The Advanced Avionics Subsystem Technology Demonstration Program

Tim Monaghan, Ghani Kanawati, Jacob Abraham, Daniel Olson, and Ravi Iyer

The Naval Air Warfare Center
Aircraft Division
Warminster, Pa 18974

Abstract

The Navy's Advanced Avionics Subsystem Technology (AAST) Fault Tolerant program is clarifying the Navy's fault tolerant avionics specifications methods and acceptance tests. The goal of the program will be clarify the Specification and Statement of Work language needed in future procurements and to demonstrate fault tolerant validation tools on an avionics design. A set of tool features will then be developed that spans the needs of fault tolerant computer system design from early concept studies to full scale production and operational support, both hardware and software. The paper will give an overview of the AAST Fault Tolerant Demonstration and focus on two tools that are being used in the demonstration: FERRARI - a software fault injector that will be used to validate the fault tolerance of the Common Integrated Processor (CIP), the F-22 Mission Processor and GRIND a concept evaluation tool that will be used to evaluate the overall CIP architecture.

Index Terms: Avionics, Fault tolerance, Fault injection, error injection.

1 Introduction

The DoD is entering a radically new era of systems procurement. The Cold War is over and new scenarios and radical force structure changes are changing the requirements of the next generation of weapon systems. Other factors in the "new world order" of weapon system procurement are: budgets are decreasing, development cycles are being stretched out, prototypes and demonstrations are being emphasized, open systems standards will dominate, and the next generation of systems will be intensively modeled and simulated before any hardware is built. Future complex weapon systems will increasingly rely on digital systems and the dependability of these digital components will play a critical role in the effectiveness of those systems in the field.

The key issue that the AAST Fault Tolerant Demonstration is addressing in this new era of defense procurement is - how can the Navy manage and procure dependable and cost effective, computer-based weapon systems? The demonstration program will investigate the timely and practical application of fault tolerant technology early in the design cycle before major resources are committed to a particular design. This application of fault tolerant technology will be balanced against the extreme time pressures of modern avionics system development.

2 The Advanced Avionics Subsystem Technology Fault Tolerant Demonstration

Funds for the research, development, transition and insertion of new technologies into the fleet are divided into 6.1, 6.2, 6.3A and 6.4 funds. The 6.1 and 6.2 funds are focused on exploring the feasibility of new technologies. The 6.3A funds are aimed at demonstrating those technologies so that program offices can specify them with confidence. In 1990, ONR's 6.1 research started the Ultradependable Multicomputers and Electronic Systems Research Initiative. This research initiative addresses a wide ranging number of fault tolerance topics including measurement and modeling of expected system fault modes, fault injection, simulation and modeling techniques, software fault tolerance approaches, as well as compiler, algorithm and hardware-based fault-tolerance techniques. ONR's 6.2 exploratory development, computer block has an effort called the "Engineering of Complex Systems Technology" whose aim is to explore the entire design and development of advanced real-time systems. The fault tolerance portion of the block plan is aimed at integrating fault tolerance into the design process of complex systems. The Advanced Avionics Subsystem Technology (AAST) Fault Tolerance Demonstration is a 6.3A project that takes the 6.2 Engineering of Complex Systems effort the next step and demonstrates the fault tolerance metrics and acceptance tests at each stage of an evolving contractor's design. The AAST work will also transition some

of the ONR 6.1 developed tools (fault injection, fault tolerance benchmarks and fault tolerance simulation techniques). The goal of the AAST Fault Tolerance Demonstration is to demonstrate the necessary and sufficient dependability metrics and validation techniques of a fault tolerant system. These requirements will be documented so that program offices can use them in their specifications and SOW packages according to their various fault tolerance and dependability needs.

3 Language and Fault Injection Tools

The two key thrusts of the AAST Fault Tolerance Demonstration program are: what legal language needs to be in the Statement of Work (SOW) and Specification to clarify to the contractor the exact fault tolerant and dependability requirements that the government expects to see in the system? And what tools and fault injection techniques are needed to validate that language?

Legal, formal, requirements are the only way the government can define computer performance and dependability requirements. More precise fault tolerant requirements would specify the system's fault containment regions, the specific faults the system will guard against, and the types of analysis and fault injection testing that shall be done at each stage of the system design.

The SOW is the requirements for the contractor design team to fulfill at each stage of the system design. Generally, the SOW should require that the error handling features of the system shall be validated at each stage of the systems evolution. The validation should be a functional fault analysis which will map the specified fault set onto each identified fault containment region and then identify the fault detection, isolation, removal and recovery mechanisms of the system that will enable the system to maintain the mission services in the presence of faults. This validation should be demonstrated with fault injection techniques on the current simulation or breadboard of the evolving design.

This precise legal language, dealing with dependability and fault tolerance, should include clear and quantifiable validation techniques to be performed at each stage of the system's design that will allow Navy to be informed customers able to quantify a design's dependability and reasonably ensure that the evolving system will be a dependable system for the Navy to own and operate. Thus, the Navy needs to support the development of fault tolerant validation tools. The following two sections describe two of the tools to be used in the laboratory demonstration of the AAST Fault Tolerant program.

4 Fault Injection Tools Supporting the AAST Fault Tolerant Demonstration

The two tools that are used in supporting the AAST Fault Tolerant Demonstration. The first is **DEPEND**, which is a simulation environment tool that is used in the design phase of a system before an actual hardware has been built. The second is **FERRARI**, which is a fault and error injection tool that injects errors into a prototype of a system to measure its ability to detect, locate and recover from errors while it is executing real application.

5 DEPEND/GRIND Overview

Commercial systems for air, ground, and space applications require innovative solutions to dependability problems. To meet these needs, we have developed a highly instrumented, simulation-based CAD environment, called **DEPEND**, which allows designers to study a system in detail. The CAD tool provides an object-oriented framework that allows the evaluation of highly dependable systems. The tool provides facilities to rapidly model components typically found in fault-tolerant systems. It provides an extensive, automated fault injection facility which can simulate realistic fault scenarios. For example, the tool can inject correlated and latent errors, and it can vary the injection rate based on the workload on the system. In addition, it provides several key features that are necessary for fault simulation:

1. It provides ways to signal a change in the status of the components due to a failure, so that remedial actions can be simulated.
2. It provides mechanisms to halt on-going processes due to faults/errors/failures. This is an extremely needed feature for fault simulations. It is also useful for incorporating importance sampling methods.
3. It has the capability to model the inter-component dependencies under fault conditions. For example, a failed server may not be able to initiate re-integration without control from a healthy control server. Such dependencies can be easily modeled with **DEPEND**.
4. It provides several automatic fault statistics collection facilities that can provide measures such as MTTF and availability. They can also provide a detailed list of every fault injected, repair action attempted and their status.

DEPEND is a powerful tool capable of modeling complex systems, but using it may be difficult for those who are new to the tool or who are unfamiliar with C++. Though the object library reduces the amount of programming that the user has to perform, models often turn out to be hundreds of lines of code. **GRIND**, a **GR**aphical **I**nterface for **DEPEND**, provides an alternative to coding C++ directly. **GRIND**

Figure 1: Common Integrated Processor block diagram.

is a menu-driven X-Windows application which facilitates the creation of DEPEND models. With this interface, one is able to visualize the architecture of the system being modeled and how it functions. Hardware components are represented using icons while the software aspects of the system are specified using a graphical flow-chart representation. Development of models can also be performed quicker because GRIND's menu structure presents most of DEPEND's features to the user so that less time is spent referring to the manual and debugging typos. While a graphical interface provides a quicker and more intuitive way of entering models, much of DEPEND's power can not be harnessed graphically, which means that direct C++ coding must be used to create especially complex models. However, since GRIND's output is a file containing a well-formatted C++ program, one can speed up the creation of a complex model by first using GRIND to create a simpler, more abstract model, and then extending it by jumping directly into the C++ code generated by GRIND.

5.1 Example Application

As an example application of GRIND, this section will present a model of a system similar to a processing module found in Hughes Common Integrated Processor (CIP). The system is a fault-tolerant element consisting of four processing elements (PEs), two redundant network interfaces (NIs), a global memory (GM), and a control processor (CP).

Refer to Figure 1. Either NI can be used by any of the PEs in order to send data to or get data from the outside world. We will assume that one NI has sufficient bandwidth to support the system, so that if one fails the system can continue to function. The CP is responsible for distributing tasks among the four PEs which communicate with each other using the GM. The reason for having multiple processing elements is for fault-tolerance as well as for increasing computing power. Let's say that three of the four processors are needed to maintain the minimum throughput require-

ments. Since tasks are likely to be running on a PE when it fails, the process of reconfiguring to use only three PEs is likely to be complex and itself prone to failure. Thus, the model will include a reconfiguration coverage for the processing elements. Given the failure rates of each of the subcomponents, an interesting analysis would be to see how sensitive the reliability of the module as a whole is to this reconfiguration coverage. This would give engineers an idea of how much effort needs to be invested in designing a robust reconfiguration process.

Constructing this model using GRIND was a straightforward process. The first step was to create a derived class for each of the different types of subcomponents in the model. GRIND allows the user to create derived objects from the classes within the DEPEND object library so that more specialized functionality can be added to the default objects. Once a derived class is created, one can create variables and methods for that class in addition to those inherited from the parent class. In this example model, a PE class was derived from the FT_kofn object. Because FT_kofn is the parent of PE, PE inherits all of FT_kofn's functionality making it able to model the processing-element k-out-of-n system. Similarly, a GM class was derived from a FT_memory object, as well as a CP class from a FT_server2 object and a NI class from a FT_link2 object. Since no workload (such as processor utilization, message passing, memory access, etc.) is incorporated into this model, there was no need to further specialize the derived classes. The derived classes were added here to demonstrate that this model can be readily extended within the GRIND environment.

Figure 2: GRIND hardware display showing the initializations methods for the processing elements.

Once the derived classes were established and objects from these classes were added to the model,

the initialization methods of these objects were set through the *Set Inits* menu. Through this menu, the fault injection rates were set as well as some of the configuration parameters. Through these 'configuration parameters', we specified that the *proc_elem* object was to be 3-out-of-4 system with a 96% reconfiguration coverage and the *net_interface* object was to have one spare.

See Figure 2 for a GRIND display showing the objects of the model and a listing of the initialization methods of *proc_elem*, an object of the class PE. The GRIND environment also allowed us to specify that the executable was to run the simulation one thousand times, outputting the time-to-failure after each run. With this information, GRIND was able to create a C++ program which could be compiled and then run. The output of the executable is simply a listing of one thousand times to failure; so with the aid of a standard statistical analysis package, one could generate a curve estimating the reliability of the system. The parameter for the reconfiguration coverage can easily be changed within GRIND so that multiple scenarios may be simulated and a sensitivity analysis can be performed.

The next tool that will be discussed is FERRARI.

6 Design and Implementation of FERRARI For the Hughes Common Integrated Processor (CIP) Module

Fault and error injection has been recognized as a powerful technique which allows the evaluation of a prototype system under faults, in particular, the measurement of the effectiveness of its error detection and correction capabilities. Another advantage of this technique is that the effects of faults in the system can be studied when it is executing realistic programs.

Hardware and software techniques have been proposed for fault injection. The motivation behind our work was the development of a flexible and an automated fault and error injection system. We concluded that hardware fault injection would be cumbersome and would not allow us to inject faults and errors inside chips, for example, change a bit or bits in an internal register of a processor. On the other hand, it was clear that simulation would be too time consuming. Our approach, therefore, is to emulate hardware faults and errors through software, by corrupting the program execution state while it is executing, so that the behavior of the system would be the same as if the internal fault had been present.

Our studies showed that the behavior of a system varies with the type of faults and errors injected. (This is described in more detail in Section 8.) We also wanted the ability to inject faults while executing a variety of applications or system functions. The injection techniques described in this paper provide the necessary flexibility.

These techniques have been incorporated into FERRARI a Fault and Error Automatic Real-time Injector. The main contributions of FERRARI are its ability to inject transient errors as well as permanent faults so that it can be used to test the effectiveness

of concurrent error detection and correction mechanisms, and its capability to perform the injection on object code. The current version of FERRARI is implemented to emulate a large number of faults and errors in the CPU circuitry, in memory, and in peripheral drivers.¹ Hardware faults as well as control flow errors (including bus errors, memory errors and processor control line errors) are emulated through software. FERRARI allows control over the time, location, type and duration of the fault or error. It can measure coverage and latency (in instruction cycles or microseconds) and is able to locate the source of a detected error or one which caused failure. It is able to automatically control a large number of experimental runs.

The current version of the CIP module was not built to be fault tolerant. Our aim in injecting fault to this module is not to evaluate its dependability properties but rather to test the capability of FERRARI to inject faults and errors on the hardware prototype. This will give us the ability to add new features to FERRARI that will be used on future versions of the CIP module which are built to be fault tolerant.

Figure 3 depicts the hardware configuration of the fault and error injection process. In this figure, a microvax workstation is connected to the CIP through the high speed data link. The microvax is configured as the host machine where the fault and error tool is executing.

The CIP contains the general purpose processing elements (GPPE's), the special signal processing elements (SPE's), and the global memory. The GPPE's execute the ADA applications while the SPE's execute the signal processing applications. Communication between the GPPE's and the SPE's is attained through the global memory shown in Figure 3. Each GPPE's has its own local memory which contains the program that is running on that GPPE and the data needed for the successful execution of the program. The "interface module" is a special purpose hardware that controls the communication between the microvax and the CIP module.

As mentioned earlier, the fault and error injection tool (FERRARI) is executing on the host microvax workstation which is running the vms operating system. The procedure for injecting errors into the CIP module is depicted in Figure 4. The "user console" is a program that can access the low level functionality of each component residing on the CIP module. It can insert software breakpoints, access any of the internal registers for any of the GPPE's, and read/write into the local memory of the GPPE's. FERRARI can inject fault and ERRORS in the CIP modules by sending a sequence of commands to the "user console" program that is running concurrently with FERRARI. This communication is attained through the use of mailboxes². After receiving the sequence of commands from FERRARI, the "user console" pro-

¹More rigorous studies to obtain the actual high level fault/error models of a particular processor should be conducted prior to using FERRARI. An example is the study undertaken in [5] where a technique for mapping real hardware failures to high level error models is presented.

²Mailboxes are used to send streams of data from one process to another.

Figure 3: Hardware configuration of error injection on the CIP module

gram executes this sequence one command at a time by sending the command through the high speed link to the CIP module as shown in Figure 4. The operating system running on the CIP module will interrupt and modify the running application accordingly.

Figure 4: Procedure of error injection on the CIP module

At the end of the fault injection experiment, the CIP module operating system will send error messages through the high speed link to the "user console" program running on the microvax workstation which, in turn, transfer these messages to FERRARI. This error message contains information that describes the execution state of the application running on the CIP module when the error was injected. FERRARI uses this information to evaluate the final coverage of the system.

6.1 Initialization and Activation Module

This module prepares the test program³ for fault and error injection. The tasks of this module are: 1) it parses the test program executable file to determine the starting address location and size of the text and data segments of the file, and 2) it extracts the execution behavior of an error free run of the test program. Extracted information includes the output of the program, referred to later as *reference*, execution time, and the address space traversed during program execution.

6.2 User Information Module

This module obtains experiment parameters supplied by the user which include: 1) the type of the dependability measurement (coverage vs. latency), 2) the duration of the fault/error (transient error vs. permanent fault), 3) the mode of the experiment which selects between user vs. random selection of the address bit position and time instance at which the fault/error is injected, 4) the fault/error type which selects one of the five types supported by FERRARI (these are XORing a bit with logic "1", resetting a bit, setting a bit, resetting a byte, and setting a byte), 5) the fault and error model (explained in the next subsection), 6) the method of fault/error injection (also explained in the next subsection).

6.3 Fault and Error Injection Module

FERRARI supports the injection of both permanent faults and transient errors. The mechanisms for fault and error injection are identical, and the only difference is the duration of the injected fault and error. For transient error injection, the duration is defined to be one instruction cycle. On the other hand, the duration of permanent faults may be several instruction cycles, or may span the entire execution interval of the application.

One of the design features of FERRARI is its capability to inject a variety of fault/error models. This feature was established when we observed the target system responding differently when different error models were injected in the system. (An example of this behavior is shown in Figure 13.) In addition, the design of FERRARI allows the inclusion of other models to its set of fault/error models.

A detailed discussion of the mechanism of some of the transient errors and permanent faults that are injected through FERRARI follows in the next two subsections.

6.3.1 Transient Errors

FERRARI supports three methods of transient fault/error injection, Memory Corruption, Spatial and Temporal. In the "Memory Corruption" method, a fault is injected in the task memory image before program execution starts. In the "Spatial" method, the

³We refer to the program whose code is mutated by FERRARI during the course of the fault/error injection process as the "test program".

fault/error injection is triggered after N occurrences of a randomly selected address line. The value of N is defined by the user. Once the program uses the erroneous value, the error is removed. Finally, in the "Temporal" method, the execution of an application is interrupted at a randomly selected instant of time and the value of the next element fetched or stored into memory is modified. An experiment to compare the three different methods will be presented later in Section 8.

Table 1: Selected Transient Error Models

Model	Description
1	AddIF address line error resulting in executing a different instruction
2	AddIF2 address line error resulting in executing two instructions
3	AddOF address line error when a data operand is fetched
4	AddOS address line error when an operand is stored
5	DataIF data line error when an opcode is fetched
6	DataOF data line error when an operand is loaded
7	DataOS data line error when an operand is stored
8	CndCR errors in condition code flags

Some of the transient errors supported by FERRARI are presented in Table 1 and are emulated as follows. When the execution reaches a specified address, the program is trapped. A selected error is injected and the current instruction is executed. The injected error is then removed and the program is allowed to resume execution. The reason for this procedure is to avoid injecting the error more than once if the selected address was in a loop. Of course, if the single execution of the instruction under the error resulted in a change of internal state, this erroneous state would remain, and may cause other execution errors subsequently. This section will present the mechanism for injecting two of the error models presented in Table 1, the AddIF and the AddOF. The mechanisms for injecting the other error models in Table 1 were presented in [1].

- Address line error while the processor is fetching an instruction

Figure 5 illustrates the mechanism to inject this error. The processor is interrupted when it reaches the selected address to be modified. The next instruction to be executed is fetched from the address pointed to by the current program counter having one of its bits (bytes) modified. After the execution of the wrong instruction, the program is trapped on the following instruction. The previous program counter value is restored before the program is allowed to proceed.

- Address line error when the program is fetching an operand

When execution reaches the address where the error is to be injected, the program is trapped (Figure 6). In SPARC machines, for example, only *load* and *store* instructions access memory.

Figure 5: Address line error when an instruction is fetched

All the other instructions use registers as their source and destination operands. The load and store instructions use registers to access memory. The effective address of the operand, for the selected load/store instruction, is modified. After the execution of the faulty instruction, the program is trapped again to restore the content of the previous program counter.

Figure 6: Transient address line error when an operand is fetched

In addition to the error models listed in Table 1, FERRARI allows the user to mutate the contents of an internal register. In addition, the user may select a combination of some of the transient errors in the Table 1. For example, it was found in [5] that a significant percentage of the injected faults inside a sample processor are manifested as "address and data line errors while fetching an operand".

6.3.2 Permanent Faults

Permanent faults supported by FERRARI are: 1) address line fault; 2) data line fault; 3) fault in condition code flags. These faults are emulated as follows:

- Address line fault

When the program execution reaches the address of the selected instruction, a bit/byte in the program counter is modified. The instruction at the modified address is executed. If the executed instruction is a branch instruction, the value of the program counter becomes the target address of the branch instruction, otherwise its value is the previous program counter, before fault injection, incremented by four, Figure 7. This procedure is

repeated N times where N is a number that determines the duration of the fault in instruction cycles. If any of the executed instructions accesses memory (load/store instructions), the effective address of the operand may be modified in the same bit/byte position used to mutate the address of the executed instruction.

Figure 7: Mechanism for permanent address fault injection

- Data line fault

When the program execution reaches the address of the selected instruction, the processor is interrupted. The value of the instruction at that address is modified. The program is then trapped N times where N is a number that determines the duration of the fault in instruction cycles. If any of the executed instructions accesses memory (load/store instructions), the operand values may be modified in the same bit/byte position used to mutate the executed instruction.

- Fault in condition code flags

This fault emulates a stuck-at fault in the condition code register. When the program execution reaches the address of the selected instruction, the selected condition code bit is modified. The program is then trapped N times where N is a number that determines the duration of the error in instruction cycles.

6.4 Data Collection and Analysis Module

This module measures and records the response of the system for every injected fault/error. For each run, the location of the fault/error (virtual address), the affected bit, and the affected register, if any, are recorded. Terminating conditions for every run are also appended to the logfile. Terminating conditions indicate whether the resulting error: 1) was dormant (did not lead to a failure and did not produce wrong output during the lifetime of program execution), 2) had led to a failure (the test program either produced wrong output, or was terminated after it timed-out, or 3) was detected. Terminating conditions that result in detected errors are due to: 1) executing "exit(cause)" statements in the test program code, where *cause* is a value that either indicates the nature of an error detected by a user detection mechanism, or signals the end of the program execution, 2) an error triggering one of the built-in error detection mechanisms of the system. Note that when the test program terminates abnormally, it returns to its parent (in this case the fault/error injection process) a flag indicating the nature of the error that caused the system to abort the execution of the test program.

The data analysis module also records the identity of the error detection mechanism and the error detection latency if the error was detected. Finally, a flag used to indicate the results of the *compare* function utilized to compare the output produced by the current run and *reference* (explained in Section 6.1) is appended. The function *compare* utilized in all the experiments presented in Section 8 uses the UNIX *cmp* function that performs a byte-by-byte comparison of two files. The user may decide to utilize another *compare* function more suitable to the test program. For example, for non-deterministic applications, *compare* would check whether the difference between corresponding elements from two files is within a specified limit. FERRARI may also be used to test systems that utilizes spatial redundancy techniques. For example in a TMR system, FERRARI can inject faults/errors into one of the TMR modules and the output of the voter is considered to be the *compare* function output.

At the end of the experiment, the collection and analysis module collects these results along with the associated status flags and calculates percentages with respect to coverage, latency, and type of error detection mechanism for each experiment.

7 Experiment Description

Experiments presented in this paper were conducted on SUN SPARC workstations running SUNOS 4.1. These experiments provide an insight of the type of faults and errors that can be injected into the CIP module. These experiments were selected to demonstrate the capabilities of FERRARI, as well as to study the behavior of the target system when injected with faults and errors. In addition, a variety of faults and errors were injected to measure and compare the effectiveness of several of the error detection and correction techniques that are either built into the operating system or are embedded into the test programs.

For every experiment, the user selects the fault type and fault model to be injected and 6.4) injection runs. In each run, the bit position(s), the selected register to be faulted (if any), and either the location (address inside the program code, including library codes) or the instance at which a fault/error is injected⁴ were randomly selected.

Results for over one million runs are presented in this paper. The criterion adopted when selecting the number of runs per experiment was based on obtaining a consistent average behavioral response of the system (e.g. percentages of "No Error", "Undetected Errors", and "Error Coverage" including the distribution of the contribution of each of the error detection mechanisms). For some of the conducted experiments, the response of the system became consistent at 10,000 runs, while for others, the behavior of the system became consistent at 20,000 runs.

The guideline followed when selecting test programs was to maintain an automated injection environment while making fault/error injection runs. This feature resulted in conducting a large number of runs for each experiment, thus providing confidence in the measurement of the response of the system. Another factor considered when selecting test programs was to evaluate several of the concurrent error detection and correction techniques that were embedded at the application-level code. As a result, test programs used in our experiments were application-level programs.

An advantage for injecting faults/errors in the system at this level is that many of these faults/errors generate traps (error conditions and exceptions) which are later detected by the built-in error detection mechanisms of the system (e.g. detecting an "illegal instruction"). Once these errors are detected, the system aborts the execution of the program and returns to its parent shell, which is in this case FERRARI.

When those same errors, on the other hand, were trapped while the system is executing in the supervisor (kernel) mode⁵, the processor enters either an endless "wait" state, or a "diagnostic" state⁶. In both cases automating the fault/error injection process becomes impossible since the system has to be reset manually in order to continue the fault/error injection experiments. The study in [6] has shown that injecting faults/errors in the system while executing in either mode generates the same traps but differs in the action taken once the error is detected. Note that injecting the system with faults/errors while it is running in the kernel mode is accomplished by running FERRARI as a daemon process⁷ in the supervisor mode. As a result, FERRARI will have access to supervisor allocation tables and would thus be able to modify the processor state while it is executing operating system code.

The procedure of fault/error injection in FERRARI

⁴during the course of execution of the test program

⁵The SPARC processor supports multi-tasking which requires the processor to be able to operate in two modes: a supervisor and a user mode.

⁶In this mode, the system dumps all its memory to a swap area and later reboots the machine.

⁷In Unix, a daemon process is one which is running silently until awakened by a request.

is shown in Figure 8. In each experiment conducted, a selected application is first run without injecting an error in the system. The output, named *reference*, is written to a file for future comparisons. A fault or error is injected into the system while running the application. If no error detection mechanism is triggered, the output of the current run is compared to *reference*. A difference between the two outputs indicates an error has resulted in a wrong output and that the error was not detected by any of the error detection mechanisms. This would contribute to the lack of coverage of the mechanism.

Figure 8: Fault/error injection procedure

The following is a list of the test applications utilized in our experiments.

1. Matrix multiplication using checksums [4].
2. Quicksort with assertions [9].
3. Robust data structures applied to modular robust binary (MRB) trees [7].

8 Empirical Results

In this section we present the results of fault/error injecting SUN SPARC1 workstations while running the above applications. Both permanent faults and transient errors were injected in our extensive studies. The results of the experiments presented in this paper, however, concentrate on transient errors, since the results were more interesting, and since we found that errors due to faults which were active for more

than a few instruction cycles were very likely to be detected by one of the error detection mechanisms. This is consistent with previous work done on permanent faults [10].

The first experiment was designed to evaluate the coverage of error detection mechanisms under different types of fault and errors. The next experiment attempted to determine the percentage of injected errors which remained latent for a particular application. The coverage sensitivity to different error models and the effect of system error detection mechanisms was studied in the next two experiments. Finally, the tradeoffs between error detection capability and performance overhead were measured.

8.1 Effect of different fault/error injection methods

Experiments were conducted to compare three different methods of fault and error injection. These are: 1) corrupting the task memory image, 2) spatial transient error injection, 3) and temporal transient error injection. In these experiments we injected over 15,000 errors in the system while running the quicksort application for two different data sizes. The purpose of these experiments was to study the variation of error coverage with data size when using these three injection techniques. Figure 9 presents coverages for the three experiments. In the first experiment, a fault was injected in the task memory image before program execution started. This was referred to previously as the "Memory Corruption" method. The injected fault, as explained earlier, remained throughout the execution sequence of the program and the faulty value was potentially used many times. This is referred to as a "memory error" in the figure. In the second experiment, an address line was randomly selected before program execution started. Once the program used the erroneous value (in this experiment a data line error), the error was removed. This is referred to as a "spatial transient error" in the figure. The "temporal transient error" injection method was used in the third experiment. In this experiment, the error injection was accomplished by interrupting the execution of an application at a randomly selected instant of time, and changing the value of the next element fetched from memory.

Results presented in Figure 9 demonstrate the variability of the error coverage of the detection mechanisms under different injection methods. From these results we deduce the following.

- Errors injected in the task memory image exhibited the highest coverages since the error, as argued before, was exercised more than once if the corrupted instruction was in a loop.
- The coverage was insensitive to the size of the sorted data when faulting the memory image of the test program and when using the spatial injection method. The coverage, on the other hand, increased as the size of the sorted elements increased from 100 to 1000 elements for the temporal error injection method.

Figure 9: Error detection for quicksort application on different data with different injection methods

This behavior becomes clear when one realizes that the quicksort application was embedded with “assertions”, a data value error detection mechanism. The quicksort application is composed of three segments: 1) initialization, 2) data loop, and 3) file closing. In the “data loop” segment, the program reads and sorts data elements in ascending order. When the size of the sorted elements increases, the execution time inside only the data loop segment is increased whereas the execution time inside the other two segments is not affected. As a result, the probability that a data error is injected inside the loop and consequently is detected by the “assertion” mechanism increases when the “temporal” injection method is employed. This behavior, however, was not observed in the other two injection methods. In those methods, the data error is injected at the first incident when the value of the “program counter” matches the value of the pre-selected address location, including any address inside the data loop segment. Consequently, the response of the system for the memory corruption and spatial injection methods is independent of the execution time of the application. Therefore, the temporal injection method would challenge the error detection mechanisms to a higher degree.

8.2 Effect of latent errors

Figure 10 shows the result for the matrix multiplication application using the checksum technique for error detection. Errors from every error model shown in Table 1 were injected into the system while running the matrix multiplication of two matrices each has 20 by 20 elements. For this experiment, we first injected the transient errors at all legal addresses traversed during the course of execution of the test program. Later, we utilized the pseudo-random number generator to select the address at which an error will be injected. We started with 1000 runs and kept incrementing the number of runs by 1000. After 10,000 runs, the system exhibited the same behavior (measured in terms of the distribution of the responses of the system, as explained later) to that observed when the test program was injected exhaustively.

Figure 10: Distribution of errors for matrix multiplication application

A large percentage(41%) of the injected transient errors were latent (“No Error”) and did not affect the program output. Latent errors include erroneous values that are overwritten before they are used in subsequent computations, and errors in the condition code that do not affect the expected course of execution of the test program. In Figure 11, we list a segment of

code, shown in both C++ and the corresponding low-level SPARC assembly language, mutated to emulate the injection of an error. In the C++ code, the algorithm checks whether the value of the variable “argc” is less than 4. In this example, the injected error modified the “immediate” operand of the instruction to the value of zero. When the “cmp” instruction is executed, the value of the internal register “O0” was greater than zero and the program branched to location “main + 120”. Thus the injected fault did not cause the program to deviate from its normal execution.

```
/* C++ code */
if(argc < 4){
    cout << "usage <qsort> <array_size> <input_file> <output_file>";
    exit(user_exit_code);
}

/* assembly code before the error was injected */
0x2ad8 <main +68>: ld [%fp + 0x44], %o0
0x2adc <main +72>: cmp %o0, 3      before error was injected
0x2ae0 <main +76>: bg 0x2b0c      <main +120>
0x2ae4 <main +80>: nop
0x2ae8 <main +84>: sethi %hi(0x18000), %o1
0x2aec <main +88>: or 0x1ec, %o1, %o0
0x2af0 <main +92>: sethi %hi(0x2000), %o2
0x2af4 <main +96>: or 0x0x2c0, %o2, %o1
0x2af8 <main +100>: call 0x3108 <op$alshift__7ostreamPCc>
0x2afc <main +104>: nop
0x2b00 <main +108>: ld [%fp + 0x48], %o0
0x2b04 <main +112>: call 0x501c <exit>
0x2b08 <main +116>: nop
0x2b0c <main +120>:

/* assembly code with the error injected */
0x2ad8 <main +68>: ld [%fp + 0x44], %o0
0x2adc <main +72>: cmp %o0, 0
0x2ae0 <main +76>: ...
```

Figure 11: An example of a latent error

43% of the errors were detected by the built-in error detection and protection mechanisms in the SPARC system. These mechanisms, which are triggered before any application-level error detection techniques, trap illegal instructions, bus errors, segmentation faults, bad system calls, interrupts, arithmetic exceptions, etc. In Section 8.4 we present the contribution of these built-in error detection mechanisms.

Of the remaining errors, 5.5% were detected by the checksum technique and 3.0% by program exit conditions (“Prog Exit” in the figure). Program exit conditions were features added to increase program robustness, such as checking the status of I/O operations when opening and closing files, and were found to enhance the error detection capabilities of the system. The “Time Out” value in Figure 10 and in all other figures indicates the percentage of errors which caused the system to enter a *wait* state. This case was observed when the injected fault caused the program to start executing a system *wait* call. Apparently the arguments passed to this system call (*wait*) were set to produce an infinite loop. For such cases, a timer was used to abort program execution. In this particular application, 6.9% of the errors were not detected (“Undetected Error”), and produced incorrect results. *Undetected Errors* are errors that are not detected by

the application program error detection mechanisms, yet they do not cause the execution of the application to terminate prematurely. The nature of such errors and techniques to prevent them will be described later in the paper.

As shown in Figure 12, the response of the system is confined to only a few categories (*Detected by the system, No Error, Undetected Error, Checksum or User, Prog Exit, and Time out*). These categories are not limited to the SPARC system; they are similar to others reported in the literature. Results from other studies have also confined the responses of the targeted system to only few categories based on service outages [3], statistical distribution of responses [8], or error manifestation of fault injection experiments [2]. Note, however, the system responses obtained through our experiments do not include the category "system crash", which is a very common error manifestation reported in other studies. As argued previously, we decided to avoid system crashes (consequently the time delay to reboot the machine once it crashes), by fault/error injecting while the processor is in the "user" mode.

In order to concentrate on detectable errors, the rest of the figures in this paper show the coverages of errors when latent errors are excluded from the calculations. Figure 12 shows the distribution of coverages for matrix multiplication using checksum only when excluding the latent errors.

Figure 12: Percentages of errors detected in the matrix multiplication application when latent errors are excluded from the calculations

8.3 Coverage sensitivity to different error models

The purpose of this experiment was to study the response of the system when injected with faults/errors that are representative of failures in the real hardware⁸[5]. Figure 13 shows the system behavior when transient errors presented in Table 1 were injected into the system while running the MRB tree test program.

Over 60% of the detected errors were trapped by the built-in system error detection mechanisms. In Figure 13, the highest coverage was obtained when

⁸As mentioned earlier, injecting faults/errors that emulate hardware failures and are 100% accurate requires conducting a study to map low level hardware failures to these fault/error models.

Figure 13: Error coverage of the MRB tree as a function of the injected transient error model(10,000 runs).

address line errors were injected while loading/storing operands (AddOF and AddOS) where the system detection techniques (elaborated in Section 8.4), the assertion technique, and other program robustness techniques have contributed to the overall coverage. The lowest coverage was obtained when data line errors were injected (models DataIF, DataOF, and DataOS). Note that although the MRB tree error detection mechanism is a data value checking technique, it was still effective for other errors caused by the injected errors. The effectiveness of the MRB tree error detection, however, was the highest when errors were injected into the data bus when operands were either fetched or stored (error models DataOF and DataOS).

8.4 Effect of system error detection mechanisms

The SPARC system has built-in error detection mechanisms that monitor address, data and control buses, Figure 14. In this experiment, the system was injected with the eight different error models (see Table 1), while running the matrix multiplication application using checksums. These coverages were found to be comparable to those obtained with the quicksort and robust data structures applications. In this figure, most of the illegal instructions produced were caused by injecting transient errors from models AddIF, AddIF2, DataIF (when the processor is fetching different instructions) and model CndCR. Segmentation faults were triggered by memory access exceptions which occur when a data memory access or an instruction prefetch fails to complete normally. Although this response is anticipated when address line errors were injected, it is less intuitive when data errors were injected (models DataOF, DataOS). Further analysis has shown that the majority of these errors were modifying operand values that specify the memory address of a data element directly or indirectly (an example is the case when the contents of the destination register for a "load" instruction is mutated). In Figure 14, "Bus error" is due to a memory misaligned access which occurs when a load, store, or exchange

instruction attempts to access a memory address that is not consistent with the size of the access. An example of a misaligned access is when a half-word access is attempted to an odd byte address.

Figure 14: Percentage of errors that were detected by the SPARC built-in error detection mechanisms. "others" = hangup, bad system calls, arithmetic exceptions (20000 runs).

8.5 Measurement of error detection latency

Figure 15 shows the average user, system, and overall error detection latency for the quicksort application. In this figure, latency is shown for selected transient error models presented in Table 1. The sorted list for this experiment consisted of ten integer elements. As shown in the Figure, the latency for the user detection mechanism was several thousands of instruction execution time compared to a few hundreds for the system error detection mechanisms. The reason is that user assertions (elements are monotonically increasing) are applied near the end of the execution of the program, whereas the system error detection mechanisms are triggered at every instruction. A timeout interval at the beginning of the experiment was set to 5000 instruction cycles. If an error was not detected during this interval, it was labeled as an "Undetected Error".

As shown in the figure, latencies for address line errors were in general smaller than those for other injected error models. This behavior is expected since most of the injected address line errors (80%), as was shown in the previous experiment, were detected by the built-in system error detection mechanisms such as "segmentation faults", "illegal instructions", and "bus error". The system error detection mechanisms trap these illegal conditions in one or few instruction cycles. Note that since the *quicksort* application occupies a small address space, the probability of inducing a "segmentation fault" exception increases, and is detected by the memory management unit in the few cycles. Similar arguments holds for "bus error" and "illegal instruction" exceptions. Also note in the figure that the latency for AddIF2 model is smaller than that for AddIF since more than one bit of the instruction address is mutated in the AddIF2 error model.

Regarding data line errors when operands are accessed, DataOS has higher latency than that of DataOF, as shown in Figure 15. A principal reason for this behavior is that some of the corrupted data operands are used in the address calculation of other operands. For the DataOF model, these corrupted data are immediately used to fetch other data, whereas for the DataOS model, some of these corrupted data are only used at a later stage during the course of execution of the test program, depending on the control flow of the program.

Figure 15: Error detection latencies for matrix multiplication utilizing checksum techniques (10,000 runs).

9 Conclusion

This paper outlined the Navy's AAST Fault Tolerant Demonstration program and described two tools being used in the demonstration. This program will clarify the precise legal language needed in future Specifications and SOW of complex, computer based weapon systems and what validation techniques are needed to support that contract language.

During the Gulf War, Naval aircraft sustained high Full Mission Capable (FMC) rates (in the high 90% range), but these high FMC rates were sustained at the cost of high Maintenance Man Hours per Flight Hour (MMH/FH), (30 to 65 MMH/FH), high spares usage and high false alarm rates, (consistently in the 30% to 35% range). Future complex weapon systems will increasingly rely on digital sub-systems and the dependability of these digital designs will play a critical role in the effectiveness of those systems in the field. In future combat situations, the rate of spares usage, the time to isolate and repair the equipment, and the false alarm rates will play a critical role in the effectiveness of those systems. As budgets decrease, fewer systems are purchased and commercial parts are used, the dependability of those systems will be a critical factor in the effectiveness of the next generation of computer based combat systems.

References

- [1] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," *Proc. 22th Int. Symp. Fault Tolerant Comput.*, Boston, Massachusetts, July 1992, pp. 336-345.
- [2] J. Barton, E. Czeck, Z. Segall, and D. Siewiorek, "Fault Injection Experiments Using FIAT," *IEEE Trans. on Comput.*, vol. 39, no. 4, April 1990, pp. 575-582.
- [3] R. Chillarege and N. Bowen, "Understanding Large Systems Failures - A Fault Injection Experiment," in *Proc. 19th Int. Symp. Fault-Tolerant Comput.*, Chicago, IL, June 1989, pp. 356-363.
- [4] K. Hua and J. Abraham, "Design of Systems with Concurrent Error Detection Using Software Redundancy", *Proceedings, ACM/IEEE Fall Joint Computer Conference*, Dallas, Texas, November 1986, pp. 826-834.
- [5] G. Kanawati, N. Kanawati, and J. Abraham, "EMAX: A High Level Error Models Automatic EXtractor," AIAA 93, San Diego, California, October 19-21, 1993, pp. 1297-1306.
- [6] N. Kanawati, G. Kanawati, "A Case for Extending FERRARI to Include Hardware-Based Fault/Error Injection," Technical Report, University of Texas at Austin, JAA-01-94, 1994.
- [7] N. A. Kanawati, G. A. Kanawati, and J. A. Abraham, "A Modular Robust Binary Tree," *4th IFIP Working Conference on Dependable Computing For Critical Applications*, San Diego, Californian, January 1994.
- [8] G. Finelli, "Characterization of Fault Recovery Through Fault Injection on FTMP," *IEEE Trans. Reliability*, vol. R-36, June 1987, pp. 164-170.
- [9] D. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," *Proc. 9th Int'l Symp. on Fault-Tolerant Comput.*, June 1979, pp. 102-105.
- [10] P. Banerjee, J. Rahmeh, C. Stunkel, V. Nair, K. Roy, V. Balasubramanian, and J. Abraham, "Algorithm-Based Fault Tolerant on a Hypercube Multiprocessor," *IEEE Trans. on Computers.*, vol. 39, no. 9, September 1990, pp. 1132-1145.