, interleaved with ordinary processing. The re-integration time is load dependent. Table 3 contains the measured re-integration times from a system with 32MB of global memory. Global memory re-integration has lower priority than CPU re-integration and is aborted and restarted in case of a CPU re-integration.

| **Percent of Time CPU is Idle** | **Re-integration Time** |
|---|---|
| 99% | 30 sec. |
| 59% | 2 min. 25 sec. |
| 37% | 3 min. 46 sec. |
| 27% | 4 min. and 5 sec. |
| 16% | 4 min. and 40 sec. |
| 0% | 5 min. and 29 sec. |

Table 3: Global memory re-integration times with varying machine idle percentages.

## 6.2 The Simulation Model Developed with DEPEND

Recall that we are interested in evaluating the Tandem TMR-based prototype system and several of its architectural features under the stress of near-coincident errors. To achieve this, several key characteristics of the Tandem system were simulated. These include:

- the loose synchronization policy of the system (the fact that the processors idle at the voters to synchronize, and the exact time needed by the voting operation),
- the CPU (with its local memory) and the global memory structure that is unique to the Tandem system,
- the functional behavior of the error-detection mechanisms of the CPU and global memory structure,
- the CPU off-line POST and the on-line re-integration process and the global memory background re-integration process that are unique to the Tandem system, and
- the behavior of the Tandem system when a CPU and global memory re-integration occurs simultaneously (prioritized re-integration).

These details of the system architecture and how it reacts to faults were determined by studying its layouts, descriptions and manuals, discussing the matter with its designers and conducting several fault injection studies (in addition to the validation experiments mentioned below). Simultaneous injections into various components of the system helped to uncover interesting characteristics of the system that were subsequently incorporated into the simulation model.

The simulation model for the Tandem system, developed with DEPEND, is shown in figure 9. The blocks on the right are the DEPEND objects used in the simulation model. The block on the left summarizes the program written to create the simulation model and control the operations of each of the components.

The **NMR object** in the DEPEND library is the primary object used in the simulation. The NMR object simulates dual self-checking, triple-modular redundant and N-modular redundant systems. The servers idle until they receive a task to process. They then execute for a specified time period and feed their results to the voter. The voter waits for all the servers and then executes a voting algorithm. A timeout condition is used to prevent hanging in cases where a server fails to report to the voter. The NMR object provides two voting algorithms: *bit stream* voting and *error-based* voting. The *bit stream* voting scheme performs a bit by bit comparison of the data deposited by the servers. The *error-based* algorithm flags a server's result as being faulty if an error has been injected into the server. This option was used in this simulation because the processors were not given real data to process. The NMR shuts down the servers with faulty results. Automatic repair schemes are not provided, but functions can be called to repair the individual servers. This feature was used to simulate the automatic re-integration feature of the Tandem system. The NMR object's fault injector injects latent and correlated errors.

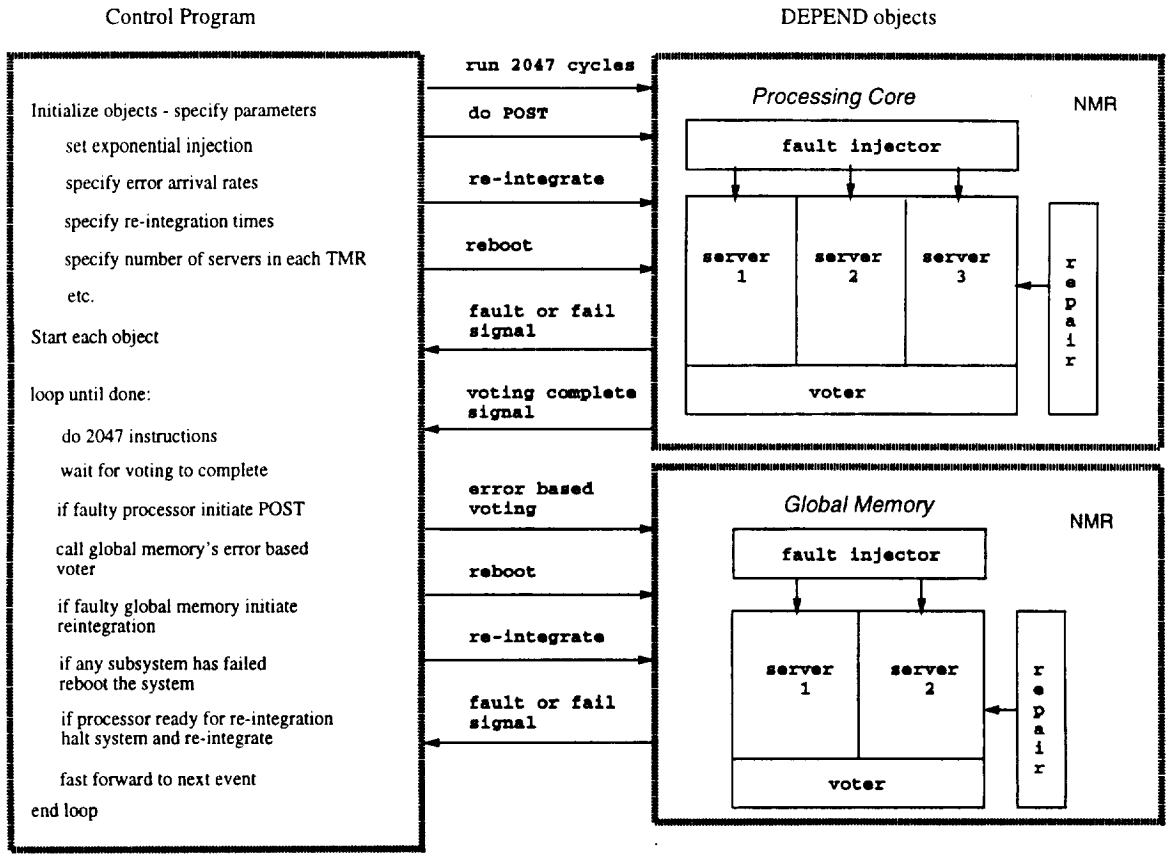


Figure 9: The simulation model of the Tandem TMR-based system developed with DEPEND.

The **processing core** of the TMR-based system is simulated with a NMR object containing 3 servers. Each server simulates a processor board containing a CPU and a local memory. The NMR's injector is used to inject errors into both the processor and the local memory.

The two **global memory** boards are simulated with a NMR with 2 servers. Every 2047 cycles, when the processors synchronize, the global memory's *error-based* voting function is explicitly invoked by the control program to check each server and shut down any that has an active error. This simulates the actual operation of the system because parity errors in the global memory are detected when the processors synchronize at the voter to access global memory.

The **control program** in figure 9 is the only part that is written by the user. It declares instances of the two DEPEND objects and initializes and customizes them. Initialization consists of specifying the error arrival distribution, the error arrival rate, the error latency distribution and so on. Then each object is "started" causing them to automatically perform tasks based on the parameters specified. All actions are automatically logged and the user can call functions to obtain a detailed report of every fault or repair. In addition, statistics such as availability, MTBF, the number of faults injected, the mean time between repair and the repair coverage are available.

The control program simulates the execution of 2047 instructions and the voting process. If any detectable errors are found in any of the components during voting, the components are shutdown. The status of the system is checked to determine whether it has failed (i.e. two CPUs or both global memory boards have failed). If the system has failed, it is rebooted and the simulation restarts. If the system has not failed, a background re-integration process is started for any component that was shutdown earlier by the voter. Finally, the control program checks to see if re-integrations initiated in an earlier cycle has completed. If so, they are handled based on the component type (CPU or global memory) as described in the previous section. Though not shown in figure 9, the memory scrubbing process is also simulated. Although the simulation can produce many results, the one we are most concerned with is the mean time between system failures (MTBF).

## 6.3 The Error Occurrence Process and Experiment Design

The simulation models errors in the storage elements (e.g. registers and memory) of the system. These errors are assumed to be caused by transient faults such as ionization radiation and are assumed to manifest as a single bit flip that can be detected by the voter if it is in the processor board, or detected by the parity error coding scheme if it is in the global memory. Compensating second errors to the same bit are not considered because it is a very rare event.

The TMR's fault injector is used to inject *active*, *latent* and *correlated* errors. Active errors are detected within 2047 cycles from the time they are injected. Latent errors may remain dormant for hours before being detected. As a result, a component may have several latent errors. This models the phenomenon observed when errors were injected into the target system; errors injected into the section of a local memory containing the exception handler, the TLB (Translation Lookaside Buffer) miss handler or the processor register space were usually detected immediately whereas errors injected into other locations in the local or global memories had a much larger latency. A correlated error is two or more errors injected simultaneously into two or more components of a subsystem. Each correlated error is either active or latent. Correlated errors with different latencies is justifiable because there is a high probability that the errors will not occur in the same location in each component and hence will produce different latencies. Furthermore, measurement studies [43, 46] show that correlated errors do not occur simultaneously as it is typically modeled with analytical tools.

The complete error occurrence process for just two CPUs is illustrated in figure 10. A similar process is used to inject errors into the global memory with one exception; only latent errors are injected. The error arrival times are exponentially distributed. When an error is injected, a probabilistic branch is used to determine whether it is a correlated error affecting 2 or more components in a subsystem, or a single error. A probabilistic branch is also used to determine whether the error is active or latent. Eventually, the error is detected by the voter and the component is shutdown and then re-integrated.

### 6.3.1 Simulating error latency

DEPEND provides a simulation-based software model that can evaluate the error behavior of programs caused by hardware faults [17]. This model can be used to obtain application dependent measurements such as error detection latency times. However, in this study, latency distributions

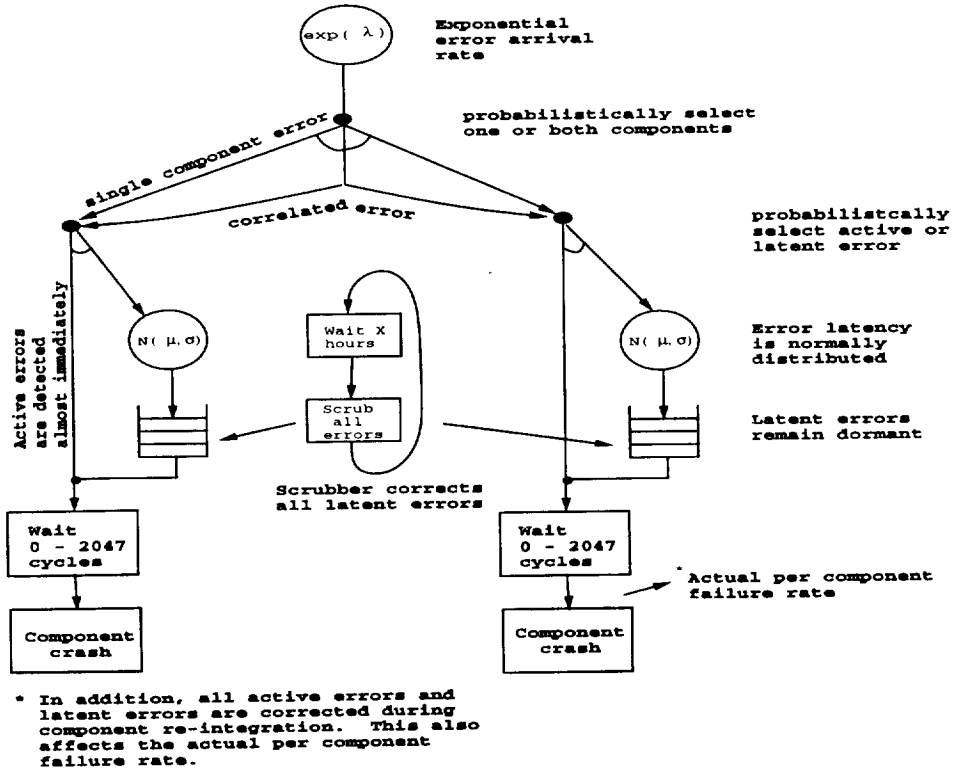


Figure 10: The error occurrence process simulated with DEPEND.

from a measurement study (see section 5.2) are used. When the NMR object injects a latent error, it determines its latency from these distributions. The NMR object uses the scheme described in section 5.2 to model latent errors. This approach provides enormous simulation speed up because the time at which the next latent error will become detectable is known *apriori* thus eliminating the need to simulate the system between occurrences of detectable errors. The time acceleration algorithm described in section 4.3 takes advantage of this property.

### 6.3.2 Assumptions and parameters used in the simulations

Error arrival times are exponentially distributed. The distribution means are based on measurements of real error data collected from a DEC VAXcluster multicomputer system[42]. The mean time between CPU errors ($1/\lambda_{CPU}$) in the system was 265.8 hours with a standard deviation of 497.6 hours. The mean time between memory errors ($1/\lambda_{Memory}$) was 27.0 hours with a standard deviation of 150.4 hours. The combined error arrival rate is approximately 1 error every 24 hours. Of this combined rate, approximately 62% of the errors are injected into the global memory and 38% are injected into the processor board containing the CPUs and the local memories. These numbers are based on the size of the memories (8Mbytes of local memory per board and 32Mbytes of global memory per board) and the contribution of the CPU error arrival rate to the combined error arrival rate. The voter is assumed to be error free. To compensate for the fact that the measurements are from a larger system, three combined error arrival rates (shown in table 4) are

used in the simulations. Still, one cannot assert that the error rate of the target system is similar to that of the VAX cluster, but since the focus of this study is on relative trends and changes to system reliability and not absolute MTBF figures, this does not pose a problem.

The error latencies used in this study are approximated by normal distributions with the means and standard deviations from the measurement study in [5]. A larger latency with a mean of 36 hours and a standard deviation of 18 hours is also used in the simulations to determine the effect of extremely large latencies.

All errors, including undetected latent errors, residing in a processor or in a global memory are assumed to be corrected when the component undergoes a re-integration, the system is rebooted, or when scrubbing takes place. Thus, not all the errors injected are detected and the actual error arrival distribution of *detected* errors depends on the scrubbing rate, the component re-integration rate, the error latency and the injection rate. Because error latencies and global memory re-integration times are workload dependent, various system workloads are implicitly modeled by varying these parameters. Finally, since we are primarily concerned with transient errors, permanent errors are not injected and the MTBFs presented do not reflect their impact on system reliability.

# 7 Validation of the Simulation Model

An important but often overlooked issue is how accurately simulation models represent the actual system being studied. This section briefly describes the fault injection experiments that were conducted on an actual Tandem machine to validate the basic simulation model developed with DEPEND.

Figure 11 shows the hybrid monitoring environment used to conduct the experiments. A detailed description of the environment can be found in [47, 16], and a description of all the experiments conducted can be found in [16]. The hybrid environment takes advantage of the target system's ability to re-integrate a failed component of a subsystem on-the-fly. The environment is automated and can execute for days repeatedly injecting errors and collecting measurements. A Tektronix DAS 9200 digital logic analyzer is used to monitor the bus activity on the CPU that is injected with errors. A *finite state machine* is used to specify the data that is collected, such as the times when an injection occurs, an exception is raised, and when POST is initiated.

A *DAS control program* running on a Sun workstation accepts *start*, *stop* and *data upload* commands from the *injection program*, translates them, and relays them to the DAS. The *injection program* runs on the Tandem TRM prototype machine and injects errors into the text region (the region containing the machine instructions) of a process. Injecting an error consists of randomly selecting a word, and randomly corrupting one bit of the word residing in the memory of CPUB. If the word resides in the cache, it is deleted to ensure that the corrupted version of the word is used. The *target applications* are Gaussian elimination programs which repeatedly generate 300-by-300 element matrices and execute the Gaussian elimination algorithm to solve the set of simultaneous equations. Two instances of the program were executed simultaneously and injected with errors.

Figure 12 shows the injection program used to collect the time to CPU shutdown distribution

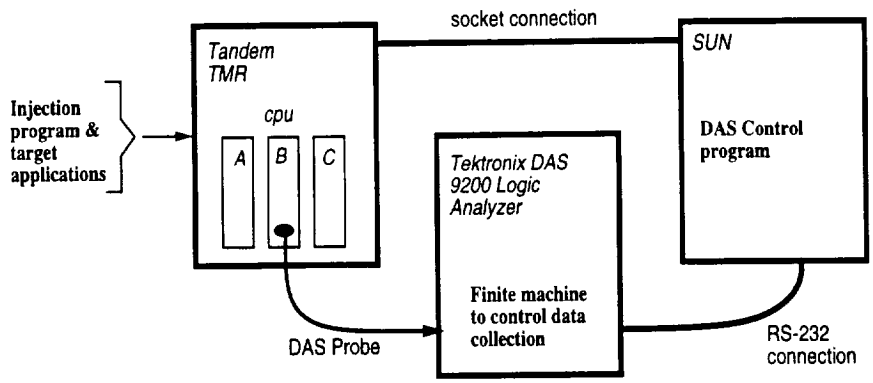


Figure 11: An injection environment using hybrid monitoring.

and the number of undetected, latent errors present in the CPU board prior to a shutdown [2]. These measured data are compared with those generated with the DEPEND simulation model. An accelerated exponential injection rate with a mean of 3 minutes is used to ensure that enough CPU shutdown events are collected for a meaningful comparison with the simulation results. The

1) Start the two Gaussian elimination workload programs.
2) Start the DAS controller and request it to start the DAS.
3) Randomly select
    the program to inject
    the address of the word to be corrupted
    the mask to use.
4) Inform the DAS of the address of the word corrupted.
5) Inject the error into the word (flip a single bit).
6) Determine time of next error, $t$ ($exp(\lambda = 3minutes)$).
7) Wait for CPU shutdown or until $t$ – whichever comes first.
8) If (CPU is shutdown before $t$ elapses)
    re-integrate the CPU
    sleep until $t$ elapses
9) Goto step 3.

Figure 12: Injection program used for the validation experiment.

experiment was conducted over a period of 28 hours, during which 414 errors were injected and the CPU was shutdown 247 times.

The experiment is then repeated with the DEPEND model of the target system. The system simulation was modified to only inject errors into one CPU. The model was executed for a simulated time period of 500,000 seconds (5.78 days). Figure 13 shows the measured and simulated CPU shutdown distributions and their means, medians, standard deviations and the sample counts. The means, medians and the standard deviations are statistically identical. Comparing the distributions, the general shape of both distributions is similar in spite of the fact that the measured distribution has 5 times fewer samples. A closer look at the two distributions reveal that many of the peaks in the measured distribution can also be found in the distribution obtained from simulation. For

[2]This is simply a count of the number of errors injected prior to a CPU shutdown. So if four errors are injected before a shutdown, it is assumed that there are three undetected errors in the CPU prior to a shutdown.

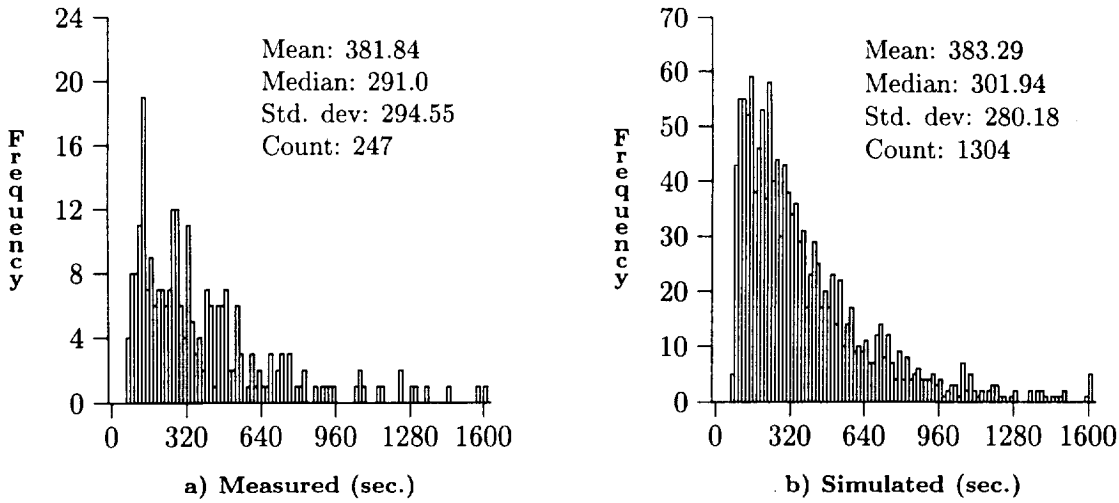


Figure 13: The time to CPU shutdown distribution (in seconds).

example, the simulated distribution captures the peaks which occur between 640 and 960 seconds and between 960 and 1280 seconds. Figure 14 contains the measured and simulated distributions of the number of undetected, latent errors in the CPU at the time of a shutdown. The distribution obtained from the simulation model tracks the one obtained from measurement very well. These

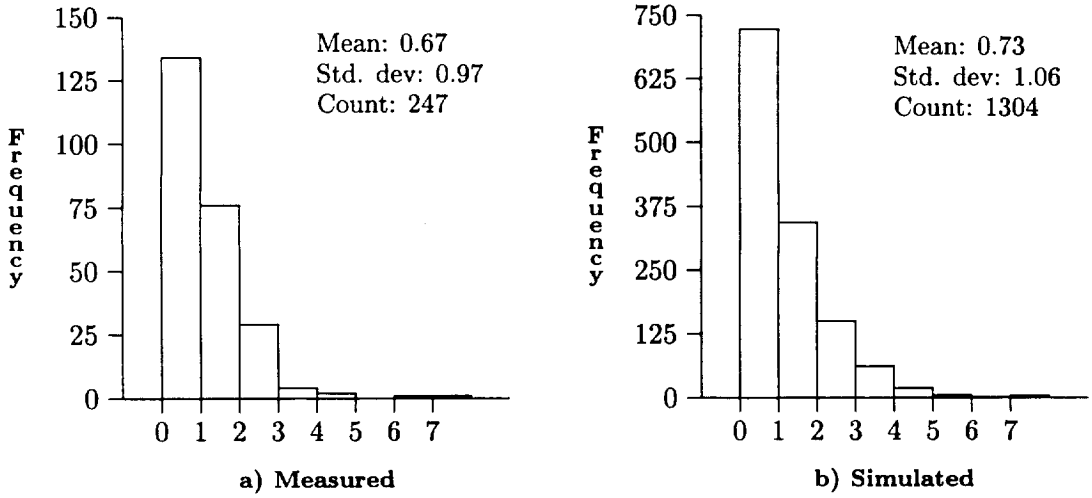


Figure 14: Distribution of the number of latent errors prior to a CPU shutdown.

results demonstrate that the simulation model of the Tandem system is valid and that DEPEND is capable of capturing the intricacies of a real system.

# 8 Simulation-based Fault Injection Study

The DEPEND simulation model is used to analyze the behavior of the TMR-based system under correlated and latent errors and with memory scrubbing and various re-integration times. The main purpose is to illustrate the sorts of studies that can be conducted with DEPEND. The parameters used in the simulation experiments are listed in Table 4. The experiments were conducted in phases to isolate and determine the impact these various parameters have on system reliability. The results shown here do not consider the degradation in recovery coverage due to latent errors. The system was simulated for time periods ranging from 30 years to 2000 years. Each simulation was run twenty times with different random seeds and the averages of these repeated executions are shown here. Except where explicitly stated, the simulations were executed with a POST time of 60 seconds, a

| | |
|---|---|
| Error Arrival Rate $\lambda_1$ | 1/24 hrs |
| Error Arrival Rate $\lambda_2$ | 1/72 hrs |
| Error Arrival Rate $\lambda_3$ | 1/120 hrs |
| Small Latency (SL) | $\mu = 44$ min., $\sigma = 29$ min. |
| Large Latency (LL) | $\mu = 8$ hrs., $\sigma = 4$ hrs. |
| Extra Large Latency (XL) | $\mu = 36$ hrs., $\sigma = 18$ hrs. |
| Percent Correlated Errors | 0, 1, 2 |
| Percent Latent errors | 85 |
| POST Time | 1, 10, 30, 60 sec. |
| CPU Re-integration | 1.5 sec. |
| Global Mem. Re-integration | 0.5, 2, 5, 10 min. |
| System Reboot time | 10 min. |

Table 4: Simulation parameters.

global memory re-integration time of 2 minutes and with memory scrubbing turned off. The mean time between failures (MTBF) is calculated by dividing the simulation period by the average number of system failures. The MTBF figures presented in this paper should not be construed to reflect the MTBF figures of an actual Tandem TMR-based prototype system because the error arrival rate and the error latency, which have a direct bearing on this measure, were *not* obtained from measurements of the Tandem TMR-based prototype but rather from other production machines. For this reason, the results should only be construed to reflect the general trend and behavior of a TMR-based system that is similar to the Tandem system.

## 8.1 Impact of Correlated Errors

TMR-based systems have been shown to be extremely effective against single, independent errors. In this experiment, active, correlated errors are injected to determine their impact on system reliability. The modeling is similar to the 'partial coverage' technique commonly used with analytical models. Correlation factors of 0, 1 and 2 percent were used. Figure 15 shows the results for error arrival rates $\lambda_1$ and $\lambda_2$ for various global memory re-integration times. The figure shows that even a small fraction of correlated errors produces orders of magnitude reduction in the MTBF. However, measurement studies [43, 46] show that correlated failures do not occur simultaneously as modeled here. The next subsection models the occurrence of correlated errors more realistically

and as observed in measurement studies.

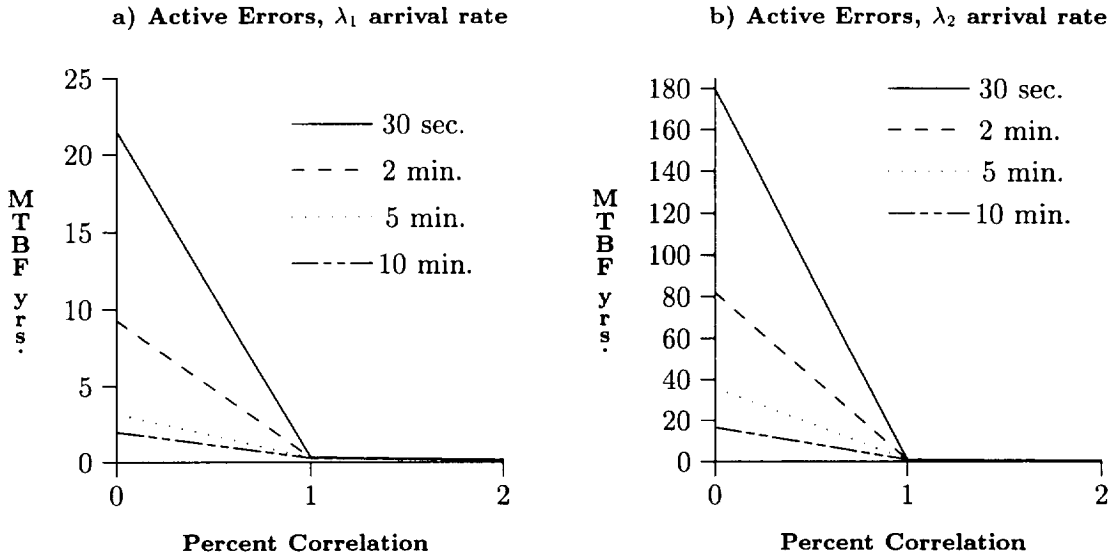


Figure 15: Active errors (no latency) with various global memory re-int. times

The figure also illustrates the degradation in reliability caused by increasing the global memory's re-integration times. A ten fold increase in the re-integration time (from 30 seconds to 5 minutes) reduced the MTBF by a factor of 5 for $\lambda_2$, and a factor of 7 for $\lambda_1$. As the error arrival rate decreases, the impact of the re-integration times will become less of a factor. However, since the re-integration time is dependent upon the size of the memory, the system reliability will decrease as memory size increases; the decrease will be more than linear because the larger memory will also increase the error arrival rate.

The performance overhead for voting was measured during the experiments and was found to be 3.36% with the assumptions that the processors only vote every 2047 cycles and that they all arrive at the voter at the same time. This is the minimum voting overhead because in actual operation, the CPUs are likely to vote more often (i.e., they vote whenever they access global memory or perform I/O) and they will typically never reach the voter at the same time, leaving the early comers to idle waiting for the slowest processor.

## 8.2 Impact of Latent, Correlated Errors

Correlated errors with latency are used to mimic the phenomenon of "staggered machine failures" found to be caused by correlated errors [46]. All errors injected into the global memory are latent; 85% of the errors injected in the CPU boards are latent and the remaining 15% are active errors. Figure 16 graphs the change in the MTBF for error arrival rates of $\lambda_1$ and $\lambda_2$. There are three things worth noting. First, the order of magnitude degradation in system MTBF, due to correlated errors seen in the previous subsection, is not as apparent when a more realistic model is used. Though the reduction in MTBF is significant, a 5-fold decrease for $\lambda_2$ with 1% correlated errors and small latency (SL), it is substantially smaller than the exaggerated 80-fold degradation obtained when error latency is not considered. Second, figure 17 shows the increase in the number of failures for

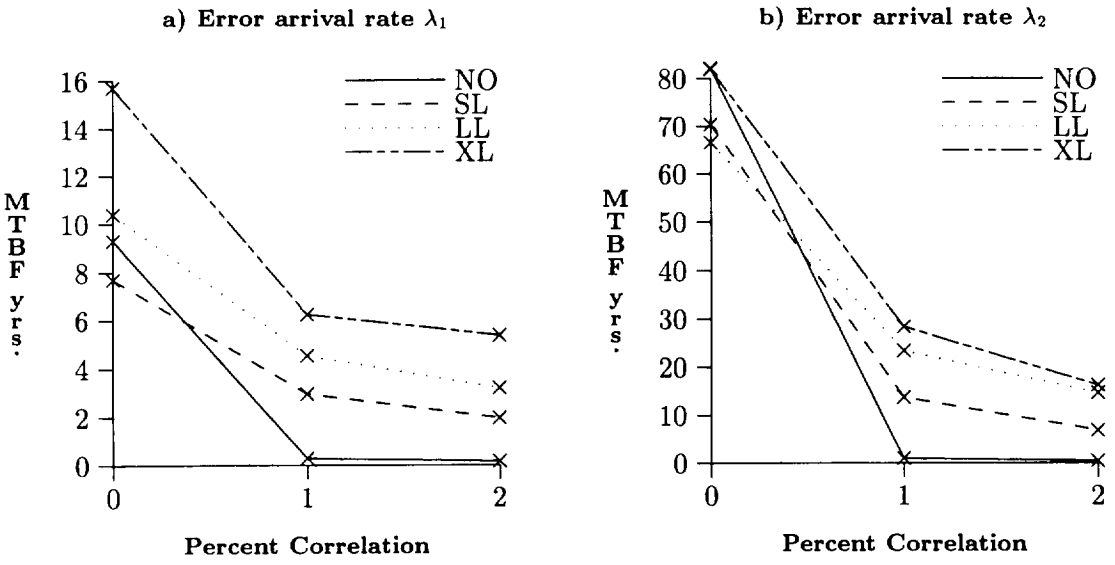


Figure 16: Latent correlated errors, Mem. re-int = 2 min., Scrubbing off.

the different correlation factors. It reveals that as the size of the latency increases, the impact of correlated errors diminishes. This still holds for the processors where 15% of the errors injected are active errors. It should be noted, however, that this finding is based on the assumption that re-integration coverage is 100%. Figure 16 also shows that there is a slight degradation in MTBF when small latent errors are injected versus the case where only active errors are injected.

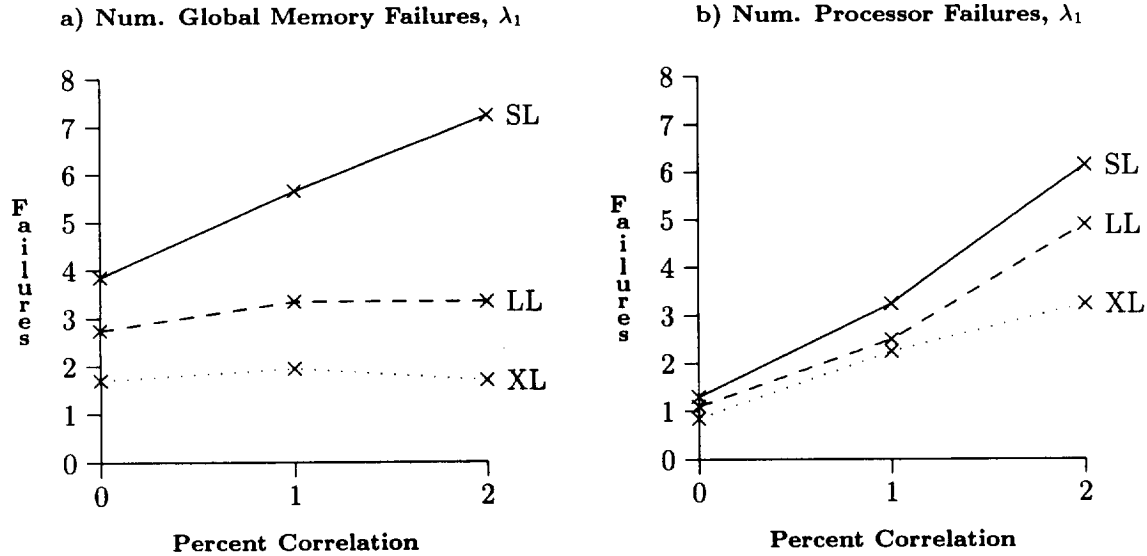


Figure 17: Number of processor subsystem and global memory subsystem failures. No scrubbing.

| Scrub Interval | Latency | | |
|---|---|---|---|
| | SL | LL | XL |
| - | 7.69 | 10.39 | 15.69 |
| 1 | 52.63 | > 2000.0 | > 2000.0 |
| 4 | 11.43 | 666.66 | > 2000.0 |
| 8 | 9.48 | 142.86 | 510.5 |
| 24 | 8.73 | 16.39 | 489.3 |

Table 5: Results with scrubbing activated. POST = 60s. Global Mem. Re-int. = 2 min.

### 8.3 Impact of Memory Scrubbing

In this experiment, memory scrubbing is activated to see how well it protects the system from correlated, latent errors. Four scrubbing intervals, 1 hour, 4 hours, 8 hours and 24 hours, are used. Table 5 shows the impact of scrubbing on the system MTBF for error arrival rate $\lambda_1$. The case where scrubbing is not activated is included for comparison purposes. The table shows that frequent, hourly scrubbing is necessary for small latencies. For SL, hourly scrubbing increases the MTBF by nearly 7 times. When the scrubbing interval is increased to once in 4 hours, their is a corresponding decrease in the MTBF. Scrubbing once every 8 hours provides good results for the larger latencies. Identical trends were seen for error arrival rates $\lambda_1$ and $\lambda_2$ and tend to indicate that the dominant factor that should determine the scrubbing period is the size of the expected latency and not on the error arrival rate. Here again there is a performance/reliability tradeoff because latency decreases with increasing workload and that is precisely when frequent scrubbing is required. Fortunately, the overhead for hourly scrubbing is only 0.34% and is not significant.

Table 6 clearly indicates that any improvement in system reliability provided by scrubbing is diminished if there is correlated errors. For example, with 1% correlated, small latent errors, the MTBF falls from 52.63 years to 8.4 years. Though scrubbing is not effective against correlated errors, the experiments revealed that it still tends to decrease the number of near-coincident faults and increase the availability of the individual components.

| Latency | Percent Correlation | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| SL | 52.63 | 8.40 | 4.29 |
| LL | > 2000.0 | 15.92 | 8.74 |
| XL | > 2000.0 | 19.39 | 9.33 |

Table 6: Impact of correlated, latent errors on scrubbing. POST = 60s. Global Mem. Re-int. = 2 min.

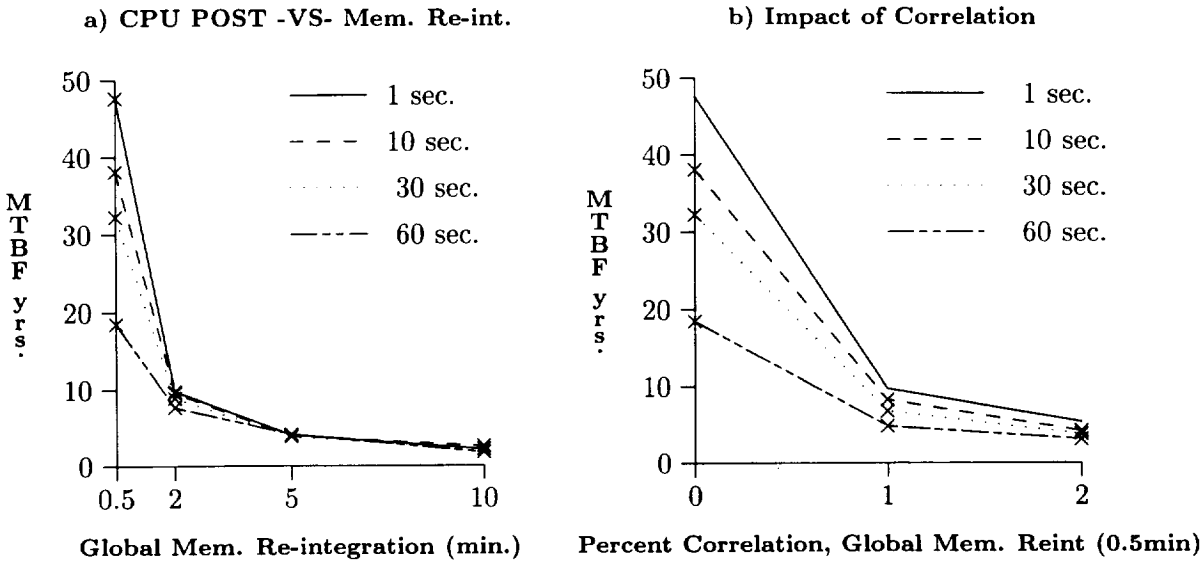


Figure 18: System reliability for various subsystem re-integration times.

## 8.4 Impact of POST Time

The system simulated is designed to tolerate single faults. For such systems, the time needed to repair a faulty component is referred to as its window of vulnerability. If a second fault arrives within this window, the system will fail. The previous experiments show that the MTBF of the system is quite sensitive to the size of this window. The CPU re-integration time consists of 60 seconds to perform a power-on self-test (POST) and 1.5 seconds to re-integrate the CPU. The re-integration time cannot be easily reduced but the POST time can be cut by using different self-checking programs. Since most errors are caused by transient faults, reliability can be improved by performing a perfunctory check that takes a few seconds and immediately initiating a re-integration. If another error is detected in the same board shortly thereafter, a more thorough POST program can be executed to check for permanent defects. The re-integration time for the global memory varies with workload, but it can be reduced by increasing the priority of the re-integration process. In this experiment, simulations are conducted with various POST times and global memory re-integration times to determine which of these repair times have the greatest impact on system MTBF. Specifically, POST times of 1, 10, 30 and 60 seconds and the global memory re-integrations times listed in table 4 are used in the simulations. Memory scrubbing was not activated.

Figure 18a plots the MTBF figures for the various POST times for small latent errors with error arrival rate $\lambda_1$. With a 30 second global memory re-integration time, reducing the POST time from 60 seconds to 1 second improves the MTBF from approximately 20 years to 50 years. However, when the global memory re-integration time is increased to 2 minutes, reducing POST times has no effect on system MTBF. According to table 3, global memory re-integration time for a 32Mbyte system is 2 minutes when the system is working at half capacity. Clearly, given the assumption that 62%[3] of all errors are injected into the global memory, the "reliability bottleneck" is the large

[3]This number was determined based on the size of the global memory and the CPU local memories. See section 6.3.2.

re-integration time of the global memory. Rather than trying to reduce CPU POST times, the designers should focus their efforts on reducing global memory re-integration times.

The results of simulations where correlated errors are injected are shown in figure 18b. For these simulations, the global memory's re-integration time was kept fixed at 30 seconds. Again, in this experiment we see that the presence of correlated errors diminishes any gains achieved by reducing the CPU POST times.

# 9 Conclusion

The objectives of this paper were to: 1) motivate the need for functional simulation tools and in particular the integrated fault injection and simulation environment of DEPEND for system level dependability analysis, 2) discuss the issues and problems of developing such general-purpose tools, 3) describe the DEPEND tool and present techniques developed to simulate real fault scenarios and reduce simulation time explosion, and 4) illustrate a few of DEPEND's capabilities via a realistic evaluation of the Tandem TMR-based prototype fault-tolerant system.

DEPEND is an integrated design and fault injection environment. It is a functional, process-based simulation tool. The system behavior is described by a collection of processes that interact with one another. DEPEND exploits the properties of the object-oriented paradigm to tackle the large component and fault model domain which is an integral part of system level analysis. Using decomposition, composition, parameterization and inheritance, DEPEND provides the skeletal foundation necessary to model an architecture and conduct simulated fault-injection experiments. A library of objects is provided to reduce the development time and effort needed to build simulation models. Acceleration techniques, such as hierarchical simulation and time acceleration, are provided to address simulation time explosion problems.

Examples were used to show why DEPEND is ideally suited for the second stage of the design cycle of a system where specific repair schemes, detailed fault scenarios such as latent errors, and software behavior due to hardware faults need to be evaluated. Through these examples, DEPEND was shown to be able to determine failure mechanisms, quantize the impact of faults, identify important failure conditions, and accurately evaluate fault tolerance mechanisms and repair schemes. For these reasons, DEPEND can also be used to pick out the key features that impact system reliability and help to specify analytical models for further system evaluations.

A few capabilities of DEPEND were illustrated by analyzing a Tandem TMR-based prototype fault-tolerant system. The effect of near-coincident errors caused by correlated and latent faults was analyzed. Issues such as memory scrubbing, re-integration policies and workload dependent repair times, which affect how the system handles near-coincident errors were also evaluated. Simultaneous, correlated errors were injected in the same fashion used as analytical models. Results showed an exaggerated degradation in system MTBF. Error latency was introduced to more accurately model the staggered failure effect of correlated errors observed in real systems. Though the reduction in MTBF was still significant with latent, correlated errors, the degradation was not in the order of magnitude seen with the simpler model. Hourly memory scrubbing was found to be very effective for latent errors but proved ineffective against correlated, latent errors. The

scrubbing rate was found to be more sensitive to the latency period than the error arrival rates. Finally, evaluation of the impact of CPU and global memory re-integration times showed that if the machine is working at half capacity, the global memory re-integration time is the reliability bottleneck. Designers should try to reduce this time rather than CPU re-integration times.

An important but often overlooked issue is how accurately a model represents the system under study. Results obtained using the DEPEND simulation model were validated by comparing them with measurements obtained from fault injection experiments conducted on a prototype Tandem system. A hybrid monitoring testbed was used to conduct the injection experiments. The results of the validation experiments demonstrated that DEPEND is capable of capturing the intricacies of a real system. To our knowledge, no other simulation-based dependability study has been validated in such a fashion.

## 10 Acknowledgments

This work would not have been possible without the help of Doug Jewett, Bob Horst and Carlos Alonso who have furnished many of the details of the Tandem system and have given useful feedback about DEPEND and the Tandem simulation. The authors would like to thank In-hwan Lee, Dong Tang, Axel Hein, Mark Boyd, and Fran Baker for their valuable suggestions regarding this paper.

## References

[1] 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. on Software Engineering*, Vol. 16, No. 2, Feb. 1990, pp. 166-182.

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

[3] J. Carreira, H. Madeira, and J. Gabriel Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," *Proc. 5th Int. Working Conference on Dependable Computing for Critical Applications*, Urbana, IL, Sept. 1995, pp. 135-149.

[4] X. Castillo and D. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," *12th Int. Symp. on Fault-Tolerant Computing*, Santa Monica, June 1982.

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

[6] R. Chillarege, "Understanding Large System Failures—A Fault Injection Experiment," *19th Int. Symp. on Fault-Tolerant Computing*, Chicago, Illinois, June 1989, pp. 356-363.

[7] G. Ciardo, J. Muppala, and K. Trivedi, "SPNP: Stochastic Petri Net Package," *Int. Conf. on Petri Nets and Performance Models,* Kyoto, Japan, Dec. 1989.

[8] J. A. Clark and D. K. Pradhan, "A Simulated Fault-Injection Testbed for Alternative TMR Architectures," *Tech. Report TR-92-CSE-1, U. of Massachusetts*, Jan. 1992.

[9] P. J. Courtois, "Decomposability, Instabilities, and Saturation in Multiprogramming Systems," *Comm. of the ACM,* Vol. 18, No. 7, July 1975, pp. 371-377.

[10] J. B. Dugan and K. S. Trivedi, "Coverage Modeling for Dependability Analysis of Fault-Tolerant Systems," *IEEE Trans. on Computers,* Vol. 38, No. 6, June 1989, pp. 775-787.

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

[12] R. Geist, K. Trivedi, "Reliability Estimation of Fault-Tolerant Systems: Tools and Techniques," *IEEE Computer,* Vol. 23, No. 7, July 1990, pp. 52-61.

[13] K. K. Goswami and R. K. Iyer, "DEPEND: A Design Environment for Prediction and Evaluation of System Dependability," *9th Digital Avionics Systems Conference*, Oct. 15, 1990.

[14] K. K. Goswami, R. K. Iyer, and M. Devarakonda, "Prediction-Based Dynamic Load-Sharing Heuristics," *IEEE Trans. Parallel and Distributed Computing*, Vol. 4, No. 6, June 1993, pp. 638-648.

[15] 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.,* Jan. 1993, San Diego, CA, pp. 197-202.

[16] K. K. Goswami and R. K. Iyer, "DEPEND: A Simulating-Based Environment for System Level Dependability Analysis," *Technical Report CRHC-92-11,* Coordinated Science Laboratory, University of Illinois, June 1992.

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

[18] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment for Distributed Real-Time Systems," *Proc. Int. Computer Performance and Dependability Symp.*, Erlangen, Germany, April 1995, pp. 204-213.

[19] H. Hecht and E. Fiorentino, "Reliability Assessment of Spacecraft Electronics," *Proc. Annual Reliability and Maintainability Symp.,* 1987, pp. 341-346.

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

[21] O. C. Ibe, R. C. Howe, and K. S. Trivedi, "Approximate Availability Analysis of VAXcluster Systems," *IEEE Trans. on Reliability*, Vol. 38, No. 1, April 1989, pp. 146-152.

[22] IEEE Standard VHDL Language Reference Manual-Std 1076-1987, IEEE Press, 1988.

[23] R. K. Iyer, S. E. Butner, and E. J. McCluskey, "A Statistical Failure/Load Relationship: Results of a Multicomputer Study," *IEEE Trans. on Computers*, Vol. SE-8, July 1982, pp. 354-370.

[24] R. K. Iyer and D. Tang, "Experimental Analysis of Computer System Dependability," *Technical Report CRHC-93-15,* Coordinated Science Laboratory, University of Illinois, June 1993.

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

[26] A. M. Johnson and M. A. Schoenfelder, "Rainbow Net Analysis of VAXcluster System Availability," *IEEE Trans. on Reliability*, July 1991.

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

[28] W. Kao and R. K. Iyer, "DEFINE: A Distributed Fault Injection and Monitoring Environment," *Fault-Tolerant Parallel and Distributed Systems* (D. K. Pradhan and D. R. Avresky, Eds.), IEEE CS Press, Los Alamitos, CA, 1995, pp. 252-259.

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

[30] E. E. Lewis, F. Boehm, C. Kirsch, B. P. Kelkhoff, "Monte Carlo Simulation of Complex System Mission Reliability," *Proc. Winter Simulation Conf.*, pp. 497-504, 1989.

[31] M. H. MacDougall and J.S. McAlpine, "Computer Simulation with ASPOL," *Symp. on the Simulation of Comp. Sys.*, ACM/SIGSIM, pp. 93-103, 1973.

[32] B. Melamed and R.J.T. Morris, "Visual Simulation: The Performance Analysis Workstation," *IEEE Computer*, vol. 18, no. 8, pp. 87-94, Aug. 1985.

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

[34] J. F. Meyer and L. Wei, "Influence of workload on error recovery in random access memories," *IEEE Trans. on Computers*, vol. C-37, no. 4, pp. 500-507, April 1988.

[35] Victor F. Nicola, Marvin K. Nakayama, Philip Heidelberger, and Ambuj Goyal, "Fast Simulation of Dependability Models with General Failure, Repair and Maintenance Processes," *Proc. of the 20st Inter. Symp. on Fault-Tolerant Computing*, England, June 1990.

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

[37] W. H. Sanders, W. D. Obal II, M. A. Qureshi, and F. K. Widjanarko, "The UltraSAN Modeling Environment," *Performance Evaluation*, Vol. 24, No. 1, October-November 1995, pp. 89-115.

[38] C.H. Sauer, E.A. MacNair, and J.F. Kurose, "RESQ: CMS User's Guide," *IBM Research Report RA-139*, Yorktown Heights, N.Y., April 1982.

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

[40] 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*, Tokyo, Japan, June 1988, pp. 102-107.

[41] SES, Inc., "SES/Sim Simulation Language Reference Manual, " Austin, TX, March 1989.

[42] D. Tang, R. K. Iyer, S. S. Subramani, "Failure Analysis and Modeling of a VAXcluster System," *Proc. 20th Int. Symp. Fault-Tolerant Computing*, England, June 1990.

[43] D. Tang and R. K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *IEEE Trans. Computers,* Vol. 42, No. 1, Jan. 1993.

[44] K. S. Trivedi and R. M. Geist, "Decomposition in Reliability Analysis of Fault-Tolerant Systems," *IEEE Trans. on Reliability,* Vol. R-32, No. 5, Dec. 1983, pp. 463-468.

[45] T. K. Tsai, R. K. Iyer, and D. Jewett, "An Approach towards Benchmarking of Fault-Tolerant Commercial Systems," *Proc. 26th Int. Symp. Fault-Tolerant Computing*, Sendai, Japan, June 1996, pp. 314-323.

[46] A. S. Wein and A. Sathaye, "Validating Complex Computer System Availability Models," *IEEE Trans. Reliability,* Vol. 39, No. 4, Oct. 1990, pp. 468-479.

[47] Luke Young, R. K. Iyer, K. K. Goswami and C. Alonso, "A Hybrid Monitor Assisted Fault Injection Environment," *Third IFIP Conf. on Dependable Computing for Critical Applications*, Sicily, Italy, Sept. 1992.