NASA Contractor Report 178423

SOFTWARE-IMPLEMENTED FAULT INSERTION: AN FTMP EXAMPLE

Edward W. Czeck, Daniel P. Siewiorek,
and Zary Z. Segall

CARNEGIE-MELLON UNIVERSITY
Pittsburgh, Pennsylvania

Grant NAG1-190
October 1987

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

List of Figures

Abstract

This report presents a model for fault insertion through software; describes its implementation on a fault-tolerant computer, FTMP; presents a summary of fault detection, identification, and reconfiguration data collected with software-implemented fault insertion; and compares the results to hardware fault insertion data.

The software-implemented fault insertion model assumes faults manifest as data errors at the output of a task. The implementation of the software-implemented fault insertion model on FTMP allows inserted faults to emulate faults in the processor data path, processor control path, system memory, and system transmit bus.

The experimental results show detection time to be a function of time of insertion and system workload. For the fault detection time, there is no correlation between software-inserted faults and hardware-inserted faults; this is because hardware-inserted faults must manifest as errors before detection, whereas software-inserted faults immediately exercise the error detection mechanisms.

Fault identification time for FTMP is a function of the system configuration and the fault manifestation pattern. The software-inserted data and the hardware-inserted data show a striking difference, thus portraying the non-unique mapping between the two fault insertion methods, but attesting to the range of the software fault library. System reconfiguration times are comparable for both hardware- and software-inserted faults.

In summary, the software-implemented fault insertion is able to be used as an evaluation technique for the fault-handling capabilities of a system in fault detection, identification, and recovery. Although the software-inserted faults do not map directly to hardware-inserted faults, experiments indicate software-implemented fault insertion is capable of emulating hardware fault insertion, with greater ease and automation.

1. Introduction

Validation procedures, such as those proposed in [Carter 86, NASA 79a, NASA 79b] include steps to characterize and evaluate the behavior of a system under faulty conditions. The means for these evaluations include the following:

1. *Computer Simulation:* Computer simulation is used to evaluate the manifestation of faults and the system's response. The simulation models range from the Processor-Memory-Switch level through the Instruction-Set-Processor level, the Register-Transfer level, the gate level, and to the device level. The drawbacks to simulation are the high cost of model development, computational needs, and the difficulty in model validation [McGough & Stern 81, Rennels 84].

2. *Physical Fault Insertion:* Physical fault insertion places faults in the hardware of a realized system. The advantages over computer simulation include speed and fidelity to actual system faults. The drawbacks to this method are two fold. First, fault insertion requires physical manipulation of components, a time consuming effort [Lala & Smith 83a]. Second, the faults are limited to pin level insertion; as realizations moves from SSI/MSI to VLSI, the fault insertion level moves from gate level to system level.

This paper discusses *Software-Implemented Fault Insertion,* in which the attempt is to emulate hardware or physical faults by modifying program data or control. The motivations for software-implemented fault insertion include speed and automation advantages over simulation and physical fault insertion; the experimental run time is near, if not the same, as the actual system and software-implemented fault insertion does not require any physical manipulation of system hardware. Additionally, software-inserted faults are repeatable across architectural and implementation boundaries, since the insertion method does not require detailed information of hardware implementation. Finally, the gap between pin level fault insertion in VLSI and software-implemented faults insertion is narrowing and may be approaching equivalence.

The literature abounds with prior work demonstrating the benefits of fault insertion and the feasibility of software-implemented fault insertion at the architectural or bus level; a sampling of this prior work includes:

- Pin-level (gate-level) fault insertion has been used in several evaluations including [Decouty et al. 80, Finelli 87, Lala & Smith 83a, Lala 83]. Observations noted in [Lala & Smith 83a, Lala 83] include difficultly caused by incorrect functioning of the test module with the test equipment attached. From these fault insertion experiments, characterization of fault-handling routines, along with preliminary, but **inconclusive** data on fault coverage were generated. These experiments demonstrate the value of using fault insertion for fault-tolerant system evaluation.

- [Schuette, et al. 86] inserted transient or soft faults in a MC68000 to evaluate software triple-modular redundancy and a signature instruction stream monitor. The MC68000 realization does not allow gate-level fault insertion, hence faults were inserted on the address, data, and control bus lines. This experiment shows fault insertion at the bus level can be used to evaluate fault-tolerance techniques.

- The Sperry UNIVAC 1100/60 [Boone et al. 80] has a built-in fault insertion capability to verify fault detection, isolation, and recovery mechanisms. This capability is activated during system idle time and can insert faults in the processor, memory, and I/O unit. The UNIVAC 1100/60 system shows fault-tolerant mechanisms can be verified using software control at the system level.

- [Yang et al. 85] inserted faults into the iAPX 432 to evaluate software-implemented triple-modular redundancy. Faults were inserted by altering bits in the program or data areas of memory using the debugger. The experiment shows fault insertion may be accomplished by altering bits in the memory.

This paper is divided into five sections. The second section gives an overview of the FTMP architecture with emphasis on the fault-handling mechanisms. Section 3 describes a model for a fault-tolerant system, a model for fault insertion at the architectural level, and the implementation of this model on FTMP. Section 4 presents data from software-implemented fault insertion experiments and provides a comparison, where applicable, to similar hardware fault insertion experiments. The last section concludes the paper with an evaluation of software-implemented fault insertion techniques.

2. FTMP Architecture

This section presents an architectural description of FTMP, the target machine for the software-implemented fault insertion. Four subsections include a general overview, followed by a detailed description of fault detection, fault identification, and fault recovery mechanisms

2.1 General Overview

The Fault-Tolerant Multiprocessor, FTMP, is a hardware-redundant multiprocessor designed for ultrareliable avionic environments [Hopkins et al. 78, Smith & Lala 83, Lala & Smith 83b]. The architecture, as seen by the programmer, consists of three virtual processors with local memory, connected via a common bus to global memory and I/O ports. Reliability is attained through hardware redundancy; each virtual processor consists of a processor triad. The memory and buses are also triplicated. Spare processors, memories and buses shadow (i.e. execute the same code as) the active units, but do not participate in voting. Each triad executes synchronously and a hardware vote occurs during data transfers. Voting is performed by each receiving unit, either a processor or memory, from data transferred over redundant buses.

The bus structure consists of four sets of serial buses, each quintuply redundant, of which three are active at any time. The bus sets are: the *Poll Bus*, which is the bus arbiter; the *Transmit Bus*, which carries address and data information from the processor; the ***Receive Bus***, which carries data from global memory or I/O ports to the processors; and the ***Clock Bus***, which carries clock signals to each processor to maintain system synchronization.

FTMP employs a realtime operating system with a basic dispatch period of 40 milliseconds, referred to as Rate-4. There are two lower rate groups, Rate-3 with a period of 80 milliseconds, and Rate-1 with a period of 320 milliseconds. Tasks are assigned to rate groups depending upon the task's execution requirement; most of the system tasks, such as the system configuration controller (SCC), a memory checker, status display and self tests are assigned to the lowest rate group, Rate-1.

2.2 Fault Detection

The fault detection mechanism for FTMP employs hardware voters residing at the receivers of each bus set. A disagreement at the voter sets an error latch associated with the bus line in error. SCC, running as a Rate-1 task, reads the error latches to check for errors and potentially faulty units. If an error is detected, the time of the error is stored and the fault identification routine is called. A time line of the events occurring in fault detection, identification, and reconfiguration is shown in Figure 2-1.

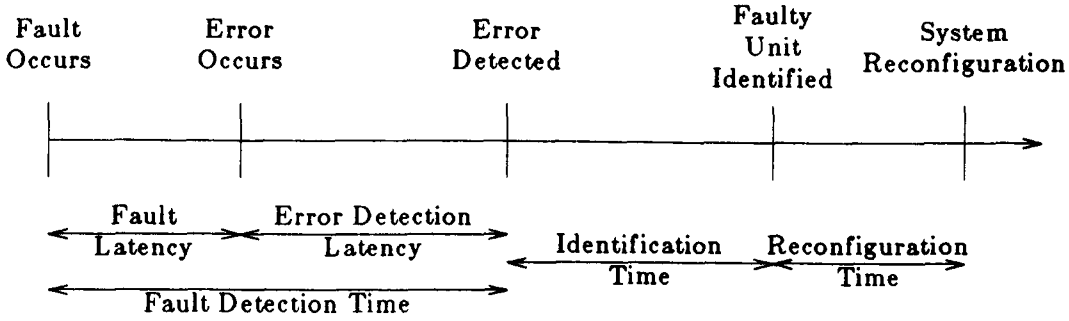

Figure 2-1: Time Line of Fault Detection, Identification and Reconfiguration

2.3 Fault Identification

The goal of fault identification is to determine from the error latch information which unit caused the error. Since an error on one bus may be attributed to multiple sources (each unit enabled on the bus), the general procedure of the fault identification routines is:

1. Determine all possible sources of the error from the error latch information. If there is more than one source, the bus assignments are switched and the identification routine waits till the next Rate-1 frame for another error to occur.
2. If another error occurs, its possible sources are identified and intersected with the previous possible sources. If the new set is not unique, this step is repeated after switching bus assignments again. If an error does not occur, a transient fault analysis routine assigns demerits to all possible sources.

The fault identification routine runs as part of SCC; hence, the identification time will be a function of the number of passes needed to identify the faulty unit.

2.4 System Reconfiguration

The system reconfiguration procedure entails the removal of faulty units either by activating a spare unit or by graceful degradation. These procedures are described as follows:

1. If there is a spare unit (Processor, Memory, or Bus) and it is shadowing the faulty unit, the bus assignments are changed to bring the spare unit active and the failed unit inactive.
2. If there is a spare processor or memory and it is shadowing a triad other than the one containing the faulty unit, then the spare is brought to shadow the triad with the faulty unit, the spare activated and the failed unit deactivated.
3. Finally if there are no spare processors, the triad is retired with its good processors assigned as spares. When memories or buses fail without spares, the triad reduces to a duplex.

The Rate-4 dispatcher executes the reconfiguration commands from the information supplied by the fault identification routine. The reconfiguration time is defined as the time from fault identification to the execution of the reconfiguration commands.

3. Software-Implemented Fault Insertion

This section describes a model for a fault-tolerant system, and a model of software-implemented fault insertion for a realtime operating system. The realization of the software-implemented fault insertion is presented on the example system, FTMP.

3.1 Fault-Tolerant System Model

Fault-tolerant systems generally use either hardware or time redundancy to achieve reliability. Under each of these schemes, there are confinement regions (hardware or time) which localize the corruption caused by a fault. Associated with the region, usually at the boundary, is an error detection and fault isolation mechanism (EDFI) which limits fault propagation and detects errors. The EDFI also generates a status report showing the condition of the region or system. Figure 3-1 presents a system model, based on confinement regions, where the regions are processors, P, memories, M, and I/O units, interconnected via buses through the EDFI interface.

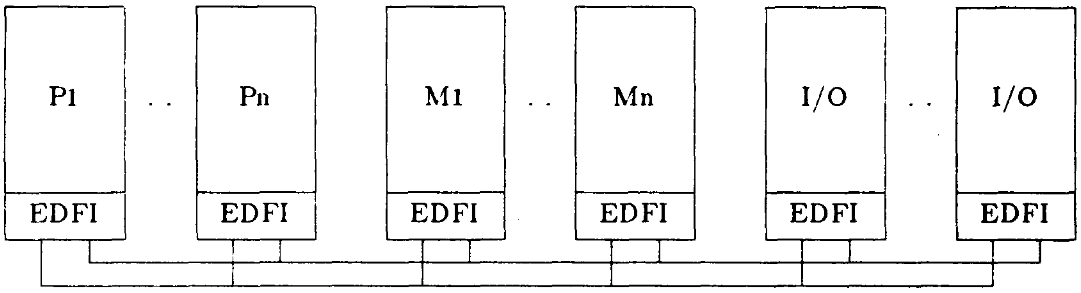

Figure 3-1: Fault-Tolerant System Model

The goal of software-implemented fault insertion is to force the system to appear faulty by exercising one or more of the EDFI interfaces by one of the following means:

1. Immediate activation where the EDFI is exercised by an error at the boundary of a confinement region, or
2. Latent activation where faults are seeded within the confinement regions.

A comparison between software fault insertion and hardware fault insertion includes:

- The goal of both fault insertion schemes is to exercise and evaluate the fault-handling mechanisms of the system.
- Software-inserted faults are inserted at the architectural (or programmers) level, several levels removed from the generation of true physical faults, but it is the level at which the system architect considered faults and errors.

- Fault propagation, the spreading of adverse physical phenomena [Laprie 85], is constrained within the *fault containment regions*, and would be tested by: failure modes and effects analysis, low level physical fault insertion, and simulation.
- Error propagation, the spreading of errors within the system [Laprie 85], is the primary area which software-inserted faults can be utilized in system evaluation.
- Software-implemented fault insertion is analogous to functional level testing of hardware or software, which has been shown feasible in the literature [Howden 80, Lai 79].
- The set of faults at the architectural level is more manageable, reduced in size and complexity, than the set of gate level or functional faults.
- Software-inserted faults may be better in triggering a specific response (such as a malicious liar), which is difficult to generate or reproduce with physical fault insertion.
- For some desired errors, physical fault insertion may be the easiest method to generate these errors.
- Physical fault insertion may be more analogous to actual faults generated in the system.

3.2 System Task Model

A model for a computational task is shown in Figure 3-2a. Data (sensors) are read at the start of the task, operations are performed on the data, and the results are written (to actuators). A fault occurring in the task would manifest as an error in the output of the results. These errors include incorrect data, no data, or late data. Hence faults in the task could be modeled as an error in the output part of the task, Figure 3-2b. Realtime execution could be modeled as a series of computational tasks with the dispatcher, D, executing between the tasks as shown in Figure 3-2c. Adjusting the task model to fit the multiple execution rates of FTMP, let L be a lower rate task, where L is all the non Rate-4 tasks[1] concatenated together to form a single task. The L task executes at the end of the Rate-4 tasks and is interrupted by the start of the next Rate-4 frame, Figure 3-3; thus, the amount of execution time per Rate-4 frame for the L task depends on the Rate-4 frame size and the execution times of the Rate-4 task and dispatcher.

[1] The lower rate tasks include a clock update, System Configuration Controller (SCC), memory checker, and status display which execute at 3.125 Hz, one-eighth of the Rate-4 tasks.

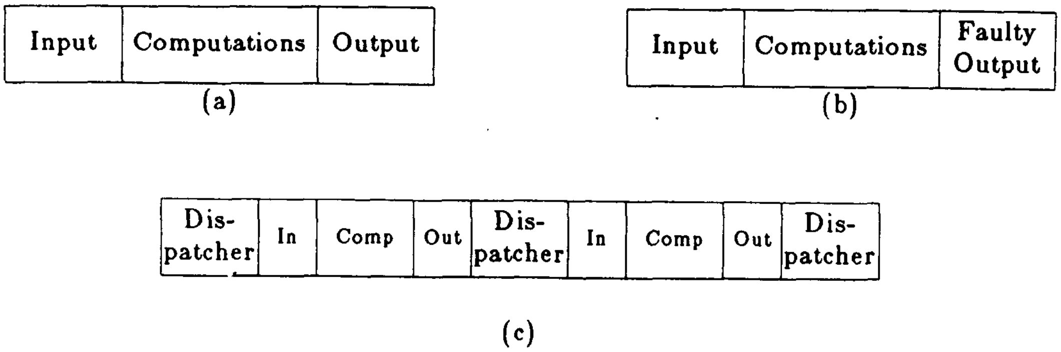

Figure 3-2: Computational Task Model

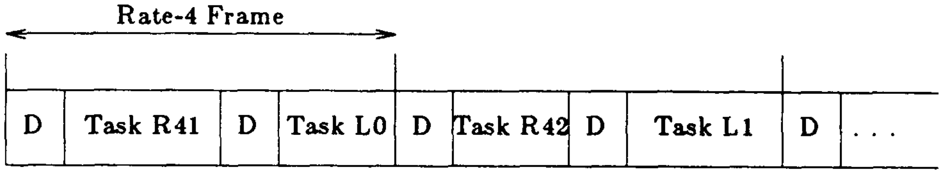

Figure 3-3: Lower Rate Task Execution Model

3.3 Software-Implemented Fault Insertion Realization

The desired abilities of software-implemented fault insertion or of any fault insertion in general are the following:

- Location of Fault: Insertion of faults should be able to model true faults which can occur throughout the system hardware.
- Timing of Faults: A fault may occur throughout any execution task of the system; a fault insertion environment should allow similar conditions.
- Duration of Faults: Faults are classified as either transient, intermittent, or permanent [Siewiorek & Swarz 82]; the fault insertion should allow the duration of inserted faults to vary accordingly.

The realization of the software-implemented fault insertion in FTMP is unfortunately limited by the controllability of hardware by the software programmer. Other highly reliable systems should provide the necessary *"hooks"* for fault insertion and fault monitoring [Decouty et al. 80].

3.3.1 Location and Generation of Faults

The fault insertion environment must be able to insert or emulate faults in different locations. The software-implemented fault insertion environment presently allows faults in four regions; these regions and the means which the faults are inserted are described as follows:

- *Processor Data Path Faults:* Faults occurring in the data path may manifest as a number of different error types. These include transmission of incorrect data, no data or late data. The software-implemented fault insertion environment assumes processor data path faults manifest as incorrect data being transmitted by the processor, causing an error on the transmit bus assigned to the processor. The incorrect data is a single word and the processor state remains good after the transmission of the bad data. This fault class assumes that faults which corrupt multiple words would be detected easier than the single word fault. This generality of this fault class does not limit its application to FTMP.

- *Processor Control Path Faults:* Faults within the control path may manifest to many different error types. These include no data transmitted, early or late data transmitted, or incorrect data transmitted. The software-implemented fault insertion environment emulates faults within this region by having the processor execute an infinite loop, resulting in no transmission of data. This causes errors on two of the buses to which the processor is assigned: the poll bus because the processor never requests the bus, and the transmit bus because no data is transmitted. Similar emulations of control faults can be used on other systems.

- *System Bus Faults:* Faults on a bus may be attributed to many sources, such as noise or a unit transmitting out of protocol. Software-implemented fault insertion emulates bus faults by having a processor transmit bad data on a specific bus. Although the processors are generating the errors, the errors map to a particular bus.

- *Global Memory Faults:* Memory faults may be attributed to decaying bits, stuck-at bits, or incorrect address decoding. Memory faults are emulated by writing bad data into one memory module of a triad, and then performing a read of the location. These type faults are the most general type and can be emulated within other systems.

A few comments on the fault insertion are in order. First, all inserted faults cause an immediate error; there is no latency between insertion and error occurrence. The present implementation does not exclude latent faults from further additions. Second, the faults are transient and cause no change in processor state or corruption of data, except for the control path fault.[2] Although this is a simplistic model, this is a minimum fault. A faulty processor, or one with a corrupted state will (most likely) have a stream of incorrect data.

[2] The memory fault is also transient; once the data is changed and read, the memory word is written over.

3.3.2 Timing of Faults

Faults may occur at any time within the execution of a program. The model assumes faults manifest as errors in the output portion of the task. The implementation of software-implemented fault insertion allows faults to be generated in the output of Rate-4 application tasks. The occurrence of a fault is specified to a particular Rate-4 frame, but not to the time within the frame. Furthermore, faults may only occur in Rate-4 application tasks and not within the dispatcher or lower rate tasks. This limits the insertion time of the faults to specific tasks.

For FTMP and other hardware-redundant systems, a fault occurring within a task, such as the dispatcher or the fault-handling routines, would be limited to one of the fault confinement regions. The occurring fault would thus, have no effect on the correct functioning of the other redundant processors.[3] Additionally the error detection mechanism for FTMP cannot distinguish the insertion time or location to a specific task.

3.3.3 Duration of Faults

Faults can be transient (soft), due to a temporary random environmental condition or permanent (hard), due to a physical change in the hardware. The software-implemented fault insertion environment generates transient faults, and to emulate permanent faults, a transient fault is repeatedly inserted in consecutive Rate-4 frames. Thus a module with a permanent fault produces a stream of faulty data, which gives the appearance of a permanent fault when viewed from the error detection and identification mechanisms.

3.3.4 Workload

A system's workload is its set of inputs received from its environment; a desirable feature within any computer evaluation environment is a controllable workload. [Feather et al. 85] developed a synthetic workload[4] generator for FTMP which has been modified to include the software-implemented fault insertion. The synthetic workload provides a means of specificing the following factors:

[3] This assumes only single faults, and the absence of simplex data or control.

[4] A synthetic workload exercises a computer system by modeling its natural workload with generic inputs and tasks.

- System Configuration, which defines the number of processor triads and spares.
- Number of Tasks for each rate group, and the inclusion of the system tasks, such as SCC and Status Display.
- Workload for each task, which includes the amount of I/O and computations per task.

3.3.5 Recovery Mechanism

To gather a large data base of fault insertion data, a recovery mechanism must augment the software-implemented fault insertion environment. Draper Labs modified the system configuration controller to repair and activate processor 3 before fault insertion.[5] This repair code was modified allowing for the repair and activation of the last unit failed (processor, memory, or bus), before each fault insertion.

3.4 Experimental Environment

The experimental environment for software-implemented fault insertion can be divided into three sections: the experimental set-up and selection of parameters, the execution of the experiment and the collection of data, and the analysis of data. This section describes these areas of the FTMP implementation.

3.4.1 Parameters

The first phase of the experimental procedure is experimental setup and selection of parameters. A program queries the user on the selection of the parameters, and from the inputs, generates a command file which properly configures FTMP and collects data. The controllable parameters include:

- Workload: The system workload includes the amount and distribution of tasks between the rate groups, the amount of I/O and computation executed by each task, the inclusion of system tasks, and the overall system configuration.
- Location: The different locations for fault insertions are described in Section 3.3.1.
- Timing: The time of fault insertion is controlled by the Rate-4 frame, hence a 40 millisecond resolution.
- Duration: Either a transient (single) fault or a permanent (repeated) fault may be inserted, as described in Section 3.3.3.
- Data Collected: The data which may be collected includes: the application tasks' execution time; the fault insertion, detection, identification, and reconfiguration time; the identification of the failed unit; and the reason code for the failure.

[5] Draper's fault insertion environment inserted faults in Processor 3; before a fault was inserted, the software assured Processor 3 was active in a triad.

3.4.2 Experimental Execution

The second phase of the experimental procedure is insertion of faults and the collection of data. During each experimental loop the following actions occur:

1. The system repairs any module which failed during the last cycle.
2. The fault inserter is started and the workload data collection cycle begins. Workload data (task execution times) are collected for one Rate-1 frame, and the inserted faults trigger the fault-handling mechanisms whose execution times are also collected.
3. The requested data is uploaded to the host VAX and the cycle repeated.

The cycle time is approximately 30 seconds: one-half is for wait states between steps one and two, and steps two and three, which may be reduced by passing ready signals. The other half is consumed during the uploading of the data, which is dependent on the data requested by the user.

3.4.3 Data Analysis

The third phase of the experimental procedure is data analysis. The data analysis program takes the absolute timer values collected from FTMP and records differences between two events as requested by the user. The average, standard deviation, minimum, maximum, and a histogram of the data are then printed.

4. Experiments

With the Software-Implemented Fault Insertion Environment implemented, the next step was to run experiments evaluating the feasibility of software-implemented fault insertion, and the capabilities of the environment. The experiments exercised most of the parameters available in the software-implemented fault insertion environment. In particular the location of the fault, time of insertion, and system configuration were the primary parameters varied. The collection of data from these experiments involves a measurement of the system workload and the times of fault insertion, fault detection, identification, and system reconfiguration. Additionally, the unit which failed and the reason code for the failure were stored for analysis of incorrectly diagnosed faults. This section details the experiments performed. Comparisons to hardware fault insertion results [Lala & Smith 83a, Wimmergren 82] are made where appropriate.

4.1 Summary of Draper's Fault Insertion Data

Draper, under contract to NASA, completed extensive hardware fault insertion experiments at the pin (gate) level [Lala & Smith 83a]. Wimmergren [Wimmergren 82] also performed fault insertion experiments on FTMP for his Master's research; his results are compared when appropriate, with the same assumptions as Draper's data. This section summarizes their experiments and presents a comparison between Draper's hardware-implemented fault insertion and the software-implemented fault insertion experiments.

Draper's experiments inserted faults at the pin level of the processor; the types of faults were single stuck-at zero, stuck-at one, or inverted. The data is divided by the fault location, where the locations are cards in the LRU's.[6] For each card, several chips were pulled and faults inserted on each of the chips. For our comparison, Draper's data from four different cards were taken: the CPU data path card (CPUD), the CPU control path card (CPUC), the bus interface transmit card (BIT), and the cache controller card (CC). These correspond to the software-implemented fault insertion locations of data path, control path, transmit bus, and data path respectively; Figure 4-1 diagrams a hypothesized mapping between hardware and software-inserted faults.

[6] An LRU is a Line Replaceable Unit; each is identical and contains a processor, memory and the necessary bus interface circuitry.

The parameters for Draper's data were many times unspecific; therefore, for the purpose of comparison to the software-inserted faults, some assumption were made:

- The time of fault insertion was random for Draper's data, whereas with the software-implemented fault insertion, the insertion time was specified to the output portion of a Rate-4 task.

- The system workload was unknown, but we will assume a light workload with the Rate-4 frame size at 40 milliseconds for Draper's data. The workload for the software-insertion was one Rate-4 task, one Rate-3 (timer) task, and three Rate-1 tasks (display, SCC, and Readall), and the Rate-4 frame size was stretched to 50 milliseconds; hence the Rate-1 frame size was 400 milliseconds.[7] The difference in frame sizes for the two insertion methods should increase the time measurements for the software-inserted faults by approximately 25% in comparison to the hardware fault insertion measurements.

- While the system configuration for Draper's data was unknown, a reasonable assumption is that three triads were executing with either zero or one spare processor. The software data lists the configuration either as two or three triads without a spare, or two triads with a spare.

Draper's Hardware Fault Injection Location	Possible Fault Manifestations	Software Fault Insertion Comparison Location
CPU Data Path Card	Bad Data from Processor	Data Path
	No Data from Processor	Control Path
CPU Control Path Card	Processor Hangs	Control Path
	No Data from Processor	Control Path
	Bad Data from Processor	Data Path
Cache Controller Card	Processor Hangs	Control Path
	No Data from Processor	Control Path
	Bad Data from Processor	Data Path
Bus Interface: T-Bus Card	Bad Data on Bus	T-Bus
	No Data on Bus	T-Bus

Figure 4-1: Possible Fault Mapping Between Hardware- and Software-Inserted Faults

[7] The frame sized was stretched by the operating system to allow for the higher workload.

4.2 Fault Detection Time

Fault detection time is the time from the insertion of a fault until an error is detected by the system. For software-inserted fault, the insertion time is at the end of a specified task, whereas with Draper's hardware-inserted faults, the insertion times are any point within the frame. Error detection, reading of the error latches, is done by SCC at Rate-1. Hence the detection time for software-inserted faults should be a maximum of one Rate-1 frame (400 milliseconds), the latency in reading the error latches. For hardware-inserted faults the detection time will include the manifestation of the fault as an error, along with the delay in reading the error latch, that is, fault latency plus error detection latency.

In predicting the detection time for software-inserted faults, the following parameters affecting the detection time are proposed:

- Workload: A large workload stretches the frame size, placing the detection point later in the frame. Likewise, a large workload limits the execution time per Rate-4 frame of the lower rate tasks (e.g. error detection). The workload function is expressed by *R4task*, the Rate-4 **task size**, and *R4Frsize*, the Rate-4 **frame size**, both measured in milliseconds.

- Time of error detection: The point at which error detection occurs within the realtime cycle affects the error detection latency. This time within the realtime Rate-1 cycle is determined by the amount of time which the lower rate task executes before the error detection routine is run. This time is represented as *LDet* and is measured in milliseconds.

- Time of Insertion: The of point of fault insertion within the realtime cycle in conjunction with the time of error detection governs the fault detection latency. The time of insertion is represented as *Tin* and measured in Rate-4 frames.

Finally, let: *LxTime* be defined as the amount of time that the lower rate tasks execute per frame, where *LxTime*=max (*R4FrmSize*–*R4Task*,10) milliseconds, where 10 millisecond is the amount of time the dispatcher will allow for the execution of lower rate tasks. The error detection time can be represented by:

$$DetTime \;=\; R4FrmSize \times [[(\frac{LDet}{LxTime}) - Tin] \bmod 8] \qquad (4.1)$$

The quotient in Equation (4.1) marks the Rate-4 frame in which the error detection task runs; the *modulo 8* term comes from the realtime cycle of FTMP (eight Rate-4 frames per Rate-1 frames).

Equation (4.1) is plotted in Figure 4-2 as error detection time versus frame of insertion for different workloads; two experimental runs of software-inserted faults are also plotted. In Figure 4-2, one Rate-1 cycle, consisting of eight Rate-4 frames, is plotted twice to show the continuity of the error detection time in the realtime cycle. The high workload data has a Rate-4 frame size of 50 milliseconds and the low workload data a 40 millisecond frame size. In comparing the data of Figure 4-2, the experimental data corresponds closely to the computed data. The reason for the multiple data points for each insertion time is that error detection can be accelerated or delayed one Rate-4 frame cycle, due to the run time task allocation of FTMP. The increase in the slope as the workload increases is due to the lengthening of the basic frame size, hence placing the error detection a further time away from the fault insertion.

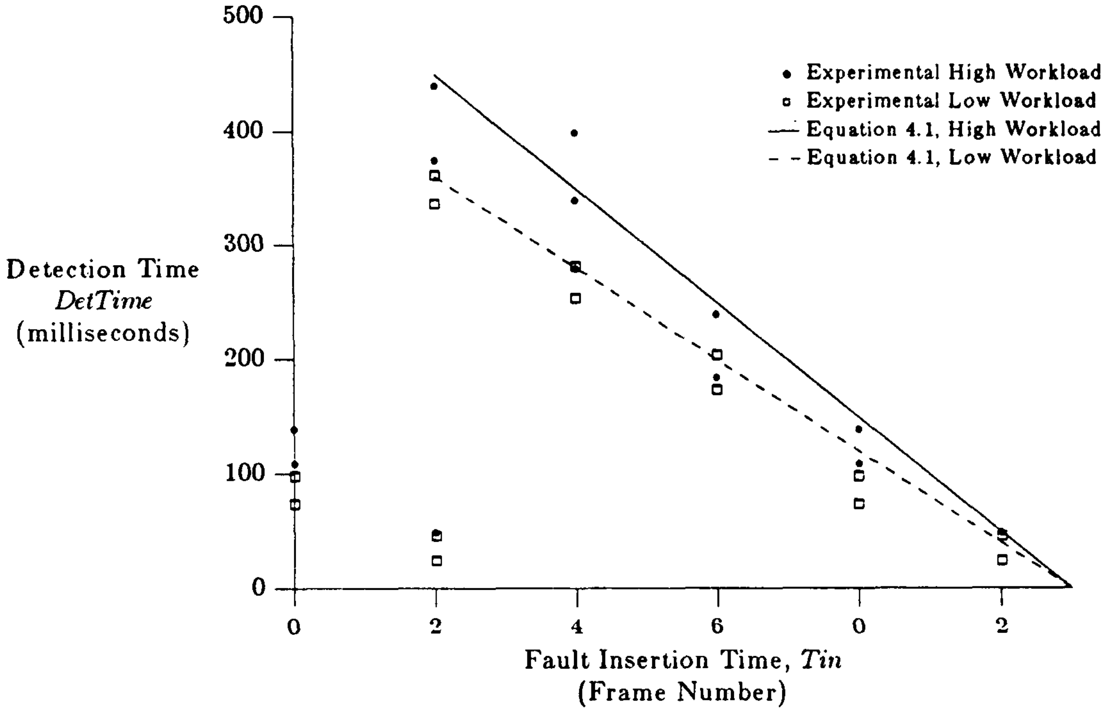

Figure 4-2: Error Detection Time as a Function of Insertion Time

Figure 4-3 shows histograms for the fault detection time with the time of fault insertion varying between graphs. The fault location is the data path, but this data is representative of the other fault locations. The time skewing between the graphs shows the lengthening of the detection time as the fault insertion time moves relative to the fixed detection time. Figure 4-4 shows histograms of fault detection time for Draper's hardware fault insertion data. The detection time is approximately two times larger than the software-inserted faults. The difference is due to the manifestation time of faults as errors, whereas with software-inserted faults, the detection time is only the latency between inserting the fault and reading the error latches (Figure 2-1). Another difference between the two data sets is the distribution of the data; the software-inserted faults fall into two or three clusters, while the hardware-inserted faults are dispersed across the whole range. The random insertion time of Draper's faults and the delay in fault manifestation contribute to the apparently more random distribution of the data.

From the fault detection time measurements, the parameters affecting the fault detection time were shown. Also achieved was the characterization of the process from the error occurrence (error latch set) to the error detection (error latch read), but not the events from fault occurrence to error occurrence (Figure 2-1).

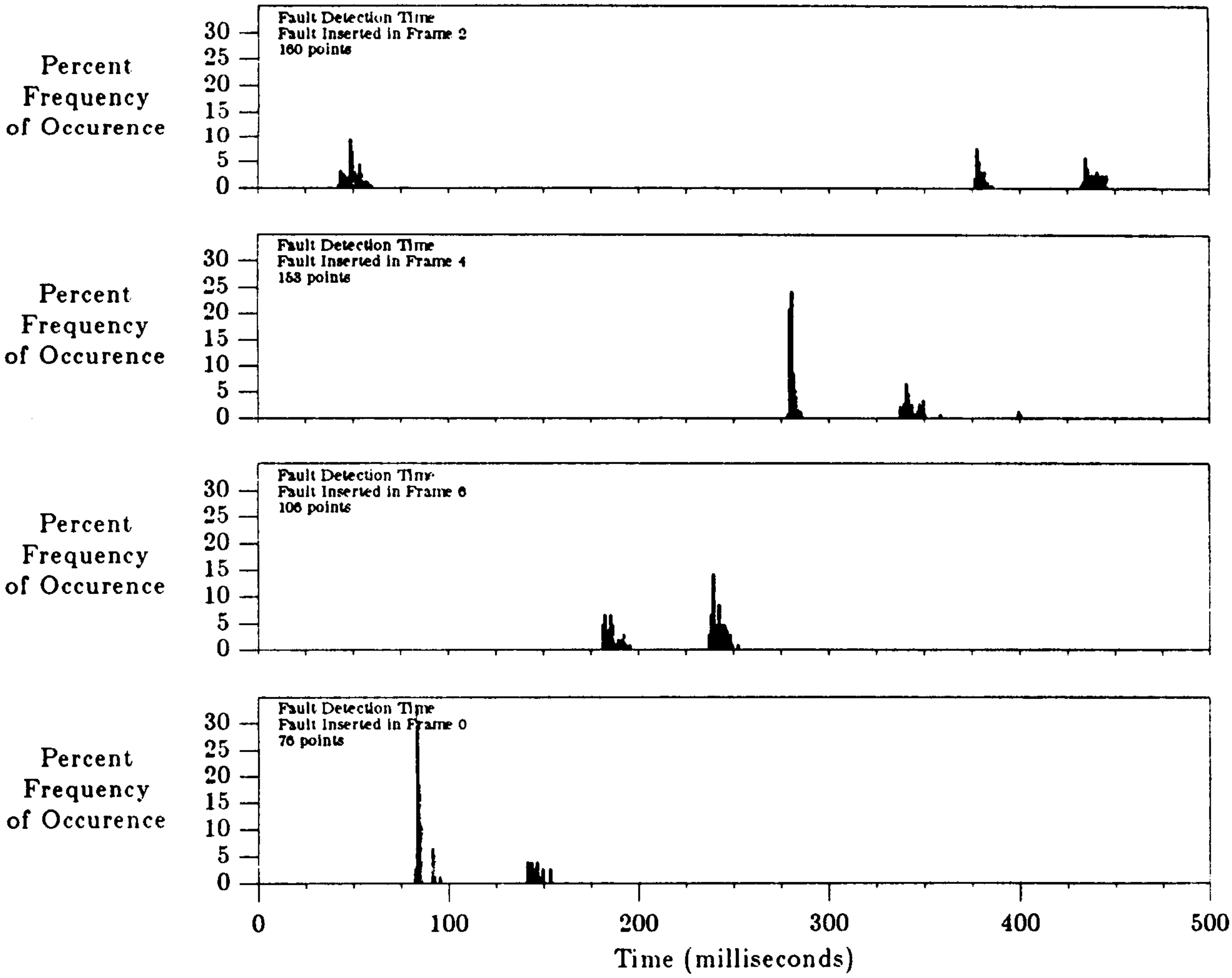

Figure 4-3: Fault Detection Time for Software-Inserted Faults

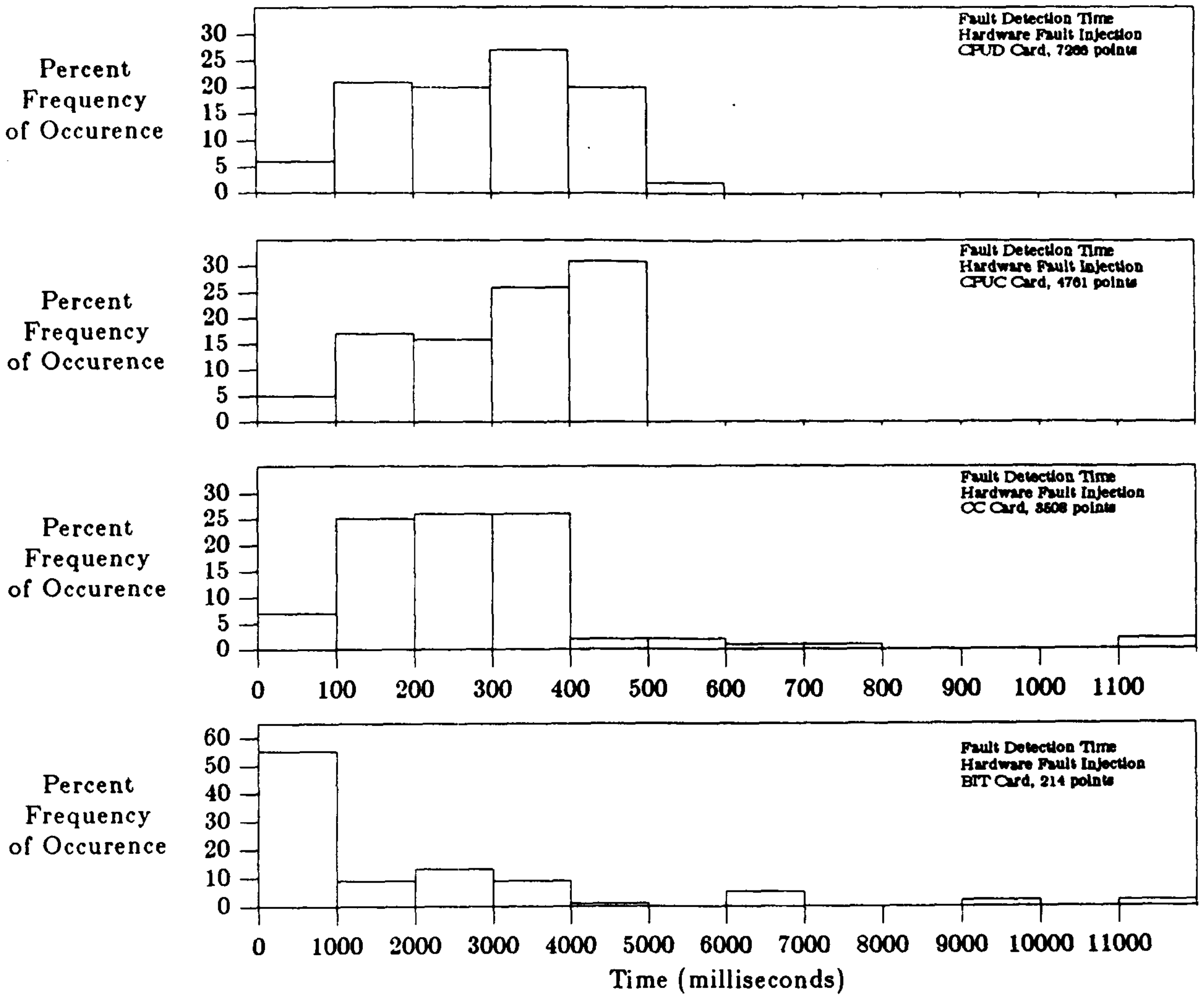

Figure 4-4: Draper's Hardware-Inserted Fault Detection Time

4.3 Fault Identification Time

The fault identification time is the time from the detection of an error by the system until the source of the error is identified. For both software- and hardware-inserted faults, the expected data should be similar; the mechanisms involved are the same (Section 2.3).

The primary parameter affecting this data is the manifestation of the fault; if a fault manifests as errors on different buses, then the possible sources of the fault is limited. A secondary parameter is the number of possible sources for the error. This is dependent on system configuration: the more processors, the more sources of errors. That is: as faults manifest to bus errors, the possible sources are all modules enabled on the bus; as errors on different buses occur from the same faulty source then the bus error "signature" will map to fewer modules. The experiments conducted varied the fault locations (manifestation), and the system configuration (possible sources).

The data should be grouped according to the execution time of the identification routine, which is dependent on the number of suspect units. The routine runs as a Rate-1 task, once per 400 milliseconds, with the data grouped according to the number of passes.

Figure 4-5 shows histograms of fault identification times for the software-inserted faults. The main feature of the data is the clustering in the distribution. This was expected; the distribution is in multiples of the Rate-1 frame size, approximately 400 millisecond. Thus Figure 4-5 also shows the number of execution cycles required for the identification routines.

Of interest is the control path fault with two triads executing; the identification time is under 50 milliseconds, hence the source was located without reconfiguring the system. The reason for this is as follows: the control path fault sends one processor of the triad into an infinite loop. As the other processors continue execution they will transmit on both the poll bus and the transmit bus causing errors to occur on each. These two errors are sufficient to determine the source and hence the faulty unit is identified without further information.

At the other extreme is the transmit bus fault with three triads executing. Here the number of error sources is four, the bus and the three processors enabled on the bus. This should require a minimum of two bus swaps to determine the source of the error. In this example three bus swaps were required. This is due to a coding error in the identification routine which does not swap buses on all triads. Related to this coding error was another error uncovered during preliminary experimentation. During these experiments, transient faults were inserted, with the response monitored at the console. Reconfiguration commands issued by the fault identification routine did not swap processor/bus assignments as documented, thus delaying system recovery.

Figure 4-6 presents histograms of Draper's fault identification time data for the four different cards in the comparison and Wimmergren's general data of identifications times. The major difference between the hardware-inserted faults and the software-inserted fault data is that a significant amount of hardware data points lie in the first bin, 0 to 100 milliseconds, with fewer outliers at the Rate-1 frame size, 320 and 640 milliseconds. As stated earlier, the fault

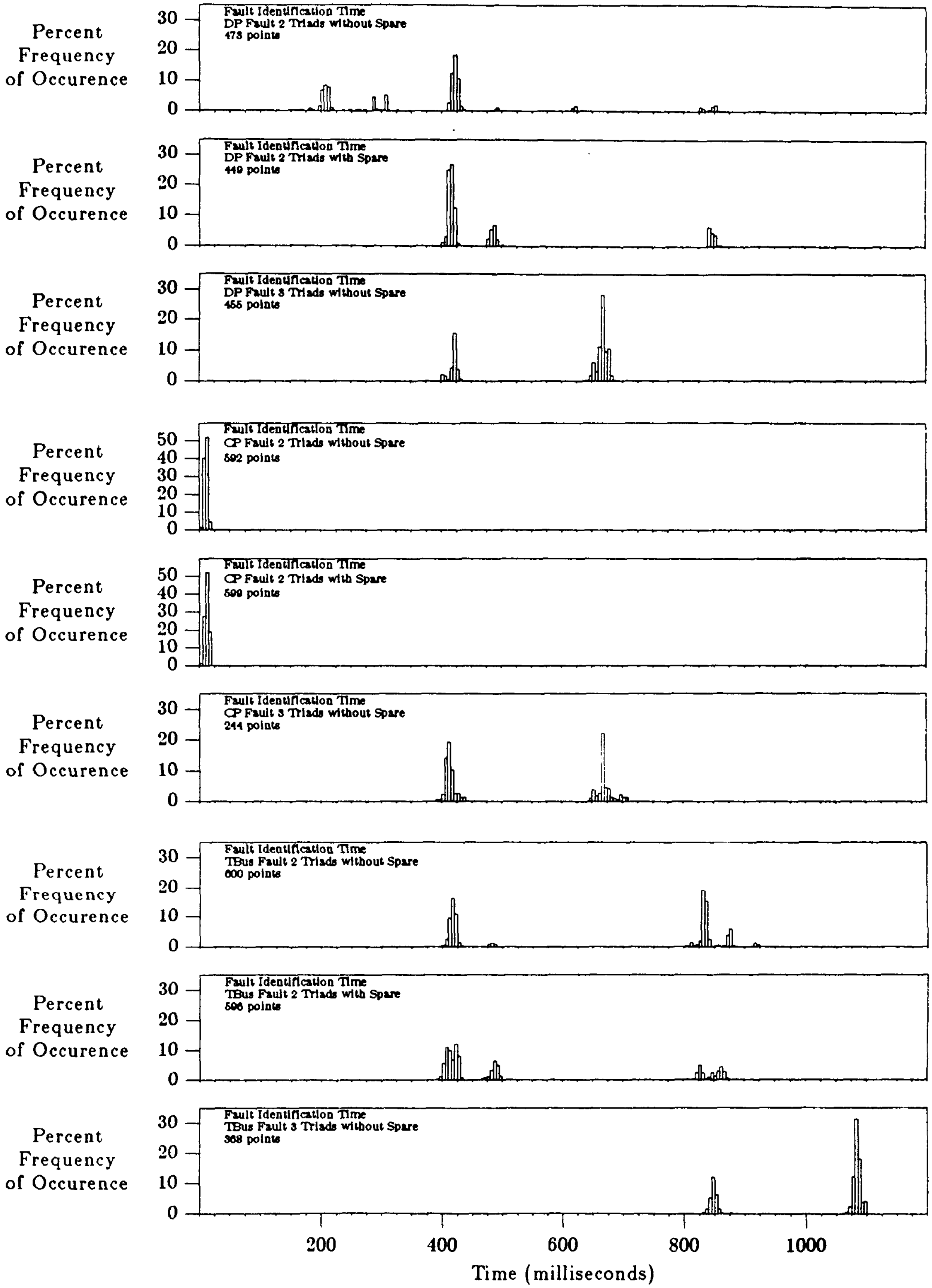

Figure 4-5: Fault Identification Time for Software-Inserted Faults

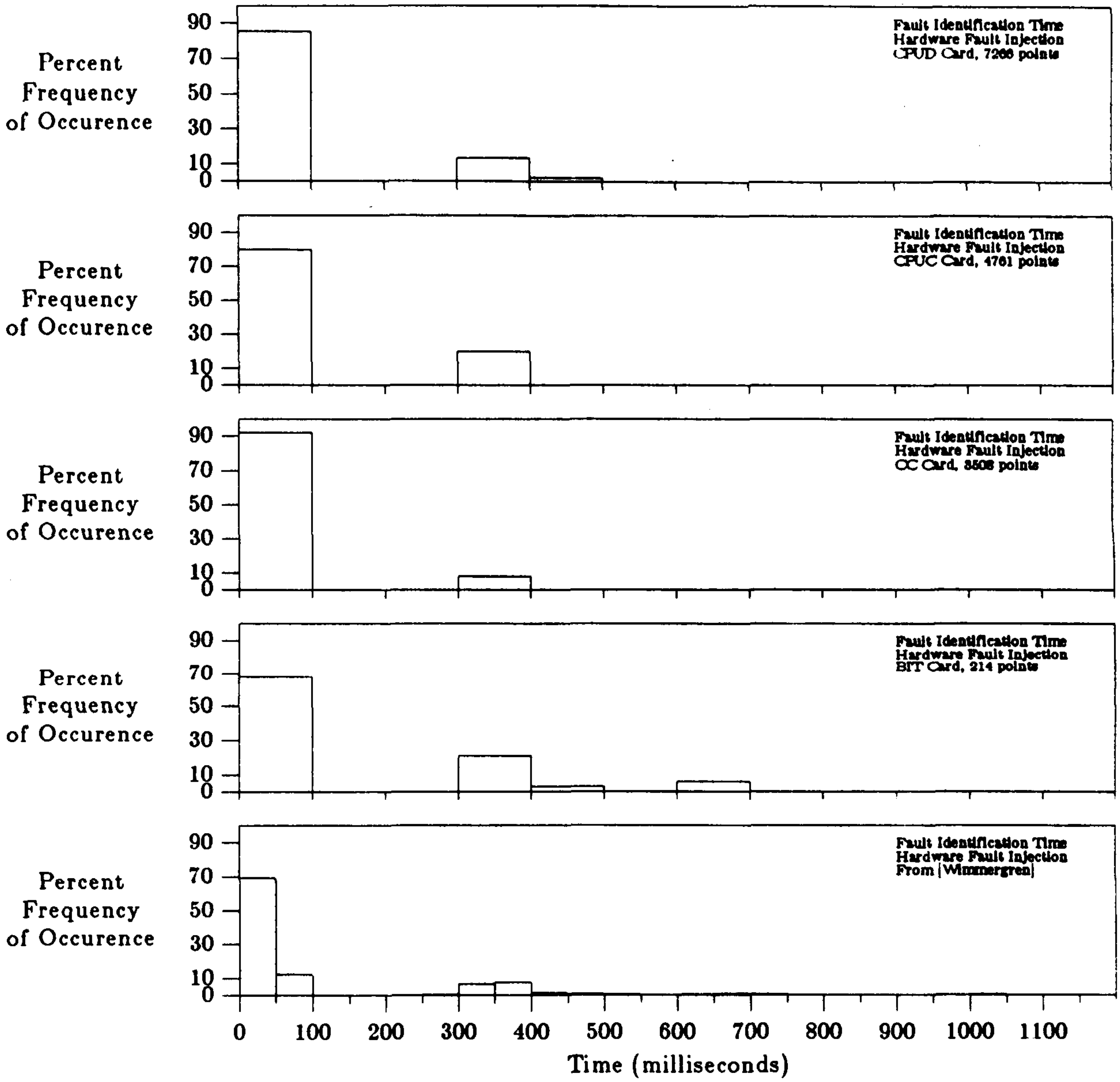

Figure 4-6: Draper's and Wimmergren's Fault Identification Time

identification time is a function of the number of suspect units to which the errors can be attributed. From analysis of data, the hardware-inserted faults manifest as errors on multiple buses which can only be attributed to a single unit.

From the experiments, the fault identification behavior was characterized with both fault insertion methods. Hardware-inserted faults manifested mostly as multiple errors, whereas the software-inserted faults allowed evaluation under single detected errors. Here software-implemented fault insertion showed an advantage over hardware fault insertion by being able to trigger specific error types necessary for recovery validation.

4.4 System Reconfiguration Time

The system reconfiguration time is the time from the identification of a faulty unit to the time when the unit is removed from the active system. The data for software-inserted faults should be similar to Draper's hardware-inserted faults. The primary parameter is the system configuration, the presense or absence of spares. The data should show an increase in system reconfiguration time when spares are not available.

Figure 4-7 shows histograms of system reconfiguration times under various system configurations and fault locations. With the data path and control path faults, the failed unit was a processor and hence the processor was retired; for the transmit bus fault a bus was marked faulty and replaced. The data show the expected increase in reconfiguration time when no spares are available, furthermore the data is clustered at 45 and 95 milliseconds. This represents the period of the dispatcher which executes the reconfiguration commands.

Figure 4-8 shows Draper's system reconfiguration data. Their data is similar to the sum of the software-inserted fault data. Draper's data lacks the resolution and specification of experimental conditions for useful comparisons, but from the two data sets, it is evident software-implemented fault insertion can be used to characterize and evaluate the fault recovery procedures of a system.

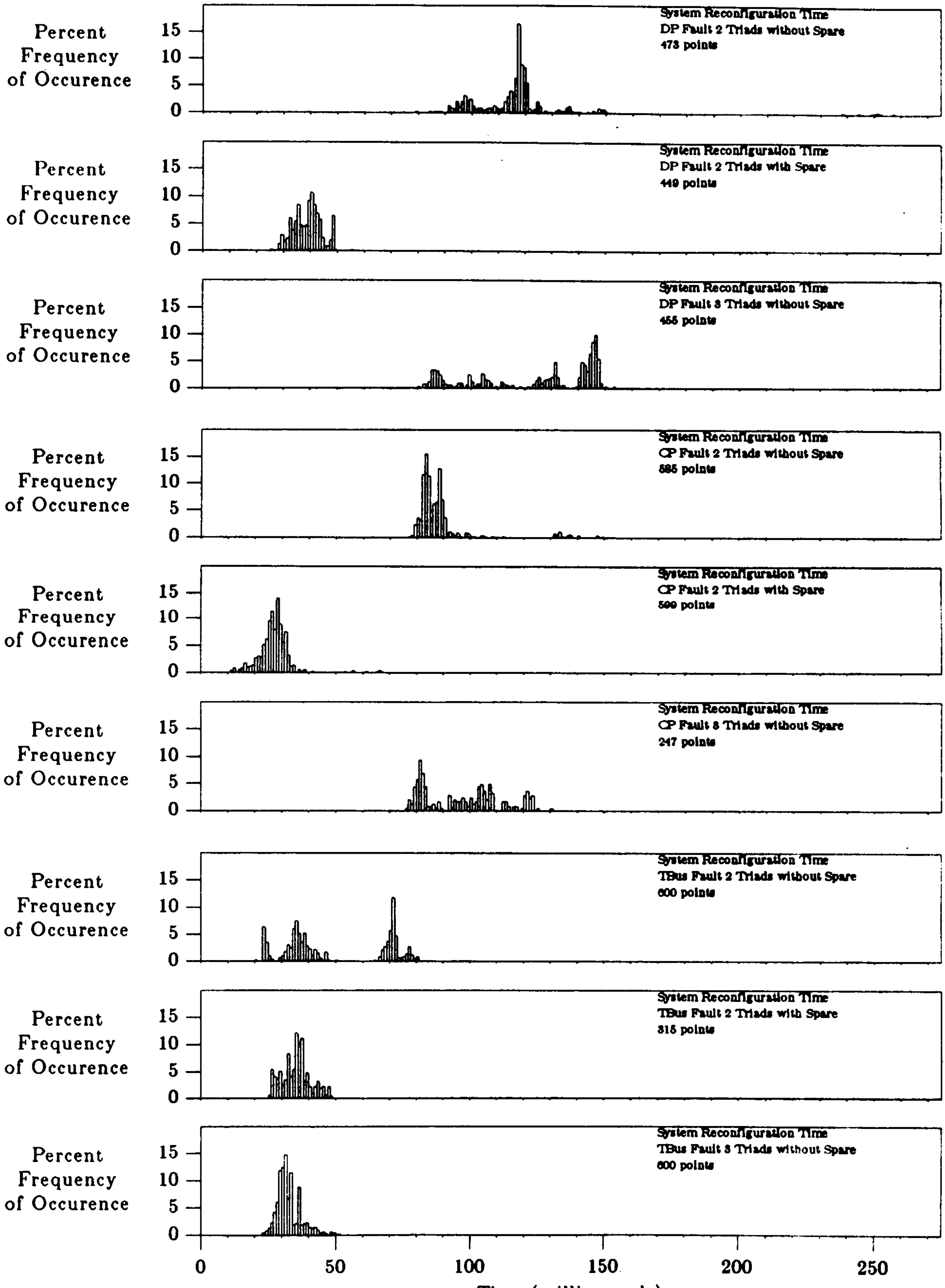

Figure 4-7: System Reconfiguration Time for Software-Inserted Faults

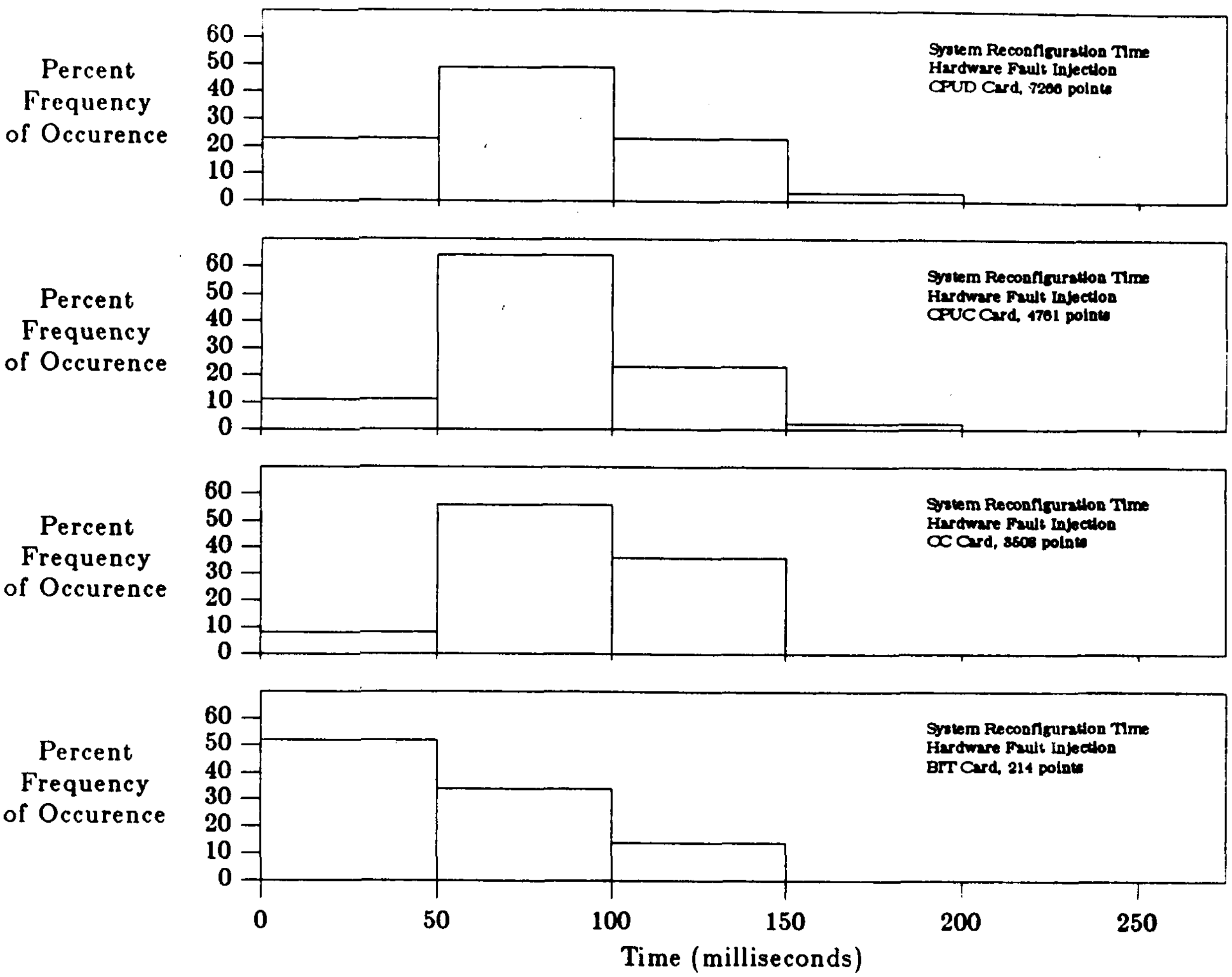

Figure 4-8: Draper's System Reconfiguration Time

5. Conclusions

This paper presented a model for software-implemented fault insertion and its implementation on a fault-tolerant computer, FTMP. Experiments were conducted to compare the software-implemented fault insertion to hardware fault insertion in the characterization of fault detection, identification, and recovery. From the experiments, the following information regarding the two fault insertion methods can be asserted:

- Both fault insertion schemes were able to characterize the fault detection, identification, and recovery times of the system.

- Hardware fault insertion places the fault at a lower level (pin level) than the software-insertion (processor level). For this reason, the detection times for the hardware-inserted faults included the fault latency times, whereas software-implemented fault insertion only included the error detection latency, Figure 2-1.

- The fault manifestation and propagation for hardware-inserted faults allows less control in the generation of specific error types than the software-inserted faults. This control was available with the software-implemented fault insertion, and was useful in discovering a bug in one of the fault-handling routines.

In summary, although software-implemented fault insertion does not fully emulate hardware fault insertion, it provides a means to evaluate the fault detection, identification, and recovery means of a system, and in some aspects provides better regulation in generating specific errors in the system. The software-implemented fault insertion can also be used for system characterization across architectural and implementation boundaries with greater ease and automation than hardware fault insertion. Furthermore, as the controllability and observability of systems decrease due to the increased use of VLSI technology, software-implemented fault insertion may be a reasonable approach to system evaluation.

Appendix A. Source Files

This appendix lists the source and executable files on the NASA VAX, System 10, which are used by the Software-Implemented Fault Insertion Environment. A brief explanation is given for each file listed.

A.1 FTMP Files

The directory containing the FTMP AED source files is **disk$MO:[cmu.aed]**; these files can also be found in **disk$MO:[ewc.swfi.wrkld.aed]**.

- **swfi.aed**: contains the procedure **swfi()** which inserts the faults into FTMP tasks.
- **nfscc.aed**: is a modified version of the system configuration controller, **FSCC**, capable of repairing any specific Processor, Memory or Bus.
- **fwrkld44.aed**: is the control task which starts the fault insertion and data collection cycle.
- **fwrkld4.aed**: contains the Rate-4 FTMP workload tasks; these tasks call the **swfi()** procedure.
- **fwrkld3.aed**: contains the Rate-3 FTMP workload tasks.
- **fwrkld1.aed**: contains the Rate-1 FTMP workload tasks.
- **fwrkld.asm**: contains the system memory tables for FTMP. This file is located in the directory **disk$MO:[cmu.asm]**.

A.2 Data Collection and Analysis Files

The directory containing the data collection and analysis programs for the *Software-Implemented Fault Insertion Environment* is **disk$MO:[ewc.swfi.wrkld.code]**.

- **wrkld.c**, **wrkld.exe**: generates the command files which configure FTMP for the experiment and collect data from FTMP during experiment.
- **anal.c**, **anal.exe**: analyzes the collected data and prints user-requested information regarding the data.
- **btree.c**: has the binary tree code used for holding data during processing by **anal.c**. **btree.c** is complied and then linked with **anal.c** to form **anal.exe**.
- **defines.h**: contains global definitions for both **wrkld.c** and **anal.c**.

References

[Boone et al. 80] L.A. Boone, H.L. Liebergot, and R.M. Sedmak.
Availability, Reliability, and Maintainability Aspects of the Sperry UNIVAC 1100/60.
In *10th International Symposium on Fault-Tolerant Computing*, pages 3-9. IEEE, June, 1980.

[Carter 86] W.C. Carter.
System Validation - Putting the Pieces Together.
In *7th AIAA/IEEE Digital Avionics Systems Conference*, pages 687-694. 1986.

[Decouty et al. 80]
B. Decouty, G. Michel, C. Wagner.
An Evaluation Tool of Fault Detection Mechanisms Efficiency.
In *10th International Symposium on Fault-Tolerant Computing*, pages 225-227. IEEE, June, 1980.

[Feather et al. 85]
Frank Feather, Daniel Siewiorek, and Zary Segall.
Validation of a Fault-Tolerant Multiprocessor: Baseline Experiments and Workload Implementation.
Technical Report CMU-CS-85-145, Carnegie Mellon University, July, 1985.

[Finelli 87] George B. Finelli.
Characterization of Fault Recovery through Fault Injection on FTMP.
IEEE Transactions on Reliability R-36(2):164-170, June, 1987.

[Hopkins et al. 78]
A.L. Hopkins, T.B. Smith, and J.H. Lala.
FTMP - A Highly Reliable Multiprocessor.
Proceeding of the IEEE 66(10):1221-1237, October, 1978.

[Howden 80] William E. Howden.
Functional Program Testing.
IEEE Transactions on Software Engineering SE-6(2):162-169, March, 1980.

[Lai 79] Larry Kwok-Woon Lai.
Error-Oriented Architecture Testing.
In *National Computer Conference*, pages 565-576. June, 1979.

[Lala 83] J.H. Lala.
Fault Detection, Isolation, and Reconfiguration in FTMP: Methods and Experimental Results.
In *5th IEEE/AIAA Digital Avionics Systems Conference*, pages 21.3.1-21.3.9. November, 1983.

[Lala & Smith 83a]
Jaynarayan H. Lala and T. Basil Smith III.
Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. III, FTMP Test and Evaluation
Charles Stark Draper Laboratories, 1983.
NASA CR-166073.

[Lala & Smith 83b]
Jaynarayan H. Lala and T. Basil Smith III.
Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. II, FTMP Software
Charles Stark Draper Laboratories, 1983.
NASA CR-166072.

[Laprie 85] Jean-Cluade Laprie.
Dependable Computing and Fault Tolerance: Concepts and Terminology.
In *15th International Symposium on Fault-Tolerant Computing*, pages 2-11. 1985.

[McGough & Stern 81]
John G. McGough and Fred L. Swern.
Measurement of Fault Latency in a Digital Avionic Mini Processor
Bendix Corp., 1981.
NASA CR-3462.

[NASA 79a] NASA-Langley Research Center.
Validation Methods for Fault-Tolerant Avionics and Control Systems - Working Group Meeting I, NASA-Langley Research Center, 1979.
NASA CP-2114.

[NASA 79b] Research Triangle Institute.
Validation Methods for Fault-Tolerant Avionics and Control Systems - Working Group Meeting II, NASA-Langley Research Center, 1979.
NASA CP-2130.

[Rennels 84] David A. Rennels.
Fault-Tolerant Computing - Concepts and Examples.
IEEE Transactions on Computers C-33(12):1116-1129, 1984.

[Schuette, et al. 86]
M.A. Schuette, J.P. Shen, D.P. Siewiorek, and Y.X. Zhu.
Experimental Evaluation of Two Concurrent Error Detection Approaches.
In *16th International Symposium on Fault-Tolerant Computing*, pages 138-143. IEEE, July, 1986.

[Siewiorek & Swarz 82]
Daniel P. Siewiorek and Robert S. Swarz.
The Theory and Practice of Reliable System Design.
Digital Press, 1982.

[Smith & Lala 83]
T. Basil Smith III and Jaynarayan H. Lala.
Development and Evaluation of a Fault-Tolerant Multiprocessor Computer, Vol. I, FTMP Principles of Operations
Charles Stark Draper Laboratories, 1983.
NASA CR-166071.

[Wimmergren 82]
Alan Lee Wimmegren.
Verification of a Fault Tolerant Multi-Processor Architecture.
Master's thesis, Massachusetts Institute of Technology, May, 1982.
CSDL-T-782.

[Yang et al. 85] X.Z. Yang, G. York, W.P. Birmingham, and D.P. Siewiorek.
Fault Recovery of Triplicated Software on the Intel iAPX 432.
In *Distributed Computing Systems*, pages 438-443. May, 1985.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.
NASA CR-178423

2. Government Accession No.

3. Recipient's Catalog No.

4. Title and Subtitle
Software-Implemented Fault Insertion: An FTMP Example

5. Report Date
October 1987

6. Performing Organization Code

7. Author(s)
Edward W. Czeck, Daniel P. Siewiorek, and
Zary Z. Segall

8. Performing Organization Report No.

10. Work Unit No.
505-66-21-01

9. Performing Organization Name and Address
Carnegie-Mellon University
Department of Electrical and Computer Engineering
Pittsburgh, PA 15213

11. Contract or Grant No.
NAG1-190

13. Type of Report and Period Covered
Contractor Report

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes
Langley Technical Monitor: George B. Finelli

16. Abstract

This report presents a model for fault insertion through software; describes its implementation on a fault-tolerant computer, FTMP; presents a summary of fault detection, identification, and reconfiguration data collected with software-implemented fault insertion; and compares the results to hardware fault insertion data.

The experimental results show detection time to be a function of time of insertion and system workload. For the fault detection time, there is no correlation between software-inserted faults and hardware-inserted faults; this is because hardware-inserted faults must manifest as errors before detection, whereas software-inserted faults immediately exercise the error detection mechanisms.

In summary, the software-implemented fault insertion is able to be used as an evaluation technique for the fault-handling capabilities of a system in fault detection, identification, and recovery. Although the software-inserted faults do not map directly to hardware-inserted faults, experiments indicate software-implemented fault insertion is capable of emulating hardware fault insertion, with greater ease and automation.

17. Key Words (Suggested by Author(s))
Fault Insertion
Software-Implemented
Fault Detection
Fault Identification
System Reconfiguration

18. Distribution Statement
Unclassified - Unlimited

Subject Category 62

19. Security Classif. (of this report)
Unclassified

20. Security Classif. (of this page)
Unclassified

21. No. of pages
33

22. Price
A03