DOCTOR: An IntegrateD SOftware Fault InjeCTiOn EnviRonment

Seungjae Han, Harold A. Rosenberg, and Kang G. Shin

Real-Time Computing Laboratory
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109–2122.
email: {sjhan, rosen, kgshin}@eecs.umich.edu

ABSTRACT

This paper presents an integrateD sOftware fault injeCTiOn enviRonment (DOCTOR) which is capable of injecting various types of faults with different options, automatically collecting performance and dependability data, and generating synthetic workloads under which system dependability is evaulated. A comprehensive graphical user interface is also provided. A special emphasis is given to the portability of this dependability experiment tool set. The fault-injection tool supports three types of faults: processor faults, memory faults, and communication faults. It also allows for injecting permanent, transient or intermittent faults. The proposed design methodology for DOCTOR has been implemented on a distributed real-time system called HARTS [1], and its capability is demonstrated through numerous experiments. Dependability measures, such as detection coverage & latency and the associated performance overhead, are evaluated through extensive experiments. Communication fault injection is used to evaluate a probabilistic distributed diagnosis algorithm. The results show that the algorithm performs better than its predicted worst case, yet is quite sensitive to various coverage and inter-processor test parameters.

Index Terms — Fault injection, communication faults, error-detection latency & coverage, synthetic workloads, dependability and performance monitor, validation and evaluation of fault-tolerance mechanisms, distributed real-time systems, dependable communications, distributed diagnosis.

The work reported here is supported in part by the Office of Naval Research under Grants N00014-91-J-1115 and N00014-92-1080, the National Aeronautic and Space Administration under Grant NAG-1493, and the National Science Foundation under Grants MIP-9012549 and MIP-9203895. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

1 Introduction

One of the major problems in developing a dependable system is how to evaluate its dependability. Generally, there are several attributes of dependability, such as reliability, availability, maintainability, and performance-related measures (performability) [2]. Numerous approaches have been proposed to validate and evaluate system dependability, including formal proofs, analytical modeling, and experimental methods.

As computer systems become more complex, evaluation by modeling may become intractable. Dependability evaluation with fault-injection experiments has therefore become a popular alternative. This approach is aimed at accelerating the occurrence of faults in order to assess the effectiveness of fault-tolerance mechanisms while executing realistic programs on the target system.

There have been several methods proposed for hardware fault injection, such as pin-level fault injection [3, 4, 5], and heavy-ion radiation [6]. However, these methods have become increasingly more difficult to use, mainly due to the complexity of contemporary computer architectures and the high-degree of integration of functions into an ever-shrinking VLSI chip. As a result, simulation approaches [7, 8, 9] and software fault-injection methods have appeared as viable alternatives to hardware fault injection.

The authors of [10, 11] introduced software fault injection (SFI), in which faults or errors are inserted into the system memory in order to produce errors or emulate actual errors. However, the memory fault model can only represent the effects of a limited number of actual faults, because some errors are hard to simulate by changing memory contents. In [12], fault injection into CPU components was emulated, and in [13], communication faults were added. A combination of SFI and hardware fault injection is utilized in [14]. In spite of these efforts, the capability of representing diverse fault types with SFI is still limited. However, SFI has been recognized as a viable means of dependability evaluation for the following reasons.

- Fault models are changing from the traditional gate-level model to a higher abstract-level model, and SFI suits this trend well.
- SFI is less expensive than hardware fault injection, because it does not require additional hardware support.
- It is much easier to repeat experiments with SFI and collect sufficient amounts of data.
- SFI always succeeds in injecting intended faults, but hardware fault injection suffers parasitic mutations caused by physical interferences.
- The risk of permanently damaging the target system under test is almost nonexistent.

All previously-implemented SFI methods have a common restriction that they were developed for specific target systems. That is, portability — an important merit of SFI — has not been figured into their design. By minimizing its dependence on the underlying hardware architecture and operating system, a SFI tool which runs on one system can be ported to another with minimal effort. Fault-injection experiments can then be performed during early design phases without developing a new fault injector for each target system. This is important, because changing system design at a later stage is usually very expensive. Another point we would like to emphasize is the necessity of other support tools for fault-injection experiments. Two main tools needed for fault-injection experiments are the fault injector itself and a data collection/analysis tool. In contrast to the fault injector, the latter tends to be built on an ad hoc basis. Moreover, the dependence of experimental results on the executing workloads has not been dealt with in a systematic manner. Using only a few application workloads is not sufficient to assess the effects of a wide range of applications on the underlying fault-tolerance mechanisms.

In this paper, we design, implement, and evaluate a generic methodology for constructing and integrating a set of tools needed for software fault-injection experiments. This methodology does not rely on any particular system, though our main interest lies in distributed real-time systems. Complete independence of a SFI tool from hardware and system software is impossible to accomplish, since in many cases the fault-tolerance features under test are implemented at the hardware or system software level. A realistic way to reduce the porting effort is to utilize "standard" features available in commonly-used systems, and to separate the system-dependent parts from the rest.

Our SFI supports three types of faults: memory faults, communication faults, and processor faults. High-precision controllability over fault injection timing is supported, and a fault's temporal behavior may be specified as transient, intermittent, or permanent. A powerful and flexible monitoring tool is also provided in order to facilitate the collection of both dependability and performance data. A synthetic workload generation tool is provided for ease in generating workloads of various operational characteristics under which system dependability may be evaluated.

All the functions mentioned above are controlled through a unified graphic user interface in an X-window environment. In addition, an automated and user-transparent multi-run facility allows experiments to be repeated any number of times with no additional user intervention.

The proposed SFI environment, called an integrateD sOftware fault-injeCTiOn enviRonmenti or DOCTOR for short, is implemented on HARTS [1], and several error detection and recovery mechanisms are evaluated using these tools. Section 2 presents the fault model of our SFI, as well as the architecture of DOCTOR. Section 3 gives a brief description of the initial target system, HARTS, and discusses the underlying implementation issues. Section 4 presents experiments designed to demonstrate the capability and usefulness of DOCTOR. Finally, the paper concludes with Section 5.

2 Design Methodology

In this section we present our methodology for building the DOCTOR and facilitating its use in a variety of system architectures.

We first describe a fault model which primarily determines the fault-injection capabilities and the controllability requirements. We then present the organization of DOCTOR. Finally, we discuss both portability and flexibility issues.

2.1 Fault Model

Actual hardware or software faults affect various aspects of the system's hardware or operational behavior, such as memory or register contents, program control flow, clock value, the condition of communication links, and so on. Ideally, SFI should mimic the consequences of actual faults without requiring additional hardware. A common weakness of SFIs is their limited ability to represent the effects of actual faults.

Modifying memory contents is the basic approach used in software fault injection. Faults are likely to (eventually) contaminate a certain portion of memory, so memory faults must be able to not only represent RAM errors but also emulate faults occurring in the other parts of the computer system. Though the memory fault model is quite powerful, there are two important points that this model cannot handle well. The first is the importance of dormant/latent faults which may take a long time to manifest themselves, depending on the underlying workload. Latent faults, however, are considered to have a higher probability of leading to critical system failures. Some faults, such as faults in the CPU circuitry, result in failures before they affect the system memory. Second, some faults may not affect memory contents at all, or change system memory in a very subtle and nondeterministic way. It is very hard to simulate such a fault behavior with memory fault injections. Communication errors are good examples of this.

A more sophisticated fault model is therefore required to get around these limitations of the memory fault model. To fully emulate the effects of various faults, a pure software-implemented fault injector may not be sufficient. However, the increasing complexity of contemporary computer architectures and the advance of VLSI technology are pushing fault models towards a higher level than the gate/chip level. Fault injection at a system component level will prove to be more beneficial, because of the increasing difficulty of accurate low-level fault injection and because of the trend that system-level components are becoming the basic units of replacement or reconfiguration. Based on this observation, we have decided to support three types of faults: memory faults, communication faults, and processor faults. Each of these types corresponds to a major system-level component. The user can also choose any combination of the three types to create appropriate abnormal conditions within a computer system. For each fault type, one can specify a number of options as shown in Tables 1, 2 and 3.

One important aspect of our fault model is the refined controllability. Evaluation of a fault-tolerance mechanism needs a systematic fault-injection plan, and thus, the capability of injecting a proper fault instance into a proper location at a proper time is essential to the evaluation experiments. Our fault model supports three temporal types of faults: transient, intermittent, and permanent. Various types of probability distributions are also provided to specify fault inter-arrival times, so that the system can determine when to inject faults. The user can directly control each experimental parameter using user-defined options; this is very useful, especially when testing application-level fault-tolerance mechanisms.

A memory fault can be injected as a single bit, two-bit compensating, whole byte, or burst (of multiple bytes) error. The contents of memory at the selected address are partially or totally set, reset, or toggled. A transient fault is injected only once, and an intermittent fault is injected repeatedly at the same location. When injecting an intermittent fault, the user can specify the distribution of the fault recurrence interval. Currently, the exponential distribution, normal distribution, Weibull distribution and binomial distribution are supported. The user can specify the necessary constants of each distribution type, and can add other distributions

Memory faults				
Fault range	Types	Duration	Interarrival time	Injection locations
Single bit	Set	Transient	Exponential	Stack/Heap
Compensating	Reset	Intermittent	Normal	Global variables
Single Byte	Toggle	Permanent	Binomial	User-code
Multi Bytes			Weibull	OS area
User defined			User defined	User defined

Table 1: Memory error options

Communication faults			
Fault types	Options	Duration	Interarrival time
Lose outgoing messages	Faulty-link selection	Transient	Exponential
Lose incoming messages	Altered location	Intermittent	Normal
Lose all messages	Delay control	Permanent	Binomial
Alter messages			Weibull
Delay messages			User defined
User defined			

Table 2: Communication error options

as needed. A permanent memory fault is more difficult, since the only way to facilitate true-permanent memory faults is to use make use of system provided memory protection support. Considering that memory protection is not supported in some real-time systems, and that the utilization of memory protection will be highly system-dependent and may require hardware assistance, we chose a pseudo-permanent memory fault approach. A permanent fault is emulated as an intermittent fault with a very small recurrence interval, but for efficiency reasons our implementation is different from a direct implementation of intermittent fault injection.

Besides the fault type and the injection timing, the location of memory fault injection is important. The injection location can either be explicitly specified by the user, or can be chosen randomly from the entire memory space. It is sometimes desirable for a fault to be injected in a memory section containing the user program code, the user stack/heap, the user global variables, or the system software. But it is still difficult to modify the program execution behavior properly or to emulate erroneous communication-related functions by just injecting memory faults. We overcome this difficulty by using the other fault types.

Communication failures are specified in a manner similar to memory errors, except with some additional options. Messages can be lost, altered, or delayed. If the node has multiple incoming and outgoing links, as in a point-to-point architecture such as HARTS [1], different fault types can be specified separately for each link. Lost messages are simply not delivered to the recipient. The user can specify whether outgoing, incoming, or all messages are lost at the faulty link. Messages can be lost intermittently, with a probability specified by the user, or alternatively, every message can be lost. Messages may be altered in the same manner

Processor faults			
Fault types	Duration	Interarrival time	Injection locations
Control flow change	Transient	Exponential	Random
Register content change	Intermittent	Normal	User defined
ALU malfunction	Permanent	Binomial	
Clock malfunction		Weibull	
User defined		User defined	

Table 3: Processor error options

as memory locations, i.e., by inserting single bit, two-bit compensating, or burst errors. The user can specify whether the error is to be injected into the body of the message or into the header, which contains routing information. The injection can be intermittent or permanent. For delayed messages, a method must be specified to determine how long each message will be delayed. The delay time can be either deterministic or can follow an exponential distribution with a user supplied mean. In addition to this set of predefined communication fault types,the user can define additional communication fault actions. These user defined faults may be combinations of the predefined fault types, and may be based on the contents of individual messages or on the past message history. This variety of communication failures, and the ability to combine existing and define new fault types, allows the injection of a variety of failure semantics, including Byzantine failures.

Processor faults reside in data registers, control registers, bus interface units, the ALU, and so on. Depending on the underlying hardware and system software, accessibility to hardware components varies widely. For example, it is impossible to directly access certain components in a data path and a control path by software. (This is also true in the case of hardware fault injection.) One way to overcome this limitation is to inject an erroneous behavior rather than a fault itself. However, the exact effect of faults in each component of a processor is highly architecture-dependent. Instead of using the detailed knowledge of specific CPU architecture in order to emulate actual faults more directly as was done in [12], we chose to emulate the consequence of actual faults. Without using any special hardware, we make use of executable image modification, which changes some existing instructions generated by the compiler or inserts extra instructions for fault-injection purposes.

The manifestations of processor faults we currently support are the alteration of control flow, register contents, clock value, and ALU results. For example, the control flow can be altered by bus line errors, instruction decoding logic errors, errors in a condition code flag, or control register errors(e.g., program counter). Also supported are three temporal behavior types: transient, intermittent, and permanent. The injection location combined with timing parameters determines the time of error occurrence. The decision of an injection location can be either application-transparent or specified in the user's source code. Note that the execution of extra instructions may affect the system performance.

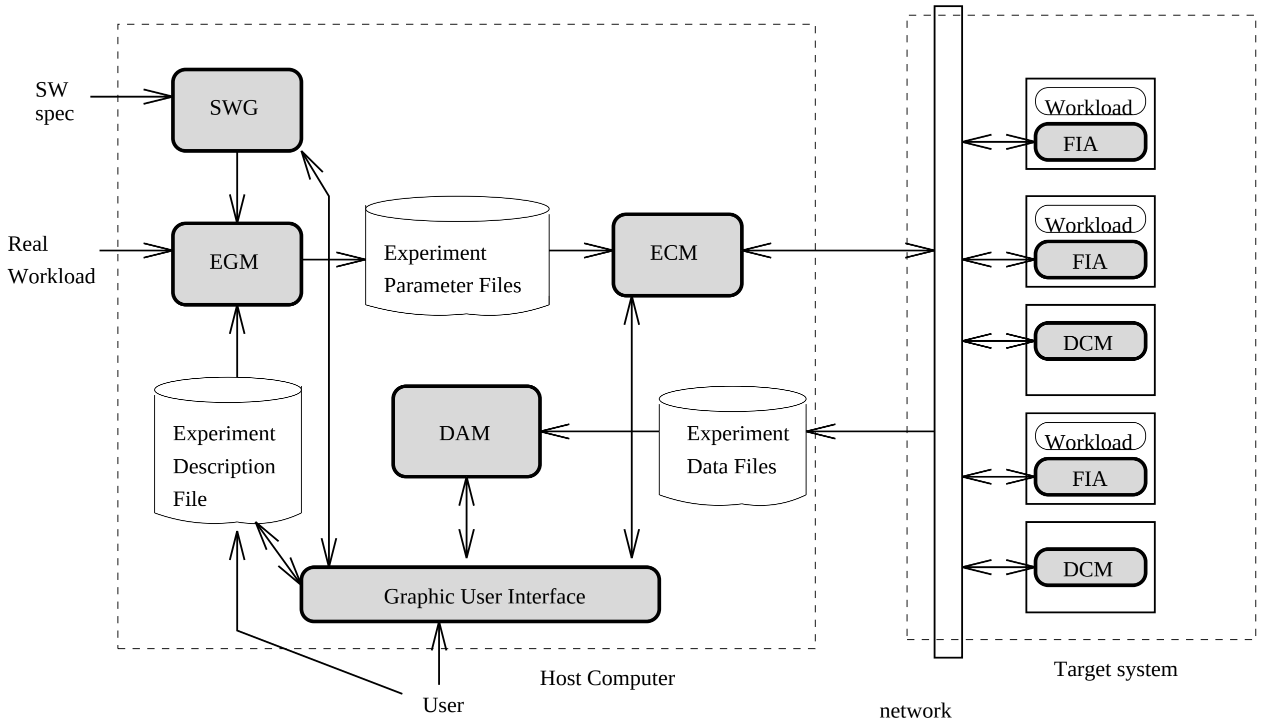


Figure 1: The organization of DOCTOR

2.2 Organization of DOCTOR

A fault-injection experiment environment consists of several components: the target system with fault-tolerance mechanisms which are to be evaluated, a software fault injector, the workloads to be run on the target system, and a dependability monitoring tool.

We provide a complete set of tools for automated fault-injection experiments, and this tool set forms a modular software architecture. Our tool set can be used in both a single processor system and a multiple processor system, but it is mainly intended for use in real-time distributed systems. Figure 1 shows the organization of DOCTOR. In this architecture, a host computer works as a console node and several processing nodes are connected with each other and linked to the host node over a communication network.

One distinct feature of the above organization is the separation of components of the host computer from those of the target system. DOCTOR organization has some innate advantages. First, it reduces the run-time interference caused by fault injection. It is important to minimize interference in order to obtain accurate timing data, such as fault-detection latency. Second, it makes the run-time control of experiments much easier, because each component can be controlled separately. Third, it can support a multi-run experiment facility and it reduces the effort needed for porting to other systems. This will be detailed in the following section.

The core part of DOCTOR is the SFI which supports the fault model described in the previous section. SFI consists of three modules: the Experiment Generation Module (EGM), the Experiment Control Module (ECM), and the Fault Injection Agent (FIA). The dependability monitor collects dependability data during each experiment, and analyzes it on the fly, or off-line after completing the experiment. The monitor is composed of two modules: Data Collection Module (DCM), Data Analysis Module (DAM). A Synthetic Workload Generator (SWG) is provided to generate various synthetic workloads under which fault-tolerance mechanisms are evaluated. In addition, an integrated Graphic User Interface (GUI) and a fully-automated multi-run experiment facility are provided to facilitate and automate the design and execution of fault-injection experiments.

The first role of EGM is to generate code executing different workloads which will be downloaded (from the host) to the target system. A workload could be run on a single processing node or be distributed among a number of nodes. The user can use real programs as workloads, or can rely on SWG for synthetically-generated workloads. In either case, when the workloads are compiled and linked, the symbol-table information is extracted to be referenced by ECM, and library programs are attached to workload executable code in order to synchronize the start and the stop of workload execution. Special instructions are inserted into workload executable code for injecting processor faults. However, the the attached library modules are transparent to the users. Only when the fault-tolerance mechanism under test is implemented within the workload itself does reporting the error detection/correction become the workload's responsibility. In this case, the user needs to know the interface functions of DCM.

The second role of EGM is parsing the experiment description file supplied by the user or generated by GUI automatically. The experiment description file contains the experiment plan and the information about the fault type and injection timing. EGM generates an experiment parameter file for each node involved in an experiment. These files are used by ECM to determine when to start fault injection, what kinds of fault are to be injected, and how many times the experiment will be run, and so on.

FIA is a separate process which runs on the same nodes as the workloads. It receives control commands from ECM via a communication network and executes them. It injects faults, and lets workloads wait/start/stop. It also reports its activities to DCM, such as the injection time, location, type, etc. A workload is controlled by FIA through shared memory and system calls. Thus, it is possible to run FIA on a different node if the underlying system architecture allows it. FIA directly controls the execution of workloads, and FIA is in turn controlled by ECM. Consequently, ECM can capture the control of entire experiment sequences.

ECM functions as a controller. It supervises all other modules, and sets up an experiment environment by downloading workload executable code, FIAs, DCMs, and even system software if needed. FIAs and DCMs pass information to and receive commands from ECM. ECM uses the experiment parameter files generated by EGM to send proper commands to FIAs. The symbol-table information can be used to decide memory fault injection locations, and a variety of random number generators are equipped to support different distribution types. It can deliver a precise independent timing control service, which means the start of each run is synchronized and the user can specify each injection start/end time on different nodes. The execution of workloads is interrupted when the run duration limit is reached.

To evaluate the dependability of fault-tolerance mechanisms, we must measure dependability parameters like detection coverage and latency while executing appropriate workloads. A workload produces demands for the system resources, so the structure and behavior of the workload may affect the result significantly. In DOCTOR, the user can use a real application program as a workload, or use a synthetic workload produced by SWG [15]. Because the synthetic workload is parameterized, the user has direct control of the workload characteristics, and so experiments can be conducted under various workload conditions. SWG compiles a high-level description of a workload to produce an executable synthetic workload, ready to be downloaded to a processing node.

One important issue in dependability experiments is the collection of relevant data. In fault-injection experiments, two types of data are needed for dependability analysis. One is the history of fault injections, and the other is the error report. A simple way of collecting data is to give the entire responsibility to FIA and the fault-tolerance mechanisms under test. This choice is simple but violates a few basic principles of monitoring. The effect of monitoring on the monitored system should be minimized because the high overhead of monitoring may distort timing measurements. Monitoring must also be transparent to the monitored system. Another important point in collecting fault-injection data is the consideration of performance monitoring. We chose to use a separate process for performance data collection so as to (i) reduce the burden of data logging on FIA and fault-tolerance mechanisms, and (ii) achieve more flexible and expandable capabilities that can be incorporated with the capability of performance monitoring. The problem of separate measurement of performance and dependability is mentioned in [16].

The basic function of DCM is to log the events generated by the monitored object. The FIA and fault-tolerance mechanisms under test generate such events, and if performance is monitored along with dependability, the event triggering instructions should be placed in the operating system kernel. These events are the categorized and time-stamped information about the activities which we want to monitor. Generation of events is the only overhead to the monitored object. DCM works continuously during experiments, and its function is fairly passive. It can be run on the same node with workloads or on a different node. The number of DCMs required in an experiment depends on the target system architecture and the complexity of experiments, and is not restricted to any particular architecture or experiment. The collected data may be stored in a number of files and used later for automatic data analysis.

DAM analyzes the data collected by DCM. For example, it calculates the coverage and latency of each detection mechanism. Since DAM has a modular structure, other analysis capabilities can be added easily. DAM has a graphic display function to present analysis results. A common step before the data analysis is the merging of files generated by DCMs according to time-stamps. In this preprocessing step, superfluous data are filtered out, and the global information about the experiment is extracted. The accuracy of analysis and the range of parameters to be analyzed are determined by the data collection process as well as DAM.

The GUI helps the user design and control fault-injection experiments. It creates the experiment description files, and allows the user to control the execution of experiments at run time. It also facilitates the use of SWG, and manages the graphic display for DAM. The GUI provides the user with an integrated and easy to use interface to DOCTOR.

Each fault-injection experiment with specific workloads is called a *run*. In a fault-injection experiment, one of the factors that determine the quality of analysis results will be the number of runs. Therefore, it is very useful to automate multi-run experiments. In a multi-run facility, the key problem is the synchronization and re-initialization of several processes, including workloads. The level of re-initialization required depends on the failure semantics of the target system. In some cases, it may be necessary to reset the whole system, and in some other cases, the restart of workloads may be enough. A typical multi-run experiment sequence is given below.

Step 1: EGM generates executable images of workloads and experiment parameter files.

Step 2: ECM sets up the experiment environment: downloading FIAs and DCMs and workloads on the proper nodes.

Step 3: ECM synchronizes all the processes.

Step 4: One run is started and relevant data are collected by DCM.

Step 5: ECM interrupts workloads, and then go to Step 2 or Step 3, depending upon failure semantics.

Step 6: DAM analyzes collected data.

2.3 Portability and Flexibility

When assessing the portability of software, one should consider the low level services required for porting. In the SFI case, it can affect the lowest layers of system software. It must sometimes penetrate the protection boundary of operating systems, and modify communication protocols of various layers. To accomplish a reasonable degree of portability in this specially complex situation, we take advantage of software engineering techniques as summarized below.

First, we used modular design and functional independence among modules. Instead of lumping all into a unified executable image, the software needed for fault-injection experiments is divided into a set of modules. In a multiprocessor or at least a multi-programming environment, each module of DOCTOR is a separate process and communicates with others through typical interprocess communication methods such as shared memory, message passing, or intermediate files. As a result, modules are functionally independent. In other words, each module can be either disabled or replaced by other programs which perform similar functions with the required interface. This flexibility allows DOCTOR to be joined with other programs which have been developed for their own special purposes.

Second, we encapsulated or isolated system-dependent parts. This information hiding leads to effective modularity. Modules are roughly divided into two groups: highly system-dependent and less system-dependent. One good example of a highly system-dependent module is the FIA. In addition to this separation, standard implementation techniques are employed to enhance portability. As will be described in the next sections, the modules which run on the host are implemented under the UNIX operating system, and the data exchange with modules on the target system relies on standard communication protocols such as the TCP/IP socket stream. Therefore, in most cases modules of the host can be ported to a different system with little effort.

Third, we chose a high-level fault injection. By abstracting the fault model, one can therefore choose less system-dependent fault-injection techniques. In fact, the portability of each injection method is different. For example, memory fault injection may require an access method if virtual memory is used. Processor fault injection requires the knowledge of the target system compiler.

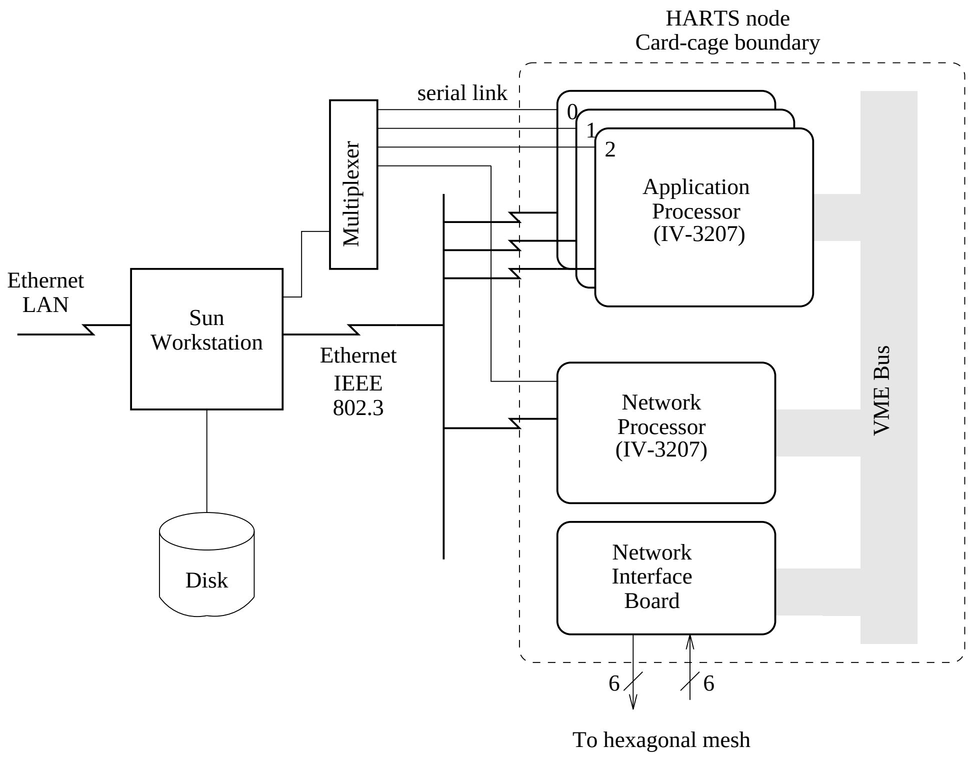


Figure 2: The HARTS software development environment.

Communication fault injection may be more complicated, depending on the structure of the communication protocol stack, and its degree of integration with the operating system.

3 Implementation on HARTS

The first target system of DOCTOR is HARTS [1], a real-time distributed system. Presented below are the relevant details of HARTS and important implementation issues.

3.1 Description of the Target System

HARTS is comprised of multiprocessor nodes connected by a point-to-point interconnection network.[1] Each HARTS node consists of several Application Processors (APs) and a Network Processor (NP). The APs are used for executing application tasks, and the NP handles most of communication processing. The NP consists of interfaces to the network as well as the APs, buffer memory, and a general-purpose microprocessor.

In the current configuration, the nodes of HARTS are VMEbus-based Motorola 68040 systems. Each HARTS node has 1–3 AP cards, an NP card, and a communication network interface board. Each AP card is the Ironics IV-3207 [17], a VMEbus-based 68040 card, and

[1]Initially it was a continuously-wrapped hexagonal mesh topology, but now it is not restricted to this topology only.

another IV-3207 card serves as an NP. The Ancor VME CIM 250 [18], which composes a fiber optic interconnection fabric through Ancor CXT 250 switch [19], works as a communication network interface board. A custom network adapter board, called SPIDER [20], is currently under development.

Each node of HARTS runs an operating system called HARTOS [21]. HARTOS is primarily an extension of the functionality of pSOS^{+m} [22], a commercial real-time executive to work in a multiprocessor and distributed environment. While pSOS^{+m} provides system support within a node, an extended version of the x-kernel [23] operating system coordinates communication between nodes. The x-kernel provides facilities for implementing communication protocols, such as a uniform protocol interface and libraries to efficiently manipulate messages and maintain mappings.

Software for HARTS is developed on Sun workstations. A Sun sparcstation serves as the main connection to HARTS. HARTS applications and system software are downloaded from this workstation through the local HARTS Ethernet. The workstation is also connected to the campus computing facilities by a separate Ethernet connection. Thus, the programs developed and compiled on other workstations may be downloaded to HARTS, but HARTS executes with a dedicated local Ethernet. The workstation also serves as the console for the HARTS nodes. It is connected via a multiplexor (MUX) to the console serial ports on each AP card and an NP card of each HARTS node. The HARTS software development environment is shown in Figure 2.

3.2 Implementation Issues

In implementing SFI on HARTS, we use three methods to inject faults concurrently with the execution of workloads. Simple memory overwrites are used for injecting memory faults, a special fault injection protocol layer is used for communication faults, and modification of the workload executable code is used for processor faults. Fault injection is controlled indirectly by ECM through FIAs. One FIA is downloaded to each AP on which a workload is running. To support the precise timing control of an experiment, ECM spawns a child process per node, and synchronizes the execution of these child processes.

For memory fault injection, each child process decides the injection location, the fault type (range/mask), and the injection timing. When a pre-determined time is reached, the process sends the corresponding FIA a message, which contains the address and the fault type. The content of the addressed location is masked by the specified pattern in the fault type using AND (reset) operation, OR (set) operation, or XOR (toggle) operation. If the fault is the permanent fault type, FIA refreshes the contents of the address periodically. As a result, no excessive communication between ECM and FIA is required to inject a permanent memory fault. HARTS does not have any memory protection,[2] so FIA can easily overwrite any memory area.

Another issue in injecting a memory fault is the problem of deciding the location of injection. HARTS does not use virtual memory, so the symbol-table information can be easily used in an absolute address form, combined with the OS memory-map information. If the target system

[2]Like most other real-time systems, HARTS does not employ virtual memory to eliminate the unpredictability in memory access caused by page faults.

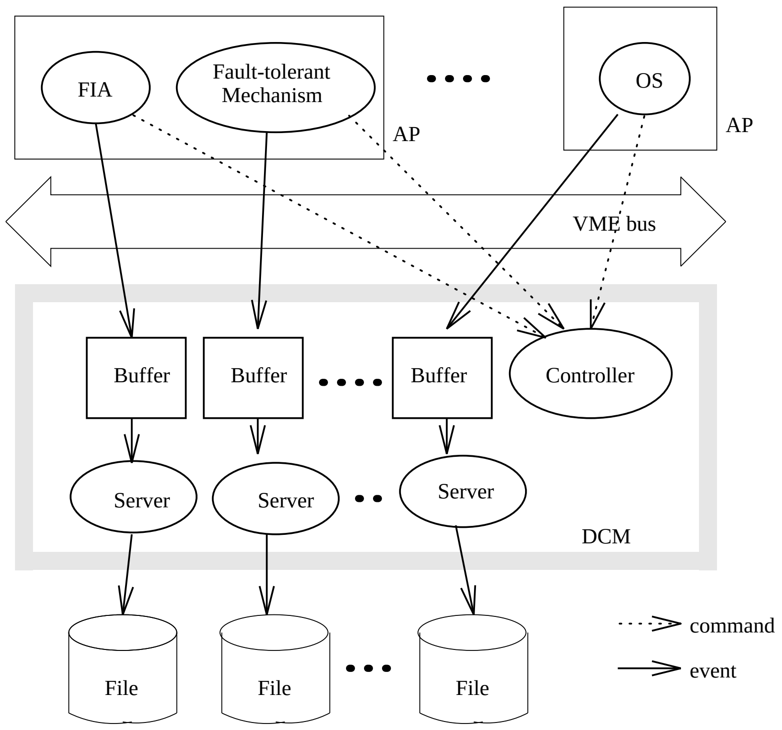
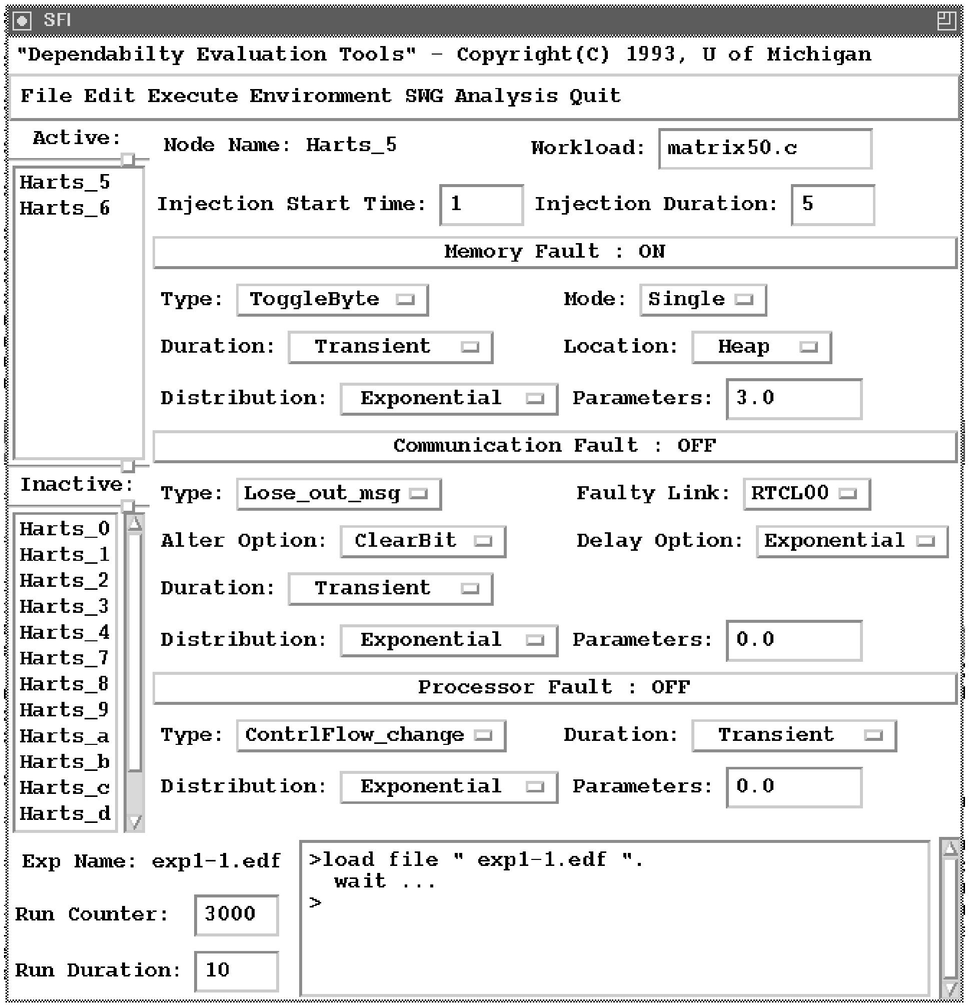


Figure 3: Data collection process

is furnished with a memory protection facility or virtual memory, one can use either software techniques like 'Trojan horse' in [10], or hardware-dependent techniques like those in [11, 12].

Communication faults are injected by a special communication protocol layer that can be inserted anywhere in the protocol stack. This fault injection layer accepts timing and fault type commands from FIA, and uses information from the parameter files generated by the EGM to initialize the message history structures. It may be placed between any two protocol layers, but is normally inserted directly below the protocol or user program to be tested. This approach is similar to that in [13]. The current implementation takes advantage of the features of the x-kernel operating system, in which our communication protocols are implemented. All protocols in the x-kernel are implemented using same interface between layers, called the Uniform Protocol Interface (UPI). As a result of the UPI, the fault-injection layer does not need to be modified when it is placed between different protocol layers. All protocol specific information needed by the fault-injection layer, such as the message header and data formats, is included using separate header files. If more complex fault scenarios are desired, copies of the communication fault-injection layer may be placed in multiple places in the protocol graph. The x-kernel also provides additional system libraries, such as an event library that is used to schedule future events and a map library that is used to maintain bindings between identifiers, that simplify the implementation of the communication fault injection layer. The communication fault injection layer operates by intercepting the UPI operations between the protocol under test and the lower layer protocols. If it detects an operation during which a fault should be injected, based on commands from the FIA, it performs the appropriate fault injection operation. All other operations are simply passed through without modification.



"Dependabilty Evaluation Tools" - Copyright(C) 1993, U of Michigan

Active: Harts_5, Harts_6

Node Name: Harts_5 Workload: matrix50.c

Injection Start Time: 1 Injection Duration: 5

Memory Fault : ON

Type: ToggleByte Mode: Single

Duration: Transient Location: Heap

Distribution: Exponential Parameters: 3.0

Communication Fault : OFF

Inactive: Harts_0, Harts_1, Harts_2, Harts_3, Harts_4, Harts_7, Harts_8, Harts_9, Harts_a, Harts_b, Harts_c, Harts_d

Type: Lose_out_msg Faulty Link: RTCL00

Alter Option: ClearBit Delay Option: Exponential

Duration: Transient

Distribution: Exponential Parameters: 0.0

Processor Fault : OFF

Type: ContrlFlow_change Duration: Transient

Distribution: Exponential Parameters: 0.0

Exp Name: exp1-1.edf

Run Counter: 3000

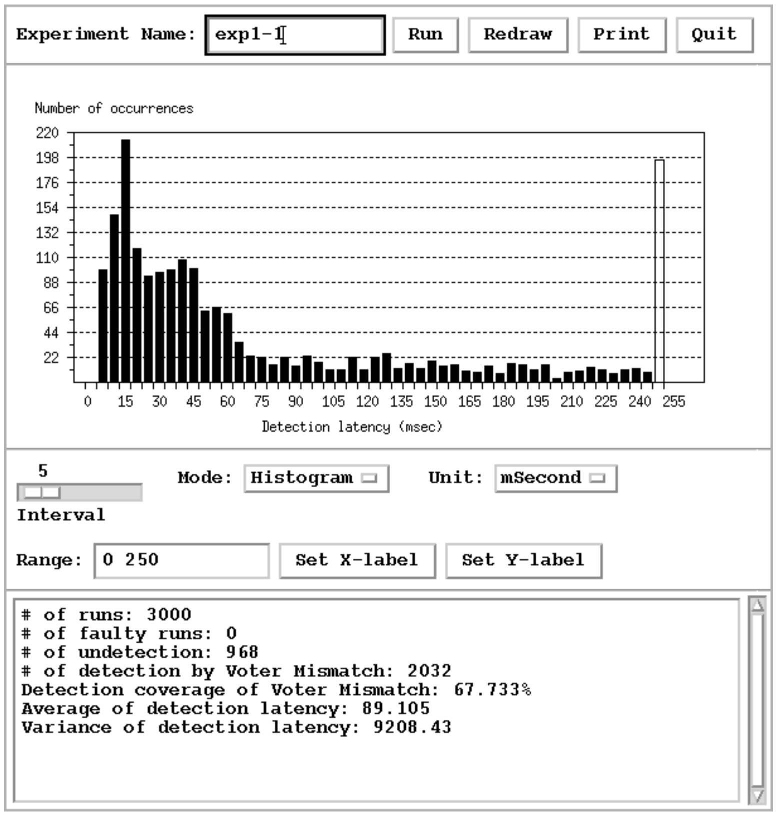
Run Duration: 10

```
>load file " exp1-1.edf ".
  wait ...
>
```

Figure 4: Main window of Graphic User Interface

Outgoing and incoming messages are lost by intercepting the appropriate send and receive operations, and then discarding the message. Messages are altered by intercepting the send or receive operation, and then changing the message contents before passing it on to the next protocol layer. Messages are delayed by stopping the current message and then scheduling a future message with the same contents, using the x-kernel event library. In order to support the user-defined fault classes, we allow messages to be stored in a message history using the x-kernel map library. All references to past messages in the user-defined fault description are translated into map library operations. In combination with the message header format and the predefined fault types, this allows complex communication fault scenarios to be specified by the user.

A permanent processor fault can be emulated by changing program instructions at compile time. For example, the control flow of a program can be altered by arbitrary jump instructions. Modifying all instructions using a faulty ALU can emulate an ALU fault, and overwriting the register contents in the middle of the program execution can emulate a register fault. By



```
# of runs: 3000
# of faulty runs: 0
# of undetection: 968
# of detection by Voter Mismatch: 2032
Detection coverage of Voter Mismatch: 67.733%
Average of detection latency: 89.105
Variance of detection latency: 9208.43
```

Figure 5: DAM window

replacing or adding instructions at the assembly language level, the other types of processor fault can be emulated.

A transient/intermittent type of processor fault requires more than compile-time efforts. We support this type of fault as follows. When the program execution reaches the specified instruction, whether or not to inject a fault is determined using one extra instruction inserted at compile time. If a prespecified fault injection time is reached, a predetermined or randomly selected fault is injected, such as changing the value of the program counter. In the other case, the program will follow the normal sequence.

Currently, the hardware-assisted clock synchronization protocol [24] of HARTS is under development, so we added a software-implemented clock-synchronization capability to ECM. The clocks of processor nodes are synchronized periodically, once at the beginning of each experiment, or once every run, or as often as needed.

In the current HARTS version, DCM runs on a dedicated processor and communicates with FIAs of different APs which are on the same VMEbus backplane. Therefore, if several HARTS nodes are used, then we need the same number of DCMs as HARTS nodes. Thus, DCM minimally interferes with the execution of application workloads. One more benefit of this separation is the isolation of DCM from other APs. Even when workloads must be re-initialized between runs or some APs must be reset due to failures, DCM can continue its function without disruption.

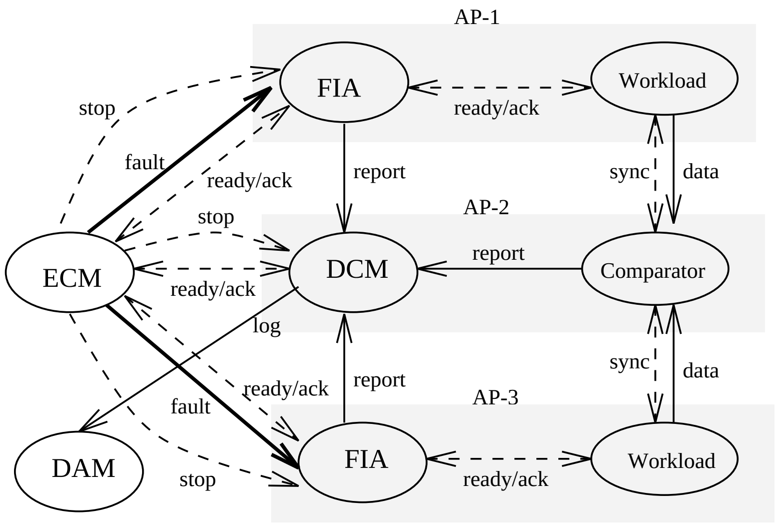


Figure 6: Example of muti-run experiment structure

Communication between DCM and log-event [25] generating processes (e.g., FIA) is done through the VMEbus on the HARTS node. DCM allocates a distinct buffer area for each client process. Logged data are stored in the file system mounted on the dedicated processor of running DCM. Similarly, one file is opened for each client process. Figure 3 depicts the basic working mechanism of data collection.

GUI is implemented based on X-window Motif [26], and runs as a separate process. Figure 4 shows the main window of GUI, and Figure 5 shows the DAM window.

One critical problem in implementing a multi-run facility is the re-initialization of both workloads and the fault-injection environment. Figure 6 shows the relationship of processes to the re-initialization sequence in the case of duplex-match detection experiment (see Section 4.1 for more on this). The first step is to synchronize all the processes involved. This step is initiated by the 'ready' signal from workloads, and concluded when each workload receives 'ack' signal from ECM. Then, workloads begin running, and FIAs inject faults as commanded by ECM. At the same time, DCM collects relevant reports from FIAs and the underlying fault detection mechanism, and logs them in files for DAM. After a specified time elapses, ECM sends a 'stop' signal to FIAs in order to interrupt the current run. These sequences will be repeated for multi-run experiments.

4 Experimental Results

In this section, we demonstrate the capability of DOCTOR by evaluating the effectiveness of error-detection and diagnosis methods with this tool set. All experiments were performed on HARTS. Since DOCTOR offers an automated multi-run facility, it is easy to run experiments as many times as needed while varying various (including workload-related) parameters. The first experiment evaluates a duplicate-match error-detection method, while the second experiment evaluates a signature monitoring detection method [27]. The third experiment use

	Case 1	Case 2	Case 3	Case 4
matrix dimension	30x50x30	30x50x30	40x80x40	40x80x40
sampling frequency	1/50	1/150	1/50	1/150
number of runs	3000	3000	3000	3000
detection coverage	67.73%	32.33%	63.52%	56.85%
latency mean (variance)	89.10msec (9208)	118.49msec (11501)	206.79msec (69348)	251.57msec (74300)
performance overhead	36.13%	34.97%	36.20%	35.02%

Table 4: Data analysis summary of memory fault injection experiments

communication faults to evaluate a probabilistic distributed diagnosis algorithm.

4.1 Memory Fault Injection Experiments

The goal of these experiments is to illustrate how the dependability parameters of a fault-tolerance mechanism can be measured with DOCTOR. Specifically, we measure the distribution of error-detection latency and the coverage of error detection. We also analyze other interesting parameters, such as the performance overhead of the error-detection mechanism, the dependence of error-detection latency and coverage on the executing workloads, and the effect of comparison granularity in the duplicate-match error-detection mechanism. Described below is the experiment specification.

A software-implemented duplicate-match error detector is run on an AP which is different from those APs executing real workloads. Two identical workloads are run on two distinct APs. They are executed independently, and their start and stop are loosely synchronized. The error detector compares the memory access operation of two identical workloads. Software-based implementation of this detection mechanism obviously limits its processing capability. To reduce its intrusiveness, we use the detection mechanism with two options. First, buffers are utilized instead of simple lock-step comparison. Second, comparison is done only on the sampled memory write operations. But it is still susceptible to buffer overflow when too many memory accesses occur in a short period. A program for floating-point matrix multiplication is used as a workload. It consists of an infinite loop of matrix data initialization and multiplication. Every write operation on matrix data is done through a function call in which some of data sampled at a certain frequency are sent to the error detector for comparison. When an overflow occurs at the error detector, the workloads must wait in the write function call. Memory-type faults are injected into the memory section allocated for matrix data in the user program stack area. For simplicity, we chose to inject one byte toggling transient memory faults.

Four cases are tested and measured, and the results are summarized in Table 4. Each experiment is repeated 3,000 times, and two different size workloads are tested with two different sampling frequencies, respectively. One observation is that the larger matrix case has a larger detection latency, even if the same sampling frequency is used. This is because it takes longer to generate each element to be compared. Increasing the sampling frequency shortens the detection latency, but it also increases the performance overhead. The analysis results also indicate that the comparison granularity determined by the sampling frequency directly affects

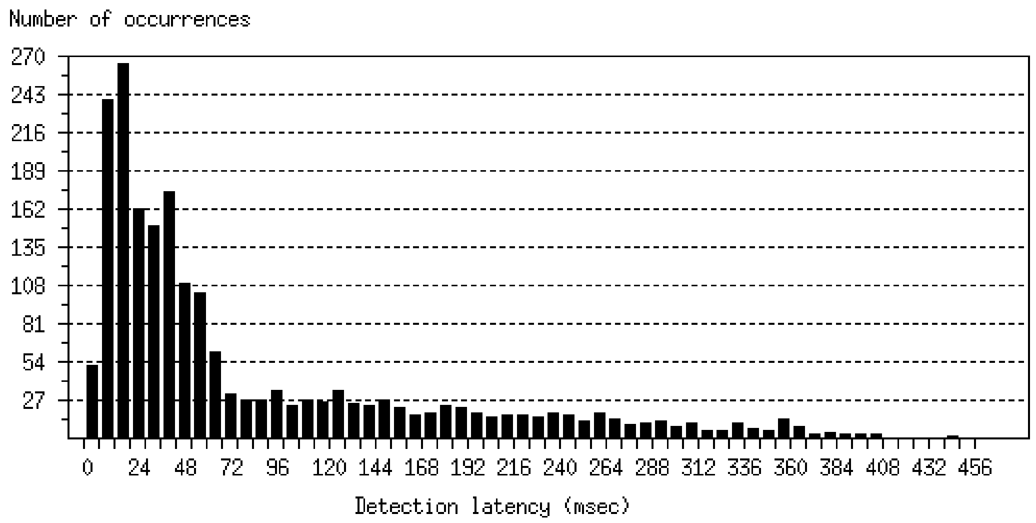


Figure 7: Fault latency distribution: Case 1

error-detection coverage.

Figure 7 shows the distribution of error-detection latency for Case 1. The results of Case 4 are presented in Figure 8.

4.2 Processor Fault Injection Experiments

In these experiments, we use synthetic workloads to observe the workload effects on the dependability measures. The synthetic workload used here simulates a structure of a typical real-time system, in which a sensor generates events periodically and the processed events are delivered to an actuator connected to a different AP. SWG is a useful tool for generating representative workloads, especially when actual application code is unavailable at an early stage of system design. A detailed account of the SW specification language can be found in [15]. Figure 9 shows the conceptual structure of these experiments.

The error-detection mechanism tested in this experiment is a simplified software-implemented watchdog monitor. It uses the asynchronous disjoint signature monitoring technique for multi-processor systems as in [28]. In our scheme, explicit signature-send instructions are inserted in the program, and the signature history of an error-free run is used as a reference structure of the watchdog monitor. Tasks on different processors transmit their signature to the watchdog monitor through dedicated signature queues. One signature-send instruction is added to the end of each program block, and each signature is randomly decided.

To evaluate this error-detection mechanism, transient changes of control flow (transient faults) are introduced. We make use of SWG to generate synthetic workloads which have blocks of different sizes. For simplicity, all instructions in a block are floating-point multiplication statements, and a 5,000-instruction synthetic workload is employed. In this experiment, we observe the effect of the average block size on error-detection coverage, detection latency, and

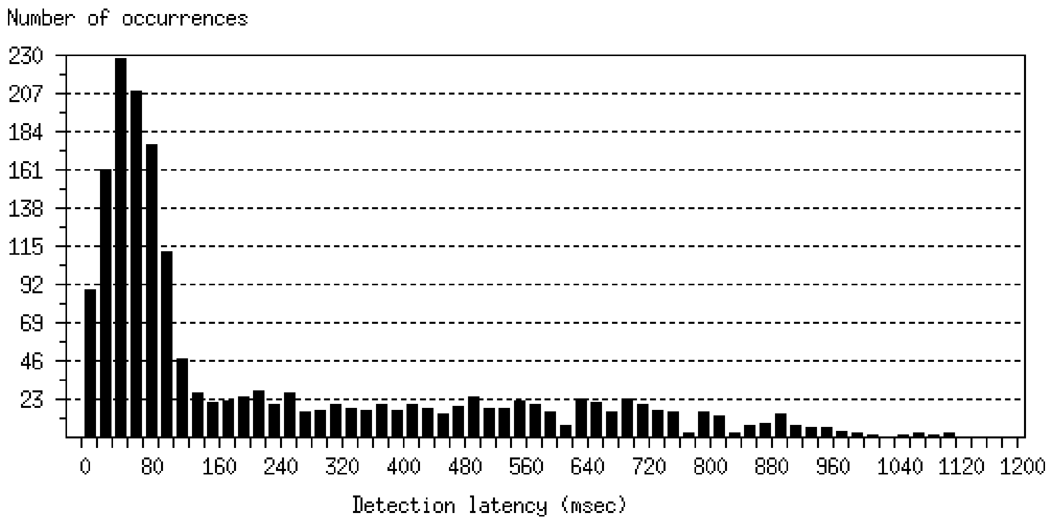


Figure 8: Fault latency distribution: Case 4

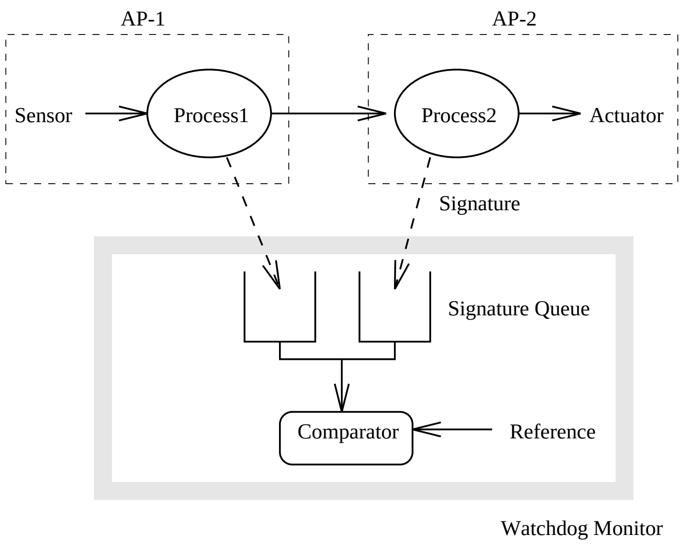


Figure 9: Watchdog monitor experiments

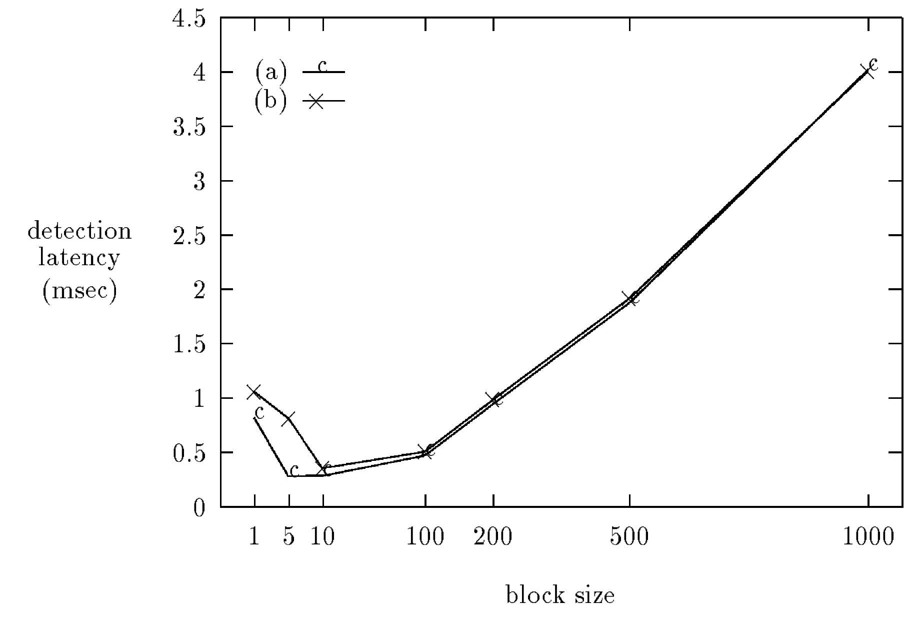


Figure 10: Dependence of error latency on the workload (a) when one signature source is used, (b) when two signature sources are used.

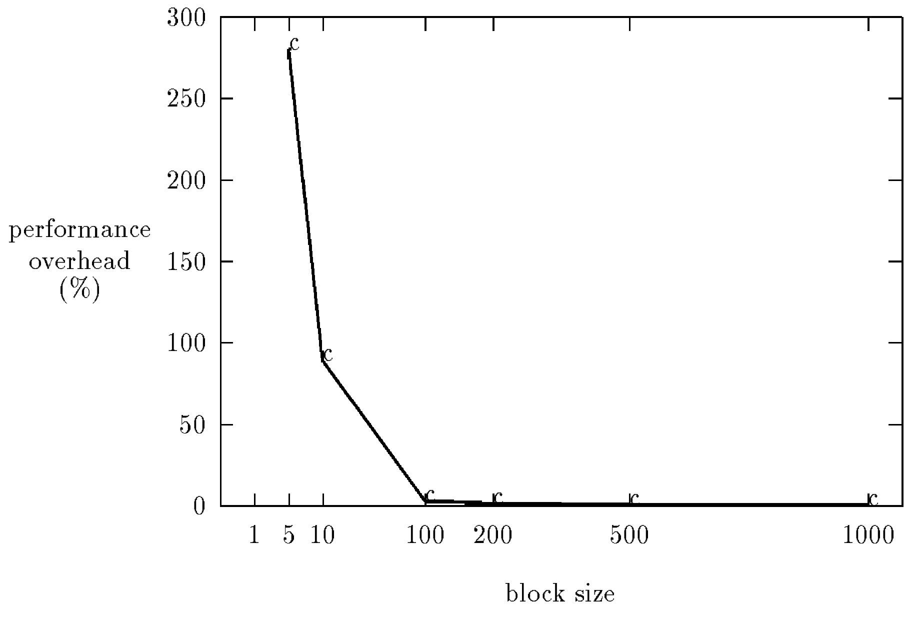


Figure 11: Dependence of performance overhead on the workload

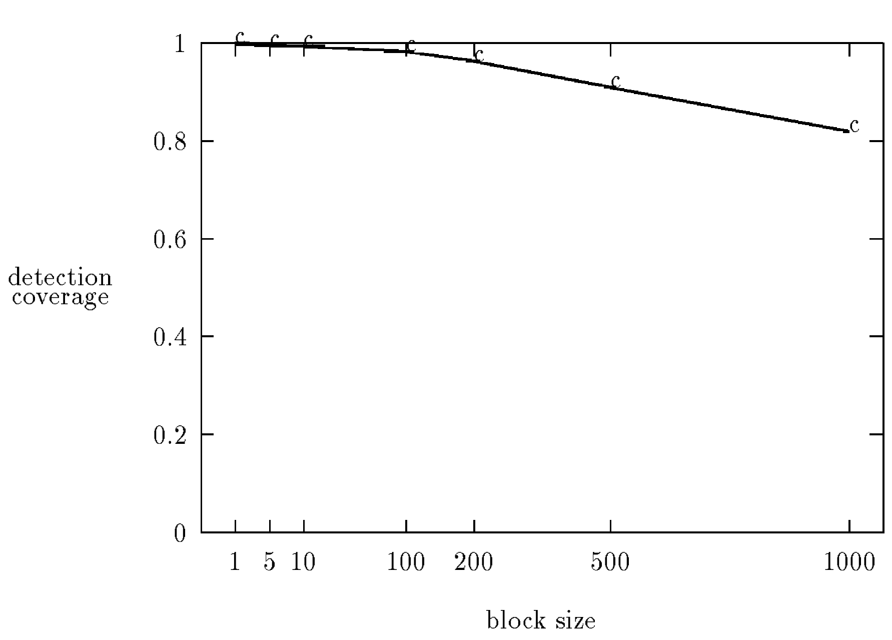


Figure 12: Dependence of error-detection coverage on the workload

the performance overhead.

Figure 10 shows the dependence of error latency on the block size. The average latency of each case is obtained through 1,000 runs. The case when two signature sources share a watchdog monitor is also evaluated. In the watchdog monitor under test, the signature of each block is not generated based on the content of the block. Therefore, when block size increases, so does error latency. However, this is not true until the block size goes past a certain point. This phenomenon is the consequence of the signature queue overflow caused by the high signature generation rate exceeding the capability of the watchdog monitor, which is measured to be able to make about 20,000 signature comparisons per second.

The performance overhead is plotted in Figure 11. Obviously, the high density of signature-send instructions worsens the performance. Figure 12 shows the dependence of error-detection coverage on the block size. The lack of consideration of block contents when generating signatures leads to the decrease of detection coverage with large blocks.

4.3 Communication Fault Injection Experiments

In this section, we demonstrate the usefulness of software fault injection as a tool for validating dependability models of distributed protocols. By using the communication fault injection capabilities of DOCTOR, we are able to collect data on the behavior of a distributed diagnosis algorithm under a wide range of conditions. This data can then be used both to validate the predicted performance of the algorithm, and to assist in the selection of various parameters used during the execution of the algorithm.

One important problem in dependable distributed computing is the diagnosis of the set of faulty nodes in the system. There have been a number of distributed diagnosis algorithms proposed to address this issue. A good survey of distributed diagnosis algorithms, and their

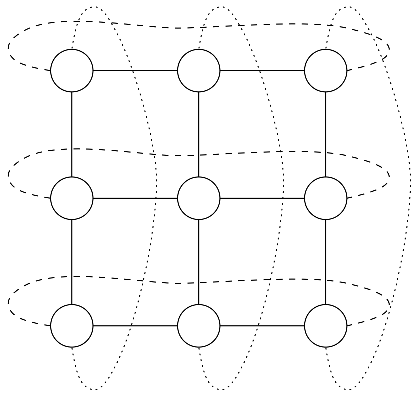


Figure 13: Diagnosis Experiment Topology

place in the broader context of consenus algorithms, is given in [29].

The algorithm we chose to implement and test is the probabilistic distributed diagnosis algorithm given in [30]. In probabilistic diagnosis algorithms, no bound is placed on the number of faulty nodes. These algorithms define procedures for using information gathered during interprocessor tests to identify a set of faulty nodes. The diagnosis can be done using either local or global information, and the analysis of these algorithms generally includes proving results about the probability of arriving at a correct diagnosis, given certain assumptions about the system. The algorithm in [30] is intended for the diagnosis of distributed systems of arbitrary connectivity, and is based on comparison testing. Each run of the diagnosis algorithm consists of a number of rounds of testing. For the purposes of the diagnosis algorithm, a test graph, which is a subgraph of the undirected processor connectivity graph, is selected. Each node runs an identical test task on each round, and then exchanges the results with its neighbors in the test graph. The local result is then compared with the results received, and, if the number of mismatches is greater than some threshold, the node is considered to have failed that round. This is repeated for some number of rounds. If the number of rounds in which the node failed is greater than a second threshold, then the node is considered to be faulty.

This algorithm has a number of parameters that alter or determine the effectiveness of the algorithm. Some of these parameters are selectable by the user, while others, such as the probability of a processor having failed, are functions of the system and its environment. The parameters that we look at in these experiments are: the number of rounds of testing (r), the coverage of interprocessor tests (c), and the number of failure modes of a test (f). The coverage of a test is the probability of a faulty processor generating an incorrect result on that test. The number of failure modes of a test is the number of possible incorrect results that a faulty processor can generate for that test. Other parameters, which we will fix for these experiments, are the probability of failure of a processor (p), the total number of processors, the interconnection topology, and the test graph.

In order to test the distributed diagnosis algorithm using communication fault injection, it was necessary to construct a user-defined fault type from the basic communication fault types provided by DOCTOR. The diagnosis algorithm has two probabilistic parameters which we wanted to be able to alter in our experiments, the probability of failure of a processor, and

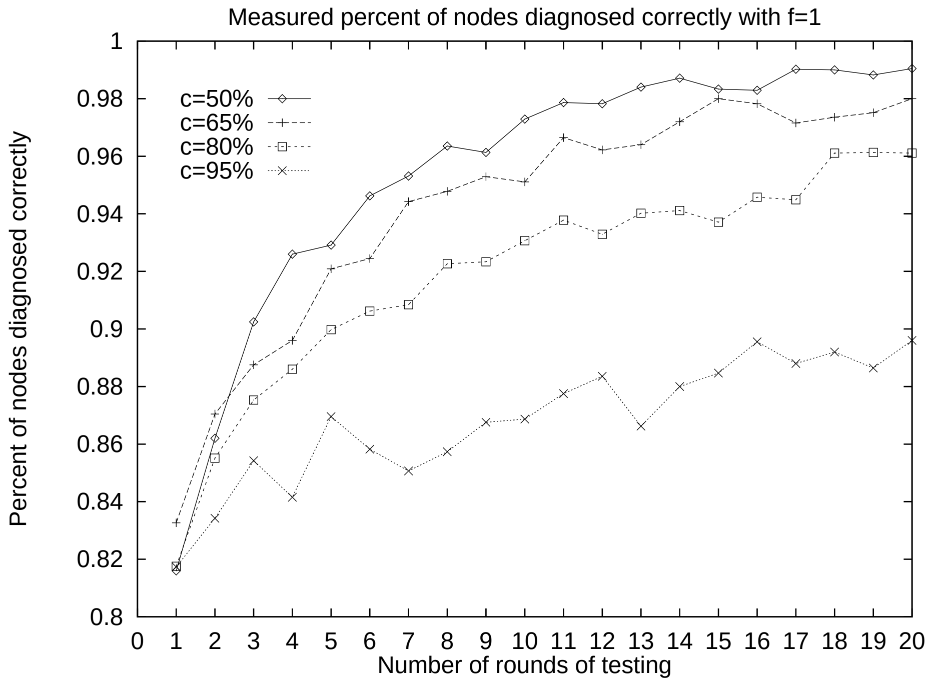


Figure 14: Percent of nodes diagnosed correctly with 1 failure mode, measured

the interprocessor test coverage. As a result, we constructed the following fault scenario. Each time the experiment is initialized, the fault status of each node is independently chosen, with a probability of failure, p, equal to a preselected value. On a faulty node, each time a message is sent by the diagnosis algorithm, the message history is checked to determine whether a previous message with the same test round number has been sent to another node. If not, then with a probability equal to the test coverage, the message contents are altered to a randomly selected value, with a range given by the number of failure modes. If a message from the same round had been sent, then information from the message history is used to ensure that all messages from the same round are altered (or not altered) in the same way. This fault scenario allows us to inject faults that emulate the fault model for which the diagnosis algorithm was designed, while still maintaining control over the failure and coverage parameters.

In our experiments, we implemented the algorithm and executed it on a nine processor subset of HARTS. One advantage of performing experiments on HARTS is that it includes multiple interconnect technologies. In addition to the fiber optic switch, all of the HARTS processors are connected by a dedicated local Ethernet. Using the Ethernet, we are able to emulate interconnection topologies that can not be implemented using the switch. In this experiment we chose to connect the processors in a nine node wrapped square mesh, as shown in figure 13. We fixed the probability of node failure at 25%. Selecting such a high figure allows us to test the algorithm under worse than expected conditions. The values of the other parameters were selected to be: c = 50%, 65%, 80%, and 90%; f = 1, 10, 20; r = [1..20]. We ran 500 iterations of the diagnosis algorithm with each combination of these parameters, for a total of 120,000 iterations.

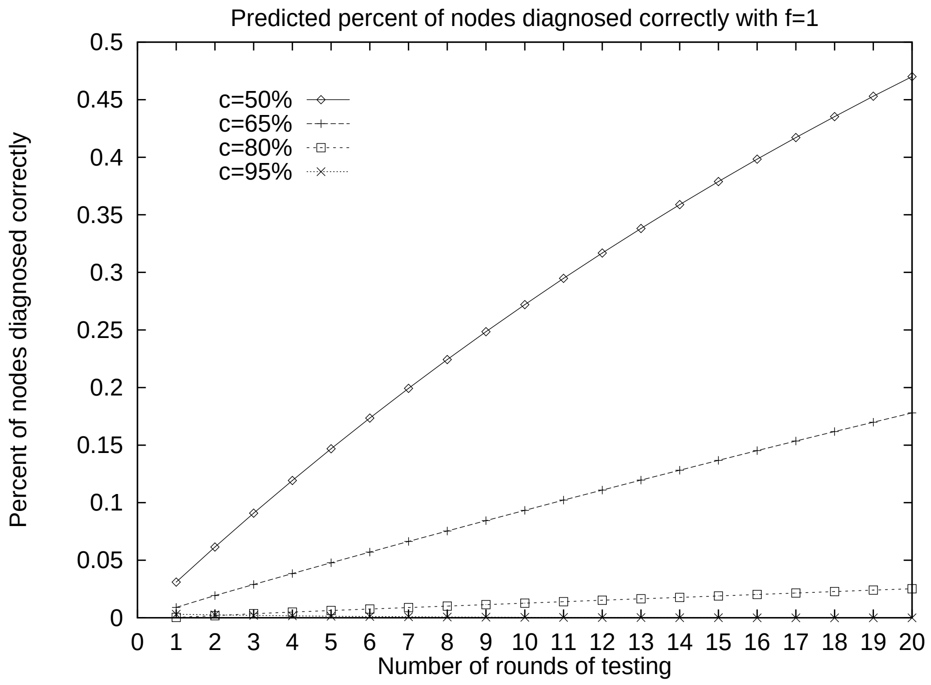


Figure 15: Percent of nodes diagnosed correctly with 1 failure mode, predicted

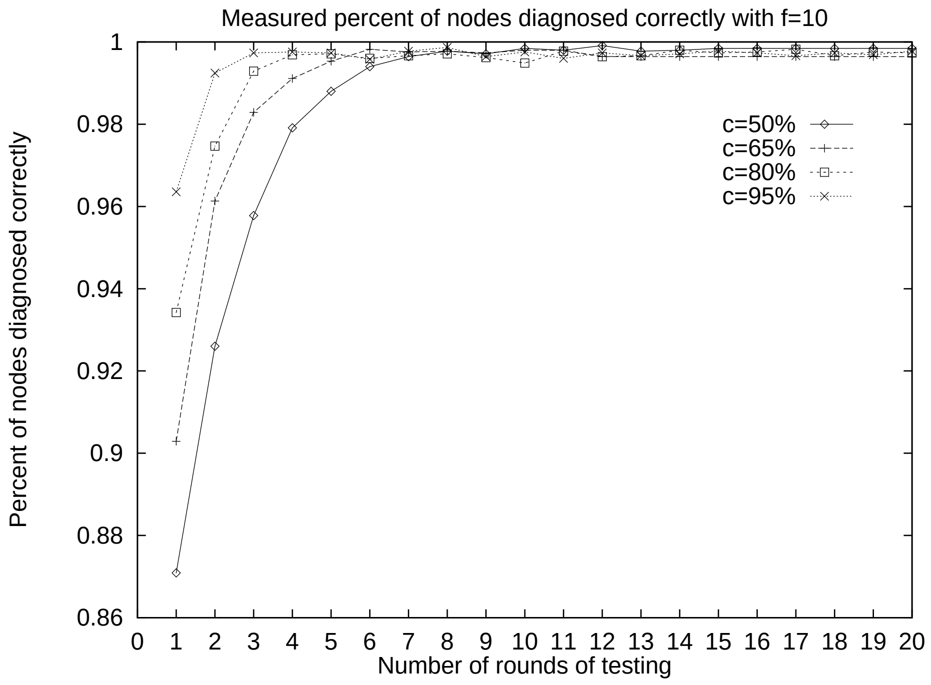


Figure 16: Percent of nodes diagnosed correctly with 10 failure modes, measured

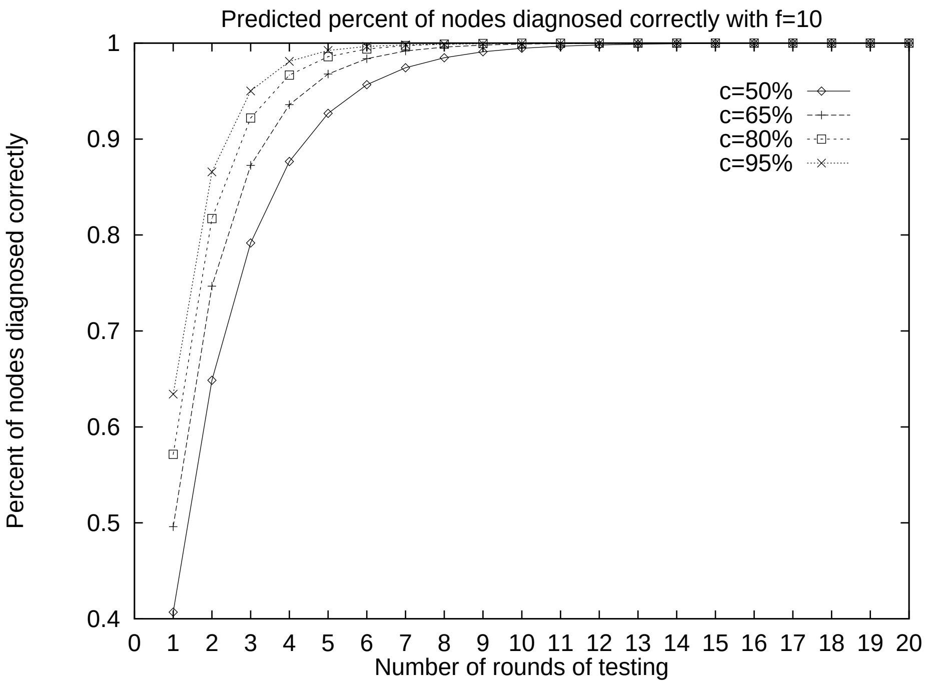


Figure 17: Percent of nodes diagnosed correctly with 10 failure modes, predicted

The results of these experiments are summarized in figures 14 through 17. There are a number of interesting observations to be drawn from this data. The first thing to notice is that in almost all cases, the measured diagnostic accuracy of the algorithm exceeded that predicted by the probabilistic model in [30], in many cases by a significant percentage. This is because the model makes a number of pessimistic assumptions, and therefore predicts only the worst case performance of the algorithm. As a result, this distributed diagnosis algorithm may actually be appropriate for use in more systems than might be expected based only on the probabilistic model. As we see in Figure 16, using tests with 10 failure modes, even with interprocessor test coverage as low as 50%, the algorithm achieves nearly 100% correct diagnosis within 7 rounds of testing. When the test coverage is 95%, only 3 rounds are required to reach 100%. As predicted by the asymptotic analysis of the algorithm in [30], both the measured and predicted diagnostic accuracy converge to 100% as the number of tests increase, but the measured accuracy starts much higher, and converges more quickly than predicted.

One other interesting observation can be made by comparing the graphs in Figures 14 and 15 to those in Figures 16 and 17, respectively. In the cases where f, the number of failure modes, is 1, we observe that the accuracy of the diagnosis actually improves as the interprocessor test coverage decreases. This is because, when f=1, the faulty processors will always match when comparing their results with other faulty processors, and thus will be more likely to diagnose themselves as correct when the test coverage is high. This effect appears both in the predicted and observed behavior of the algorithm. When f is increased to 10, this effect disappears. These results indicate that tests with simple binary (e.g. good/bad) results are not a good choice when using comparison-based distributed diagnosis algorithms.

5 Conclusion

In this paper, we have proposed a design methodology for building an integrated flexible fault-injection environment, and discussed the portability of this tool set. The proposed methodology was implemented on a real-time distributed system, HARTS, and extensive experiments conducted, demonstrating its power and utility.

The initial work on SFI described in [31] has been expanded significantly. We implemented a wide range of fault type and injection options, and developed an injection control mechanism as well as a data collection mechanism which minimizes the interference with the objects under test. Addition of synthetic workloads using SWG facilitates experiments for evaluating the dependency of performance and dependability on the executing workload. An important contribution of DOCTOR is its consideration of portability issues, an essential requirement to eliminate/reduce the excessive duplication of effort and cost.

To demonstrate the portability of DOCTOR, we intend to port this tool set to additional distributed systems. We also plan to measure the fault injection and data collection overhead added by our tools, and will explore new methods of reducing and controlling this overhead. In addition, we are currently exploring the issues involved in formalizing both the specification of fault injection experiments, and the systematic selection of the faults to be injected.

References

[1] K. G. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer*, vol. 24, no. 5, pp. 25–35, May 1991.

[2] J. C. Laprie, "Dependability: Basic concepts and terminology," IFIP WG 10.4, October 1990.

[3] K. G. Shin and Y. H. Lee, "Measurement and application of fault latency," *IEEE Trans. Computers*, vol. C-35, no. 4, pp. 370–375, April 1986.

[4] G. Finelli, "Characterization of fault recovery through fault injection on ftmp," *IEEE Trans. Reliability*, vol. 36, no. 2, pp. 164–170, June 1987.

[5] J. Arlat, Y. Crouzet, and J.-C. Laprie, "Fault injection for dependability validation of fault-tolerant computing systems.," in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 348–355, June 1989.

[6] U. Gunneflo, J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 340–347, June 1989.

[7] G. Choi, R. Iyer, and V. Carreno, "Simulated fault injection: A methodology to evaluate fault tolerant microprocessor architectures," *IEEE Trans. Reliability*, vol. 39, no. 4, pp. 486–490, October 1990.

[8] K. Goswami and R. Iyer, "Simulation of software behaviour under hardware faults," in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 218–227. IEEE, 1993.

[9] E. Czeck and D. Siewiorek, “Effects of transient gate-level faults on program behaviour,” in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 236–243. IEEE, 1990.

[10] Z. Segall et al., “Fiat – fault injection based automated testing environment,” in *FTCS-18*, pp. 102–107, 1988.

[11] R. Chillarege and N. S. Bowen, “Understanding large system failures — a fault injection experiment,” in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 356–363, June 1989.

[12] G. Kanawati, N. Kanawati, and J. Abraham, “FERRARI: A tool for the validation of system dependability properties,” in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 336–344. IEEE, 1992.

[13] K. Echtle and M. Leu, “The EFA fault injector for fault-tolerant distributed system testing,” in *Workshop on Fault-Tolerant Parallel and Distributed Systems*, pp. 28–35. IEEE, 1992.

[14] T. Dilenno, D. Yaskin, and J. Barton, “Fault tolerance testing in the advanced automation system,” in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 18–25, 1991.

[15] D. L. Kiskis, *Generation of Synthetic Workloads for Distributed Real-Time Computing Systems*, PhD thesis, University of Michigan, August 1992.

[16] J. Meyer, “Performability: a retrospective and some pointers to the future,” *Performance Evaluation 14, North-Holland*, pp. 139–156, 1992.

[17] *IV-3207 VMEbus Single Board Computer and Multiprocessing Engine User's Manual*, Ironics Inc., 1991.

[18] *VME CIM 250 Reference/User's Manual*, Ancor Communications Inc., 1992.

[19] *CXT 250 16-Port Switch Installer's/User's Manual*, Ancor Communications Inc., 1993.

[20] J. Dolter, S. Daniel, A. Mehra, J. Rexford, W. Feng, and K. G. Shin, “Spider: Flexible and efficient communication support for point-to-point distributed systems,” Technical report, University of Michigan, October 1993.

[21] K. G. Shin, D. Kandlur, D. Kiskis, P. Dodd, H. Rosenberg, and A. Indiresan, “A distributed real-time operating system,” *IEEE Software*, pp. 58–68, September 1992.

[22] *pSOS+/68K User's Manual*, Integrated Systems Inc., 1992.

[23] N. C. Hutchinson and L. L. Peterson, “The x-Kernel: An architecture for implementing network protocols,” *IEEE Trans. Software Engineering*, vol. 17, no. 1, pp. 1–13, January 1991.

[24] P. Ramanathan, D. Kandlur, and K. G. Shin, “Hardware-assisted software clock synchronization for homogeneous distributed systems,” *IEEE Trans. Computers*, vol. C-39, no. 4, pp. 514–524, 1990.

[25] D. Haban and D. Wybranietz, "A hybrid monitor for behavoir and performance analysis of distributed systems," *IEEE Trans. Software Engineering*, vol. 16, no. 2, pp. 197–211, Februay 1990.

[26] D. Heller, *Motif Programming Manual*, O'Reilly & Associates Inc., 1991.

[27] A. Mahmood and E. McCluskey, "Concurrent error detection using watchdog processor-a survey," *IEEE Trans. Computers*, vol. C-37, no. 2, pp. 160–174, 1988.

[28] S. Tomas and J. Shen, "A roving monitoring processor for detection of control flow errors in multiple processor systems," in *Proc. Int'l Conf. on Computer Design: VLSI Comput.*, pp. 531–539, 1985.

[29] M. Barborak, M. Malek, and A. Dahbura, "The consensus problem in fault tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, June 1993.

[30] D. Fussell and S. Rangarajan, "Probabilistic diagnosis of multiprocessor systems with arbitrary connectivity," *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 560–565, 1989.

[31] H. Rosenberg and K. G. Shin, "Software fault injection and its application in distributed systems," in *Proc. Int'l Symp. on Fault-Tolerant Computing*, pp. 208–217. IEEE, 1993.