

# **Fault-Tolerant Systems in A Space Environment: The CRC ARGOS Project**

Philip P. Shirvani and Edward J. McCluskey

CRC Technical Report No. 98-2

(CSL TR No. 98-774)

December 1998

## **CENTER FOR RELIABLE COMPUTING**

Computer Systems Laboratory

Departments of Electrical Engineering and Computer Science

Stanford University, Stanford, California 94305

### **Abstract**

This report describes the ARGOS project at Stanford CRC. The primary goals of this project are to collect data on the errors that occur in digital integrated circuits in a space environment, to determine the tradeoffs between fault-avoidance and fault-tolerance, and to see if radiation hardening can be avoided by using fault tolerance techniques. Our experiments will be carried out on two processor boards on the ARGOS experimental satellite. One of the boards uses radiation-hardened components while the other uses only commercial off-the-shelf (COTS) parts. Programs and data can be uploaded to the boards during the mission. This capability allows us to evaluate different software fault-tolerance techniques.

This report reviews various error detection techniques. Software techniques that do not require any special hardware are discussed. The framework of the software that we are developing for error data collection is presented.

**Key Words and Phrases:** ARGOS, fault tolerance, control flow error detection, software error detection, COTS in space.

1. Introduction.....	1
2. The CRC ARGOS Project .....	4
2.1 Experiment Setup.....	4
2.2 Goals.....	6
2.3 Features of the Processor Boards .....	7
2.3.1 Components.....	7
2.3.2 Interface .....	9
2.3.3 Fault Tolerance Features.....	9
2.3.4 Operating System.....	10
2.4 Fault-Tolerance Software.....	11
3. Error Detection Techniques.....	14
3.1 Control Flow Error Detection.....	15
3.2 Execution Time and Address Methods.....	21
3.3 Pure Software Methods.....	22
3.3.1 Block Signature Self-Checking & Error Capturing Instructions.....	22
3.3.2 Executable Assertions.....	23
3.3.3 Stutter Step Mode .....	25
3.4 Summary .....	29
4. Fault Injection .....	30
5. Summary.....	33
Acknowledgments.....	34
References.....	35

# 1. INTRODUCTION

Space missions require systems that can operate reliably for long periods with little or no maintenance. In the harsh environment of space [Ritter 90], this is only possible if the systems are designed to either (1) prevent the occurrence of failures by using shielding, radiation-hardened components, highly-reliable parts, etc., or, (2) tolerate failures so that the system operation continues undisturbed in their presence. The decision of which strategy, or combination of strategies, to adopt for a particular design is presently made very informally. We hope to contribute to improving this decision-making process. The objective of the research described here is to gather data (1) on the occurrence of disturbances in electronic equipment in actual space missions as well as the effects of these disturbances and (2) on the efficacy of various fault-tolerance techniques.

Our general approach focuses on space missions involving equipment that combines the two basic approaches of fault avoidance and fault tolerance along with facilities to detect and record the occurrence of any errors. Very little data on the effectiveness of fault tolerant computing in the space environment currently exists, and without data it is difficult to make decisions about the appropriateness and effectiveness of various fault-tolerance schemes. What types of disturbances actually occur, and how often do they occur, in a space environment? Which schemes detect those errors? These are the types of questions that need to be answered if reliable systems are to be designed.

The terms used in this report in relation to fault tolerant computing are defined as follows. A *defect* is the physical anomaly present in a device that may or may not cause a failure. A *failure* is the deviation of a device from the specified characteristic. A *fault* models the effect of failure on logical signals. An *error* is the manifestation of a fault within a program or data structure. *Transient* errors occur in the system temporarily and are usually caused by interference. *Permanent* errors happen when a part fails for good and needs to be replaced. *Fault-tolerant computing* is the correct execution of a specified algorithm in the presence of failures [Siewiorek 92]. The errors that are caused by these failures can be overcome by the use of *redundancy*.

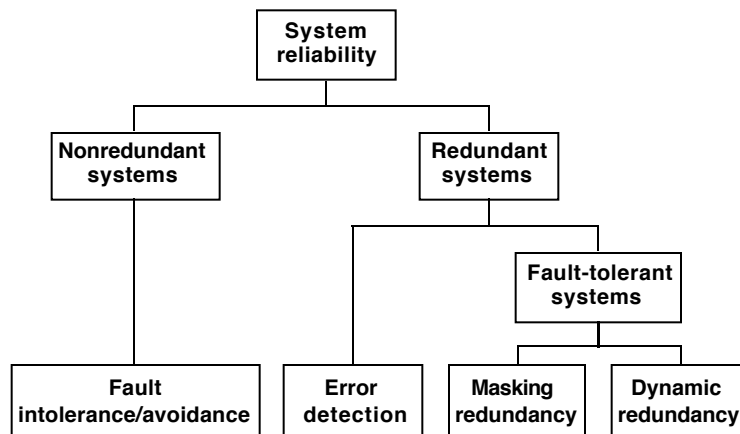
Redundancy can be either temporal (repeated in time) or physical (replicated hardware or software). The redundant information that is produced in either method can be used to detect and possibly correct errors in the outputs of the system. By observing an inconsistency among the outputs, we can detect errors. With enough redundancy, errors can be corrected, the system can be reconfigured or the errors can be masked and correct operation can be continued. Redundancy translates to more cost. In addition, more

components consume more power, which can be a scarce resource (e.g., in space applications).

There are different strategies in designing a reliable system (Fig. 1.1) [Siewiorek 92]:

- 1- *Fault-avoidance* – attempts to reduce the probability that a fault occurs by using conservative design practices, parts with high reliability, special fabrication techniques, radiation hardening, and shielding,
- 2- *Error-detection* – attempts to detect the errors so that the system can be stopped from producing erroneous outputs and a repair procedure can be initiated,
- 3- *Fault-tolerance* – attempts to add enough redundancy to keep the system operational. It either masks the results of faults and simply ignores their occurrence (masking-redundancy), or reconfigures itself to bypass the faulty part (dynamic-redundancy).

Even with the most thorough fault-avoidance schemes, a system can still experience faults. Many systems are designed with a combination of the three approaches.



**Figure 1.1** Strategies in designing a reliable system.

This report presents an overview of the CRC ARGOS project and our objectives in this project, plus a summary of the studies and some of the work done during two years of research. The CRC ARGOS project involves fault tolerance experiments conducted on a couple of processor boards on board the ARGOS experimental satellite. The goal of this project is to collect data on the errors that occur in microprocessors in a space environment, to determine the tradeoffs between fault-avoidance and fault-tolerance, and to see if radiation hardening can be avoided by using fault tolerance techniques.

In Sec. 2, we will talk about the ARGOS project in detail. In Sec. 3, we review some error detection techniques with emphasis on control flow error detection using

signature analysis. Some software error detection techniques that are suitable for our project are presented.

One way to evaluate the fault tolerance techniques implemented in a system is fault injection — as opposed to putting the system in its real environment and waiting for a real fault to happen. Disturbing the signals on the pins of the ICs, radiation, power supply disturbance, and logic simulation are the main fault injection methods. These methods are briefly reviewed in Sec. 4. Section 5 summarizes the report.

## 2. THE CRC ARGOS PROJECT

This project is an experiment that will be carried out as part of the NRL-801: Unconventional Stellar Aspect (USA) experiment on the Advanced Research and Global Observations Satellite (ARGOS). The USA Experiment, one of the eight experiments that will fly on the ARGOS satellite, is primarily a low-cost X-ray astronomy experiment. The opportunity to perform some experiment in fault-tolerant computing has evolved out of both the need for a processor to analyze the X-ray data on-board and autonomous navigation —to make the navigation of the space vehicle independent of the ground station. With processor boards available on board the satellite, the opportunity to gather data on faults and fault-tolerance is excellent.

Section 2.1 explains the experiment setup. In Sec. 2.2, we will discuss the goals of our research in ARGOS. The specifications of the processor boards are presented in detail in Sec. 2.3. Section 2.4 presents the software that we plan to develop and run on the processor boards for collecting error data.

### 2.1 Experiment Setup

The objective of the computer testbed in the USA Experiment on ARGOS is the comparative evaluation of approaches to reliable computing in space, including radiation hardening of processors, architectural redundancy and fault tolerance software technology. These goals are met by flying processors and comparing performance on orbit during the ARGOS mission. The experiment utilizes two 32-bit processors, the RH3000 and the IDT3081. Each of the processor modules is integrated as one double-sided 6U VME board containing the processor chip set, EEPROM, and 2M bytes of RAM. The Hard board, built around the Harris RH3000 radiation-hardened chip set, features a self-checking pair configuration [Harris 93]. The COTS board, built around the 3081 microcontroller from IDT, uses only Commercial Off-the-Shelf (COTS) components. Both boards have access to the full downlink science telemetry stream, and the COTS board has a direct connection to the raw science data collected from the X-ray detector. Data can be downloaded and uploaded on both boards during the mission. The indications are that there will be sufficient power to operate both the COTS board and the Hard board simultaneously on orbit. This means that we will be able to carry out the first example of a so-called "McCluskey test", i.e., the simultaneous operation of commercial and hardened processors

of the same class in the same orbital environment. It will be possible to uplink software and test fault tolerance technology in either of the processors.

Earlier experiments to gather fault-tolerance data have been limited in their scope. They either implemented only one fault-tolerance technique and collected very limited data [Takano 91], or they artificially injected faults [Miremadi 95b] [Shaeffer 92] [Kaschmitter 91] [Kaul 91] [Worley 90] [Hass 89] [Berger 85] [Koga 84] that may not fully represent the condition in an actual space environment (these artificial techniques are briefly discussed in Sec. 4). ARGOS has a Sun-synchronous, 450 mile altitude orbit with a mission life of three years. A variety of radiation environments are encountered during this mission, providing a rigorous test.

Radiation —such as alpha-particles, cosmic rays and solar wind flux— is a major cause of transient faults in electronic systems used in space. For example, an alpha-particle can change the logic value of a node inside an integrated circuit [Lantz 96]. Such errors are called single-event upsets (SEUs) [Messenger 91]. SEUs are the main type of errors that we are expecting to see in ARGOS.

The boards will be running programs and collecting data on the errors that occur during the mission. The programs will have software fault-tolerance techniques added to them. In this research, we will implement multiple techniques, with the ability to modify some techniques in flight, and will gather data in an actual space environment thereby avoiding the necessity of relying on questionable fault injection. The data gathered from this experiment will help in making decisions about the effectiveness of various fault-tolerance techniques.

We can not change any hardware feature of the boards before or during the mission (except for limited changes in an FPGA), but the software routines can be modified during the mission, allowing us to adapt techniques to the data received. Hence, despite the fixed hardware of the boards, there is still much room for experimentation with the software fault-tolerance techniques. The hardware on the Hard board uses many circuit level and system level error detection methods. Our effort is to use the available methods as well as introduce additional software techniques and compare their effectiveness. The COTS board does not have any hardware fault-tolerance features. Therefore, software techniques will be used for detecting errors. We will be able to determine the tradeoffs between fault-avoidance and fault-tolerance by comparing the behavior of these two boards.

Further explanation of our research and its goals are presented in Sec. 2.2.

## 2.2 Goals

Our goal in this project is to collect as many in-flight transient errors as possible, exercise different error detection techniques and finally come up with an efficient blend of techniques suitable for space applications. To reach this goal, we have studied available hardware and software fault tolerance techniques and we plan to come up with some new techniques along the way. There are many features incorporated into the processor boards and we plan to take full advantage of them. The programs that we will run on the boards will have additional error detection techniques, e.g., Stutter-Step Mode execution (executing pieces of code two or more times), assigned signature control flow checking, and assertions (checking the validity of data at different points), to name a few. These programs will try to exercise all the circuitry on the processor board, collect information on the errors, store them in a redundant format and use the telemetry system of the satellite to send them to the ground. A local program will receive this data and put it in a database for analysis.

Some of the areas in which we hope to gather data are discussed below. This is not an exhaustive discussion, but it should give an idea of the experiment's goals.

**Logging of disturbances and SEUs detected during flight:** Correlation may be made between the type and frequency of the detected anomalies with the orbital position, the position in the magnetic and radiation belts of Earth, and solar flares that will occur during the mission.

**Occurrence of common-mode failures:** Many fault-tolerant schemes detect errors by using a self-checking pair (SCP): two processors running in lock-step and comparing outputs (implemented in the Hard board). This method assumes that common-mode errors do not occur, i.e., a fault will not cause the same error in both processors or in the comparison circuitry. By using fault-tolerance schemes that do not depend on this assumption (examples given in Sec. 3.3), faults missed by the SCP may be detected.

**Effectiveness of radiation-hardened hardware in a space environment:** It is currently believed that by using special fabrication processes, susceptibility to SEUs is eliminated. Error detection techniques can be used to collect data on the effectiveness and necessity of radiation-hardening.

**Effectiveness of software fault-tolerance schemes:** By employing both hardware and software fault-tolerance schemes in a redundant manner, comparisons may be made on the relative effectiveness of different techniques.

**Effects of SEUs on microprocessors:** It is known that SEUs do affect hardware systems in space. By using multiple methods to detect errors, data can be

collected on what errors SEUs cause in microprocessors. A better understanding of the mechanisms involved will benefit the design of fault-tolerance techniques.

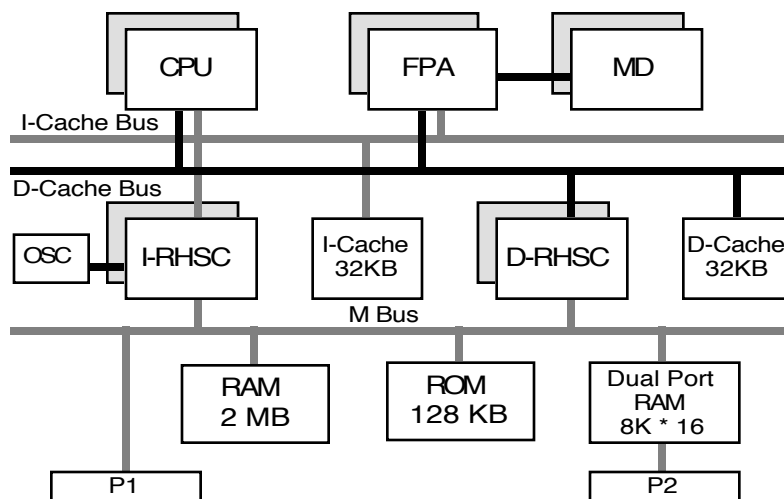
**Reconfigurable Logic:** FPGAs will be another part of our project. The COTS board has a Xilinx 4003 FPGA [Xilinx 96] that can be reprogrammed during the mission. We will use this feature for testing the FPGA, testing other parts of the system if possible and tolerating the faults occurring in the FPGA. Using FPGAs adds a lot of flexibility to the system, and in the meantime, it will be a good opportunity to test these devices in a space environment.

## 2.3 Features of the Processor Boards

Here is a summary of the configuration of the processor boards. In Sec. 2.3.1, the hardware components on the boards are described. Section 2.3.2 presents the interface specification of the boards to the rest of the satellite. Fault-tolerance features of the Hard board are summarized in Sec. 2.3.3. Finally, Sec. 2.3.4 talks about the operating system of the boards and some of its useful features.

### 2.3.1 Components

The major components used in the RH3000 processor module are (Fig. 2.1) [STI 94] [Harris 93]:

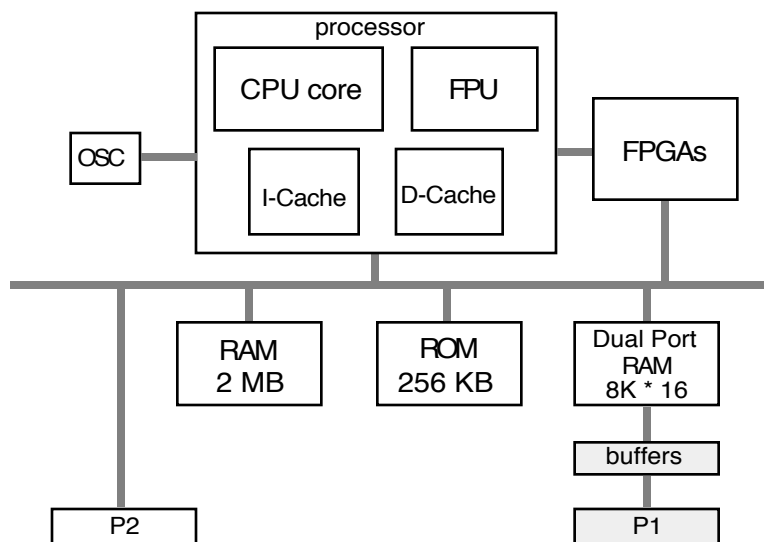


**Figure 2.1** Block diagram of the RH3000 processor module (Hard board).

- RH3000 CPU which runs at 10MHz and is software compatible with the MIPS R3000 microprocessor.

- RHFPA (Floating Point Accelerator) and RHMD (Multiplication and Division).
- RHSC cache controller. There are separate controllers for the instruction cache (I-Cache) and the data cache (D-Cache).
- 32KB I-Cache, 32KB D-Cache made out of 32K×8 SOI (Silicon On Insulator) SRAM. Due to some unresolved design bugs, these caches are currently disabled.
- Memory is composed of 2MB of SRAM for local memory and 128KB of EEPROM that stores the boot program and the operating system in a compressed format. An IDT7025 provides 16KB of I/O implemented as a dual port RAM, and 16 bytes of I/O for semaphores. The EEPROM has a parallel programming interface via an external test adapter interface module that connects to the P2 connector. It contains power-on/reset initialization routines and diagnostic routines for the CPU functional test, FPU functional test, and tests for memory, wait-state, EDAC, cache refill/invalidate, timers and rollback buffers. The 28C256 is used for the EEPROM. It has an internal timer for write (program) time (~10ms) and auto-erase-before-write features.

Figure 2.2 shows the block diagram of the COTS board. All the components on this board are commercial off-the-shelf, so they are not radiation-hardened.



**Figure 2.2** Block diagram of the IDT3081 processor module (COTS board).

The IDT3081 microcontroller contain an R3000 CPU core, an R3010 FPU, 8KB of instruction cache and 8KB of data cache with the corresponding controllers. There is the same amount of memory on this board as there is on the Hard board. It also uses the

IDT7025 for the dual-port memory. This board contains a Xilinx XC4003 FPGA that can be reprogrammed during the mission.

### **2.3.2 Interface**

Each processor module has the following interface specification for communicating with the rest of the system:

It has a VME bus that interfaces through a 16KB dual-ported RAM. It complies with the IEEE 1014 backplane standard that has two 96-pin connectors (P1 and P2 in Fig. 2.1 and 2.2). However, it does not use the VME bus pin assignment. It has a read/write rate of 666KB/s.

The dual ported RAM (IDT7025) provides 16KB of addressable space for buffering data messages, plus eight 16-bit semaphore locations. It also provides an interrupt signal from bus master to the processor. The VME bus controller can send an interrupt signal to the processor by writing to a fixed address in the dual-port memory. The processor resets it by reading a similar fixed address.

The reads are partial (16-bit) word. The writes are implemented as a read-modify-write operation.

### **2.3.3 Fault Tolerance Features**

Several fault tolerance techniques have been implemented in the Hard board. The processor chip set (RH3000 CPU, RHFPA, RHMD, I-RHSC and D-RHSC) is radiation-hardened. These chips are duplicated and act as self-checking pairs. They run in lock-step. There is a master/slave control line on one of the connectors to set the active unit in the self-checking pair (a reconfiguration capability); 0=active, 1=shadow. The active chip (master) drives the outputs and the shadow chip (slave) reads them and compares them to the outputs it has produced. The outputs from the slave are suppressed and do not go on the bus. On error, a 2-rail miscompare signal is generated; the miscompare signals merge in the RHMD chip and the RHMD drives a 2-rail signal to the I-RHSC, which generates a restart event (module reset and microboot). Finally, software initiates a recovery operation in response to a restart event. The miscompare signals are encoded in 2-rail format. Therefore, a single error on these lines will not cause an erroneous miscompare signal.

Main memory is protected with a single-byte-error-correcting double-byte-error-detecting (SBC-DBD) code, where each byte is 4 bits. A spare column is also provided such that an area of memory can be deselected in case of a permanent failure.

Interfaces between chips are checked by parity bits, generated at the outputs and checked at the inputs every clock cycle. A parity error results in an “abort” event that is a maskable high priority exception.

Rollback buffers are provided to store old data and the address being written by a store instruction to help with error recovery (“restore” operation).

There is a set of diagnostic tasks in the EEPROM for testing various parts of the system (Sec. 2.3.1). They are executed on power-up and can be called by the programs running on the board.

Several user-programmable timers, as well as a watchdog timer, are provided to do time checking. There are eight user programmable timers used as four pairs. Each pair can have three modes: 32-bit counter, 2×16-bit counter or 16-bit counter with auto-reload. In addition, a 32-bit watchdog timer generates a reset event (microboot) on time-out.

There are several health and status semaphores: heartbeat (indicates that system is up and running), SEU (Single Even Upset) counter, task switching and task exception counter, and a counter for the number of messages sent and received.

A memory scrubbing technique has been implemented in the system. It acts similar to the refresh mechanism in DRAM memories and scans all memory locations periodically but passes the contents through the error detecting and correcting (EDAC) logic before writing it back to memory. Therefore, single errors are corrected and the probability of double errors is reduced.

There is no hardware fault tolerance technique implemented in the COTS board. Therefore, we are restricted to software techniques on this board. It is our goal to implement software fault tolerance techniques on the two boards and compare their performance in the same radiation environment.

#### **2.3.4 Operating System**

A customized version of the VxWorks operating system is used in both processor boards. Several system routines have been provided for memory load, checksum and dump that are used for uploading data or program and downloading the telemetry data. VxWorks and the existing flight software provide task management routines, e.g., exception handling, task control (running and aborting different tasks), link and unlink of object codes, and task response telemetry. As explained in Sec. 2.3.1, there is a set of diagnostic tasks that run on power-up. Each test can be masked to stop it from executing. For memory tests, the test area can be changed. At the end of the diagnostic tasks, results

of the tests are sent to the ground. On reboot, a self-test program is run to see why the system was rebooted.

Synchronization of tasks is done by using flags and by synchronous and asynchronous message passing using mailboxes.

The operating system provides dynamic linking capability both for library functions and for user functions. That means new versions of functions and libraries can be uploaded on the boards, on-the-fly. By linking the programs again, the programs will start using the new functions without having to upload the programs. We intend to make full use of this feature as will be explained in Sec. 2.4.

With the above background on the hardware and the operating system, we now look at our plans for the software that we are going to implement and run on the boards to enhance the error detection coverage and to collect error data.

## **2.4 Fault-Tolerance Software**

A set of programs is being written to exercise the two processor boards and collect data on the errors that occur in them. The collected data should be safely stored along with the corresponding diagnostic information. One constraint on our program is the amount of memory available to us. There is only 2MB of memory on each board. This memory is shared by the operating system's code and programs that are run by other researchers involved in the ARGOS project.

The other constraint is the communication bandwidth between Earth and the satellite. The uplink bandwidth is 1.1 kbps (kilo bits per second). The downlink bandwidth is switchable between 40kbps and 128kbps, which will be selected depending on the atmospheric conditions. As mentioned before, the satellite has a Sun-synchronous orbit. It rotates around Earth once every 101.6 minutes, out of which only 8 minutes can be used for transmissions (when it is above the ground station). This translates to about 64KB of uplink per rotation that is shared between the satellite command and the researchers.

As explained in Sec. 2.3, the operating system provides dynamic linking facilities. We have designed the framework of our programs such that they can be upgraded incrementally. If a new version of a function is written or one function needs to be uploaded because an error has corrupted its code, there will not be any need to load the whole program again. That function will be temporarily disabled and the memory assigned to it will be freed. Once the code is uploaded, it is linked properly and enabled again.

Here are the classes of onboard programs that are being written:

- *Main Control*: The main program will control all the programs that are explained below. It will have a command interpreter that will process the commands sent from Earth for adding or deleting functions, changing the parameters of the program, and requesting a re-transmission of download data.
- *Collector*: This is the program that will collect all the errors detected by hardware detection mechanisms and by the software mechanisms that we will add. This data along with its diagnostics information will be *safely* stored (multiple copies or error correcting codes) and given to the Telemetry program for transmission. This program may have to do some data compression during the periods of time that telemetry can not be done due to the position of the satellite.
- *Diagnostic*: Upon error detection, a program will try to diagnose the error to find the source that caused it and the mechanism that detected it. This complete information will be given to the Collector to be stored and sent to the ground.
- *Profiler*: A program will keep track of the time slots spent in each program. This is needed because the operating system, as well as the programs from other researchers, will be running on the boards at the same time. This data will be provided by the operating system of the module and will be organized here and given to the Telemetry program for transmission. This information will be used in the analysis of the data gathered by the Collector program. The errors detected during the execution of other programs will be reported to ground independent of our programs, but will be included in our error data analysis.
- *Telemetry*: All of our uploads and downloads will be done through this program. It will use system routines for the actual communications. It will be able to add some extra error correction codes to the data if necessary.
- *Computation*: Vulnerability of each functional unit can be estimated if we exercise it so that a possible fault will manifest itself as an error and appear at a point that can be detected by one of the detection mechanisms. Several SPEC benchmarks, Power-On Self-Test (POST) codes, and sample codes from DSP applications, e.g., FFT algorithms, are being put together to exercise all functional units of the system. This program will incorporate all the previously mentioned detection techniques. Most of the time slots allocated to us will be spent in this set of programs.

The software explained above will be run on the two boards simultaneously. We will try to keep the programs on the two boards identical so that the test conditions are as close as possible.

In addition, a program will be written that will run on the ground. Its job will be receiving all the transmitted data, checking their integrity, logging them and putting them in

appropriate format to be fed into a database or spreadsheet for analysis. We assume that the programs needed for remotely reconfiguring the FPGA on the COTS board will be available as system maintenance software.

Our plan is to implement as many software fault-tolerance techniques as possible and compare the effectiveness of the different techniques. There are two basic steps in adding software fault-tolerance to a program: determining what information is going to be checked for correctness and determining how that information is going to be checked. In order to achieve fault-tolerance in a program some redundancy must be added to it. This redundancy can be in the form of some information about the program that is obtained and stored along with the program during compilation. As the program is executed, the same information is recalculated and compared to the stored values. Information that can be checked includes control flow, reasonableness of data values, address and execution time, and stability. We will present several techniques for each case in Sec. 3.

The second step is determining how to check the information. These are some of the techniques that we are planning to use:

- Hardware duplication detects all the single faults but it fails to detect identical faults that happen in both units (common-mode failures). Time-redundancy will be added to the present physical-redundancy by executing each program segment twice. This will help with detecting transient common-mode failures that escape the hardware duplication technique.
- Different control-flow checking techniques will be added. The program can be modified so that it is self-checking, or a separate task can be run on the same processor to monitor the program of interest [Ersoz 85]. We will use the multitasking capability of the VxWorks operating system to run a watchdog task.
- Timers will be used to check the timing behavior of the program [Madeira 93]. This will check that each piece of code will execute within a certain amount of time.
- The watchdog timer will be used to verify that the system is up and running and not stuck in a piece of code. The program should reset this timer periodically. If it fails to do so, we assume that there has been an error in the execution.
- Assertions will be added to each function to check the validity of the input and output parameters.

Since the main goal in this project is to collect data on the occurrence of faults, coverage will have the highest priority in enhancing the error detection capabilities of the system. The memory and performance overhead will not be important in selecting different techniques. This gives us flexibility in fine tuning the techniques to our purpose.

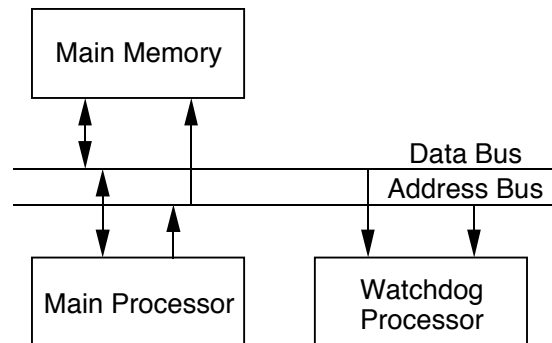
In the next section, several error detection techniques are discussed.

### 3. ERROR DETECTION TECHNIQUES

Error detection methods are divided into two classes [Mahmood 88]:

- 1- *Circuit level* (hardware), e.g., parity codes, SEC/DED (Single Error Correction / Double Error Detection) codes, residue codes and self-checking circuits.
- 2- *System level* (software and/or hardware), e.g., watchdog timer, capability-based addressing, duplication, N-version programming and watchdog processors.

As mentioned in Sec. 2.4, one method for system level error detection is to gather some information about the program during compilation and compare it with the information gathered during execution. The question here is what information can be checked and how it can be checked. Examples of things that can be checked are: control flow of the program, control signals coming out of different units, memory access behavior, and reasonableness of results. This checking can be done by writing self-checking programs [Lala 91], running a separate task to do the checking [Ersoz 85], or having a watchdog processor [Mahmood 88] to minimize the performance overhead. A watchdog processor is a small and simple processor that sits on the busses, passively observes the bus transactions generated by the main processor, and detects errors by monitoring the behavior of the system (Fig. 3.1).



**Figure 3.1** A system with watchdog processor.

We have done a survey of different system-level error-detection techniques. The following section gives a brief explanation of each of them. Only on-line error detection (error detection while the system is up and running) is discussed. Both permanent and transient errors are considered. Many of the techniques presented in the following sections use special hardware for error detection. We reviewed these techniques in this report because, even though these hardware are not available on the processor boards on

ARGOS, the concepts of these techniques are very useful in developing new pure software techniques for our project.

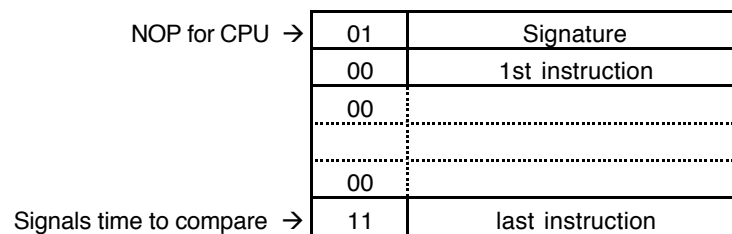
### 3.1 Control Flow Error Detection

By *control flow error detection*, we mean checking the correct sequencing of the instructions. The program is divided into *basic blocks*. A basic block is a branch-free sequence of instructions, i.e., there should be no jumps to any instruction of the block except for the first instruction and there is no jump out of the block except for the last instruction of the block. A directed graph is constructed for the program with the nodes representing the basic blocks and the edges representing the possible transitions between the blocks. This graph is called the *control flow graph*. Control flow error detection assures that blocks are executed in an allowed sequence (interblock), the sequencing of the contents of a block is correct (intra-block), or a mixture of both. The most popular techniques do signature analysis, but execution time and address information have also been used. *Signature analysis* is a method in which a signature (a specific bit pattern) associated with a block of instructions (one or more nodes in the control flow graph) and saved somewhere at compile-time. Then during run-time, the same calculation is done and the generated signature is compared with the saved one. A miscomparison indicates an error in the flow of the program. There are two types of signatures: assigned and derived. In the *assigned signature method*, each node is given an arbitrary number (with least correlation between the numbers). One technique that uses this method is called Structural Integrity Checking (SIC) [Lu 82]. It recognizes the high-level control flow structures (e.g., *if-then-else*, *for* and *while* structures) during compile-time and labels them with signatures or labels, and checks for integrity of these structures at run-time using a watchdog processor. One advantage is that the watchdog processor program is generated directly from the high-level language program by replacing the computations in the main program by “receive-label” and “check-label” instructions. The control structures remain the same. Another advantage is that the two processors (main and watchdog) can operate asynchronously without making the watchdog complex. The disadvantages are that the signatures have to be explicitly transferred to the watchdog and it only does interblock checking.

In the *derived signature method*, the signature is derived from the binary code of the instructions. This method assumes that the program is not run-time modifiable. Most signature analysis methods use derived signatures because it has higher error detection coverage and it can do both inter- and intra-block checking. It should be mentioned that

semantics or correct execution of instructions can not be directly checked by this analysis. For example, an ALU may become faulty and produce wrong sums and this will not be detected by these methods unless it affects control flow.

*Path Signature Analysis* (PSA) is a derived signature method for control flow error detection [Namjoo 82]. In the basic form of PSA, a signature is derived for every basic block. Let us assume that the binary codes of the instructions in a basic block are  $W_0, W_1, \dots, W_{n-1}$ . The initial signature, the signature at the first location in the block, is set to  $S_0$  (usually all zeros). The intermediate signature formula after instruction  $k-1$  is:  $S_k = f(S_{k-1}, W_{k-1})$ , i.e., the signature at location  $k$  ( $S_k$ ) is a function of the signature at location  $k-1$  ( $S_{k-1}$ ) and the binary bit pattern of instruction  $k-1$  ( $W_{k-1}$ ). Examples of the function ( $f$ ) are ‘xor’ and ‘addition’. The signatures are calculated at compile-time and inserted at the beginning of each basic block (Fig. 3.2). Two tag bits are used to differentiate signatures from the rest of the instructions in the code. During program execution, the same signature is generated by a watchdog processor based on the same instruction stream as it is fetched from the memory. The watchdog monitors the data bus and captures the signatures. When a signature is reached, the main processor executes a NOP instruction. A special tag signals the time to compare the computed signature with the embedded one. A mismatch generates an interrupt that initiates a recovery procedure. The time between the occurrence of the error and when the error is detected (mismatch signal in this case) is called error detection *latency*.

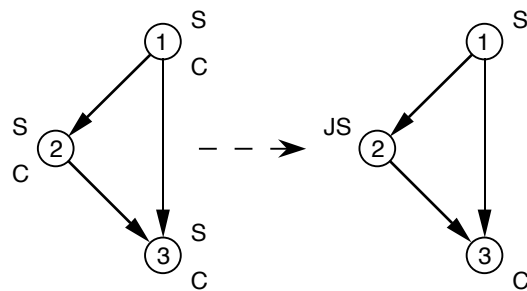


**Figure 3.2** PSA signature embedding.

One way to simplify the checking circuitry is to add signatures to the node (representative of the basic block in the control flow graph) so that the correct computed signature at the end of the node is all-one [Sridhar 82]. This will reduce a comparator to a single gate.

Since a basic block is 7 to 8 instructions long on average [Hennessy 96, Sec. 4.1], the memory overhead of this scheme is high. In generalized PSA [Namjoo 82], signatures are computed for sequences of nodes, i.e., paths rather than a single node. Paths are grouped into sets (paths starting from the same block) and each set has a single signature.

To do this, justifying signatures may have to be added to some paths in a set so that the ending signature will be the same. The left diagram in Fig. 3.3 shows the control flow diagram of three basic blocks 1, 2 and 3, each having a signature at the beginning and a checkpoint at the end. To reduce the overhead, the checkpoints at the end of blocks 1 and 2 as well as the signature at the beginning of block 3 are removed. Then the signature at the beginning of block 2 is adjusted so that the signature for path 1-3 is the same as the signature for path 1-2-3 (the right diagram in Fig. 3.3).

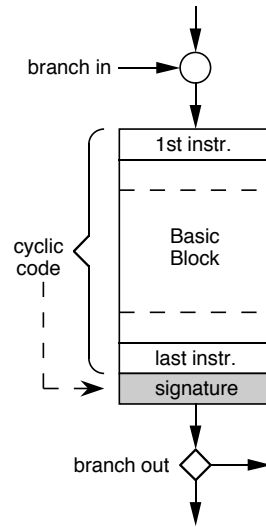


S = signature , C = check point , JS = justifying signature

**Figure 3.3** Adding justifying signature to reduce memory overhead in PSA.

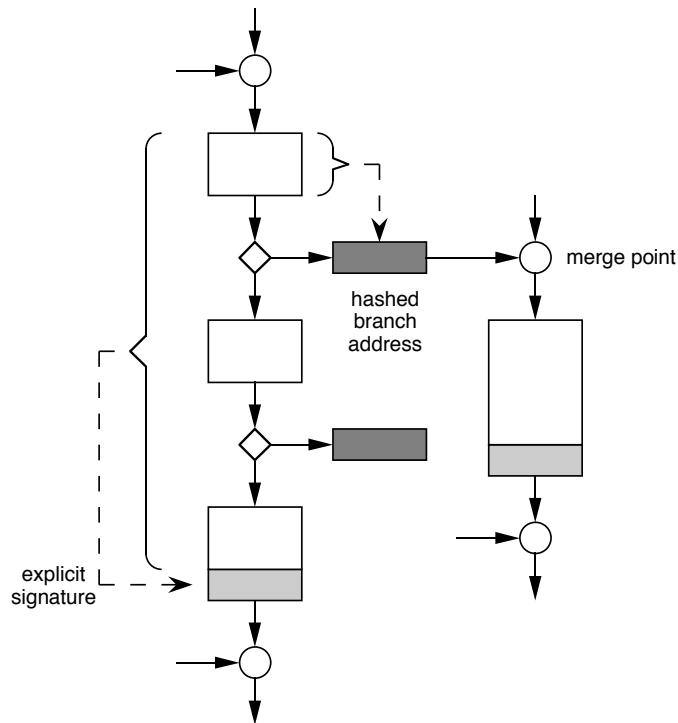
We should note that as the number of signatures reduces, the distance between the checkpoints increases, and consequently, the error detection latency increases.

In another derived signature method, called *Signed Instruction Streams* (SIS), a cyclic code (signature) is generated for every basic block at assembly time (Fig. 3.4) [Shen 83]. The signatures are inserted at the end of each block. During execution, the same signature is generated by the monitoring hardware as the instructions are fetched. When the embedded signature is reached, it is compared with the generated signature, resulting in a recovery interrupt in case of a mismatch.



**Figure 3.4** Embedding a signature for a basic block.

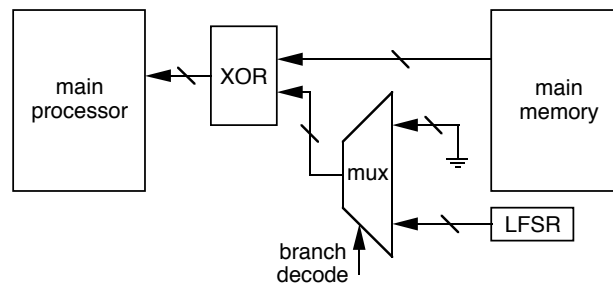
This basic scheme has a high memory overhead, just as the basic form of PSA. In order to reduce the number of signatures embedded in the code, a technique called *Branch Address Hashing* (BAH) can be used. Instead of embedding a signature before each branch instruction (as shown in Fig. 3.4), at compile time, the target address of the branch instruction is hashed with the signature accumulated up to that location (Fig. 3.5), producing an incorrect target address in the code.



**Figure 3.5** Branch Address Hashing (BAH).

At run time, when a branch instruction is encountered (recognized by the monitoring hardware), the target address of the branch is inverse hashed or rehashed using the generated signature to get the correct target address back. In case of an error, the branch address would be wrong and execution would continue from an erroneous destination. This error will be caught by the next embedded signature with some latency penalty. When a branch is not taken, the final signature calculated up to that branch is used for the initial signature of the next basic block and this procedure is continued until a merge point is reached. A merge point is the beginning of a basic block whose corresponding node in the control flow graph has more than one predecessor. Explicit signatures are inserted right before the merge points. When a branch is taken, BAH sets the current signature to  $S_0$ , i.e., the initial signature after all the merge points is  $S_0$ . Thus if a branch is taken but the processor jumps to an incorrect merge point, the error is undetectable.

Figure 3.6 shows an implementation of BAH. The branch instructions are decoded and the decode signal selects between the signature calculated by a Linear Feedback Shift Register (LFSR), or all zeros, to be XORed with the address part of the branch instruction.



**Figure 3.6** Implementing BAH.

BAH can reduce the number of signatures by 50% comparing to simple SIS, but an incorrect branch to a merge point will not be detected. A program bound detector is also needed to detect jumps to the areas in memory that are not part of the program. In systems with no memory protection scheme, a simple way to do this for the areas which are not data segments of the program (unused space), is to fill the memory with instructions that will cause an exception or interrupt [Miremadi 92]. We will look at this in Sec. 3.3. However, in modern microprocessors, hardware facilities are provided for bound detection and modern operating systems take advantage of that for memory protection.

We should also notice that in BAH the distance between the signatures (checkpoints) is increased. That increases the average error detection latency. For more details on the SIS method see [Schuette 87]. For the implementation discussed there, this approach had 10 percent hardware overhead for error detection, plus 10 percent more for error recovery.

The performance penalty was expected to be less than 10 percent. A fault injector (Fig. 4.1) was used to inject faults on the external data, address, and control busses of an MC68000 microprocessor system. SIS showed an error coverage of 98 percent for instruction-type errors (errors that changed the bit pattern of the instructions). The overall coverage for all the errors was 82 percent, which is the combined coverage of the error detection mechanisms of the MC68000 and SIS.

The asynchronous SIS method was presented in [Eifert 84]. It differs from the SIS scheme in the use of a watchdog type processor (referred to as Roving Monitoring Processor - RMP) and is capable of concurrently monitoring multiple processors in a system.

The effectiveness of a signature monitoring technique can be characterized by five properties: (1) error detection coverage, (2) memory overhead, (3) processor performance loss, (4) error detection latency, and (5) monitor complexity [Wilken 90]. Several techniques to improve these properties over the basic SIS scheme are discussed in [Wilken 89] and [Wilken 90]. In Continuous Signature Monitoring (CSM), signatures are derived for paths rather than each basic block to reduce the memory overhead. To reduce the error detection latency in this scheme, horizontal signatures are added to the instructions. Horizontal signatures give us low latency while vertical signatures give us high coverage. The word “continuous” in CSM comes from the fact that the horizontal signatures are checked *continuously* and justifying signatures maintain vertical *continuity* among maximal paths. When horizontal signatures are combined with SEC/DED bits, transient errors in the monitor are detected. This combination has some big drawbacks. It requires generating and storing the SEC/DED code with the program and the meaning of this code will be different for data and code segments in memory. Therefore, it requires a new compiler, linker, loader and memory system [Saxena 90].

The *Extended-Precision Checksum* is another derived signature method that uses a watchdog coprocessor with explicit extended instructions defined for it [Saxena 90]. With n-bit instruction words, a simple n-bit checksum code is calculated by adding the instruction words modulo  $2^n$ . In order to have the checksum and the opcode in one word, the instructions are space compacted (number of bits is reduced, as explained in [Saxena 90]) and the extended-precision checksum (addition without any loss of precision) is calculated from this compacted code. This method has a big advantage on a sequence of errors. For other signature-based method discussed above, the sequence error coverage remains relatively constant ( $2^{-L}$ , L being the length of the signature), and the latency increases as the number of errors grows. For this method, coverage approaches unity

(because we get closer and closer to zero as we are always subtracting) and latency remains bounded by the average block length (when a zero check is signaled).

The *On-line Signature Learning and Checking* (OSLC) is another method that uses special hardware to eliminate any compiler or assembler modifications [Madeira 92]. Block identification and reference signature generation are done in a normal program execution called the learning phase. Having the hardware calculating the signatures instead of the compiler, has a significant advantage. Operand size, address and control signals can be included in the signatures very easily. Including these extra inputs requires considerable complexity on the compiler side because they will have to be simulated. These extra inputs to the signature function will uncorrelate the signatures, increase the type of errors that can be detected and hence increase error detection coverage.

### 3.2 Execution Time and Address Methods

The time it takes to execute a piece of code is another property we can check for detecting errors. This timing can be exact, an estimate or an upper bound like a time-out. Address information can also be checked. For example if the size of a basic block is known, the exit address should be the sum of the start address and size of the block.

[Miremadi 95a] explains different timing and address checks. In his paper, the program is divided into two types of blocks. *Protected basic blocks (BB)* are basic blocks (a branch-free sequence of instructions) with extra instructions added to the beginning and the end of them in order to send information to a watchdog processor. To keep the memory and performance overhead low, basic blocks with fewer than a certain number of instructions are left unprotected. Instruction blocks appearing between protected basic blocks are called *partition blocks (PB)*. There are several time checks:

- *BB-timer*: Exact execution time of a BB is checked by inserting store instructions at the beginning and at the end of the BB to start and stop a timer.
- *PB-timer*: PBs are checked for an upper bound of execution time because they contain conditional branches and it is either hard or impossible to determine their exact execution time at compile time. The timer is started and stopped by the same store instructions used above.
- *WL-timer*: A traditional watchdog timer is used to check for hang-ups caused by infinite loops. This timer should be reset by the program at regular intervals, otherwise it is counted as a sign of error.

To detect the errors missed by the above timing checks, the following checks can be added:

- *BB-address*: Uses the address and the size information to create a unique tag for each basic block. The size is sent to the watchdog processor upon entry to the basic block and the watchdog processor checks for “exit\_address=start\_address + size” as mentioned earlier. This technique checks for illegal jumps between BBs.
- *Phase*: Entering and exiting a basic block should always occur in the correct order (entry followed by exit). A simple state machine checks for correct sequencing of these events.
- *CRC*: This is a checksum on instructions of basic blocks. The checksum is sent to the watchdog processor at the entry to a basic block.
- *BB-OP-counter*: This mechanism uses a counter to count the number of instructions executed in a basic block. Initialization of the counter is performed by the same store instruction that sends the precalculated signature to the watchdog processor.

More detailed implementation issues are explained in [Miremadi 95a]. Two combinations of the above techniques are compared by experiments: TTA (Time-Time-Address) and STA (Signature-Time-Address) as shown in Table 3.1.

Schemes	Mechanisms Included in the Schemes						
	BB-timer	PB-timer	WL-timer	BB-address	Phase	CRC	BB-OP-counter
TTA	✓	✓	✓	✓	✓	-	-
STA	-	✓	✓	✓	✓	✓	✓

**Table 3.1** Techniques used in TTA and STA.

The TTA method can be adapted for external monitoring of processors with internal caches and nondeterministic execution time. However, STA can not be used for external monitoring of processors with internal caches.

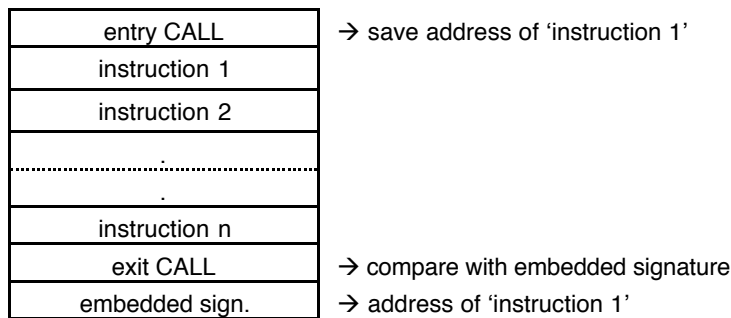
### 3.3 Pure Software Methods

If the hardware is fixed and cannot be changed, software methods have to be developed to detect errors. Two software techniques are proposed in [Miremadi 92]. These techniques are described in Sec. 3.3.1. There are other techniques that we are going to use in our project and they are explained in Sec. 3.3.2 and 3.3.3.

#### 3.3.1 Block Signature Self-Checking & Error Capturing Instructions

Block Signature Self-Checking (BSSC) is an assigned signature technique [Miremadi 92]. The program is divided into basic blocks and partition blocks as described in Sec.

3.2. Each basic block is *assigned* a signature. The signature is the address of the first instruction of the basic block. In this method, the job of the monitoring hardware or the watchdog processor is done by a monitoring subroutine (Fig. 3.7). The signature is sent to the monitoring subroutine at the beginning of the block (entry call) and stored in a local variable. At the end of the block, the same signature — which is embedded after the exit call instruction— is sent to the monitoring subroutine for comparison with what was stored in the local variable by the entry call. A miscomparison will indicate a control flow error. A drawback to this technique is that the code becomes position-dependent as the embedded signature consists of an absolute address. This technique was further improved in [Miremadi 95b] to check the control flow both on entry and on exit to a basic block. The *Block Entry Exit Checking* (BEEC) improves the detection coverage and has a position-independent code.



**Figure 3.7** Signature embedding in BSSC.

Error Capturing Instructions (ECI) is a very simple technique to detect erroneous jumps to locations of the memory that are not used during normal execution of the program [Miremadi 92]. It simply fills those locations with error capturing instructions, e.g., a divide by zero or some special software interrupt.

### 3.3.2 Executable Assertions

Executable assertions are known to be very effective in program testing, validation and fault tolerance [Andrews 79] [Ersoz 85]. Assertions are added to programs mainly for debugging during software development. For example, a null pointer may be produced because of a bug in the program. The programmer may add an assertion statement before the pointer is used to check that it is not null. During program execution, if the assertion fails the programmer can use that information to trace the bug. As another example, at some points in the program after a result is calculated, the reasonableness of that result can

be checked, e.g., checking that it is within an acceptable range. An out-of-range result indicates that either the specification or the implementation is wrong.

Assertions can be used for concurrent error detection, too. The difference is that they target different errors. The debugging assertions target programming errors. Most of these assertions are taken out in the final program to reduce the performance overhead. In concurrent error detection, the goal is to detect run-time errors due to hardware malfunctions. Therefore, while keeping the debugging assertions, more assertions are added to the program to detect the hardware errors.

In a "for" loop structure, the value of the loop counter can be checked after the loop finishes to make sure there was not a false jump out of the loop. Similar checks can be done for the "while" constructs on their loop conditions. In addition, some variables remain unchanged during the execution of a loop. These variables are called *loop invariants*. After the execution of a loop, assertions can be added to check that these values have remained unchanged.

As a range check example, the variable that is used to index into an array can be checked for being within the size of the array.

Some assertions may be application dependent: (1) After sorting a list, a quick traverse of the list can make sure that the sort algorithm was executed correctly [Saxena 94], (2) After encoding some data, the decoding function can be run on the result to see if we get the original data back. A mismatch will indicate an error in execution of either the encoding or the decoding function. In either case, an error has occurred and been detected. Assertions can also be added to ensure that the results of calculations are within valid range.

The first three examples represent cases that can be done automatically by running a preprocessing program on the source code or by modifying the compiler to add the necessary code. The application dependent examples represent cases where the assertions should be added manually by the programmer. *Algorithm-Based Fault Tolerance* (ABFT) is another example of application specific techniques where the extra computations added for error detection/correction use the special properties of the main algorithm [Huang 84].

For a discussion on writing self-checking programs using assertions see [Mili 82] [Mili 90].

### 3.3.3 Stutter Step Mode

In *Stutter Step Mode* (SSM) execution, each group of instructions is executed twice or more and the results are compared, i.e., temporal redundancy is used as opposed to structural redundancy to provide extra information for detecting errors [Ignatushchenko 94]. This software technique has, of course, a big disadvantage in performance (at least twice the normal execution time plus comparison time). It may also have memory overhead if we duplicate the data structures (as we will see in some implementations later in this section). When the primary goal is to detect the errors missed by other techniques (as is the case of our project), stutter step mode has its advantages. It can detect transient errors that do not affect the control flow of the program, e.g., ALU miscalculations. It can check for correct execution of the data transformation instructions (instructions that change the data, e.g., addition, rather than instructions that change the control flow of the program). It will also detect common mode errors when used in a dual system (which is what we have in the Hard board). Note that this technique will not detect permanent faults.

The size of the blocks of instructions that are repeated is a parameter to be chosen. At source code level, the block can be the instructions that evaluate an expression. For example, if we have:

```
x = (a×b)+(c×d);
```

in the source code, we can convert it to:

```
x1 = (a×b)+(c×d);
```

```
x2 = (a×b)+(c×d);
```

```
if (x1<>x2) error();
```

The same thing can be done when calling a function. For example:

```
x = f(a, b, c);
```

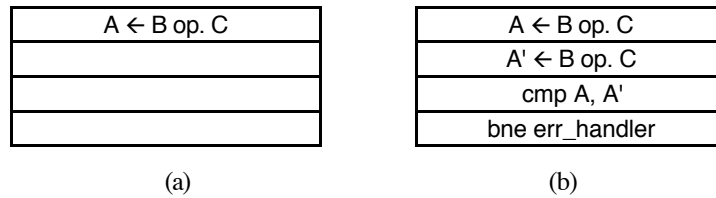
→

```
x1 = f(a, b, c);  
x2 = f(a, b, c);  
if (x1<>x2) error();
```

It should be noted that function *f* should not write to the memory addresses that it reads from, otherwise the second execution will have a different result. This simple scheme can be applied to a `const` function (a function that does not modify anything other than its own local variables) in C language. If the function does modify some non-local variables, it should be duplicated, with each instance having its own copy of the memory.

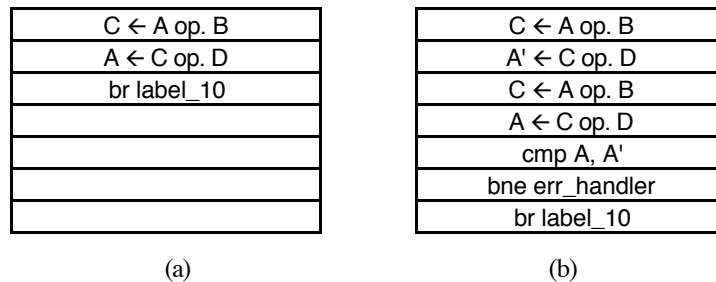
This method can be applied instruction by instruction. Fig. 3.8 (a) shows a single ALU operation that can be duplicated with a different destination register followed by compare and branch (Fig. 3.8 (b)). To be more careful, we should make sure that the

initial values of registers  $A$  and  $A'$  are different. This can be done by loading  $A'$  with the bitwise complement of  $A$ .



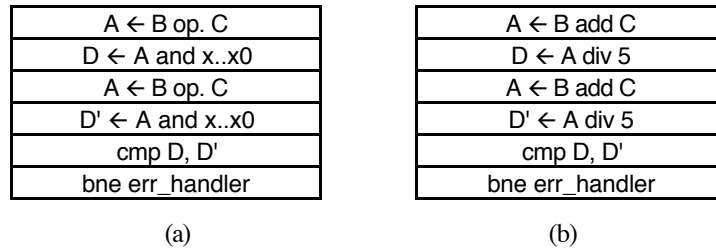
**Figure 3.8** SSM applied to one instruction: (a) original code, (b) SSM code.

The overhead of this example is 300% if a compare and branch is added for every instruction. To reduce this overhead, the duplication can be extended to a basic block (defined in Sec. 3.1). The instructions inside each basic block are duplicated, excluding its possible final branch which will remain as a single copy. Fig. 3.9 (a) shows a series of computation — only two operations shown here for the sake of simplicity — which can be duplicated, followed by a comparison of their final results (shown in Fig. 3.9 (b)). Notice that registers that are used as the operands of the instructions should not be overwritten by the first set of computations. This has been shown for register  $A$  in Fig. 3.9.



**Figure 3.9** SSM applied to a basic block: (a) original code, (b) SSM code.

There is a pitfall that can reduce the error detection coverage. When the size of the blocks that are duplicated is extended, errors can be masked and hence escape detection. Fig. 3.10 shows two examples where errors can be masked. In Fig. 3.10 (a), the second operation is a logical operation. Register  $A$  is ANDed with a binary number whose least significant bit (lsb) is zero. Since the logical AND of zero with any value yields zero, regardless of the value of the lsb of register  $A$ , the lsb of register  $D$  will be zero (assuming there is no error in the second operation). Therefore, if the results of the two duplicates of the first operation (instructions 1 and 3) differ in that bit due to some error, this error will be masked (same final values in  $D$  and  $D'$ ) and won't be detected.



**Figure 3.10** Error masking in SSM: (a) masking in a logical operation, (b) masking in an arithmetic operation.

The second example (Fig. 3.10 (b)) is a similar sequence of instructions but with arithmetic operations. Assume the values of registers *B* and *C* are 10 and 7, respectively. In case of no error, registers *A* and *D* will have the values 17 and 3, respectively. However, the result of dividing any number between 19 and 15 by 5 is 3. Hence, if one of the duplicated addition operations produces an erroneous result that lies in this range, the error will not be detected. This kind of error masking is not important for correct operation of the system, but the goal of our project is to detect all the errors that happen so that we can get an estimate of the frequency of SEUs that occur in space. Therefore, we need to be careful in selecting the error detection techniques for our research.

To apply the SSM method, the compiler should be modified. The additional procedures in the compiler will identify the basic blocks, duplicate the instructions and add the comparison code. A liveness analysis should be done in the compiler to handle operand overwriting cases (Fig. 3.9 (b)). The register allocation algorithm can also be modified to use different registers for the intermediate and final results in the duplicated block so that permanent faults in the register file can be detected.


Similar to the function call example, we have the problem of memory being modified by the first copy of instructions. By duplicating the data segment of a program, two copies of data will be available for the two executions of instructions. This can be done by running two instances of the same program in a multitasking operating system. VxWorks is a multitasking operating system. By running two instances of a program, the code duplication and memory problem will be solved and only the comparison code has to be added. This comparison can be done by calling a function. This function will get the calculation results as its input, send the results to the other copy of the program, receive the duplicate results, do the comparison, and return true or false in case of a match or a mismatch, respectively. An error may happen in the execution of a program and it may hang (get stuck in an infinite loop) or skip a transmission, keeping the other program

waiting forever. Therefore, a timer should be used during reception to put a bound on the waiting time and detect this error.

The exchanged results can be a copy of the CPU registers plus the data written to the memory. The general comparison function will receive the written data or their addresses and create the packet to be sent. A number can also be assigned to each checkpoint and sent to this function to be added to the packet. This number can be used for control flow checking as in assigned signature methods.

In this method, the two copies of the program will run in a synchronized fashion. Each copy will execute up to a checkpoint and wait for the other to catch up, do a comparison and then continue. The flags or mailboxes available in the system (Sec. 2.3.4) can be used for this synchronization.

All programs produce outputs unless they are run only to exercise the functional units (e.g., some benchmarks). For example, the output of a program may be a set of bits that open or close a set of switches. For each switch, one bit should be written to a particular memory location. With two copies of a program running at the same time, the output of one should be suppressed just as the output of the slave processor in a self-checking pair is suppressed. In order to do this, there can be a flag in the program which is set according to a parameter passed to the program by the operating system when the program is started. This flag can make that copy of the program master or slave. The produced results will be checked between the master and slave before the master sends them out. The master will generate the actual output and the slave program will simply discard its results.

Another issue is that reading input from I/O may be destructive. Reading from an I/O may change the state of the system. The input data may be time dependent and hence the next time it is read it may be a different value. In these cases, the slave program should not read  the I/O. The master program should make a copy of the data for the slave program.

By adding the checkpoints manually or by doing some preprocessing on the source code, SSM can be implemented without modifying the compiler. The disadvantage of this method is that it has a high error detection latency because of large number of instructions between the checkpoints.

Another disadvantage is the loss of detection coverage due to error masking as explained before. By doing SSM both at instruction level and in a multitasking fashion, we can estimate the percentage of errors that are masked.

### **3.4 Summary**

Most of the control flow error detection techniques discussed above use some dedicated hardware. In the processor boards in ARGOS, the hardware is fixed and there is no watchdog processor or special hardware to generate signatures. Still, the ideas in those techniques can help in developing new software techniques. Some pure software techniques were also discussed that can be added to ARGOS to enhance the error coverage of its current hardware error detection mechanisms. Common-mode failures and errors in the unduplicated components of the system are some of the errors that the software techniques will try to detect.

In the next section, we will look at some fault injection techniques that are used to evaluate the efficiency of error detection techniques.

## 4. FAULT INJECTION

One way to validate the fault-tolerance mechanisms of a system is to inject faults into either a prototype or a software simulation model of the system [Schmid 82]. Designers and researchers disturb the signals on the pins, put the chip under radiation, disturb the power supply, or flip bit values during a logic simulation, to inject faults into the system. The extra advantages of fault injection are flexibility, controllability, and predictability, which are not available in a real environment. Moreover, it is a way to accelerate time because faults can be injected at a much higher frequency than they would occur in the real environment. However, whether the injected faults are a good representation of the faults that happen in the real environment is questionable. The unique opportunity that we have in ARGOS is that we will have the system tested in space; in the environment that it is intended operate. This opportunity is rarely provided to the designers and researchers.

Fault injection is done for different purposes. It can be used for removing faults in fault tolerance mechanisms. By observing the system responses, designers can debug and fine-tune a design before the actual system is built.

Fault injection can also be used for increasing software test coverage [Bieman 96]. During testing, it is hard to run all the statements or branches of a program. This is especially true for code used in handling exceptions, e.g., the error recovery code. This untested code will tend to be an error prone part of a system. However, it can be exercised and tested using fault injection.

There are several different approaches to fault injection in electronic systems. A detailed discussion can be found in [Iyer 93]. In this section we will briefly look at some methods:

**Disturb the signals on the pins of the IC:** This is the simplest way to introduce errors in a chip. For example, The signals can be controlled by a general purpose fault inserter (GPFI) [Scheutte 87]. The signal values can be changed at random or according to a defined sequence and timing, based on several parameters. This method was used for evaluation of SIS in [Scheutte 87]. It has the best control over the injected faults and can evaluate the response of system to several kinds of errors, but it has no control over the internal nodes of the chips.

**Radiation:** To induce faults inside the chips and simulate a space environment, the chip can be put under heavy-ion radiation or a high energy proton beam. The angle of incidence can be changed but it is usually 90 degrees for maximum penetration. For example, heavy-ion radiation has been used to evaluate error detection schemes [Gunnflo

89] [Karlsson 94]. Two CPUs were run in lock step and one of them was subjected to radiation. The output signals were compared and logged by a monitoring computer.

One parameter that is of concern for chips that are used in a space environment is their *total dose hardness*: the total dose of radiation that the chip can receive before it stops functioning correctly. High energy protons have been used to measure the total dose hardness of commercially available MIPS R3000 microprocessors [Kaschmitter 91] [Shaeffer 91] [Shaeffer 92]. The researchers looked at proton-induced SEUs in unhardened R3000s from four different vendors and also observed the total dose response. Two processors functioned at about 1 Mrad and all others, except one, remained functional above 40 krad. According to these papers, the anticipated total dose for a well-shielded processor in low Earth orbit (LEO) at 600km and 60 degree inclination is about 200 rad per year. The experiment results show that the processors are suitable for multi-year operation at LEO altitude in space with 300 mils of aluminum shield.

In this technique, the beam can be focused on only one chip, so that the chip is isolated from the errors happening in other chips. The behavior of the chips can be observed one at a time to measure their hardness and to see which chips need more hardening.

Of course, this technique does not represent all sorts of radiation that the system will be subject to in a space environment.

**Power supply disturbance:** Another simple way to cause errors in the system is to disturb the power supply [Cortes 86]. This actually models some of the errors happening on Earth due to power surges and disturbances common in industrial applications. This method was used in [Miremadi 92] and [Miremadi 95b] in conjunction with heavy-ion radiation on a MC6809E processor. Short voltage drops were caused at the power supply pin of the CPU using an MOS power transistor. A test CPU and a reference CPU were run in lock-step and the external buses were compared while the power supply pin of the test CPU was being disturbed.

When doing fault injection using radiation or power supply disturbance, we do not know where the fault is going to happen. Therefore, all parts of the chip should be exercised by using all the functional units, e.g., floating-point as well as integer units. If we fail to do this, a fault may happen somewhere and never show up as an error. If we use all parts and read the results back, we can detect errors happening almost anywhere on the chip and come up with an estimate for its sensitivity to radiation.

**Logic simulation:** Using the HDL (hardware description language) model of a system, e.g., VHDL or Verilog, errors can be introduced in the source code, and the system can be simulated to see if the detection mechanisms will detect the error. This error

can be in the form of a change in the functionality of a module or in the stimulus to the system. The same thing can be done with the netlist generated from the HDL or any other netlist at the gate-level or transistor level. Stuck-at faults can be simulated by slight modifications in the netlist; connecting nodes to power or ground. This technique is limited to the accuracy of fault models used. Some techniques for injecting non stuck-at faults are discussed in [Cortes 87].

Simulation can also be used at higher levels of abstraction for fault injection studies. DEPEND [Goswami 97] is an integrated design and fault injection environment that does simulation at system level. It uses functional fault models to simulate the system level manifestation of gate-level faults such as stuck-at faults.

## 5. SUMMARY

Our goal in this project is to collect as many in-flight transient errors as possible, exercise different error detection techniques, determine the tradeoffs between fault-avoidance and fault-tolerance techniques, and finally come up with an efficient blend of techniques suitable for space applications.

To reach this goal, we have studied available hardware and software fault tolerance techniques and we plan to come up with some new techniques along the way. There are many features incorporated into the processor boards and we plan to take full advantage of them. The program we will run on the boards will have additional error detection techniques, e.g., stutter step mode execution, control flow checking and assertions, to name a few. This program will try to exercise all the circuitry on the processor board, collect information on the errors, store them in a redundant format and use the telemetry system to send them to the ground. A local program will receive this data and put it in a database for analysis.

With an FPGA on one of the processor boards, we plan to take the opportunity to survey their behavior in a space environment and use their unique features in fault tolerance techniques. In this regard, we studied the testing techniques of FPGAs. We can reconfigure them on the fly into a BIST mode, locate the faulty cells and upload a new configuration which isolates the faulty cells and uses spare cells on the chip.

## **ACKNOWLEDGMENTS**

This work was supported in part by the Ballistic Missile Defense Organization, Innovative Science and Technology (BMDO/IST) Directorate and administered through the Department of the Navy, Office of Naval Research under Grant Nos. N00014-92-J-1782 and N00014-95-1-1047. The authors wish to thank Dr. Nirmal Saxena for his invaluable suggestions, and also Dr. Nur Touba, Dr. Robert Norwood and Nahmsuk Oh for their careful reviews and helpful comments. The project described here would not have been possible without the support and help of Louis Lome from BMDO, Kent Wood from NRL and Alan Ross from NPS.

## REFERENCES

- [Andrews 79] Andrews, D., "Using executable assertions for testing and fault tolerance," *9th Fault-Tolerance Computing Symp.*, Madison, WI, June 20-22, 1979.
- [Berger 85] Berger, E.R., M.K. House, G.J. Manzo, and A.H. Taber, "Single Event Upsets in Microelectronics: Third Cosmic Ray Upset Experiment," *IBM Tech. Direct.*, Vol. 11, No. 1, pp. 33-40, 1985.
- [Bieman 96] Bieman, J.M., D. Dreilinger, and L. Lin, "Using Fault Injection to Increase Software Test Coverage," *Proc. IEEE Int'l Symp. On Software Reliability Engineering*, pp. 166-174, 30 Oct.-2 Nov. 1996.
- [Cortes 86] Cortes M., et al., "Properties of Transient Errors Due to Power Supply Disturbances," *Center for Reliable Computing Technical Report, No. 86-1*, Stanford University, 1986.
- [Cortes 87] Cortes M., et al., "Techniques for Injecting non-stuck-at faults," *Center for Reliable Computing Technical Report, No. 87-21*, Stanford University, 1987.
- [Eifert 84] Eifert, J.B., and J.P. Shen, "Processor Monitoring Using Asynchronous Signed Instruction Streams," *Dig. 14th Annual Int'l Conf. on Fault-Tolerant Computing*, pp. 394-399, Jun. 1984.
- [Ersoz 85] Ersoz, A., D.M. Andrews, and E.J. McCluskey, "The Watchdog Task: Concurrent Error Detection Using Assertions," *Center for Reliable Computing Technical Report, No. 85-8*, Stanford University, 1985.
- [Goswami 97] Goswami, K.K., R.K. Iyer, and L.Y. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," *IEEE Trans. On Computers*, Vol. 46, No. 1, pp. 60-74, January 1997.
- [Gunnflo 89] Gunnflo, U., J. Karlsson, and J. Torin, "Evaluation of error detection schemes using fault injection by heavy-ion radiation," *19th Int'l Symp. on Fault Tolerant Computing*, Chicago, IL, pp. 340-347, Jun. 21-23, 1989.
- [Harris 93] Harris Corporation, "Development Specification for the RH3000 processor VME module," rev. B, Sep. 1993.
- [Hass 89] Hass, K.J., R.K. Treece, and A.E. Giddings, "A Radiation-Hardened 16/32-Bit Microprocessor," *IEEE Trans. on Nuclear Science*, Vol. 36, No. 6, pp. 2252-2257, Dec. 1989.
- [Hennessy 96] Hennessy J.L., and D.A. Patterson, *Computer Architecture, A Quantitative Approach*, Second edition, Morgan Kaufmann Pub., Inc. 1996.
- [Huang 84] Huang, K.H., and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Computers*, Vol. C-33, No. 6, pp. 518-528, June 1984.
- [Ignatushchenko 94] Ignatushchenko, V.V., et al., "Effectiveness of temporal redundancy of parallel computational processes," *Automation and Remote Control*, Vol. 55, No. 6, pt. 2, pp. 900-911, Jun. 1994.
- [Iyer 93] Iyer, R.K. and D. Tang, "Experimental Analysis of Computer System Dependability," *Center for Reliable and High-Performance Computing, Technical Report CRHC-93-15*, University of Illinois at Urbana-Champaign, 1993.
- [Jordan 93] Jordan, C., and W.P. Marnane, "Incoming inspection of FPGAs," *Proc. of ETC '93, Third European Test Conf.*, Rotterdam, Netherlands, pp. 371-377, Apr. 19-22, 1993.
- [Karlsson 94] Karlsson, J., P. Liden, P. Dahlgren, R. Johansson, and U. Gunnflo, "Using Heavy-Ion Radiation to Validate Fault-Handling Mechanisms," *IEEE Micro*, Vol. 14, No. 1, pp. 8-23, February 1994.
- [Kaschmitter 91] Kaschmitter, J.L., et al., "Operation of commercial R3000 processors in the Low Earth Orbit (LEO) space environment," *IEEE Trans. on Nuclear Science*, Vol. 38, No. 6, pt. 1, pp. 1415-1420, Dec. 1991.

- [Kato 94] Kato H., et. al., "Implementation and Verification of High Reliability Computer for Satellites," *Transactions of Information Processing Society of Japan*, Vol. 35, No. 9, pp. 1936-1948, Sep. 1994, (in Japanese).
- [Kaul 91] Kaul, N., B.L. Bhuvra, and S.E. Kerns, "Simulation of SEU Transients in CMOS ICs," *IEEE Trans. on Nuclear Science*, Vol. 38, No. 6, pp. 1514-1520, Dec. 1991.
- [Koga 84] Koga, R., and W.A. Kolasinski, "Heavy Ion-Induced Single Event Upsets of Microcircuits; A Summary of the Aerospace Corporation Test Data," *IEEE Trans. on Nuclear Science*, Vol. 31, No. 6, pp. 1190-1195, Dec. 1984.
- [Lala 91] Lala, P.K., et al., "On self-checking software design," *IEEE Proc. of SOUTHEASTCON '91*, Williamsburg, VA, pp. 331-335, Apr. 7-10, 1991.
- [Lantz 96] Lantz II, Leon, "Soft Errors Induced by Alpha Particles," *IEEE Trans. Reliability*, Vol. 45, No. 2, pp. 174-199, June 1996.
- [Lu 82] Lu, D., "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. on Computers*, Vol. 31, No. 7, pp. 681-685, Jul. 1982.
- [Madeira 92] Madeira, H., and J.G. Silvia, "On-line Signature Learning and Checking," *Dependable Computing for Critical Applications*, Springer-Verlag, J.F. and R.D. Schlichting (eds), Vol. 6, pp. 395-420, 1992.
- [Madeira 93] Madeira, H., M. Rela, and J.G. Silvia, "Time Behavior Monitoring as an Error Detection Mechanism," *Dependable Computing for Critical Applications*, Springer-Verlag, J.F. and R.D. Schlichting (eds), 1993.
- [Mahmood 88] Mahmood, A., and E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors—A Survey," *IEEE Trans. on Computers*, Vol. 37, No. 2, pp. 160-174, Feb. 1988.
- [Messenger 91] Messenger, G.C., and M.S. Ash, *The Effects of Radiation on Electronic Systems*, 2nd ed., New York, Van Nostrand Reinhold, pp. 416-493, 1991.
- [Mili 82] Mili, A., "Self-Stabilizing Programs: the Fault-Tolerant Capability of Self-Checking Programs," *IEEE Trans. on Computers*, Vol. C-31, No. 7, pp. 685-689, July 1982.
- [Mili 90] Mili, A., *An Introduction to Program Fault Tolerance: A Structured Programming Approach*, Prentice Hall, 1990.
- [Miremadi 92] Miremadi, G., J. Karlsson, J.U. Gunneflo, and J. Torin, "Two Software Techniques for On-line Error Detection," *Digest of Papers, 22nd Annual Int'l Symp. on Fault-Tolerant Computing*, pp. 328-335, Jul. 1992.
- [Miremadi 95a] Miremadi, G., J. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time, Location and Instruction Signatures for Control Flow Checking," *Proc. of the DCCA-5 Int'l Conf.*, Springer-Verlag Series for Dependable Computing Systems, Sep. 1995.
- [Miremadi 95b] Miremadi, G., J. Torin, "Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection," *IEEE Trans. Reliability*, Vol. 44, No. 3, pp. 441-453, Sep. 1995.
- [Namjoo 82] Namjoo, M., "Techniques for Concurrent Testing of VLSI Processor Operation," *Dig. 1982 IEEE Test Conf.*, Philadelphia, PA, pp. 461-468, Nov. 16-18, 1982.
- [Ritter 90] Ritter, J.C., "Radiation Effects in Space Systems," *Naval Research Reviews*, pp. 25-37, 1990.
- [Saxena 90] Saxena, N.R., and E.J. McCluskey, "Control-Flow Checking Using Watchdog Assists and Extended-Precision Checksums," *IEEE Trans. Comput.*, Vol. 39, No. 4, pp. 554-559, Apr. 1990.
- [Saxena 94] Saxena, N.R., and E.J. McCluskey, "Linear Complexity Assertions for Sorting," *IEEE Trans. On Software Eng.*, Vol. 20, No. 6, pp. 424-31, June 1994.
- [Schmid 82] Schmid, M.E., R.L. Trapp, A.E. Davidoff, and G.M. Masson, "Upset exposure by means of abstraction verification," *Proc. 12th Int'l Fault-Tolerant Comput. Symp.*, pp. 237-244, Jun. 1982.

- [Schuette 87] Schuette, M.A., and J.P. Shen, "Processor Control Flow Monitoring Using Signed Instruction Streams," *IEEE Trans. Comput.*, Vol. C-36, No. 3, pp. 264-276, Mar. 1987.
- [Shaeffer 91] Shaeffer, D.L., et al., "High energy proton SEU test results for the commercially available MIPS R3000 microprocessor and R3010 floating point unit," *IEEE Trans. on Nuclear Science*, Vol. 38, No. 6, pt. 1, pp. 1421-1428, Dec. 1991.
- [Shaeffer 92] Shaeffer, D.L., et al., "Proton-induced SEU, dose effects, and LEO performance predictions for R3000 microprocessors," *IEEE Trans. on Nuclear Science*, Vol. 39, No. 6, pt. 2, pp. 2309-2315, Dec. 1992.
- [Shen 83] Shen, J.P., and M.A. Schuette, "On-Line Self-Monitoring Using Signed Instruction Streams," *Dig. 1983 Int'l Test Conf.*, Philadelphia, PA, pp. 275-282, Oct. 18-20, 1983.
- [Siewiorek 92] Siewiorek, D.P., and R.S. Swarz, *Reliable Computer Systems*, Burlington, Digital Press, 1992.
- [Sridhar 82] Sridhar, T., and S.M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," *Dig. 1982 IEEE Test Conf.*, Philadelphia, PA, pp. 191-199, Nov. 16-18, 1982.
- [STI 94] *RH3000 DPB Flight Software Document*, Software Technology, Inc., Aug. 1994.
- [Takano 91] Takano, T., T. Yamada, K. Shutoh, and N. Kanekawa, "Fault-Tolerance Experiments of the 'Hiten' Onboard Space Computer," *Dig. Fault-Tolerant Computing: Twenty-First Int'l Symp.*, Montreal, Que., Canada, pp. 26-33, Jun. 25-27, 1991.
- [Wilken 89] Wilken, K., and J.P. Shen, "Concurrent Error Detection Using Signature Monitoring and Encryption: Low-Cost Concurrent-Detection of Processor Control Errors," *Dependable Computing for Critical Applications*, Springer-Verlag, A. Avizienis, J.C. Laprie (eds), Vol. 4, pp. 365-384, 1989.
- [Wilken 90] Wilken, K., and J.P. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control Errors," *IEEE Trans. on Computer-Aided Design*, Vol. 9, No. 6, pp. 629-641, Jun. 1990.
- [Worley 90] Worley, E., R. Williams, A. Waskiewicz, and J. Groninger, "Experimental and Simulation Study of the Effects of Cosmic Particles on CMOS/SOS RAMs," *IEEE Trans. on Nuclear Science*, Vol. 37, No. 6, pp. 1855-1860, Dec. 1990.
- [Xilinx 96] Xilinx Inc., *The Programmable Logic Data Book*, 1996.