Simulation of Software Behavior Under Hardware Faults

Kumar K. Goswami & Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
University of Illinois
1308 W. Main St.
Urbana, Illinois 61801

Abstract

In this paper, we introduce a simulation-based software model that permits application specific dependability analysis in the early design stages. The model represents an application program by decomposing it into a graph model 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 software model simulates the execution of the program while errors are injected into the program's memory space. The model provides application dependent parameters such as detection and propagation times and permits evaluation of function and system level error detection and recovery schemes. The paper illustrates, via a case study, the interaction between an application program and two detection schemes. Specifically, Gaussian elimination programs running on a Tandem Integrity S2 system with memory scrubbing are studied. Results obtained from the simulation-based software model are validated with data measured from an actual Tandem Integrity S2 system. Application dependent coverage values obtained with the model are compared with those obtained via traditional schemes that assume uniform or ramp memory access patterns. For our program, some coverage values obtained with the traditional approaches were found to be 100% larger than those obtained with the software model.

1 Introduction

The impact of software on system dependability (e.g. reliability, availability) is a primary concern because software is a major component of a system. One aspect is software reliability, which is concerned with design errors in the software. Another aspect, and the focus of this paper, is software behavior under hardware faults. In the early design stages of a system, a designer has a general idea of the structure and the underlying algorithm of the application software, or has access to the software that will be ported to the new system. An effective design strategy at this early stage is to evaluate the system while executing an abstract representation of the application software. This can provide key application dependent parameters that impact system dependability. It permits meaningful application specific evaluation and trade-off analysis of function and system level detection and recovery mechanisms.

In this paper, we introduce a simulation-based software model that facilitates application specific dependability analysis in the early design stages. The model represents an application program by decomposing it into a *graph model* 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 software model simulates the execution of the program while errors are injected into the program's memory space. An error is discovered when the corrupted memory location is accessed. A high-level description language is used to specify a graph model abstraction of a program. A multiprogramming environment allows many programs to be executed simultaneously, permitting evaluation of their combined impact on dependability parameters. An advantage of using a high-level abstraction is that it reduces simulation time explosion permiting hundreds of error injections within seconds. Also, large observation periods are possible allowing measurement of large detection times that are typical of real programs.

The paper illustrates one use of this model with a case study. The model is used to obtain error detection latency times of Gaussian elimination programs running on a Tandem Integrity S2 system and to evaluate the coverage of two memory scrubbing schemes. Error detection latency times obtained with the model are validated with measurements from an actual Integrity S2 system. Application dependent coverage values obtained with the model are compared with those obtained via traditional schemes that assume uniform or ramp memory access patterns. For our program, coverages obtained with the traditional approach were found to be 100% larger than those obtained with the software model.

This research was supported in part by the NASA grant NAG 1-613, in cooperation with the Illinois Computer Laboratory for Aerospace Systems and Software (ICLASS), by NASA Graduate Student Researchers Fellowship, the Alexander Von Humboldt Foundation, and the Computer Sciences Corporation under contract GSA CSC 468969.

2 Related Research

Many studies have evaluated the behavior of software under hardware faults [1, 2, 4, 8, 12, 14, 17, 19]. Actual programs are executed on simulated or real processors in which errors are injected. These studies require an actual system or a detailed model of the hardware architecture. Tools like FIAT and FERRARI verify and study software systems by injecting software-implemented faults into actual programs. DEPEND [5] and REACT [3] provide a simulation-based fault injection environment to analyze the dependability of hardware architectures. As such, there are no mechanisms to study, at the early design phase, software behavior under hardware faults. The software model presented runs within the DEPEND environment and extends DEPEND's ability to study software issues.

Laprie [9] uses a Markov model to evaluate the reliability of software systems during their operational life. The focus of the work is on software design errors. In [18], a stochastic model of error propagation is developed with a digraph that represents the interconnections between the hardware and software models. The propagation times are based on random variables with general distributions. In [15], a control flow graph automatically generated from syntactic analysis of a program, is used to estimate execution times.

The contribution of this work is a method to:

1. Create a high-level abstract representation of application software and provide an environment that can execute one or more simultaneously to provide application dependent parameters.

2. Evaluate function and system level detection and recovery schemes within the context of application programs.

3. Provide feedback early in the design phase.

3 The Software Model

3.1 Overview

The software model environment is shown in Figure 1. The model consists of programs that are represented by probabilistic control flow graphs, G, and a virtual memory system, M, within which the programs execute. Contention for resources is modeled using the first-come-first-served or round-robin scheduling discipline. Errors are injected into the physical memory space.

The distinguishing feature of the software model is the virtual memory system M and the link between G and M. The virtual memory provides two functions. First, it permits simultaneous execution of several programs allowing evaluation of their combined impact. Second, it

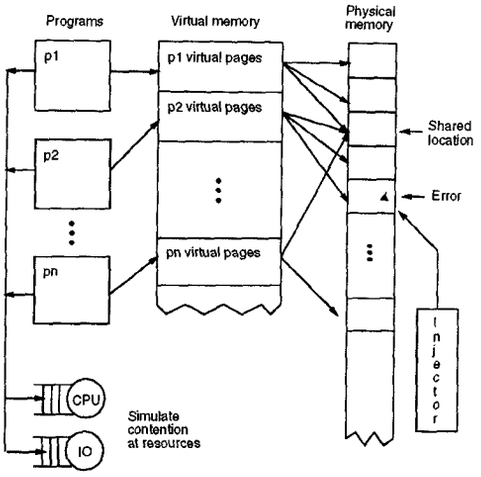

Figure 1: The complete software model execution environment.

allows these programs to share pages in memory. Inter-program communication, and hence propagation of errors among programs, is modeled with shared pages. In fact, the shared memory primitive can be used to model various communication paradigms. In [10], the authors give a convincing argument for the claim that "variable sharing" is a close reflection of computer hardware and it is a primitive upon which all other realizable distributed models can be described.

The error occurrence process is modeled by injecting errors into the physical memory and allowing the simulated execution of G, to discover the error. Manifestation of transient hardware faults are modeled as memory errors. A similar approach was taken in [12], in which the authors model all control flow errors as memory errors. The model is realistic for faults in the memory subsystem, however, it can also be applied to several types of faults in the processor. For instance, a fault in the ALU can propagate, eventually manifesting as incorrect data written to memory. Similarly, errors in the bus can cause incorrect data to be written to memory or written to the wrong location. Errors in memory mapped registers and I/O devices can also be represented by memory errors.

The user provides a control flow graph of an algorithm, the location of the memory accessed by each node and the total virtual memory used by the graph. The memory access pattern is derived from the graph. The model is hierarchical allowing designers to refine their abstract program models as more information is available. The software model simulates the execution of the graph on the underlying hardware. Outputs of the model include detection latency times, error propagation times, the probability of error propagation, and detection coverages. The software

model is intended for use in the early design stages when an actual system does not exist and details of the applications are not known. It is not meant to replace fault injection studies of existing systems.

3.2 Detailed Description

The software model SW is defined as

$$SW = (M, (G, S, Ex)^+, I)^1$$

where:

- M is the virtual memory system.
- G is a *probabilistic control flow graph* (PCFG).
- S is the subspace in M, allocated to G.
- Ex is the execution environment given by:

$$Ex = \begin{cases} Ex_{no\_error} & \text{Simulates execution of } G \text{ in a multiprogramming environment.} \\ Ex_{text\_error} & \text{Executes } G \text{ and simulates detection of errors in the text space of } G. \\ Ex_{text\_data} & \text{Executes } G \text{ and simulates detection of errors in the data space of } G. \\ Ex_{propagate} & \text{Executes } G \text{ and simulates detection and propagation of errors.} \end{cases}$$

- I is an injector which injects errors into M.

3.2.1 The Memory System

A paged virtual memory system is simulated and is defined as $M = (V_m, P_m)$, where, V_m is the virtual memory and P_m is the physical memory. A virtual memory system allows more than one G to execute simultaneously and share pages. The functions that operate on M are:

- **access_memory(PID,** v_{beg}, v_{end}) – Maps the virtual block $< v_{beg}, v_{end} >$ of a PCFG indicated by **PID** to a physical block $< p_{beg}, p_{end} >$ in P_m. The address of the first error encountered in the range $< p_{beg}, p_{end} >$ is returned. If there are no errors, -1 is returned.
- **inject_error(PID,** v_a) – The virtual address v_a is mapped to a physical address, p_a, in P_m. A bit is flipped and location p_a is marked as corrupted.
- **inject_error(**p_a) – A bit in the word addressed by p_a is flipped and p_a is marked as corrupted.
- **erase_error(PID,** v_a) – The virtual address v_a is mapped to a physical address, p_a, in P_m. The word is corrected and p_a is marked as uncorrupted.
- **erase_error(**p_a) – The word addressed by p_a is corrected and marked as uncorrupted.

[1] The unary operator '+' means "one or more instances of."

Function **access_memory** is used by Ex to detect errors in the memory accessed by a PCFG. Functions **inject_error** and **erase_error** are used by the injector to introduce and correct errors in the physical memory, P_m.

3.2.2 The Probabilistic Control Flow Graph

G is a probabilistic control flow graph, $G = (N, E)$, where N is the set of nodes and E is the set of edges.

Definition 3.1: The node set $N = \{n_1, n_2, ..., n_k\}$ where n_i is an abstract definition of a program segment consisting of:

- Required:

name	name or identification of the node.
text space	location of node's text in virtual memory, denoted by $< \mathbf{tx}_{beg}, \mathbf{tx}_{end} >$, the address of the first and last words.
exec time	number of cycles to execute the node.
ρ_t	probability of detecting an error in the text space on the first encounter.
function	function performed by the block (**Read, Write, Read/Write, IO, Err_correct, Err_detect, Checkpoint**)

- Optional:

data space	location of data blocks processed by the node denoted by $(< \mathbf{dt}_{beg_1}, \mathbf{dt}_{end1} >, \ldots, < \mathbf{dt}_{beg_k}, \mathbf{dt}_{endk} >)$ where $< \mathbf{dt}_{beg_i}, \mathbf{dt}_{endi} >$ is the address of the first and last words of data block i. Data can be located in shared memory space.
ρ_d	probability of detecting an error in data space.
ρ_{prop}	probability of propagating an error in data space.

The optional parameters are needed only if the Ex_{text_data} or the $Ex_{propagate}$ execution environment is used.

Definition 3.2: The edge set $E = \{e_1, e_2, \ldots, e_k\}$ where e_i is a directed edge defined as $e_i = ((n_i, n_j), \alpha_i, m_i)$:

- (n_i, n_j) is an ordered pair of any two nodes in N with n_i as the tail and n_j as the head of the directed edge,
- α_i is the probability of traversing the edge,
- m_i is a count of the number of times the edge is to be traversed.

Definition 3.3: A simple edge e_i is defined such that n_j is not an ancestor of n_i, $n_j \neq n_i$, $\alpha_i > 0$, and $m_i = 0$.

Definition 3.4: A loopback edge e_i is defined such that n_j is an ancestor of n_i or $n_j = n_i$, $\alpha_i = 0$, and $m_i > 0$.

Definition 3.5: The set $\xi_p^* \subseteq E$ is the set of directed edges from node n_p such that

$$\sum_{i=1} \alpha_i = 1, \ \forall_i : e_i \in \xi_p^*$$

3.2.3 The memory subspace

S is the subspace of the total text, data and shared space in virtual memory allocated to G and is given by:

$$S = (<S_b(text), S_e(text)>, <S_b(data), S_e(data)>, \\ <S_b(shared), S_e(shared)>)$$

where,

- $S_b(x)$ = address of the first word of space x in virtual memory.
- $S_e(x)$ = address of the last word of space x in virtual memory.

3.2.4 The execution environment

A discrete-event simulator [16, 11, 5] is used to execute G in a simulated multiprogramming environment. Ex can be one of the following paradigms.

Execution without error detection: Ex_{no_error} models the probabilistic execution of G and is the basis upon which other execution paradigms are built.

Input: cycle_time, IOserver_ID, CPUserver_ID
Function:

```
Assign PID to PCFG
Allocate text, data, and shared
memory    space in M
Traverse the PCFG
   I = top most node in PCFG
   while (I <> terminal node)
      texterror_addr = memory_access(PID,
                          text space_I)
      if (data space_I specified)
         dataerror_addr = memory_access(PID,
                             data space_I)
S1:   server_time = exec time_I × cycle_time
      if (function_I = IO)
         reserve(IO_server_ID)
         hold(server_time)
         release(IO_server_ID)
      else
         reserve(CPU_server_ID)
         hold(server_time)
         release(CPU_server_ID)
S2:   I = get_next_node()
   end while
```

where,

`reserve()` & `release()`	simulate the queueing activity at a server.
`hold()`	increments the system clock.

```
get_next_node()
  if (ξ_I^* = {e_l, e_s}, where e_l is a loopback edge and e_s is a
     simple edge) then
      if (m_l > 0) then
          decrement m_l
          node_addr = head(e_l)
      else
          node_addr = head(e_s)
  else if (ξ_I^* = {e_1, e_2, ..., e_j}, and ξ_I^* contains no loopback
          edges) then
      select e_i ∈ ξ_I^* with probability α_i
      node_addr = head(e_i)
  return(node_addr)
```

Execution with detection of text errors: Ex_{text_error} is an extension of Ex_{no_error} which models the probabilistic detection of errors present in the text space. It requires three additional inputs:

Additional input: detect_function, record_error, ν

where,

`detect_function`	models the probabilistic detection process (see below).
`record_error`	records pertinent statistics, such as the time of detection. Execution can halt or continue after detection as specified by the user (default: continue).
ν	parameter used by the detection function, $0 \leq \nu \leq 1$.

Statements **S1** and **S2** of Ex_{no_error} are replaced with the following:

```
S1:   error_detected = detect_function(
                          texterror_addr, ν)
      if (error_detected)
         server_time = (texterror_addr − tx_beg_I) /
                       (tx_end_I − tx_beg_I + 1)
                       × exec time_I × cycle_time
      else
         server_time = exec time_I × cycle_time

S2:   if(error_detected)
         record_error()
         use server for the remaining time
      I = get_next_node()
```

In S1 above, the time to detection (server_time) is computed using a linear relationship based on the location of the error within the text space. The detection function returns a **FALSE** if the `text_error_addr` is -1. Otherwise it models error detection as follows:

```
detect_function:
    increment err_pass
    γ = U[0,1) -- a Uniform distribution
    if(ρ_t · ν^(err_pass-1) < γ AND err_pass < LIMIT)
        return TRUE
    else
        return FALSE
```

where,

`err_pass`	is the number of times the specific error is encountered.
`LIMIT`	is the maximum number of encounters within which the error must be detected.

The detection function decreases the probability of detection exponentially for $\nu < 1$, as the number of encounters increases. In section 4, we show that this accurately captures the behavior of the error detection process for the program modeled. The total probability of detecting an error, D, within K encounters is:

$$D = \sum_{\kappa=1}^{K} D_\kappa \tag{1}$$

where,

$$D_\kappa = \begin{cases} \rho_t \nu^{\kappa-1}(1 - D_{\kappa-1}), & \kappa > 1 \\ \rho_t, & \kappa = 1 \end{cases}$$

Due to lack of space, not all aspects of the model have been discussed. For instance, the remaining execution paradigms and node functions **Err_detect** and **Err_correct** are not presented.

3.2.5 The Injector

The injector injects errors into the physical memory, P_m, using the functions described in Section 3.2.1. The injector can be specified to inject one or many errors. If many errors are to be injected, the user specifies one of the three error arrival distributions: constant, exponential or Weibull. The range of locations to be injected must also be specified. The words injected are selected randomly from this range.

3.2.6 Specification Language

A simple program and the language used to specify a PCFG of the program are illustrated in Figure 2. The "`memory{}`" construct specifies the virtual space required by the text, data and shared segments. The "`bind{}`" construct binds constants to identifiers. The structure of the program is defined with the "`node{}`" construct. A node can be a simple node containing items listed in definition 3.1. It can also be a "meta" node consisting of one or more simple and meta nodes and one "`flow{}`" construct that defines the control flow within that meta node. In the figure, node `init`'s function is **Write**. Its **exec time** is 19×1000 cycles, its **text space** is $< 1001, 1018 >$ and it has one data block $< 0, 1000 >$.

The level of abstraction used will depend on the nature of the experiment and the information available about the program. As more information is known, it can be added by converting simple nodes to meta nodes. In this example, each node represents a set of program statements [2]. Some nodes are more "abstract" than others. For instance, node `init` represents the entire "for loop" used to initialize the array, whereas the loops in `sort()` are modeled in more detail. For this level of abstraction, a node's **text space** can be determined based on the number of instructions it represents. For a RISC processor, where the instructions are of fixed length, the size can be estimated based on the statement to instruction ratio. The number of cycles per node is 1 to 1.4 times the number of instructions. Once the PCFG is specified, it is compiled to create a binary file that is executed by Ex.

4 Model Application

One application of the model is illustrated by using the model to obtain detection latency times of a program and to evaluate two memory scrubbing schemes. The purpose of the experiment is twofold: 1) show that the model can provide application dependent parameters such as latency times, and 2) show that the coverage of the scrubbers are application dependent and that coverage values can be highly inaccurate if they are determined using simple estimations of application behavior. The latency times obtained with the model are validated with those obtained from an actual system.

4.1 Hardware Architecture

The underlying hardware platform is based on the Tandem Integrity S2 [7] fault-tolerant system. The Integrity S2 is a Unix-based TMR system. Each of the three processor boards consists of a RISC R3000 processor and 8Mb of local memory. The processors work asynchronously and periodically synchronize at a voter located off the board. If errors are detected by the voter or by the inherent detection mechanisms of the R3000 (e.g. illegal instruction), the processor board in question is shutdown. The others continue to operate while the failed board performs a power-on-self-test (POST) that takes 60 seconds. If it passes the test, the board is re-integrated back into the system. Re-integration takes 1.5 seconds, and all applications are suspended during this time. The local memory has no error correcting circuitry and relies on memory scrubbing to detect and correct all errors in memory. A scrubber cycles through all three

[2] Typically, the level of abstraction is much higher than shown here, but this serves well as a simple illustration.

```
main()
    for i = 1 to 1000
        list[i] = 1000 - i
    end for

    input(k)

    if (k = ``sort'')
        call sort()
    else
        call print()
end main

sort()
    for i = 1 to 1000
        for j = i to 1000
            find min entry
        end for
        exchange list[min] with list[i]
    end for
end sort

print()
    for i = 1 to 1000
        output(list[i])
    end for
end print
```

a. Original program

```
bind { pt 0.8; pd 0.5; sz 1000; sz_half 500 }
memory{ data 0 1000; text 1001 1500 }
node main {
   node init    WRITE 19 * sz pt 1001 1018 data pd 0 1000
   node input   READ  10      pt 1019 1029
   node sort
   node print
   flow{
                init input
                input sort BRANCH 0.5
                input print BRANCH 0.5
   }
}
sort{
   node fori    READ  4       pt 1030 1034
   node forj    READ  4       pt 1035 1038
   node min     READ  10      pt 1039 1048 data pd 0 1000
   node endj    READ  2       pt 1048 1050
   node xchng   WRITE 12      pt 1051 1062 data pd 0 1000
   node endi    READ  2       pt 1063 1064
   flow{
                fori forj
                forj min
                min endj
                endj min LOOP sz_half xchng
                xchng forj LOOP sz endi
   }
}
print{ ... }
```

b. PCFG definition of program

Figure 2: Example program and corresponding PCFG.

memory boards, comparing and correcting 32-byte blocks at a time. The time to make one pass through the memory is adjustable, but is typically set to 1 or 2 hours. Errors detected by the scrubber are corrected without shutting down the processor board.

4.2 Application Programs

A Gaussian elimination program with partial pivoting and back substitution is used as the application program. This program was chosen because it is representative of a large class of numerical and statistical applications. The Gaussian elimination program executes indefinitely. Each iteration takes 35.6 seconds and consists of filling a 300-by-300 matrix with randomly generated numbers, solving the matrix and printing the result. The program contains 280 lines of C code and was modeled with a 23 node PCFG. The PCFG is executed with the Ex_{text_error} paradigm.

4.3 Experiment Setup

The experiment is as follows:

1. Start the PCFG(s).
2. Randomly select a PCFG and a word within the PCFG's virtual memory space to be injected.
3. Inject transient error – flip a bit of the selected word.
4. Execute the PCFG(s) until the error is detected or LIMIT has expired.
5. Record the detection time and goto step 2.

There are two possible outcomes of an injection: 1) a PCFG detects the error, or 2) the scrubber detects the error within an hour. An 8MB physical memory is used and one to sixteen instances of the PCFG are executed simultaneously. A procedure to simulate the scrubber was added to sweep through P_m every hour. Several terms used throughout this section are now defined.

Definition 5.1: An *active error*, e_a, is an error that is detected by the program (i.e. detected by the process of program execution) assuming there is no scrubber.

Definition 5.2: The *program detection latency* is the time from injection to the time the error is detected by the program.

Definition 5.3: The *scrubber detection latency* is the time from injection to the time the error is detected and corrected by the scrubber.

A Tandem Integrity S2, instrumented with a fault injection and a monitoring device [19], was used to verify the detection times obtained from the software model. The original C version of the Gaussian elimination programs were executed while injecting errors into their text space.

4.4 Program Detection Latency

One thousand errors were injected into an executing PCFG to obtain program detection latencies. Three different detection models, obtained by changing the parameters of the detection function (see Section 3.2.4), were used and are listed in Table 1. Parameter ρ_t was selected so that D

Name	ρ_t	ν	LIMIT
SIM1	0.67	1.0	1
SIM2	0.425	1.0	2
EXP2	0.589	1/3	2

Table 1: Parameters for the three detection functions.

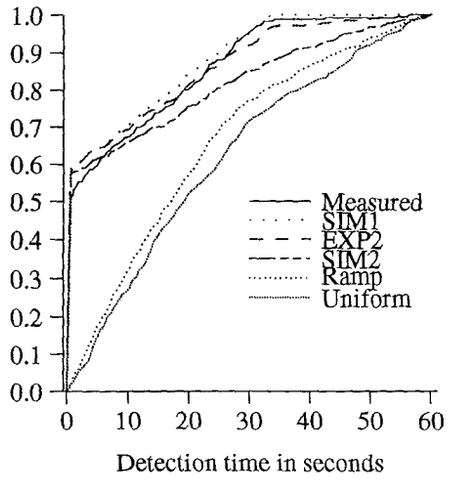

Figure 3: Cumulative detection latency distribution functions.

(Equation 1) is 0.67 [3], and it is assumed to be the same for all nodes in the PCFG. SIM1 detects errors on the first encounter only. SIM2 detects errors on second encounters but the probability of detection is the same for each encounter because ν is one. EXP2 reduces ρ_t exponentialy with each encounter. Figure 3 shows the cumulative distribution functions (CDF) of the measured and simulated detection latency. The figure also contains the CDFs obtained from using two distributions that estimate the spatial distribution of the memory accesses [13]. One is the uniform distribution where all addresses are accessed with equal probability. The other is a ramp distribution where the probability of accessing an address increases as the order of the address increases. Table 2 lists the mean, standard deviation and the sum of the square of the residuals, R,

$$R = \sum_i (y_i - \hat{y}_i)^2$$

where, y_i is the ith entry from the measured CDF
$\hat{y}_i$ is the ith entry from the simulated CDF

for all methods. The graph shows that the uniform and ramp functions, which do not model the control flow of the program, fail to capture the spatial locality exhibited by the program. The three functions which model the program's control flow captures the program detection latency distribution well.

[3] D = 0.67 was determined from measurements, however, later it is shown that D (which is a function of ρ_t) is not the dominant factor in determining latency times.

Name	Mean	Std.	R	Normalized R w.r.t. EXP2
SIM1	6.9	10.2	0.066	3.00
SIM2	11.7	17.36	0.345	15.68
EXP2	9.1	12.74	0.022	1.0
MEASURED	8.8	12.55	-	-
Uniform	37.64	34.68	7.58	344.55
Ramp	37.13	39.73	5.68	258.18

Table 2: Statistics of the measured and simulated program detection latency.

The histogram of the measured detection latency (Figure 4b) shows that a number of errors are detected after 35.6 seconds. Since one iteration of the program takes 35.6 seconds, these errors are detected on the second encounter (no errors were detected on third or more encounters). This phenomenon is due partly to data dependencies. For instance, if the instruction **bgt r5, 0x4000** is corrupted and becomes **bgt r7, 0x4000**, the error may not be detected if the contents of register r7 is greater than zero. On a second encounter, the value of r7 may be less than zero, thus potentially causing an error.

For SIM1, the detection times are limited to the execution time of the program (LIMIT = 1). SIM2 overestimates the number of errors detected on the second encounter. Only EXP2 accurately models the observed behavior by reducing the probability of detection (ρ_t) exponentially with the number of encounters. Comparison of Figure 4a and 4b , shows that EXP2 captures the tail of the latency distribution reasonably well. This result indicates that Equation 1 is an accurate depiction of the error detection process for this application.

How sensitive are the results to ρ_t? If they are highly sensitive, this approach is not applicable when an exact value of ρ_t is not known – for example, when the machine is still being designed. Table 3 contains the mean and standard deviation of the program's detection latency for varying values of ρ_t. The exponential detection model with a large LIMIT set to 5 was used to obtain the worst case results. The table indicates that there is a linear relationship between ρ_t and the mean detection latency. Clearly, the dominant factor determining the latency is the program's control structure and not ρ_t.

4.5 Coverage of the Memory Scrubber

This section defines and computes the *active coverage* of the scrubber. The scrubber coverage, SC, is:

$$SC = Pr[e_a]Pr[S_l < P_l] + (1 - Pr[e_a])$$

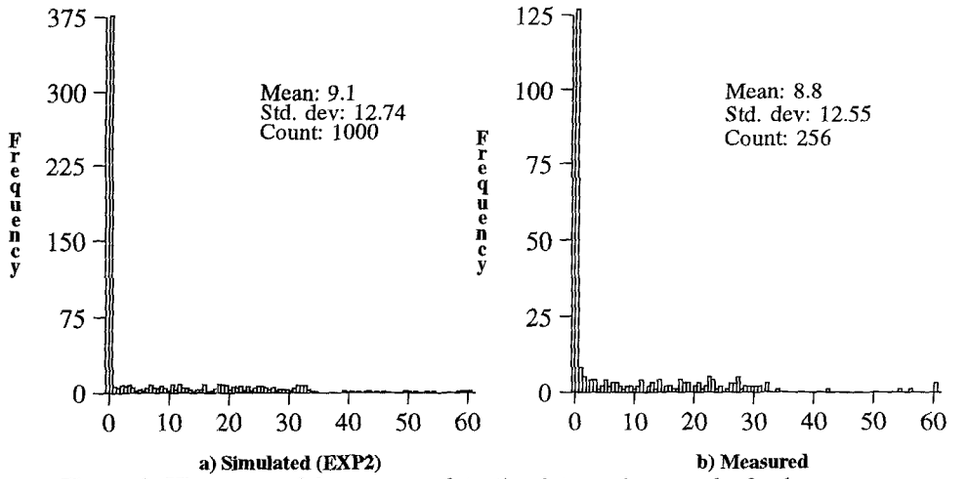

Figure 4: Histogram of the program detection latency in seconds, for 1 program.

ρ_t	Mean	Std. Dev.
0.2	12.42	21.7
0.4	11.40	19.6
0.6	10.02	16.79
0.8	8.83	14.14
1.0	7.41	10.4

Table 3: Sensitivity of the mean detection latency to varying values of ρ_t (LIMIT = 5).

The scrubber's *active coverage* is $Pr[S_l < P_l]$, and is the probability that the scrubber will detect an active error before the program. S_l is a random variable denoting the detection latency of the scrubber. P_l is a random variable denoting the detection latency of the program.

Emphasis is placed on the active coverage because active errors not caught by the scrubber cause a processor board to be shutdown followed by a lengthy POST and re-integration period. This time is a "window of vulnerability" within which a second error will cause the entire system to fail. Increasing the active coverage of the scrubber will reduce the probability of processor board failures and increase the reliability and availability of the system.

The latency distribution of the scrubber, $f_{S_l}(s)$, can be shown to be uniformly distributed, $f_{S_l}(s) = 1/T, 0 \leq s \leq T$, where T is the time required to complete one sweep through the memory. If $f_{P_l}(p)$ is the distribution of the program's detection latency, then the joint distribution of the two detection latencies is $f_{SP}(s,p) = f_{P_l}(p)/T$, by stochastic independence and:

$$\begin{aligned} Pr[S_l < P_l] &= \int_{p=0}^{T}\int_{s=0}^{p} f_{SP}(s,p)\,dsdp + \\ &\quad \int_{p=T}^{\infty}\int_{s=0}^{T} f_{SP}(s,p)\,dsdp \\ &= 1/T \int_{p=0}^{T} p \cdot f_{P_l}(p)\,dp + (1 - F_{P_l}(T)) \end{aligned} \quad (2)$$

No. Programs	$Pr[S_l < P_l]$	
	Simulation	$E[P_l]/T$
1	0.0025	0.00246
2	0.005	0.0048
4	0.009	0.0087
8	0.016	0.0165
16	0.032	0.0325

Table 4: The *active coverage* of the scrubber.

For T much larger than the mean program detection latency time, the approximate active coverage is:

$$Pr[S_l < P_l] \cong E[P_l]/T \quad (3)$$

One hundred thousand errors were injected to determine the active coverage of the scrubber for $T = 3600$ seconds. Table 4 contains the results obtained with simulation and with Equation 3. Equation 3 provides accurate active coverage values for large T. Table 4 shows that the active coverage is extremely small. It improves with increasing number of programs because the combined program detection latency times increase.

The coverage of the scrubber can be improved by using two scrubbers: one to scrub only unused memory while another scrubs the allocated pages in memory. This heuristic can be implemented easily because the memory system keeps track of allocated and unallocated pages. The unused memory must also be scrubbed to prevent latent errors from thwarting a CPU re-integration. The overhead of the original scrubbing scheme is:

$$OVHD_{old} = Nk/T$$

where, N is the number of 32-byte blocks in the memory and k is the overhead for scrubbing one block. The overhead per hour, for $T = 3600$ seconds is approximately 1.5

Active Coverage of New Scrubber			
$OVHD_{new}/OVHD_{old}$	1 Prog.	4 Prog.	16 Prog.
1	0.33	0.32	0.31
2	0.41	0.40	0.38
4	0.45	0.43	0.41
8	0.51	0.45	0.44
16	0.61	0.49	0.46

Table 5: Active coverage of the new scrubber.

seconds. The overhead of the new scrubbing scheme is:

$$OVHD_{new} = k(N_u/T_u + N_a/T_a) \quad (4)$$

where, N_u is the number of unallocated 32-byte blocks, N_a is the number of allocated blocks, and $N = N_u + N_a$. T_u and T_a are the scrubber sweep times for the unallocated and allocated pages, respectively. Table 5 contains the active coverage of the new scrubber for various overheads, with $T_u = 4T$. T_a is determined from Equation 4 for a specified overhead. The executable image of one Gaussian elimination program is 32KB (N_a = 1K blocks). The new scrubber improves the active coverage by one or two orders of magnitude without increasing the overhead. However, since T_u is four times slower than T, the impact of the two scrubbers on overall system dependability must also be evaluated before one is selected.

Since the dominant factor of the new scrubber is T_a, Equation 2 can provide an approximate active coverage if $f_{P_l}(p)$ can be estimated. Using a statistical package, the measured program detection latency shown in Figure 4b was found to best fit a 2-phase hyperexponential distribution, HYPER($\alpha_1, \lambda_1, \alpha_2, \lambda_2$). The fitted curve is HYPER(0.5, 0.045, 0.5, 5.435) with an r^2 value of 0.99. Using the 2-phase hyperexponential and substituting T_a for T in equation 2, the active coverage is:

$$Pr[S_l < P_l] = \alpha_1\left(\frac{1}{T_a\lambda_1} - \frac{e^{-\lambda_1 T_a}}{T\lambda_1}\right) + \alpha_2\left(\frac{1}{T\lambda_2} - \frac{e^{-\lambda_2 T}}{T\lambda_2}\right) \quad (5)$$

Table 6 contains the T_a values and the coverage estimated with the above equation. The results match those shown in columnn 2 of Table 5 well and verify Equation 5.

To show the impact of using simple estimations of application behavior, the active coverage is computed using the ramp function and also assuming $f_{P_l}(p)$ is exponentially distributed with a mean of 8.8 seconds (the measured mean for 1 program). Table 7 contains the coverages obtained. The error column compares the difference in coverage with those obtained using the software model (column 2 of Table 5). The error is due to an overestimation of the number of large detection latency times. Figure 3 shows that over half of the measured detection times are less than a second, whereas the 50th percentile for ramp is approximately

$OVHD_{new}$ / $OVHD_{old}$	T_a (sec.)	Estimated Active Coverage
1	19.62	0.34
2	8.41	0.43
4	3.93	0.48
8	1.90	0.53
16	0.935	0.59

Table 6: Estimated active coverage of the new scrubber for 1 program using a fitted $f_{P_l}(p)$.

Active Coverage of New Scrubber for 1 Program				
$OVHD_{new}$ / $OVHD_{old}$	Ramp		Exponential	
	Cvrg.	% Error	Cvrg.	% Error
1	0.70	112.1	0.41	24.2
2	0.85	107.3	0.65	58.5
4	0.93	106.6	0.81	80.0
8	0.97	90.2	0.90	76.5
16	0.99	62.3	0.95	55.7

Table 7: Active coverage of the new scrubber using ramp and an exponential detection latency distribution.

18 seconds. The coverages are very misleading and can produce extremely optimistic dependability figures causing the designers to falsely assume that dependability specifications are being met. These results emphasize the importance of application dependent evaluation – especially when studying application specific systems.

5 Conclusion

In this paper, we introduced a software model that provides a framework to evaluate the behavior of software due to hardware faults. The model represents an application program by decomposing it into a *graph model* 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 software model simulates the execution of the program while errors are injected into the program's memory space. The result provides application dependent parameters such as detection and propagation times. The model is especially useful in the early design stages because it allows designers to make application dependent evaluation of function and system level error detection and recovery schemes. The paper illustrated one use of the model with a case study. The model was used to obtain error detection latency times of Gaussian elimination programs running on a Tandem Integrity S2 system and evaluate the coverage of two memory scrubbing schemes. Error detection latency times obtained with the model were validated with measurements from an actual Integrity S2 system. Formulae which were derived to estimate appli-

cation dependent *active coverage* values of the scrubbing schemes were verified with the software model. The application dependent coverage values obtained with the model were compared with those obtained via traditional schemes that assume uniform or ramp memory access patterns. For our program, some coverage values obtained using the traditional approach were found to be 100% larger than those obtained with the software model. This result emphasizes the need for application dependent evaluation – especially when evaluating the dependability of application specific systems.

6 Acknowledgements

The authors would like to thank Krishna Subramanian for her help and suggestions in describing the model. We would also like to thank Dong Tang and Nancy Warter for comments and ideas that helped to improve the paper.

References

[1] J. Arlat, Y. Crouzet, and J. Laprie. Fault-injection for dependability validation of fault tolerant computing systems. In *Proc. 19th Int. Symp. Fault Tolerant Computing*, pages 348–355, Jun. 1989.

[2] 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. In *IEEE Inter. Conf. on Computer Design (ICCD)*, pages 561–564, Oct. 1989.

[3] J. A. Clark and D. K. Pradhan. React: A synthesis and evaluation tool for fault-tolerant multiprocessor architectures. In *Proc. Ann. Reliability and Maintainability Symp.*, pages 428–435, 1993.

[4] E. W. Czeck. On the prediction of fault behavior based on workload. Technical report, Ph.D. Thesis – Dept. of Electrical Eng., Carnegie Mellon University, April 1991.

[5] 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, Jun. 1992.

[6] J. L. Henessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.

[7] D. Jewett. Integrity S2: A fault-tolerant unix platform. In *Proc. 21st Int. Symp. Fault-Tolerant Computing*, pages 512–519, Jun. 1991.

[8] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: A fault and error automatic real-time injector. In *Proc. 22nd Int. Symp. Fault-Tolerant Computing*, pages 336–344, Jul. 1992.

[9] J. Laprie. Dependability evaluation of software systems in operation. *IEEE Trans. on Software Engineering*, SE-10:701–714, Nov. 1984.

[10] N. A. Lynch and M. J. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[11] M. H. MacDougall and J.S. McAlpine. Computer simulation with ASPOL. In *Symposium on the Simulation of Comp. Sys., ACM/SIGSIM*, pages 93–103, 1973.

[12] A. Mahmood and E. J. McCluskey. Watchodg processors: Error coverage and overhead. In *Proc. 15th Int. Symp. Fault-Tolerant Computing*, pages 214–219, Jun. 1985.

[13] J. F. Meyer and L. Wei. Influence of workload on error recovery in random access memories. *IEEE Trans. on Computers*, C-37:500–507, Apr. 1988.

[14] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. In *Proc. 22nd Int. Symp. on Fault-Tolerant Computing*, pages 328–335, June 1992.

[15] R. R. Oldehoeft. Program graphs and execution behavior. *IEEE Trans. on Software Eng.*, SE-9:103–108, 1983.

[16] H. Schwetman. CSIM: A C-based process-oriented simulation language. In *Proc. Winter Simulation Conf.*, 1986.

[17] Z. Segall and et. al. FIAT - fault injection based automated testing environment. In *Proc. 18th Int. Symp. Fault-Tolerant Computing*, pages 102–107, Jun. 1988.

[18] K. G. Shin and T. Lin. Modeling and measurement of error propagation in a multimodule computing system. *IEEE Trans. on Computers*, 37:1053–1066, Sep. 1988.

[19] L. Young, R. K. Iyer, K. K. Goswami, and C. Alonso. A hybrid monitor assisted fault injection environment. In *3rd IFIP Conf. on Dependable Computing for Critical Appl.*, pages 163–174, Sep. 1992.