Software-Based Fault-Tolerance

David Holdt

Preface

This report is the results of a masters thesis project entitled ”Software-Based Fault-Tolerance”. The project was carried out at the Technical University of Denmark, Department of Informatics and Mathematical Modeling under the supervision of associate professor Hans-Henrik Løvengreen. The project period was 31. august 2003 through 15. February 2004.

David Holdt

Abstract

This Masters Thesis investigate the possibility of using software-based Error Detection And Correction (EDAC) to protect the storage of space-borne computers from Single Event Upsets (SEU's). Two software-based fault-injectors are implemented, which simulate the effects of SEU's as stochastic processes, and a number of experiments are performed on programs running on prototypes of the onboard computer of a small satellite, DTUSat

The faults are injected in a continuous manner until a failure is detected (i.e. test until 'destruction'), after which the state of the target is examined.

One of the targets programs is the DTUSat boot-software, while the other is an program resembling the application software for the DTUSat. In order to perform a large number of experiments, two harnesses have been implemented, which allow batches of experiments to be carried out automatically. The results of the experiments are analyzed, an a number of metrics are measured (time of first error, runtime before failure, cause of failure etc.).

The injectors are designed to be quite flexible, in that the user can control the fault-rates and injection model. Also, a number of experiments are performed with different fault-rates, and it is found that the results obtained are representative over a large interval of injection-rates.

In order to carry out the experiments in a resonable amount of time, the injection-rates used are somewhat higher than what can be expected in a low earth orbit (LEO).

Keywords: Fault-tolerance, fault-injection, stochastic simulation, seu, leo, edac, hamming code

Contents

Chapter 1

Introduction

As more and more devices become computer-controlled, fault-tolerance in software plays an ever increasing role. The application of fault-tolerance has extended to cover a large set of methods and many areas of use. However, in this report, the focus will be on problems related to the space-domain.

1.1 Space issues

When deploying computers and electronics in space, there are many diverse problems to address, due to the harsh environment. Typically, the equipment has to be able to function in large temperature intervals and under the influence of radiation, while having only a minimal power-consumption. At the same time, the equipment is typically expected to be highly autonomous, i.e. offering high level of reliability and availability.

One of the main problems is radiation, which has various effects on electronics. The effects are generally categorized as either total dose effects or single events effects. The total dose cause a gradual performance-degradation until the point of destruction, while single events are of temporary nature. An important type of single events is the Single Event Upset (SEU), which causes the contents of storage (RAM) to be altered.

The traditional solution to the general radiation problem has been to use shielding, and the specific problem of SEU's has been addressed by using hardware-based error-correcting codes. However, the use of hardware-based error-correcting codes requires special hardware, which may not be appropriate to use in especially smaller systems.

1.2 Problem Statement

The purpose of this masters thesis is to investigate the effects of SEU's on a small satellite system, the DTUSat onboard computer. This is accomplished by implementing fault-injectors, which simulate SEU's as stochastic processes. Further, error-detecting and correcting codes

(EDAC) are implemented in software, and it is investigated to which extend these code can counter the effects of the simulated SEU's. A large numbers of experiments are performed, and the results are analyzed.

1.3 Organization

The rest of the report is organized as follows:

Chapter 2, Fault-Tolerance Primitives : In this chapter, a number of traditional fault-tolerance primitives and methods are presented, and discussed. The primitives and methods are categorized.

Chapter 3, Theory and Basics : This chapter documents the theoretical bases on which the following chapters rely. The theory for the stochastic simulations and the error-correcting code and are developed in this chapter, and the platform on which the experiments are performed is presented.

Chapter 4, Failsafe Injector : The failsafe injectors injects faults into the DTUSat boot-software (failsafe software). The development and implementation of the injector, and the experiments on the failsafe software are documented in this chapter.

Chapter 5, The Application : The application is implemented as a set of loosely-coupled modules, which communicate through a shared packet-router. which has a structure similar to that of the DTUSat application.

Chapter 6, Correctors : The implementation and use of the EDAC-system, and the experiments on the application are covered in this chapter.

Chapter 7, Conclusion : The final remarks discussed some common issues of the project, and contains the conclusion of the report.

Appendix A, Modeling in SPIN : An appendix describing how to use the SPIN modeller to model problems of the type described in chapter 3.

Appendix B, Results of the Failsafe Experiments : Tabulated results of the Failsafe Experiments.

Appendix C, Results of the Application Experiments : Tabulated results of the Application Experiments.

The sourcecodes for the implemented injectors and programs can be found in the DTUSat CVS repository, accessible through the url:
`http://cvs.dtusat.dtu.dk/cgi-bin/viewcvs.cgi/fts/?cvsroot=Development`.

Chapter 2

Fault-Tolerance Primitives

In this section, a number of traditional fault-tolerance primitives are discussed. The functionings of the primitives are described and, where applicable, the primitives are categorized. The last section of this chapter contains a summary of the primitives, in which they are partitioned into detectors and correctors.

2.1 Primitives

There are several ways of categorizing primitives, for instance [5] categorizes primitives as either detectors or correctors. While most primitives will fit into one of these categories, the practical applications of some of the primitives can actually be described as both detecting and correcting. For instance, if a watchdog-timeout causes a system (or processor) reset, it can rightfully be categorized as both a detecting and correcting primitive.

2.1.1 Partitioning

The purpose of partitioning is to prevent that faults in one part of a system propagate to other parts of the system. Partitioning can take place on many levels in both hardware and software.

Examples of Usage

A radical use of partitioning on the hardware level is to allocate different tasks to different CPU's, thus minimizing the shared resources to communications channels. This approach is often used in aircrafts and satellites.

A more common example is by means of an MMU (Memory Management Unit), by which the memory-contents of a task are protected from malicious accesses of other tasks running on the same CPU. Note that this example requires support from both the hardware (MMU) and the software (the operating system).

On the software level, partitioning is typically implemented by the compiler, the operating system or by design. For instance, the functional decomposition into layers, modules and libraries are examples of partitioning.

Partitioning of data typically involves data hiding, for instance: Object-Oriented Programming, the use of resource handles etc.

Classification

As partitioning in itself cannot by used to detect or correct faults, it is better described as a good practice than as a fault tolerance primitive. As a design-rule it is applicable at many levels of both hardware and software, and as such it helps to decompose the system in modules, thus bringing down the complexity of the individual components, and thereby making the system easier to implement.

Partitioning is therefore classified to addresses failures in external devices and in software.

2.1.2 Assertions

Assertions are checks on the state of the variables at given points in the program. For instance, variables can be range-checked, and relationships between variables can be validated.

Examples of Usage

The following example shows the use of assertions, where the state of a variable is checked both before and after its use:

```
1  int calculation(float factor)
2  {
3    assert(pre_predicate(factor));
4    // use of factor
5    assert(post_predicate(factor));
6  }
```

The assertion can be seen as expressing invariants of the program. If the assertion expresses a predicate, $pred(v_1,...,v_n)$, on the variables of the program, and the label l_{assert} corresponds to the location immediately following the assertion, it is invariant that: $at(l_{assert}) \Rightarrow pred(v_1,...,v_n)$. Of course, if the assertion fails, an appropriate action is taken.

Classification

As they do not correct any errors, assertions are detecting primitives.

2.1.3 Input Checks

Input checks is a technique that aims to prevent subprograms from malfunctioning due to improper inputs. The technique can be used to express limits or relationships in the inputs, which cannot be expressed in the used programming language. Depending on the exact implementation, input checks can be recognized as an instance of ”assertion programming”, ”defensive programming” or even ”roll-forward recovery”.

Examples of Usage

Assertion Programming and Defensive Programming require that values are validated before use. The following example demonstrates this use of Input Checks:

```
1  int calculation(float factor)
2  {
3    assert((0.0<=factor) && (factor<=1.0));
4    // use of factor
5  }
```

If corrective actions are implemented, the use of Input Checks is somewhat similar to Roll-Forward Recovery:

```
1  int calculation(float factor)
2  {
3    if(factor<0.0)
4      factor=0.0;
5    else if(factor>1.0)
6      factor=1.0;
7    // use of factor
8  }
```

Classification

Input checks is a detecting primitive, as it does not correct any errors. The corrective parts of the example above are not strictly parts of the input checks

2.1.4 Safety Supervisor

A Safety Supervisor is a control-layer, which is used to ensure that the values output from functions are within the valid range. However, the Safety Supervisor is concerned only with the value domain, i.e. it does not validate the calculations performed by the functions. When an out-of-range value is detected, it should be corrected to a safe (in-range) value; on a small scale, this behavior is similar to Roll-Forward Recovery. A safety supervisor can also be seen as a device which renders possibly unsafe values into safe ones.

Examples of Usage

The Safety Supervisor is usually implemented as a unit separated from the function or component it supervises, for example:

```
1   float safety_supervisor(float input)
2   {
3     float output;
4
5     if(input<minimal_value)
6       output=minimal_value;
7     else if(input>maximal_value)
8       output=maximal_value;
9     else
10      output=input;
11
12    return output;
13  }
14
15  void control(void)
16  {
17    float input, unsafe_output, safe_output;
18
19    input=read_device();
20    // calculate unsafe_output, based on input
21    safe_output=safety_supervisor(unsafe_output);
22    write_device(safe_output);
23  }
```

Classification

The Safety Supervisor is traditionally categorized as a detecting primitive, even though it also corrects faulty values.

2.1.5 Defensive Programming

Defensive Programming is a method that addresses unexpected data and usage by a number of guidelines. Many of the guidelines are specific to a given programming language, but [2] numbers the following general rules:

- Variables should be range checked.
- Where possible, values should be checked for plausibility.

- Parameters to procedures should be type, dimension and range checked on entry.
- Read-only and read-write parameters should be separated and their access checked. Functions should treat all parameters as read-only. Literal constants should not be write-accessible. This helps detecting accidental overwriting and erroneous use.

Note that strongly-typed languages such as Ada and Java satisfy some of these rules intrinsically, while others (C and C++) require some manual effort from the programmer. Furthermore, [2] mentions the following techniques:

- Input variables and intermediate variables with physical significance should be checked for plausibility.
- The effect of output variables should be checked, preferably by direct observation of associated system state changes.
- The system should check its configuration. This could include both the existence and accessibility of expected hardware, as well as the fact that the software itself is complete (this is particularly important for maintaining integrity during and after maintenance procedures.)

Some Defensive Programming rules overlap with rules that might be used in coding standards. For instance, when iterating over a fixed-size array, it is common practice to use a compiler-calculated expression for the array-size, rather than a manually calculated constant. Other rules might address shortcomings or undesired aspects of the language. For instance, a standard 'C' compiler will silently synthesize function declarations, when they are absent from the programmers hand, but this can be a very unpleasant property. Some compilers can be configured to issue warnings or errors, whenever it is about to do something which is potentially dangerous.

Classification

As defensive programming is a proactive method, it cannot be described as a detecting or a correcting primitive.

2.1.6 Checkpointing and Rollback

The Checkpointing and Rollback -technique revolves around the state of the software: The internal state of the software is checked regularly at certain locations of the program, and if it is found to be safe, the state is stored; this is known as a checkpoint operation. However, if the state is found to be unsafe, the last safe state is resumed; this is the rollback operation.

As [5] points out, Checkpointing and Rollback benefits from support from the operating system, but most operating systems do not provide these features. Also, for multi-process

systems, it might be difficult to restore a valid global state, because it is based on saved local states that are independent from each other. This technique is also known as Temporal Redundancy in [3].

Classification

The checkpointing and rollback -method is able to correct faulty states, so it is described as a correcting primitive.

2.1.7 Roll-Forward Recovery

Roll-forward Recovery is a lightweight alternative to Checkpointing and Rollback. Like Checkpointing and Rollback, Roll-forward Recovery checks the state of the program at certain locations, but rather than relying on saved states, it relies on predefined valid states. I.e. the implementor of the software defines which states to roll-forward to. As [5] points out, this method requires detailed knowledge of the failure-modes of the particular system, and is thus specific to each system.

Examples of Usage

As Roll-forward Recovery is system specific its implementation will be dictated by the system. A simple example is the one given in section 2.1.3.

Classification

Like checkpointing and rollback, roll-forward recovery has correcting capabilities, so it is described as a correcting primitive.

2.1.8 Recovery Blocks

Recovery Blocks consist of a number of different implementations of the same calculation. The calculations are performed in a sequence, where each calculation is followed by an acceptance test. If the test accepts the result, the calculated result is used, but if the tests rejects the result, any side-effects of the calculation are undone and an other calculation/test pair is performed. There can be any number of these calculation/test pairs, but if they all fail, the calculation cannot be performed, and is thus aborted. If the implementations of the variations are of different quality or efficiency, they can be numbered and sorted accordingly. A disadvantage of Recovery Blocks is that the execution time of the calculation can vary significantly, depending on the number of blocks executed.

Examples of Usage

```
1   int calculation(args)
2   {
3     res=calculation_1(args);
4     if(correct_1(res))
5       return res;
6
7     undo_1(args, res);
8     res=calculation_2(args);
9     if(correct_2(res))
10      return res;
11    ...
12    res=calculation_n(args);
13    if(correct_n(res)
14      return res;
15
16    return abort_calculation(arg);
17  }
```

Classification

Recovery-blocks are classified as a corrective primitive.

2.1.9 N-Version Programming, N-Selfchecking Programming

Like Recovery Blocks, N-version software (in both flavors) is based on different implementations of the same calculation. Unlike Recovery Blocks, all versions are executed when a calculation is made, which provides for better estimates of the execution time. The versions in N-Version Programming return the result of the calculation, and when all versions have been run, the results are collected and examined by a voter. The voter then decides which result is ”correct”.

The versions in N-Selfchecking Programming perform a check on the calculation before returning the value – then the results are collected and examined by the voter, only the result thought to be valid are considered. The N-Selfchecking method requires that each version is capable of evaluating its own result. As [3] suggests, this can be implemented by calculating the inverse of the result, or by examining intermediate results.

A major weakness of N-version Programming (in both flavors) is that it is difficult to produce truly diverse versions, because the manners in which a given algorithms can be practically implemented is limited. This means that supposedly independent versions can have common-mode failures.

Examples of Usage

The implementation reflects the basic structure, where the results of the independent versions are examined by the voter.

```
1   int calculation(args)
2   {
3     res1=calculation_1(args);
4     res2=calculation_2(args);
5     ...
6     resn=calculation_n(args);
7
8     res=voter(res1, res2, ... , resn);
9
10    return res;
11  }
```

Classification

Both flavors of N-version programming are proactive, in the sense that they address problems of possibly unknown nature before they arise. They both have the ability to correct these problems, so they are categorized as correcting primitives.

2.1.10 Exception Handling

As the name suggests, Exception Handling is a way to handle exceptions in the software. An exception is an unexpected error-condition, that cannot be handled by the subprogram which detected the condition. For instance, this could be caused by unexpected input parameters or an unexpected state of the program environment. It is common that programming languages implement the exceptions mechanisms by the constructs `throw` and `catch`. The principle is, that when an exception is detected, it is ”thrown” at the point of detection, and ”caught” at an appropriate enclosing exception handler. The exception changes the flow of execution, because when thrown, it will cause the execution to transfer to the handler, and the execution does not transfer back again. The handler is found among ancestor functions and enclosing blocks, and the most recently encountered handler is used. When defining an exception handler, its scope is usually confined to a single block of execution; C++ and Java defines the keyword `try` for this, while the same functionality is achieved in Ada, using general block-construct.

Examples of Usage

```
1   int calculation(args)
2   {
3     ...
4     throw(Exception);
5     ...
6   }
7
8   int function(args)
9   {
10    try
11    {
12      res=calculation(args);
13    }
14    catch (Exception E)
15    {
16      res=-1;
17    }
18
19    return res;
20  }
```

Classification

Exception handling addresses foreseeable problems in a structured manner, and is categorized as a correcting primitive.

2.1.11 Wrappers

When using 3rd party libraries or programs, it is not possible to guarantee the quality of the products or its impact on the system. One way of controlling unwanted side-effects from 3rd party components, is by using a wrapper. A wrapper is a subprogram, that monitors and verifies all inputs and outputs from the component. If a sufficiently precise description of the component is available, the wrapper may be able to detect and correct erroneous outputs from the component. However, implementing detection and correction might come very close to a parallel implementation, i.e. effectively being a two-version implementation.

Examples of Usage

```
1  int wrapper(args)
2  {
3    args'=transform(args);
4    res=wrapped_function(args');
5    res'=transform(res);
6
7    return res';
8  }
```

Of course, the transforms are depending on the function being wrapped, and on the application.

Classification

As wrappers are meant to correct problems in 3rd party components, they are categorized as corrective primitives.

2.1.12 Watchdog

A common fault tolerance primitive is a Watchdog timer. In its simplest implementation, the watchdog is basically a free-running timer, which performs some corrective action when it times out. However, it can be reset from the software, thus preventing it from timing out. In normal operation the timer should never timeout, as the software resets it on regular intervals. The Watchdog is normally used to detect timing and liveness problems.

It is common to set up the Watchdog to reboot the system on time-out. However, this use requires careful analysis of the timing-properties of the system, because it is easy to create an endless reboot-loop, thereby rendering the system inoperable.

Examples of Usage

```
1  int function(args)
2  {
3    for(;;)
4    {
5      input=read_device();
6      // calculate output from input
7      write_device(output);
8      watchdog_reset();
9      delay();
10   }
11 }
```

Classification

As the watchdog typically implements some kind of corrective action when it times out, in practice it is both a detecting and correcting primitive.

2.1.13 Execution Flow Check

An Execution Flow Check is somewhat similar to a watchdog processor, in that it is usually implemented with the support from hardware (i.e as a form of co-processor). While the processor executes the program, the Execution Flow Check checks that the instructions executed are consistent with predefined legal execution traces. If the execution differs from the expected paths, some kind of corrective action is taken. As [5] points out, Execution Flow Checks may be difficult to implement because of the need to establish all legal execution traces. This technique is sometimes referred to as ”Memorizing Executed Cases”.

Classification

The Execution Flow Checks share some similarities with the general watchdog. Like the watchdog, in order to be useful, it should implement some kind of corrective action, when an illegal state is detected. It is therefore categorized as a corrective primitive.

2.1.14 Software rejuvenation

As [3] points out, running software tends to age due to resource depletion (e.g heaps being fragmented or corrupted, and objects not being deallocated). A common solution to this problem is to reload the software on regular intervals – in [3] this is called Software Rejuvenation. Software Rejuvenation is a type of preventive maintenance, because it addresses problems that have yet to be detected.

Classification

As Software Rejuvenation addresses problems not yet arisen, it is a proactive method. It is best described as a corrective primitive.

2.1.15 EDAC and Scrubbing

EDAC (Error-Detection-And-Correction) and scrubbing aims at preventing unwanted changes to the memory, i.e. due to SEU’s (Single Event Upsets). Traditionally, this is a hardware-assisted method, which is based on RAM protected by error-correcting codes. A commonly applied code is the general Hamming-code, which can correct a single bit-error per codeword.

An important reason for choosing this code is its simplicity, which makes it easy to implement in hardware. However, in recent years more advanced codes have been used for these purposes (for instance [40]).

The hardware is assisted by a scrubbing process, which continuously read-out and write-back the values of the memory. This process prevents correctable single bit-errors from evolving into non-correctable multi-bit ones. There are several complications when applying this method. As [5] points out, the scrubbing process would typically run with low priority, to prevent it from affecting the schedulability of the system. Furthermore, because the memory is shared between the scrubbing process and the other processes, some type of synchronization is needed.

Examples of Usage

```
1   scrubber(region)
2   {
3     do forever
4     {
5       for(address=region.begin; address<region.end; address++)
6       {
7         lock();
8         value=*address;
9         *address=value;
10        unlock();
11      }
12    }
13  }
```

Classification

As the name suggests, EDAC has the ability to both detect and correct problems, and therefore it is described as both a detecting and correcting primitive.

2.1.16 Graceful Degradation

Graceful Degradation is a method of dealing with multiple levels of service by disabling the less important tasks. For instance, the software may handle that certain external devices fails, or that the processor becomes overloaded. As [5] points out, Graceful Degradation is commonly implemented as different modes of operations, where each mode corresponds to a given level of service.

Classification

Graceful Degradation is mostly a design method, but if it is employed in an automatic manner, it can be described as a correcting primitive.

2.2 Classification of Primitives

In this section, the classifications of the primitives are summarized. As previously mentioned, a classification as either correcting or detecting is not appropriate for all primitives. Therefore, the primitives can be classified in a total of four categories, corresponding to all four combinations.

Primitive	Detect	Correct
Partitioning	÷	÷
Defensive Programming	÷	÷
Input checks	√	÷
Assertions	√	÷
Safety Supervisor	√	÷
Execution Flow Checking	√	÷
EDAC and Scrubbing	√	√
Checkpointing and Roll-back	÷	√
Recovery Blocks	÷	√
Graceful Degradation	÷	√
Roll-forward Recovery	÷	√
Exception Handling	÷	√
Wrappers	÷	√
Software Rejuvenation	÷	√
N-Version Programming N-Selfchecking Programming	÷	√
Watchdog	√	√

Chapter 3

Theory and Basics

In this chapter, the theory on which the project depends, and the theory developed during the project is presented. This chapter also provides some background information on the hardware and platform used for the experiments presented in this.

3.1 Abstract Models

The articles [33, 34, 35] provide abstract models for design of fault tolerant software. In the following section, these principles are described, and two examples of their use is given in appendix A, where they are also simulated in SPIN.

A central term in the treatment of the abstract models is the use of predicates, which are used to define and prove key properties of the algorithms. Two other central terms are closure and convergence, which signifies that the behavior of a system is well-defined, given a specific set of circumstances, and that the system is able to recover from faults. Also, a path for incremental developments and improvements is shown.

3.1.1 Closure and Convergence

A program is said to be closed under a predicate, if the predicate is always true under a given set of circumstances. For instance, if the system does not incur any faults, its behavior can be described by a predicate, under which the execution is closed. Likewise, if the system does incur faults, its behavior can be described using a different predicate. Predicates can be used to describe the properties of the system, such as correctness, safety and liveness.

In this way [35] defines a set of predicates, which describe the behavior of the system both in absence and presence of faults. If the predicate S describes the behavior of the system in absence of faults, and T describes the behavior in presence of faults, the system is said to converge, if the behavior of the system eventually can be described by S, when the faults stop occurring.

This way, the state-space of the system is divided into regions, of which S is the smallest, and T (or T_n) are larger regions encompassing S. Figure 3.1 depicts the concept of multiple predicates dividing the state-space. In the state S_a where S holds, the system experiences a fault (marked by !), which brings the system into state T_a, where T holds. After some time where no additional faults have occurred, the system enters state T_b and then S_b, where S holds again. Note that faults are modeled as a special set of state-transitions, that may occur as certain points of the execution.

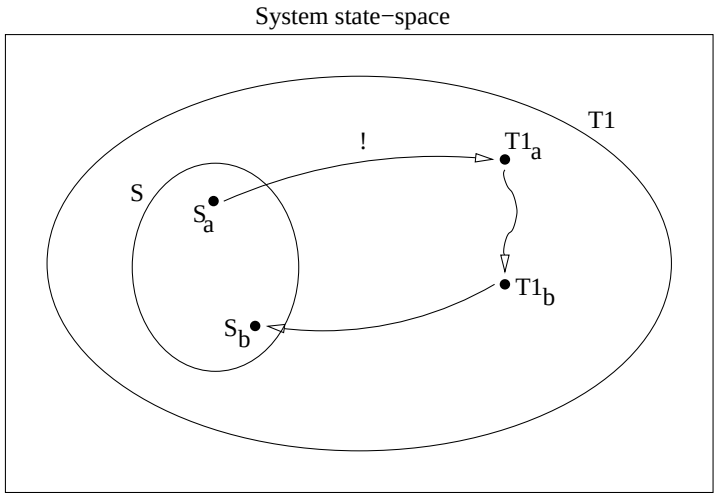

Figure 3.1: Regions of the state-space covered by different predicates.

3.1.2 Incremental Development

The system may be described by more than two predicates, corresponding to its ability to handle more than one type of fault. The development can be done incrementally, in which support for new types of faults is added without the need for redoing the proofs on the already implemented parts. This is visualized in figure 3.2, in which a new predicate $T2$ is added, corresponding to a new type of fault.

3.2 Practical Application

The abstract models rely on the program having an accurate and bounded state-space, even in the event of failures. This implies that all actions are well-defined – even the fault-actions. A special problem in space-applications is that radiation may cause unpredictable changes to the state-space of the program, which makes it difficult or impossible to describe the properties of the program.

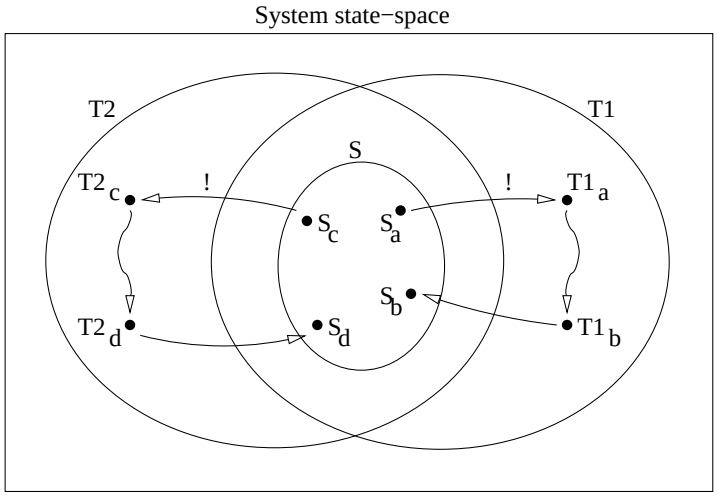

Figure 3.2: Regions of the state-space covered by multiple predicates.

The traditional solution to this problem has been to employ special hardware, which has higher resistance to radiation, but in the following it will be investigated if similar methods can be implemented in software.

3.3 Fault-injection Model

As mentioned above, when deploying electronics into space, one of the issues to address is radiation. Of course, a general remedy to radiation is shielding, but due to constrains on weight and volume, it is only feasible to provide a certain amount of shielding. Therefore, is it not possible to totally eliminate the effects of radiation (i.e. to the level found on earth). Radiation causes several different effects on electronics, but the effects are generally categorized as either total dose effects or single events effects. The total dose effects are caused by the accumulated amount of radiation, which gradually degrades the performance of the equipment until failure. Single events are of temporary nature, as they do not cause lasting damage[1] to the affected device.

In the following a special type of single events, known as Single Event Upsets (SEU) will be studied. SEU's cause bits in memory to alter their contents – this is also known as bitflips. The traditional remedy to bitflips has been to implement error detecting and correcting (EDAC) storage, which is based in a hardware-implemented error-correcting code. The intensity of bitflips caused by radiation has been studied in previous satellite missions, for instance [43, 44, 45, 46]. As a significant amount of the bitflip-causing radiation originate from the Sun, the radiation-intensity experienced by an earth-orbiting satellite will vary with the

[1] Provided they are handled appropriately: Single event latchups can cause fatal damage to a device, if not detected and handled in due time.

intensity of the solar-wind, the hight of the orbit and the position in the orbit (due to inhomogeneities in the magnetic field of the Earth). Furthermore, the susceptibility to bitflips varies with both the employed technology and the scale of the technology (the higher the integration scale, the higher the susceptibility). All this means, that the intensity of bitflips is hard to estimate with great accuracy, but a cautious estimate is made, based on intensities publicized in previous studies.

3.3.1 Defining the Model

The effects of bitflips is investigated using fault-injectors, which simulate bitflips. The fault-injectors are implemented as high-priority processes, which alters the contents of the storage at random positions at random moments in time. The effect of the bitflips can be modeled as a number of independent Bernoulli-processes (one for each susceptible bit), but rather than simulating a large number of processes running in continuous time, the injectors are activated on regular intervals. On each activation, the injectors decide if an injection should be made and if so, where it should be made.

The injection model is based on the following assumptions:

- Each failure can be simulated as a separate event.
- Each bit has an equal probability of failure.
- There is a minimal timespan between two events.

The first assumption assures that the processes can be modeled as Bernoulli-processes. Together with the second assumption, they provide the basis for modeling the process as a binomial-process. The third assumption provides for a discrete approximation of the modeled processes. The implemented injection model assumes that faults occur only in regular memory, and not in registers or devices.

3.3.2 Evaluating the Model

In the following it is assumed that bitflips occur with an intensity of $r = 1 \cdot 10^{-5} \frac{1}{\frac{bit}{day}}$ – this is a pessimistic prediction of the fault rate in LEO [2]. As the injectors run with a granularity of 100 Hz, the unit tick is defined to be: $1\ tick = \frac{1}{100}\ s$. Furthermore, d is defined to be the number of ticks per day ($d = 8.64 \cdot 10^6 \frac{tick}{day}$). The number of bits considered in the following is $N = 1 \cdot 10^6$. This is fewer than available on the DTUSat, but it is about the number of bits used for vulnerable storage in the application. The failsafe mode of operation uses a maximum of $3\ K \approx 2.5 \cdot 10^4\ bits$. Refer to section 3.7.1 for a discussion of the two targets.

[2] Low Earth Orbit: Typically LEO's are 600-800 km circular orbits

In any given time-frame, the number of bits that flip can be modeled as a number of individual Bernoulli experiments, with parameters N and $p = \frac{r}{d}$. Let X be a stochastic variable, which models the number of bitflips per tick, then:

$$X \in b\left(N, \frac{r}{d}\right)$$

But since N is large and p is small, the distribution of X can be approximated with a Poisson distribution with parameter $\lambda = N \cdot p$, thus:

$$X \in b\left(N, \frac{r}{d}\right) \approx P\left(N \cdot \frac{r}{d}\right)$$

An important question is whether a granularity of 100 Hz is reasonable. The assumption is that at most one bitflip can occur during each cycle. The probability of more than one bitflip during a cycle can be expressed using X:

$$P\{X > 1\} = 1 - P\{X = 0\} - P\{X = 1\} \approx 7 \cdot 10^{-13}$$

So it is concluded that the probability of having more than one bitflip per cycle is highly unlikely, and the principle of injecting at most one bitflip per cycle is therefore accepted. The time between two consecutive Poisson distributed events is modeled by an exponential distribution with the same intensity. Thus, the time between bitflips should be modeled with:

$$T \in exp\left(N \cdot \frac{r}{s}\right)$$

The estimated value of exponential distribution is:

$$E\{T\} = \left(\frac{s}{N \cdot r}\right)$$

For $N = 1 \cdot 10^6$, corresponding to the application, this gives an estimated time between faults of about $\frac{1}{10}d$, corresponding to about 10 faults per day. For $N = 2.5 \cdot 10^4$, the estimated time between faults is about 4 d, which is approximately one fault per four days. However, in order to carry out the simulations in a fairly limited time-frame of about 250 CPU-hours, the experiments are run with (aggregate) rates in the range of about once per second to once per minute.

3.4 Pseudo-Random Number Generation

As described in the earlier sections, the injectors simulate the stochastic processes of bitflips. The stochastic processes are simulated using a PRNG (Pseudo Random Number Generator). A PRNG is a function which returns a number which, on successive calls, generate a sequence

of numbers that appear to be random. The function maintains an internal state, which dictates the values of the numbers to be returned. A central concept in PRNG's is seeding: Before using the PRNG, the internal state must be initialized. This is done by the user, who supplies a seed-value. How the seed-value is actually used within the PRNG is dependent on the implementation, but a key property of PRNG's is that given a specific seed, the PRNG will always generate the same sequence. This last fact also justify why the generators are called "pseudo random": They are in fact deterministic, but in a seemingly random fashion.

A popular type of PRNG is known as "Linear Congruence Random Number Generators". The sequences of these generators are defined by an equation on the form: $r_{n+1} = (a\, r_n + b)\, mod\ N$. It can easily by shown that this sequence (for the chosen values of a, b and N) has alternating odd and even values, and generally that the lower order bits are not very random at all. However, the ANSI C standard recommend that this type of generator is used to implement the `rand`-function. For this reason, it has been chosen not to use the compiler-supplied PRNG.

The PRNG chosen instead is called "the Mersenne Twister". The reasons for choosing this PRNG is that it is well-renowned [36], it is reasonably fast and that it is portable. The Mersenne Twister comes in two variants: A floating point version and an integer version. Because the chosen architecture does not support floating points in hardware, it has been chosen to use the integer version.

3.4.1 Distributions

The chosen PRNG generates uniformly distributed integer values in the range $\left[0; 2^{32}-1\right]$, but this distribution is rarely needed. Instead this distribution (the general distribution) is used to create specific distributions. Two types of specific distributions are used in the injectors: Uniform and exponential distributions. Intuitively, what is needed is a mapping that maps from the domain of the general distribution to the domain of the specific distribution. In the following it is shown how the general distribution can be mapped into these specific distributions. The notation $E(\lambda)$ will be used to represent an exponential distribution with parameter λ. Note, that the parameter for the exponential distribution is traditionally called α, but in this context, the symbol λ is used to emphasize the relation to the Poisson distribution (which parameter traditionally is called λ). The uniform distribution is represented by the symbol U. In order to distinguish between discrete and continuous uniform distributions, either a set or a range is used as argument to the distribution, for example $U\left\{0...2^N\right\}$ or $U[0;1]$.

Uniform Distribution

A tempting way to implement a specific uniform distribution, $X \in U\{0...N-1\}$, is to use the modulus to implement the mapping, such as: $r_{specific} = r_{general}\ mod\ N$. However, unless the size of the domain of the general distribution is divisible by N, it will not create a uniform distribution. This is caused by the $2^{32}-1\ mod\ N$ largest values of the general distribution

being mapped to the $2^{32}-1$ *mod* N smallest values of the specific distribution (thus over-representing these values).

Instead of using the modulus mapping, a different technique known as ”the rejection method” is used: A value is generated from the general distribution. If this value is less than N, it is used as a value of the specific distribution, else a new value is generated, until a usable value is found. Of course, this would be very wasteful if the distribution of the PRNG was used directly. Instead, an intermediate uniform distribution can be created, which serves as input to the rejection algorithm. This can be done efficiently simply by masking off bits of the PRNG distribution. Thus, a mask is created, which maps $G \in U\left\{0...2^{32}-1\right\}$ to $I \in U\left\{0...2^{L}-1\right\}$, where $L = \lceil log_2(N) \rceil$. Then the algorithm can be described by the following pseudo-code:

$$\begin{array}{l} do \\ \quad u \in I; \\ while\ (u \geq N) \\ return\ u; \end{array} \tag{3.1}$$

Exponential Distribution

The cumulative function of the exponential distribution is defined as: $F(t) = 1 - e^{-\lambda t}$. An exponential distribution can be created simply by ”running it backwards”, thus isolating t gives:

$$t = \frac{1}{\lambda} \cdot -ln(1 - F(t)) \tag{3.2}$$

When running it backwards, $F(t)$ should be substituted with a continuous uniform unity distribution. Because $F(t) \in U[0;1]$, then $1 - F(t) \in U[0;1]$. Note that the expression $-ln(x)$ can be tabulated, thus eliminating the need to calculate the logarithms.

3.5 Error-Correcting Codes

The theory of error-correcting codes is comprehensive, and this presentation will not try to cover the general theory. However, a brief overview of the applied theory will be given in the following sections.

Error-correcting codes is a technique used to protect data from corruption due to errors in transmission or storage. In this context, an error means that the value of one or more of the symbols in the data has been changed. I.e. the described code cannot handle erroneous exclusion or inclusion of symbols.

3.5.1 General Codes

Error-correction is accomplished by two types of transformation: Encoding and decoding. Encoding is the process of transforming a data-word into a code-word, and decoding is the reverse process. A data-word and its associated code-word represent the same information, but the data-word has a representation which is directly usable, while the code-word has a representation which is tolerant to one or more errors. Thus, the code-word representation is used in contexts where errors are anticipated.

The theory of error-correcting codes is based on linear algebra, in that data-words and code-words are treated as vectors, and the encoding and decoding operations are based on matrix-multiplications. The matrices used for encoding and decoding are known as the generator-matrix (traditionally called $\underline{\underline{G}}$) and the parity-matrix (traditionally called $\underline{\underline{H}}$). Using these matrices, the encoding and decoding operations are defined as follows (the notation is due to [1]):

$$\underline{c} = \underline{d} \times \underline{\underline{G}} \tag{3.3}$$

$$\underline{s}^T = \underline{\underline{H}} \times \underline{c}^T \tag{3.4}$$

Where d is a data-word, c is a code-word and s is the syndrome for c. The syndrome is an encoding of the state of the code-word – from this, one can tell how many and which bits in $\underline{c}$ have been changed. Note that the vectors used in equation 3.4 are columns, but in the following, all vectors are assumed to be rows. Transposing equation 3.4 yields:

$$\underline{s} = \underline{c} \times \underline{\underline{H}}^T \tag{3.5}$$

The matrices $\underline{\underline{G}}$ and $\underline{\underline{H}}$ are bound together by the relation, that if $\underline{\underline{G}} = \left[\underline{\underline{I}} : \underline{\underline{A}}\right]$ is a generator matrix for a code, then $\underline{\underline{H}} = \left[-\underline{\underline{A}}^T : \underline{\underline{I}}\right]$ is a parity check matrix for the code (due to [1]). When the matrices are on the form $\underline{\underline{G}} = \left[\underline{\underline{I}} : \underline{\underline{A}}\right]$ and $\underline{\underline{H}} = \left[-\underline{\underline{A}}^T : \underline{\underline{I}}\right]$, they are said to be in standard form. Inserting $\underline{\underline{G}}$ in normal form into equation 3.3 yields the following:

$$\underline{c} = \underline{d} \times \left[\underline{\underline{I}} : \underline{\underline{A}}\right] = \left[\underline{d} : \underline{d} \times \underline{\underline{A}}\right] = \left[\underline{d} : \underline{p}\right] \tag{3.6}$$

That is: The code-word $\underline{c}$ consists of the data-word $\underline{d}$ appended with parity-bits $\underline{p} = \left[\underline{d} \times \underline{\underline{A}}\right]$. The code-word may be subject to bit-errors – let $\underline{c'} = \left[\underline{d'} : \underline{p'}\right]$ denote a code-word created using equation 3.6, subject to zero or more bit-errors. Inserting $\underline{\underline{H}}$ on normal form and $\underline{c'}$ into equation 3.5 yields the following:

$$\underline{s} = \left[\underline{d'} : \underline{p'}\right] \times \begin{bmatrix} -\underline{\underline{A}} \\ \underline{\underline{I}} \end{bmatrix} = -\underline{d'} \times \underline{\underline{A}} + \underline{p'} \tag{3.7}$$

Note that 3.6 and 3.7 eliminate use of $\underline{\underline{G}}$ and $\underline{\underline{H}}$, but require that $\underline{\underline{A}}$ is available (i.e. that $\underline{\underline{G}}$ and $\underline{\underline{H}}$ are on standard form).

3.5.2 Binary Codes

The previous section describes how encoding and syndrome-decoding of codes with arbitrary bases are performed. However, in practice the only relevant codes are binary, and this fact can be used to simplify some of the calculations. Note, that in binary $a = -a$ and that multiplication and addition amounts to logical **and**, and logical **exclusive-or** respectively. Let $\otimes$ denote logical and, and let $\oplus$ denote logical exclusive-or, then 3.6 and 3.7 can be expressed as:

$$\underline{c}_b = \left[\underline{d}_b : \underline{d}_b \otimes \underline{\underline{A}}_b\right] = \left[\underline{d}_b : \underline{p}_b\right] \tag{3.8}$$

and

$$\underline{s}_b = \left(\underline{d'}_b \otimes \underline{\underline{A}}_b\right) \oplus \underline{p'}_b \tag{3.9}$$

3.5.3 Example: The Hamming (7,4) Code

For example, let the following generator and parity matrices specify a binary code (due to [1]):

$$\underline{\underline{G}}_H = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \text{ and } \underline{\underline{H}}_H^T = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

Encoding a data-word, say $[0101]$ yields: $[0101] \times \underline{\underline{G}}_H = [0101010]$. Decoding the code-word yields: $[0101010] \times \underline{\underline{H}}_H^T = [000]$, signifying no errors. Introducing a bit-error in the code-word, for instance in bit 3, yields the following decode: $[0111010] \times \underline{\underline{H}}_H^T = [011]$, signifying an error in bit 3.

The reason that the syndrome directly evaluates to a bit-number is that each row in the parity check matrix encodes its number in binary. As a consequence of this, the parity check matrix is not in standard form. Using Gauss-elimination, the parity check matrix can be brought into standard form:

$$\underline{\underline{H}}_H = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix} \sim \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} = \left[\underline{\underline{A}}_H^T : \underline{\underline{I}}\right]$$

Then, note that $\underline{\underline{G}}_H = \left[\underline{\underline{I}} : \underline{\underline{A}}_H\right]$

3.6 Creating a Hamming (12,8) Code

In this section it will be described how the Hamming code used in chapter 6 is created. The classification (12,8) means that the code uses 12 bit code-words and 8 bit data-words, thus leaving 4 bits for parity. As the syndrome is encoded with the same number of bits as the parity, there are 16 possible syndromes, one of which encodes ”no error”. Therefore the syndrome can encode errors in 15 different bits, but because the code-words are only 12 bits, the code is said to be in-perfect. The Hamming (12,8) is actually a trimmed-down version of the perfect Hamming (15,11) code. Note that the Hamming (7,4) code of section 3.5.3 is also perfect.

3.6.1 Constructing the Code

As [1] states, a Hamming-code with r parity bits has a parity matrix with columns that are all different non-zero r-tuples. But as the (12,8) code is in-perfect, only 12 of the 15 r-tuples are selected.

$$\underline{\underline{H}} = \begin{bmatrix} 0&0&0&0&0&0&0&1&1&1&1&1 \\ 0&0&0&1&1&1&1&0&0&0&0&1 \\ 0&1&1&0&0&1&1&0&0&1&1&0 \\ 1&0&1&0&1&0&1&0&1&0&1&0 \end{bmatrix} \sim$$

$$\begin{bmatrix} 0&1&1&1&1&0&0&1&1&0&0&0 \\ 1&0&1&1&0&1&0&1&0&1&0&0 \\ 1&1&0&1&0&0&1&1&0&0&1&0 \\ 0&0&0&1&1&1&1&0&0&0&0&1 \end{bmatrix} = \left[\underline{\underline{A}}^T : \underline{\underline{I}}\right]$$

From $\underline{\underline{A}}$ the generator is trivially constructed:

$$\underline{\underline{G}} = \begin{bmatrix} 1&&&&&&&&0&1&1&0 \\ &1&&&&&&&1&0&1&0 \\ &&1&&&&&&1&1&0&0 \\ &&&1&&&&&1&1&1&1 \\ &&&&1&&&&1&0&0&1 \\ &&&&&1&&&0&1&0&1 \\ &&&&&&1&&0&0&1&1 \\ &&&&&&&1&1&1&1&0 \end{bmatrix}$$

Since the generator and parity check matrices are in standard form, encoding and decoding can be done using $\underline{\underline{A}}$ (using equation 3.8 and 3.9), instead of $\underline{\underline{G}}$ and $\underline{\underline{H}}$ (using equation 3.3 and 3.5). This simplifies the calculations, and saves some space.

3.6.2 Decoding the Code

Assume that the data-word $\underline{d}$ is encoded using equation 3.8. This produces:

$$\underline{c} = \left[\underline{d} : \underline{d} \otimes \underline{\underline{A}}\right]$$

Decoding $\underline{c}$ amounts to:

$$\underline{s} = \left(\underline{d} \otimes \underline{\underline{A}}\right) \oplus \underline{p} = \underline{p} \oplus \underline{p} = \underline{0}$$

That is: When the codeword has incurred no errors, the syndrome is $\underline{0}$. Then assume that the codeword has incurred a single bit-error; let $\hat{\underline{d}}$ denote a codeword in error. Let $\hat{\underline{e}}_i$ denote an error bit-vector, which is all zeroes except for position i, which is a one. Now assume that the codeword has sustained a single error in bit number i in the datapart, then $\hat{\underline{d}} = \underline{d} \oplus \hat{\underline{e}}_i$ and $\hat{\underline{p}} = \underline{p}$. Then the calculation of the syndrome becomes:

$$\begin{aligned}
\hat{\underline{s}} = \quad & \left(\hat{\underline{d}} \otimes \underline{\underline{A}}\right) \oplus \hat{\underline{p}} = \\
& \left(\left(\underline{d} \oplus \hat{\underline{e}}_i\right) \otimes \underline{\underline{A}}\right) \oplus \underline{p} = \\
& \left(\left(\underline{d} \otimes \underline{\underline{A}}\right) \oplus \left(\hat{\underline{e}}_i \otimes \underline{\underline{A}}\right)\right) \oplus \underline{p} = \\
& \underline{p} \oplus \left(\hat{\underline{e}}_i \otimes \underline{\underline{A}}\right) \oplus \underline{p} = \\
& \hat{\underline{e}}_i \otimes \underline{\underline{A}}
\end{aligned}$$

That is: When an error occurs in the datapart of the codeword, the syndrome takes on the value of the row in $\underline{\underline{A}}$, which has the same number as the bit in error in the datapart. Now assume that an error has occurred in the parity-part, that is $\hat{\underline{d}} = \underline{d}$ and $\hat{\underline{p}} = \underline{p} \oplus \hat{\underline{e}}_i$, then the syndrome decodes as follows:

$$\begin{aligned}
\hat{\underline{s}} = \quad & \left(\hat{\underline{d}} \otimes \underline{\underline{A}}\right) \oplus \hat{\underline{p}} = \\
& \left(\underline{d} \otimes \underline{\underline{A}}\right) \oplus \left(\underline{p} \oplus \hat{\underline{e}}_i\right) = \\
& \underline{p} \oplus \underline{p} \oplus \hat{\underline{e}}_i = \\
& \hat{\underline{e}}_i
\end{aligned}$$

Thus, when an error occurs in the parity, the syndrome has the value of the bit in error. Note that due to the structure of the parity-check matrix, the syndrome takes on the value of a row in $\underline{\underline{H}}$, which has the same number as the bit in error in the codeword.

3.7 Experiments

A general fault-injection experiment is carried out using two entities: A target and a fault-injector. The target is the entity in which fault are injected, while the injector causes the fault. Generally, both entities can be implemented in both hardware and software, but it was chosen to implement both in software, running on the same processor. The major advantage of this

setup is that it is quite simple to implement the injector. In the following, ”target application” denotes an entity used as a target for fault injections.

There are two principal methods in which fault injection experiments can be carried out: Single injection or continuous injection. In the single injection method, an experiment is composed of a single injection, followed by an examination of the effects. In the continuous injection method, faults are injected continuously, until the system is affected. While both methods are destructive, the former method does not necessarily lead to a failure, while latter does (it can be described as test-until-destruction). Of course, in either case the system must be restarted between successive experiments to reverse the effects of the previous fault injections. In the following experiments, the latter method was used.

An important aspect of fault-injection experiments is not to cause more disturbance to the target, than intended by the fault-injections. As the fault injector is running on the same processor as the target application, this will inevitably cause some overhead, and if the overhead is large enough, it may affect the scheduling properties of the target application, thus influencing the validity of the experiment. Therefore, emphasis has been put on reducing the resources used by the injectors.

The experiments were performed on a prototype of the DTUSat onboard computer (OBC), connected to a PC running Linux using two serial cables. The target application and the injector were run on the OBC, while the PC was used to load and monitor the progress of the application/injector. The combination of the OBC and the software running on it is called ”the clientside”, while the PC and its software is known as ”the hostside”.

3.7.1 Targets

What the above section refers to as a ”target application”, is actually two different targets. The first target known as ”failsafe”, is the flight-version of DTUSat boot-software. The second target, known as ”the application”, is an application which resembles the structure of the DTUSat flight application.

The two targets use different firmware: On the failsafe experiment, the firmware is the failsafe software, which is also the injection target. On the application, the firmware is a debug-stub (GDB), which is used to load and run the application. The debug-stub is only used for support, and is not an injection target itself.

3.7.2 Failsafe Software Injection Experiments

The failsafe software implements a number of telecommands, for instance for uploading and executing of software and inquiry of the status of the software. In the failsafe experiment a fault-injector is uploaded in the form of a small application. When the injector has been started, it runs in the background, thus letting the failsafe software resume its normal operation. In order to detect when the operation of the failsafe software is affected by the injected faults, its status is continuously queried during the experiments.

The objectives of these experiments are to investigate how many injections the failsafe software can resists, and to determine its failure modes and their frequencies.

3.7.3 Application Injection Experiments

The application is run on eCos, an embedded realtime operating system. The application is implemented as a number of loosely-coupled modules, which communicate by passing messages trough a shared packet-router. The fault-injector used for the application, is implemented as a separate high-priority thread, which is a part of the application. As in the failsafe experiment, the status of the application is continuously monitored, in order to detect when it has been affected by the injections.

The objectives of these experiments are the same as for the failsafe experiment, and in addition to investigate how countermeasures to the faults can be implemented in software. The effects of the countermeasures are measured.

3.8 Hardware

This section presents the platform on which the experiments are run. As mentioned above, the hardware is a prototype of the DTUSat onboard computer (OBC). The computer comprises the following components:

- CPU: Atmel AT91 (ARM 7 TDMI variant).
- RAM: 1 M.
- FLASH: 2 M.
- ”PROM”: 1 M.

The CPU is a 32-bit RISC processor clocked at 16 MHz. On the prototype, the PROM is actually a 1 M FLASH – on the flight version, this was replaced with a 32 K PROM. The PROM contains the firmware of the system – in this context, the firmware is the software used to boot the system. The flash is not used in the experiments.

The AT91 comprises a number of peripheral devices, which simplify interfacing to other equipment:

- 8K Internal RAM
- 2 USARTs
- An Advanced Interrupt Controller (AIC)
- 3 Timer/Counter devices (TC)

- A Watchdog timer (WD)

The first 32 bytes of the internal RAM is reserved for the exception vectors (see section 4.2.4), but otherwise it is available for general use. The two USARTs are connected to a level-converter, implementing two RS232 ports. In the original (DTUSat) configuration, port 0 was used to interface to the radio-modem and port 1 was used for debugging purposes, using a cable. On the flight version software, the radio-modem port is configured to 2400 baud, while the debugging port is configured to 9600 baud, but otherwise they are interchangeable.

However, as the prototypes are not equipped with radio-modems, the radio-modem port is instead used to facilitate the experiments. The port is reworked such that the OBC can be reset from the RS232 port – this is done by connecting the RTS signal to the Reset pin through the level-converter. This leaves the Rx-signal unconnected, but this is not a problem as it is not used. The Tx-signal remains, which allows status information to be output from OBC. In the following, port 1 is known as the debugging port, while port 0 is known as the auxiliary port.

The use of the AIC, TC's and the WD is explained in sections 4.2.4, 4.2.5 and 4.2.6. In the application, the AIC and TC is handled by the operating system (eCos).

Chapter 4

The Failsafe Injector

In this chapter the failsafe injector is discussed. First the environment in which the injector is running is described, followed by a discussion of the implementation the injector. Then the harness used for performing automated experiments is discussed, and the results gathered from the experiments are presented. Finally, some observed weaknesses are discussed, and the testing of the implementation is described.

4.1 Environment

In the following, the environment in which the failsafe-injector runs is described.

As previously explained, the injector is run on a prototype of the DTUSat onboard computer, loaded with the flight version software. Except for a few voltage and current-sensors being absent, the computer is functionally identical to the DTUSat onboard computer. However, since the failsafe software does not make use of these sensors, it is compatible with their absence. The payload and support modules are also missing from the prototype, but the only modules used by the failsafe software is the power and radio -module. The DTUSat and the prototype has a switch, that selects between the ”docked” and ”non-docked” modes of operation. The major difference between the modes is which port is used for communications: In docked mode a local serial port is used, while in non-docked mode the radio-modem is used. As docked mode is selected during the tests, the failsafe software does not try to use the (absent) radio module. The failsafe software will try to read a few voltage- and current- sensors on the absent power-module, but due to their absence, they will all read 0. The sensor-values are used for returning status-information over the communications interface, and are not used for calculations or regulations, so the failsafe software is compatible with the absence of the sensors. The failsafe software can handle a number of temperature sensors, but the number of sensors and their hardware addresses have to be declared in a special flashblock, but since none are declared, none are used. As justified above, the flight version of the failsafe software is compatible with the prototype OBC, provided that it is docked. Furthermore, aside from the

absent devices, the computer and the failsafe software is functionally equivalent to that found on the DTUSat.

4.1.1 FSTERM

The failsafe software implements a simple protocol, which allows commands to be executed on the DTUSat using the communications interface (whichever is chosen). The protocol implements a simple request/response interface, in which requests are sent from the hostside to the satellite, and responses are sent from the satellite to the hostside. Apart from beacons, all communication is initiated by a request, which must be followed by a response. Thus, there is no queuing of requests or responses.

The requests are used to implement telecommands, and the responses are used to implement statuses for telecommands. Using the telecommands, it is possible to upload and run software on the OBC. A program which is compatible with the failsafe environment is known as a failsafe program. The telecommands used for uploading and running failsafe programs are: `upload`, `call_function` and `get_status`. The uses of these telecommands are explained in section 4.4.

The hostside program `fsterm` implements a general purpose terminal interface to the telecommands implemented by the failsafe software. The program can communicate with the satellite over either interfaces, and in both interactive and batch mode. Refer to [13] for a detailed description of the commands offered by `fsterm`. The term ”fs-compatible” will be used to denote a hostside program, which can communicate with the failsafe software using the failsafe telecommands.

4.1.2 Creating a Failsafe Program

As with all embedded software, the programming model of failsafe programs is limited compared to that of, say, a desktop system. Beside the support for binary upload, and invocation of functions on arbitrary addresses, the failsafe software does not offer any services to failsafe programs.

Using the fsterm command `call_function`, it is possible to call functions on arbitrary addresses in memory. A failsafe program may implement functions intended to be activated using this mechanism. Such functions are known as exported functions. In a trade-off between generality and simplicity, it has been decided that exported functions must take an `unsigned int` argument, and return a `unsigned int` value. Due to the watchdog, exported functions should return within 4 seconds. When an exported function is invoked through the functions call telecommand, it executes using the stack of the failsafe software. As the failsafe stack is of limited size, care should be taken not to overrun the stack.

Under normal conditions, the failsafe software will allocate its stack in the internal memory of the processor. As the failsafe software at most will use 3K of stack-space, there is 5K left on the stack for failsafe programs. The failsafe software does not make use of the external

memory, so this is free to be used by failsafe programs. Failsafe programs (code and data) will normally be uploaded to, and run from the external memory (which is also the case for the failsafe injector).

As the failsafe software does not offer any runtime services, a failsafe program must supply its own runtime support. For instance, the C-library is not directly supported and all input/output must be implemented from scratch. As floating points are not implemented in the processor, all floating point operations are emulated in software, thus being quite time-consuming. For a more complete coverage of the execution model of failsafe programs, refer to [14].

4.2 Implementing

As mentioned earlier, the failsafe-injector is implemented as a failsafe program, which is uploaded and run using failsafe telecommands. The failsafe software runs as a single task, which contains a simple command-loop (interpret command, carry out command, return status), and as such the injector has to run in the background, if it is to inject faults at random moments in time. This is achieved by attaching the injector to a timer interrupt, by which it is activated on regular intervals. The injector implements a number of detectors, which are used to detect if the failsafe software performs erroneously. In order to support the functionality of the injector, the second serial port is used for injector status information.

4.2.1 Injector Callback

As mentioned above, the injector is attached to a timer interrupt, and is activated as a callback function on regular basis. The timer generates interrupts at 100 Hz, but as this is a somewhat higher rate than needed, so the rate is subdivided to generate lower rates. The injector implements this by means of a decrementing counter. On each invocation of the injector the counter is checked, and if the counter is non-zero it is decremented and the injector function returns. When the counter reaches zero, a fault is injected and the counter is reset to a non-zero value. Thus, the basic injector callback function looks as follows:

```
1  void injector_cb(void)
2  {
3    if(count--<=0)
4      {
5        injector(base,length);
6        count=dist_exp(lambda);
7      }
8  }
```

Here `count` is the global decrementing counter, used to subdivide the rate. The address-range used for fault-injection is pointed out by `base` and `length`. The fault-rate is given by `lambda` (on average `lambda` invocations per fault).

Injection

The `injector` part of the fault-injector simply works by selecting a random bit in the range `base` and `length`. The bit-address is uniformly distributed over the `length`. Thus, the basic injector looks as follows:

```
1  void injector(void *base, unsigned int length)
2  {
3    unsigned char *fault;
4    fault=((unsigned char*)base)+dist_uniform(length);
5    *fault ^= 1 << dist_uniform(8);
6  }
```

4.2.2 Exported Functions

The failsafe injector implements a number of exported functions, which are used to initialize, configure and start the injector. In the following subsections, the usages of these functions are described. All functions return a dummy value (0), and the functions whose arguments are not described (`start` and `init`), take dummy arguments.

Entrypoint: `init`: The function `init` is used to initialize the failsafe program. Because the operations carried out in `init` are rather time-consuming, the first thing `init` does is to disable the watchdog. As it is the responsibility of a failsafe program to clear its own BSS, this is done next. Then all bytes of unused internal and external memory is initialized to `0xEF`. This is used to detect runaway executions, as a word of all `0xEF` bytes triggers an exception when executed. Lastly, the on-chip interrupt controller is initialized.

Entrypoint: `set_seed`: The function `set_seed` is used to seed the PRNG. It is good practice to use a seed derived from the wall-clock time, but since the OBC only contains timers which are reset on processor reset, using these would always yield the same seed value (or values within a small range). The solution is to use a seed supplied from the hostside, which is the purpose of this function.

Entrypoint: `set_mode`: The injector supports two modes in which the time between events can be modeled: ”constant time” (mode=0) and ”exponential stochastic time” (mode=1). Constant time -mode was used during development because of its simplicity, however if Poisson distributed events are to be modeled, the exponential stochastic time -mode must be used.

Entrypoint: `set_rate` : The average time between events is selected with this function. Note, that this is actually the reverse of the rate, i.e. not events per unit of time, but units of time between events. Due to details of the implementation, the time-unit of the constant time mode is $\frac{1}{100}$ *s*, while the time-unit of exponential stochastic mode is $\frac{1}{1600}$ *s*.

Entrypoint: `start` : As the name suggests, this function starts the injection. However, before the injector is started, the other major module of the failsafe injector, the detector, is started. The detector is used to detect if and when the failsafe software performs an illegal execution. The detector implements several means of detection, which are discussed in section 4.2.6.

4.2.3 Stack Usage

As mentioned in section 4.1.2, failsafe programs initially use the same stack as the failsafe software itself, which limits the amount of memory that can be allocated off the stack. However, the exported functions of the failsafe injector are ”small” functions, which will not overflow the failsafe stack, but some stack management is still needed, as interrupt-handlers have to run on separate stacks on the ARM architecture. Therefore, an interrupt stack is set up as a part of the function prologue to `start` – the function also uses this stack, even though it should be ”small” enough to run, using the failsafe stack.

4.2.4 Vectoring

The ARM architecture implements a number of so-called vectors, which are used to activate interrupt and exception handlers. The first 8 words of the address-space are used for the vectors, of which there are 7 defined (leaving one vector reserved for future extensions). As the failsafe injector is event-driven, it depends heavily on interrupts and exceptions. The following table briefly describes the vectors used in the injector.

Undefined Instruction : As the name suggests, this exception is raised if the processor attempts to execute an undefined instruction.

Software Interrupt (SWI) : Normally software interrupts are used to implement system calls. A software interrupt is raised upon execution of a `SWI` instruction.

Prefetch Abort : A prefetch abort exception is raised if the processor tries to read an instruction from an address, which lies outside the address-space.

Data Abort : A data abort exception is similar to the prefetch abort, but is raised when the processor tries to make a data access (read or write) outside the address-space.

Interrupts Request (IRQ) : Interrupts are raised by peripheral devices (internal or external to the processor). Note that since only one interrupt request vector is defined, a request from any device will raise this exception. It is the responsibility of the interrupt-handler to determine which device raised the interrupt.

Interrupts

The AT91 processor implements a number of independent interrupt sources, each of which can be independently prioritized and masked by means of the Advanced Interrupt Controller. The AIC also provides the source of the interrupt to the interrupt-handler, thus simplifying the handler.

As the injector make use of more than one interrupt source, the common code to attach and handle interrupts has been implemented as a common module. The interface to the module consists of two functions:

- `void aic_init(void)`: This function is used to initialize the AIC, and must be called before any interrupts can be used.
- `void aic_register(void (*handler)(void), unsigned int mode, unsigned int id)`: This function attaches a callback 'C' function to the interrupt with number `id`. The `mode` controls the interrupt priority, and whether the interrupt should be level or edge triggered.

The premise of the failsafe software was that it was implemented as simple as possible, which means that all device-accesses was implemented using polling. As the failsafe software does not use interrupts, the failsafe injector find the AIC is in its reset state. The function `aic_init` first disables all interrupts in the AIC [1], and then attaches the interrupt-callback (`aic_cb`) to the IRQ vector.

For each interrupts-source, the AIC has a register which holds the address of the associated handler. The AIC assists the interrupt handling by means of the "IRQ vector register", which, on the activation of an interrupt, is loaded with the address of the handler of the corresponding interrupt. Therefore, in order to activate the callback of an interrupt, the interrupt-handler only has to call the function pointed out by this register. The interrupt mechanism is used for generation of the timer clock, which drives the fault-injection, and for watchdog timeout detection.

4.2.5 Timer Generation

The AT91 processor implements three timer-counter devices, which can be programmed to generate a combination of waveforms and interrupts. The waveforms can be generated on

[1] In the reset state the interrupts should be disabled, but it does not hurt to do it again, just to be safe.

both the external pins of the processor, and on internal 'pins' connected to the other counters.

The failsafe software uses the first timer (timer 0) to create a 100 Hz clock, which is used to clock the two other timers. The two other timers are then used by the failsafe software to create precise delays. Since timer 0 is already programmed to generate a 100 Hz clock, it can be reprogrammed to also generate interrupts with this rate. As the interrupt-generation is independent from the waveform generation, this does not interfere with the normal operation of the failsafe software.

The injector callback function is attached to the interrupt of timer 0.

4.2.6 Detectors

An important component of the failsafe injector is the detectors, which are used to detect if and when the failsafe software performs illegal executions. Of course, the failsafe software does contain detectors of its own, but these are not really suitable for carrying out experiments. Therefore, the injector replaces the existing detectors with its own. The replaced detectors correspond to the following exceptions:

- Undefined Instruction
- Prefetch Abort
- Data Abort
- Software Interrupt (SWI)
- Watchdog Timeout (IRQ)

The reasons for catching the exceptions "Undefined Instruction", "Prefetch Abort" and "Data Abort" should be obvious.

As neither the failsafe software nor the failsafe injector use or define software interrupts, the processor should never execute `SWI` instructions. This fact is utilized to detect runaway execution, i.e. if the processor starts to execute where there should be no code: The unused bytes of RAM is filled with the value `0xEF`, which will trigger a software interrupt if executed. An unconditional `SWI` instruction has the general format `0xEF******`, where the asterisks may be any 3 bytes value. So generally, any word starting with `0xEF` should trigger the software interrupt. However, it is possible that the injected faults cause the execution to transfer to an unaligned address. If unaligned data (of any kind) is read by the processor, the RAM interface will rotate the data one or more byte-positions, presenting corrupted data to the processor. For instance, if the word `0x12345678` is placed on an aligned address, a read on address+1 will yield `0x78123456` (if the target is little-endian). However, the processor can have only one interpretation of the word `0xEFEFEFEF`, no matter how many byte-positions it has been rotated.

The failsafe software uses a watchdog timer to ensure that the software does not deadlock. If the watchdog is not reloaded on regular intervals, it will timeout and cause one or more events, depending one how it is programmed. In the failsafe software, the watchdog is programmed to perform an internal reset of the processor in the event of a timeout. However, as this behavior is not suitable when performing experiments, the watchdog is reprogrammed to raise an interrupt instead.

4.2.7 UART

As described in section 3.8, the AT91 contains two serial ports. The debugging-port is used to communicate with a fs-compatible hostside program, while the auxiliary-port is used for returning information from the injector. As the failsafe software does not provide a serial driver for failsafe programs, the injector has to implement its own. The interface to failsafe UART-driver consists of three functions:

- `void uart_start(void)`: This function is used to initialize the UART, and must be called before any bytes are sent or received.
- `void uart_sendarray(unsigned int count, const char *buffer)`: This function transmits the contents of the buffer on the port.
- `char uart_receivebyte(void)`: This function receives (polls) a byte from the port.

In order to cause the least possible disturbance to the failsafe software, the UART-driver is optimized to use as little CPU-time as possible. This is done by two means: First, the UART is set up to run fairly fast (19200 baud), and secondly, the driver uses the hardware-provided ”Peripheral Data Controller”. Each UART has an associated Peripheral Data Controller, that can automatically transfer blocks of data between the UART and the memory, without intervention from the CPU (besides the initiation of the transfer).

While the UART-driver does contain the function `uart_receivebyte`, it is not used by the injector.

Status Information

The status information returned on the auxiliary port is formated as lines of text. For each event, the failsafe injector returns one or more lines of information. For each injection, the injector returns two lines of status information:

- **Injection:** `I<address> <before> <after>`
  `delay=<time>`
  `address` is the address of the fault-injection, `before` and `after` are contents of the 32-bit word covering `address` before and after injection. `time` is the time (in number of ticks) before the next injection.

Each detector returns a single line of information when activated. The format of the information returned by the detectors share the first two values – these are the `time` in which the injector has been active, and the `count` of faults injected. Following these two values, zero or more detector-specific values can follow, as described below:

- **Undefined Instruction :** `U<time> <count> <address>`
  `address` is the address of the offending instruction.
- **Prefetch Abort :** `U<time> <count> <address>`
  `address` is the address of the offending instruction
- **Data Abort :** `U<time> <count> <address1> <address2>`
  `address1` is the address of the offending instruction, while `address2` was intended to be used for the offending address (i.e. address of the datum being attempted loaded or stored). However, this requires that the offending instruction is decoded, and was not implemented.
- **Software Interrupt :** `S<time> <count> <address>`
  `address` is the address of the offending instruction
- **Watchdog :** `W<time> <count>`

All values are returned as 32 bit unsigned hexadecimal values.

4.2.8 Injector Main Components

Figure 4.1 shows the main components of the failsafe injector, and their internal communications. As can be seen, all components are driven by events, originating from hardware interrupts or exceptions.

4.2.9 Creating the Failsafe Program

An important aspect of embedded programming is linking. As mentioned in section 4.1, the failsafe software does not provide support for loading of executables, which complicates the loading of the program. In the following it is described how the failsafe injector is linked, and how the binary image of the program is created.

Linking

As the platform does not support virtual memory, the linking of the program, has to take this into account, and place the executable on the correct address.

There are basically two methods of controlling the location of a program: Either through a linker-script, or through the commandline options. A linker-script allows for detailed control

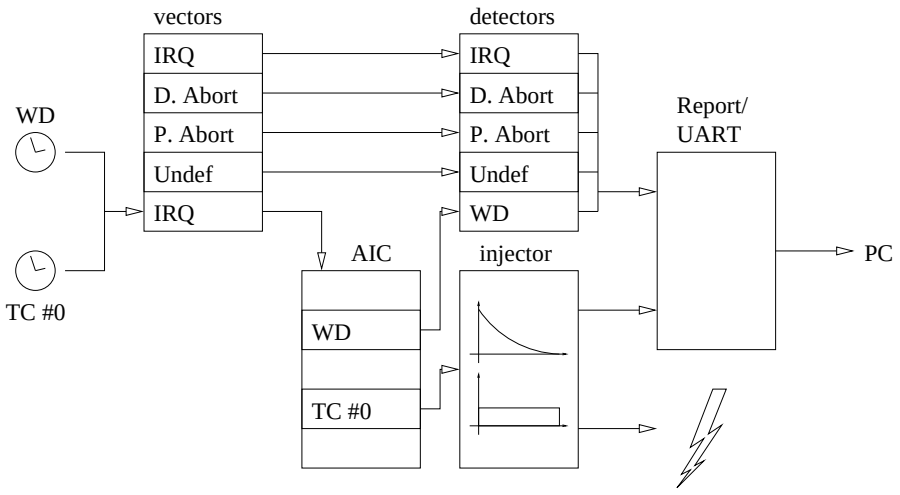

Figure 4.1: Main components of the failsafe injector, and the communication between these.

of the individual sections and symbols of the program, while the commandline options only allows the base-address of each section to be specified. As the failsafe injector does not need fine-grained control over the linking process, the latter method is used.

When the commandline method is used, the linker allows specification of base-addresses of the three major program sections: `text`, `data` and `bss`. If only the base-address of the `text` section is specified, the linker will place the `data` and `bss` sections immediately following the `text` section.

Therefore, the failsafe injector is simply linked by setting the base-address of the `text`-section to `0x02020000`. The linker outputs the executable in elf-format, but since the failsafe software can only handle raw data, a binary image must be generated, which is done using the `objdump` utility. The binary image generated by the `objdump` utility can be directly loaded using the failsafe software and an fs-compatible program.

Exported Functions

From a practical point of view, it is an advantage if the exported functions are located on preselected locations in memory. Below is shown the exported functions and their respective entrypoints:

Entrypoint	Function
`0x2020000`	**init**
`0x2020004`	**start**
`0x2020008`	**set_seed**
`0x202000c`	**set_mode**
`0x2020010`	**set_rate**

Note that the exported functions seemingly only occupy a single instruction each. This is because the initialization-code (`init.S`) contains a table of stubs, which dispatch the control to the real functions. When the linker links a program, it generally emits the symbols in the order in which they are encountered in the objects, which means that the initialization-code must be linked first, in order for the table of stubs to be placed correctly.

4.3 Distributions

As stated in section 3.4, the chosen PRNG is the ”Mersenne Twister”. The Mersenne Twister is seeded by an `unsigned long` value, and generates `unsigned long` values. The seeding function is called `sgenrand`, and the randomizing function is called `genrand`.

4.3.1 Uniform Distribution

Specific uniform distributions can be created using the function `dist_uniform(last)`, which creates uniformly distributed values in the range $[0; last - 1]$. Recall from equation 3.1 that a specific uniform distribution can be generated from a general uniform distribution by using the rejection method. The rejection method requires a mapping, which can create a intermediate distribution, and a predicate, which can determine whether a value from the intermediate distribution is located in the specific distribution.

For the mapping a bitmask is chosen, which maps to the smallest n-bit integer-type, which can represent integers in the range $[0; last - 1]$. Thus, a sequence of bitmasks $mask_n$ is created, such that the binary representation of $mask_n$ has the n least significant bits set to one (or: $mask_n = 2^n - 1$, $n \geq 0$). The smallest n, for which $last < mask_n$ is then sought for.

The ”accept and reject” iterations can then be performed: A general random number is generated, masked with $mask_n$, and compared to $last$. If the number is larger or equal to $last$, it is rejected, and the algorithm performs another iteration. If the number is less than $last$, it is accepted, and the number is returned.

The specific uniform distribution is implemented as follows:

```
1  unsigned long dist_uniform(unsigned long last)
2  {
3    unsigned long mask, srn;
4    unsigned int numbits;
5
6    // first find the smallest bitmask, such that last&mask==last
7    for(mask=0, numbits=0; last >= mask; mask=(mask<<1)|1, numbits++);
8
9    for(srn=(genrand()>>(32-numbits))&mask; srn >= last;
10       srn=(genrand()>>(32-numbits))&mask);
```

```
11
12      return srn;
13  }
```

Note that the implementation uses the most significant bits of the general distribution, rather than the more intuitive least significant ones. The reason for this is that the more significant bits are generally more random than the least significant ones (according to [6]). Since the number of rejectable values in the intermediate distribution is always less than the number of acceptable values, the probability of selecting an acceptable value is greater than 0.5. Therefore, the estimated number of iterations it takes to produce a value in the intermediate distribution is between one and two.

4.3.2 Exponential Distribution

Recall from equation 3.2, that an exponential distribution can be created from its potential function, by isolating the argument, and applying a uniform unity distribution in the place of $F(t)$, thus:

$$e = \frac{1}{\lambda} \cdot -ln(u)$$

Where $u \in U]0;1]$. As the ARM processor does not incorporate a floating point unit, floating point operations must be emulated in software, making them quite slow. Of course, this applies to both the logarithm and the division/multiplication of floating points. Note also, that the ARM processor does not support integers division, but it does support integers multiplication. As the overhead of the injector should be kept at a minimum, it is desirable to try to optimize the calculation.

As it was noted in section 3.4, the calculation of the logarithm can be tabulated, and since the ARM processor has a general problem with floating points, why not tabulate it in an integer representation? The obvious problem is lack of precision, but this can be overcome.

Multiplying the right hand side of the equation by $64K/64K$ yields:

$$e = \frac{1}{\lambda} \cdot -ln(u) \cdot \frac{64K}{64K} = \frac{16}{\lambda} \cdot -4K \cdot ln(u) \cdot \frac{1}{64K} \tag{4.1}$$

Substituting with the expressions $a = 16/\lambda$ and $h(u) = -4K \cdot ln(u)$ yields:

$$e = a \cdot h(u) \cdot \frac{1}{64K} \tag{4.2}$$

Note that u should be a continuous uniform distribution over the interval $]0;1]$, but as the PRNG generates discrete integer values, there is a need for a mapping, which can approximate the desired distribution. It has been chosen to use the following mapping:

$$m(\hat{u}) = \frac{\hat{u}+\frac{1}{2}}{2K} \tag{4.3}$$

Thus, if $\hat{u} \in U\{0,...,2K-1\}$ then $m(\hat{u}) \in U\{\frac{1}{2},...,\frac{2K-\frac{1}{2}}{2K}\} \approx u$, and $h(m(\hat{u})) \approx h(u)$. Substituting $\hat{h}(\hat{u}) = \lfloor h(m(\hat{u}))\rfloor = \lfloor -4K \cdot ln\left(\frac{\hat{u}+\frac{1}{2}}{2K}\right)\rfloor$ leads to:

$$h(u) \approx \hat{h}(\hat{u}) \tag{4.4}$$

As $\hat{u}$ has a specific discrete uniform distribution, it can be created directly. The function $\hat{h}(\hat{u})$ is easily tabulated for $\hat{u} \in \{0,...,2K-1\}$. The reason for adding $\frac{1}{2}$ in the mapping is that when tabulating $\hat{h}$, the value $\hat{h}(v+\frac{1}{4K})$ is a more precise representation of $\hat{h}$ in the range $[v; v+\frac{1}{2K}$, than $\hat{h}(v)$ is. Note, that the mapping also makes sure that $u \notin 0$, which neatly makes sure that the tabulation of $\hat{h}(\hat{u})$ does not include a calculation of $ln(0)$.

If $\hat{a} = \lfloor \frac{16}{\lambda} \rfloor \approx a$, then

$$e \approx \hat{a} \cdot \hat{h}(\hat{u}) \cdot \frac{1}{64K} \tag{4.5}$$

As each experiment uses a constant fault intensity, λ will be constant, and therefore it it only necessary to calculate $\hat{a}$ once per experiment. As $\hat{a} \cdot \hat{h}(\hat{u})$ evaluates to an integer, the multiplication by $1/64K$ can be implemented as a simple shift operation. The function $\hat{h}(\hat{u})$ is implemented as an array of `unsigned short int`'s called `dist_exponential_array`. The implementation of the exponential distribution function looks as follows:

```
1  unsigned int dist_exponential(unsigned int a)
2  {
3    unsigned int u, h;
4
5    u=dist_uniform(EXP_DIST_OPT_SAMPLES);
6    h=dist_exp_array[u];
7
8    return (a*h) >> EXP_DIST_OPT_SCALE_SHIFT;
9  }
```

where `EXP_DIST_OPT_SAMPLES` $= 2K$ and `EXP_DIST_OPT_SCALE_SHIFT` $= 64K$.

Note that the exponential distribution implementation is used in both the failsafe and the application injectors.

4.4 Harness

In order to perform experiments with the failsafe-injector, an fs-compatible hostside harness is produced. The harness comprises three threads: A main thread, which initiate the experiments, and two control-threads, which are created for each experiment. The first control-thread, called `fs_thread`, communicates with the failsafe software, and the second thread, called `log_thread`, logs the output of the injector.

4.4.1 Main thread

The main thread parses the command-line parameters, and creates the two control-threads. The harness supports the following command-line parameters:

Parameter	Description
-p	Selects name of binary image to run. Defaults to `injector_failsafe.bin`.
-m	Selects injection mode (i.e. 0=constant rate, 1= exponential stochastic). Defaults to 0.
-r	Selects injection rate. Defaults to 100 ticks per injection.
-c	Selects number of experiments to run. Defaults to 10.
-l	Selects the basename of the logfiles. Defaults to on-screen logging.
-df	Selects the device-name of the serial-port to use for fs-communication. Defaults to `/dev/ttyS0`.
-dl	Selects the device-name of the serial-port to use for injector logging. Defaults to `/dev/ttyS1`.

As can be seen, the harness support batch-operation using the **-c**-parameter. If more than one experiment is selected, the harness runs a number of identical experiments, which only differ in the logname (if any) and the seed-value (see section 4.4.2). The logfile basename allows the output from each experiment to be placed in separate files. For each experiment, two logfiles are created – one that logs the injector status information, and one that logs the communication with the failsafe software. The names of the logfiles is the logfile basename appended with the suffix `_<c>.fs` and `_<c>.inj` respectively (where `c` is the number of the experiment). The typical use of the logfile basename is to create a directory for a series of experiments. Note that while the harness does parse the **-df** and **-dl** parameters, it does not respect the names supplied (i.e. the default values are always used).

For each experiment, the main thread creates two control-threads, which control the serial communications. After the creation, the main thread awaits the termination of the control-

thread before continuing.

Each control-thread is passed a set of arguments, corresponding to the commandline parameters needed by the thread. The `fs_thread` is passed the filename of the injector and its configuration parameters (i.e. the rate and the mode). The `inj_thread` is passed a timeout-value (i.e. the maximal expected time between injections). Both threads are passed the name of the logfile to use (if any), and a flag-variable and a mutex.

The communication between the control-threads is kept at a minimum, but due to details in the implementation of the serial-port driver on Linux and the failsafe software, some communication is needed.

As the failsafe software needs to be presented a ”password” on the debug-port in order to use it for fs-communication, the debug-port on the PC must be ready for communication before the OBC is reset. In order to prevent a race-condition between opening the debug port and resetting the OBC (using the auxiliary port), a mutex is used for synchronization.

Both threads are passed a reference to a flag-variable, which is polled regularly. The variable is initially false, but if it becomes true, the thread is supposed to exit. Each control-thread is passed a reference to the other threads flag-variable, such that each thread can request the completion of the other thread. This way both threads can be orderly terminated when one detects that the experiment has ended.

4.4.2 FS Logging and Load-Generation

The thread `fs_thread` is responsible for the communication over the debug-port. This comprises answering the prompt; uploading, configuring and starting the injector and reading the status of the failsafe software.

As explained above, the debug port must be ready for communication before the OBC is reset. Therefore, the debug-port is opened, after which the mutex is unlocked. Then the prompt is awaited and the password is given. If the prompt/password procedure fails, the thread requests the other thread to exit, and exits itself.

Otherwise, the experiment continues: The status of the failsafe software is queried and saved for later reference, because this is a known-good reply to the query. Then the image of the failsafe injector is uploaded, and the initialization function is called. A seed for the PRNG is generated using the `usec` value returned by the `gettimeofday` system call. Then the seed, mode and rate parameters are set, and the injector is started.

Then the status of the failsafe software is queried continuously, until the other thread request the `fs_thread` to exit. The reply to the query is checked on several levels: If a message is successfully received, it is checked if the size of the reply is valid. Then the contents of the reply is compared to the saved known-good reply.

check protocol status			
some *error*	result: ”*error*”		
no error	**compare size to ”known-good”**		
	differ	result: ”Wrong size of reply”	
	same	**compare contents to ”known-good”**	
		differ	result: ”Reply corrupted”
		same	result: ”OK”

The harness produces text-based logfile of each attempted status inquiry. The format of the logfile is `get_status: <result>`, where `result` corresponds to one of the results mentioned above. The logfile does not convey any timing-information, but as each result takes a fixed amount of time to complete, the relative time of a result in the logfile can be derived simply by counting the number of prior results.

Generally, it takes 1 second to send a status inquiry and get the result back, even if the result is corrupted. The only deviation to this rule-of-thumb is when an inquiry times out. In this case it takes about four seconds to come to this conclusion. A simple measure of the performance under fault-injection, is the time from the beginning of the experiment to the first sign of failure (i.e. the first time a non-OK result is returned). As the time it takes to produce an OK result is assumed to be constant, the number of OK results before the first non-OK result is a directly proportional to elapsed time.

4.4.3 Injector Logging

The thread `log_thread` is responsible for communication over the auxiliary-port. This comprises resetting the OBC and logging the injector status information. As noted above, the two control-threads need to synchronize in order to prevent a race-condition. This is done as part of the OBC reset: First the OBC is held in reset state for 10 ms, then the threads synchronize, and then the OBC is released from the reset state. Then the logging of the failsafe-injector starts. The input is read line-by-line, and written verbatim to the logfile and screen. The line-read function is subject to a timeout, and the loop exists if the read times out. The loop also exits if an exception on the OBC is detected, or if the other thread requests so using the exit-flag. When the loop terminates for whatever reason, the other thread is requested to exit.

As injections happen regularly (even when using the stochastic model), it is possible to define a threshold, which can be used to determine if the injector has become unresponsive. This situation is referred to as ”Communication Lost” in their results-section.

Refining the Log-files

This section will not go into details of how the logfiles are processed, but for completeness the matter is touched briefly here. Each experiment results in two log-files, one logging the output of the failsafe software, and one logging the output of the injector. The log-files are

plain text-files formatted as one line per event, and as such they can be processed with standard text-tools such as: `grep`, `sed`, `nl`, `head` and `tail`.

For instance, finding the number of the last occurrence of a given word can be done by using `grep` to find each line with the word, using `nl` to number every such line and using `tail` to isolate the last one. Using an appropriate regular-expression substitution, `sed` can be used to isolate the number-part of the output from the `grep | nl | tail`-command.

4.4.4 Resetting the Clientside using RTS

As described in section 3.8, the RTS signal on the auxiliary port is used to reset the OBC. On the PC/Linux, the RTS-signal is normally handled automatically by the UART if hardware flowcontrol is enabled. However, by using ioctl's, it is possible to manipulate the signal directly. The RTS is connected such that, when RTS is set, the OBC is held in reset state; and when RTS is cleared, the OBC is running. On the PC, the port is opened as a device-file. The following snippet will create a pulse which resets the OBC:

```
1  int bits;
2  int fd;
3
4  bits=TIOCM_RTS;
5  ioctl(fd, TIOCMBIS, &bits);
6  delay(pulse);
7  ioctl(fd, TIOCMBIC, &bits);
```

Here, the codes `TIOCMBIS` and `TIOCMBIC` means set, respectively clear the indicated bits, and `TIOCM_RTS` means the RTS-signal. Of course, the `fd` is the filedescriptor of the device. Note that if the port is closed, the RTS will return to the set state (i.e. the OBC is held in reset state).

This procedure for resetting the target is used in both the failsafe and the application harness.

4.5 Results

A number of experiments have been carried out using the failsafe injector – in this section, the results of these experiments are presented. The experiments were all performed using exponential stochastic simulation with four different rates. In the table below, the number of experiments carried out with each rate is shown [2].

[2] Note that the result of some experiments have been discarded, due to problems in the injection logs: Sometimes the OBC seemingly begins to transmit `0xFF` bytes.

For each experiment, a number of figures are calculated based on the failsafe and injector logfiles. From the failsafe logfile, the following figures are extracted:

last OK : The number of the last successful status query. Beyond this count, the failsafe software is unable to provide the desired service.

first non-OK : The number of the first failed status query. This marks the first time a problem is encountered.

number of good replies : The number of successful status queries.

number of bad replies : The number of failed status queries.

Figure	**Rate 10**	**Rate 100**	**Rate 1000**	**Rate 6000**
# experiments	96	97	97	47
⟨**last OK**⟩	8.9	166	3060	15064
⟨**first non-OK**⟩	2.3	18	193	885
⟨**# good replies**⟩	5.7	158	3042	14997
⟨**# bad replies**⟩	16.2	53	934	4848
⟨**runtime**⟩	2072	9570	102834	509871
⟨**# injections**⟩	198	96	103	85
runtime to # inj ratio	10.5	99.7	998	5998

As can be seen, the **runtime** to **# inj** ratios are approximately equal to the injection rates, which supports that the stochastic simulation is correct (i.e. that the injection-rates are correct).

From the injector logfile, these figures are extracted:

cause of failure : The cause of the failure corresponds to either one of the detectors, or a harness-timeout (”Comm Lost”).

runtime : The number of ticks in which the injector has been active.

number of faults : The number of injections made by the injector.

exception address : The exception address. Note that the failure-causes ”Comm Lost” and ”Watchdog” does not have associated addresses.

For each rate, the frequency of each cause of failure is calculated. The extracts of the experiments are listed in appendix B.

Cause of Failure	Rate 10		Rate 100		Rate 1000		Rate 6000	
	#	%	#	%	#	%	#	%
Data Abort	29	30	36	37	35	36	23	49
Prefetch Abort	24	25	22	23	17	18	10	22
SWI	7	7	5	5	4	4	0	0
Undef. Instr.	1	1	2	2	1	1	1	2
Watchdog timeout	21	22	16	16	22	23	10	22
Comm. lost	14	16	16	16	18	19	3	6

4.5.1 Causes of Failure

In the figure below, the frequencies of failure-causes are plotted against the average time between injections (λ^{-1}). As can be seen, the predominant failure-cause is *Data Abort* with a frequency of about 30% to 50%, while the frequencies of *Undefined Instruction* and *SWI* are quite low. Also, note that the frequency of Data Aborts seem to rise with higher λ^{-1}.

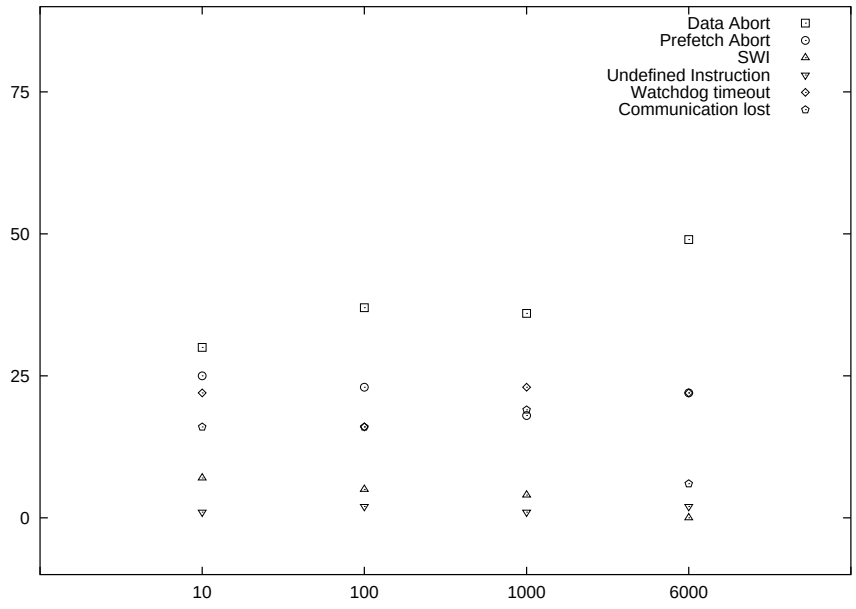

Figure 4.2: Frequency of failure-causes.

As explained in section 4.4.2, the number of the first non-OK status inquiry can be used as a measure of when the failsafe software experiences the first effects of the fault injections. Intuitively, this measure should be proportional to λ^{-1}. Each number of the first non-OK inquiries divided by the corresponding λ^{-1}, is plotted against λ^{-1}. The average values are marked diamonds and, as the figure shows, it has a value of about 0.2 regardless of the value of λ^{-1}. This is equivalent to about 20 injections before the first sign of failure.

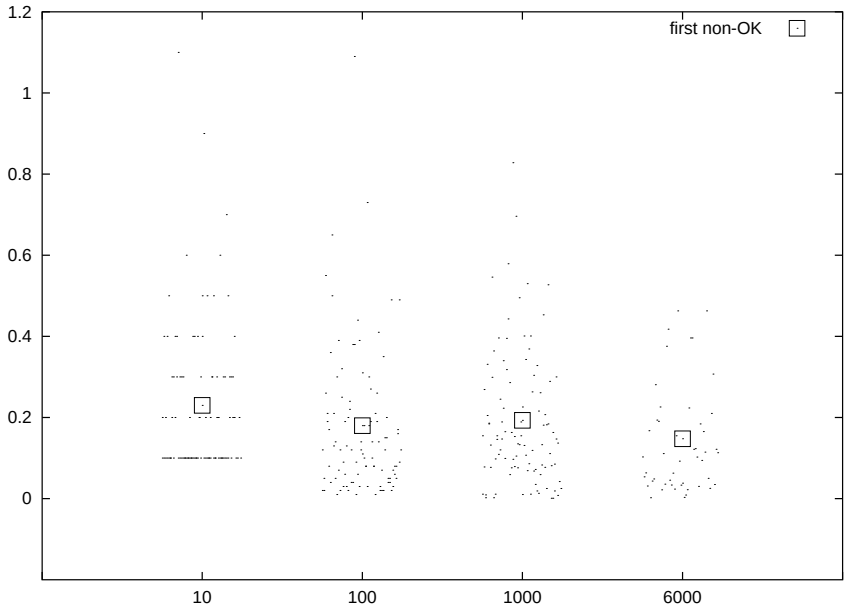

Figure 4.3: Time of first observation of problem.

As with the time to first failure, the time to destruction should intuitively be proportional to λ^{-1}. The figure below shows the total runtime of the injectors divided by λ^{-1} against λ^{-1}. Again, the average values are marked with diamonds. As can be seen, the average values for $\lambda^{-1} \in \{100, 1000, 6000\}$ are all about 100 – corresponding to about 100 injections before total failure. For $\lambda^{-1} = 10$ the average value is about 200. This indicates that for $\lambda^{-1} \geq 100$, the simulations provide results which are independent of the rate – it is merely a scalefactor. A similar pattern is seen in figure 4.3.

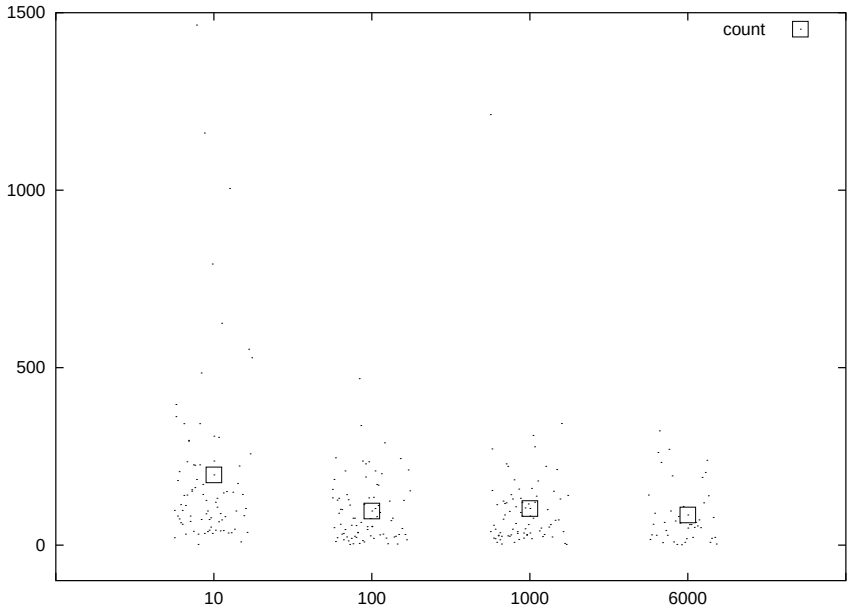

Figure 4.4: Number of injections before failure.

4.6 Discussion

While the failsafe injector has proved quite effective, there is always room for improvements. In this section, some improvements to the injector are suggested.

One of the problems with the injector is that the resolution of the timer used for injections is quite low. The reason for this is that the injector is implemented to be as little intrusive as possible. As there are no free timers available, it reuses the one timer which has a usable setup by default.

The fact that the communication is sometimes lost is not very satisfying, as all failures ideally should be detected on the clientside. And as can be seen in the results section, the frequency of experiments, which end with loss of communication is rather high for some injection rates. A reason for the loss of communication, is thought to be that the detector-part of the injector has been damaged by secondary effects of the injections, and is therefore unable

to detect (or to convey) the fact that a failure has occurred. Of course, damage to the failsafe injector can be prevented by hardening: The code of the injector can be effectively hardened by placing it in FLASH, and the global data of the injector can be hardened by unmapping its memory, when it is not in use. Mapping and unmapping the memory as needed is of course challenging, but expected to be entirely feasible.

Another weak point of the simulation is the gathering of statistics. In the current implementation, the injection-counter and the injector clock is implemented on the target, and therefore also sensible to secondary effects of the injections. A safer way to obtain the same statistic, is to simply count the number in the injector logfile. Also, the logfiles data suffer from the absence of timestamps. If the individual entries of the logfiles had been timestamped, the runtime statistics would have been more precise.

If the failure-modes of the failsafe software was to be further examined, it would be useful to add instruction decoding to the data abort exception handler. This way, the handler could provide the address of the offending access.

A last issue is whether the performed experiments provide results, which are representative of lower injection rates. The results presented in figure 4.3 and 4.4 suggests that this is the case, but recall from section 3.3.2, that the upset-rates which can be experienced in LEO are more than a factor 1000 lower than the simulated rates.

4.7 Testing

The failsafe software has an associated test-suite which tests certain key functionalities of the failsafe software. This includes creation and loading of failsafe programs, setup of the debug-port and testing of exception-handlers.

The failsafe injector is based on this existing code-base, and for the interrupt mechanisms, the major new developments are setup of interrupt-sources (WD and TC's) and programming of the AIC. However, these mechanisms are quite easy to test, as they are completely linear (i.e. there are no conditions or branches in the execution). The only state maintained by the interrupt-handling is the end-of-interrupt flag, implemented in the AIC. This flag must be cleared following the handling of the interrupt. Failing to do so will prevent future interrupts from being handled, which means that it is quite simple to test if this part works.

Also, the injector logging can be used for testing, as it provides detailed information about the injector operation (i.e. the injection address, and the contents before and after). This information is also useful for assessing the probable damage caused by the individual injections, as knowledge about the memory layout the failsafe software allows the observer to identify, for instance code-pointers (i.e. addresses in the PROM) and data-pointers (i.e. addresses in the internal RAM).

The other new major part in the failsafe injector is the implementation of the exponential distribution. The exponential distribution has been extensively tested during development, to control that the approximations does not cause unacceptable errors. The exponential distribu-

tion has been tested by calculating histograms of the synthesized distributions, and comparing these with the theoretical values obtained from the definition of the stochastic mass function. Practically the tests were performed by defining a number of buckets in the expected output range, and then do a simulation in which each output is placed in the appropriate bucket. In principle, the exponential distribution can produce infinitively large values, but for practical reasons, $N+1$ buckets are defined: The first N buckets are equally sized, and cover the range $]0; 2.5 \cdot \lambda^{-1}]$; and the last bucket covers the rest, i.e $]2.5 \cdot \lambda^{-1}; \infty[$.

The simulation of the distribution can be done in two different ways: Either one can use the full implementation of the distribution, i.e pass a $U\{0...1\}$ distribution through the mapping; or one can pass a number of equidistant numbers in the range $]0;1]$ through the mapping. The difference is that the former approach tests both the PRNG and the mapping, while the latter approach tests only the mapping. Both approaches have been used, and performing 1000000 experiments, using 20 buckets and $\lambda^{-1} = 100$ (corresponding to one event per second) gives deviations between 1 and 5 percents. Using $\lambda^{-1} = 1000$ (corresponding to one event per ten seconds) gives deviations of at most 2.5 percent, but most below 1.5 percent, and some below 1 percent.

Chapter 5

The Application

In this chapter, the ideas behind the structure and implementation of the application is explained. Opposed to the failsafe software, the application is implemented specifically for experimental use, but its basic structure is similar to that of the DTUSat flight application.

As previously mentioned, the application is implemented as a set of loosely-coupled modules, which communicate through a shared packet-router. The use of a packet-router is actually an application of the partitioning principle, described in 2.1.1. By using this principle, the software is effectively divided into modules with well-defined interfaces.

The application is running on an embedded real-time operating system called eCos. As it is customary for these kinds of operating systems, it is realized as a library which the application is linked against. The fault injector for the application is implemented as a thread – it is thus an integral part of the application. The experiments assume that the application has already been hardened, by placing all code and static data in non-vulnerable storage. The faults are therefore only injected into the assumed vulnerable storage.

5.1 Environment

The environment in which the application runs is provided by the clientside firmware and a hostside debugger. The firmware is a Gnu debugging stub (GDB-stub), using a 57600 baud RS232 connection on the debugging port. The debugger is the basic Gnu debugger used in commandline-mode.

The GDB-stub is a relative unintelligent device, which provides a series of low-level primitives to the debugger. The debugger uses these primitives to provide complex functionality, such as breakpoints, inquiry of memory contents and upload of executable files. In this way, the debugger is the brains, providing the intelligence; while the GDB-stub is the arms and legs, providing the hard work. In the following, the term ”GDB” will be used to signify the combination of the stub and the debugger (or their combined capabilities etc.)

Like a fs-compatible program, the GDB provides the means of uploading, configuring and

running the application, but opposed to using an fs-compatible program, the application does not have to be made into an image before upload, as GDB can handle ELF-files directly.

In the following experiments, the auxiliary port is only used for resetting the clientside, as the injector information is transmitted to the hostside using the GDB-connection.

5.1.1 eCos

As previously mentioned, the application is run on eCos, an embedded RTOS. eCos is a highly configurable and modular OS, which supports many popular microprocessors and devices/services. eCos provides a range of base-services, such as:

- Multi-threading and synchronization-primitives (e.g. mutex and semaphores)
- Console input/output (using a serial connection).
- C-library support.
- Floating point emulation[1].

The eCos configuration is a default configuration, with the option ”Decode Exception types in kernel” enabled. This enables the detector system to separate exception types from each other.

5.2 Implementing

The application is implemented as a set of modules, which implement tasks typically found in an onboard satellite-application. The modules communicate with each others each with virtual devices, simulating devices commonly found in a satellite. Furthermore, the application provides support for fault injections and EDAC, which are discussed in the following chapter.

5.2.1 Module-based Implementation

The application consists of a number of modules which, in this context, is an entity which has a unique address, and which is capable of communicating with other modules using the shared packet-router.

The communication between modules consist of messages, and from a packet-centered point of view, a module is an endpoint of communication. In the following, a message is a specific instance of a packet.

The services of the packet-router consist of a set of queues, and the primitives `sendMsg` and `getMsg`. The queues are known as ”inboxes”, and each inbox has an associated address

[1]Note that even though floating point emulation is available, it is not used in the injector.

(i.e. module). Using the `sendMsg`-primitive, modules can enqueue messages on inboxes, and using the `getMsg`-primitive, a module can retrieve the first message on its inbox.

In order to be able to communicate, each module must at least implement a thread, which monitors its inbox. This thread is known as the modules inbox-thread, and the modules are implemented following the rule that only the inbox-threads are allowed to retrieve messages. However, the other threads are allowed to send messages.

5.2.2 Selection of Modules

The application is designed with the following two objectives in mind: It should be complex enough to simulate a realistic scenario, but also of limited size and complexity. In this context, it is considered a realistic scenario, if the application accesses a couple of devices, and does some inter-module communication.

A module can be classified according to whether or not it accesses devices, and if the accesses comprise reading, writing or both. This gives rise to four classes of modules, and for systematism it was decided to specify one of each type. In addition to these four modules, a special watchdog-module was also specified. In the following, the main objectives of the modules are described:

Module Non Comm: The *Non Comm*-module does not access any devices, but implements a database used to store samples provided by the *In*- and *InOut*- modules. This resembles the housekeeping-module in the DTUSat application.

Module In: The *In*-module reads samples from *B*-device and sends the samples to the *Non Comm*- and *Out*- modules.

Module Out: The *Out*-module receives samples from the *In*-module, and writes the samples back to *B*-device.

Module InOut: The *InOut*-module implements a regulation-loop, which tries to prevent changes to the values sampled from the *A*-device by applying opposite directed ”forces”. The samples read, are sent to the *Non Comm*-module. The implemented algorithm resembles the $\dot{B}$-algorithm employed in the attitude-control module of the DTUSat application.

Module WD: The *WD*-module implements a watchdog algorithm, which is used to ensure that all modules are operative, by querying each on a regular basis. The *WD*-module is supported by the *WD*-device, which implements a watchdog, that must be ”touched” on regular intervals.

5.2.3 Simulated Devices

As mentioned above, the application uses a number of simulated devices. The device-names *A* and *B* are quite imprecise, but they are chosen in order to emphasize, that the devices are not meant to provide a detailed simulation of any actual device. The communications between modules and devices are implemented as simple function-calls. The devices are described in the following:

Device A: The *A*-device simulates a virtual object, which can move on a single axis. The current position of the object can be obtained using the `read`-primitive, and a force can be applied to the object using the `write`-primitive. Furthermore, the object is subject to external forces of random size and direction at random moments in time.

Device B: The *B*-device provides random samples using the `read`-primitive. After a sample has been read, the device blocks the `read`-primitive, until a value has been written to the device using the `write`-primitive.

Device WD: The *WD*-device provides only a single primitive: `touch`. This primitive is used to reset the watchdog timer, and must be called on regular intervals in order to prevent the watchdog to time out. On timeout, the watchdog causes the application to halt with an error-message. Also, the *WD*-device provides a timer, which is used to limit the runtime of the application. When this timer times out, the application halts with a special message. This timer is used in the experiments to ensure that they do end at some time.

5.2.4 Logical Connections in the Application

Figure 5.1 shows which logical connections exist in the application, and by which means they are provided. The packet-router provides logical interconnection between the modules, while some of the modules communicating with devices.

5.2.5 Packet communication

In this section, the basis for packet-based communication is described. First the general structure of PDU's (Protocol Data Units) is described, and then the actual passed PDU's are described.

PDU-format

All messages passed between modules have a common header, which is used to handle the message. The information in the header can be divided in two categories: The information used to transport the message, and some generic information used by the receiving end. The

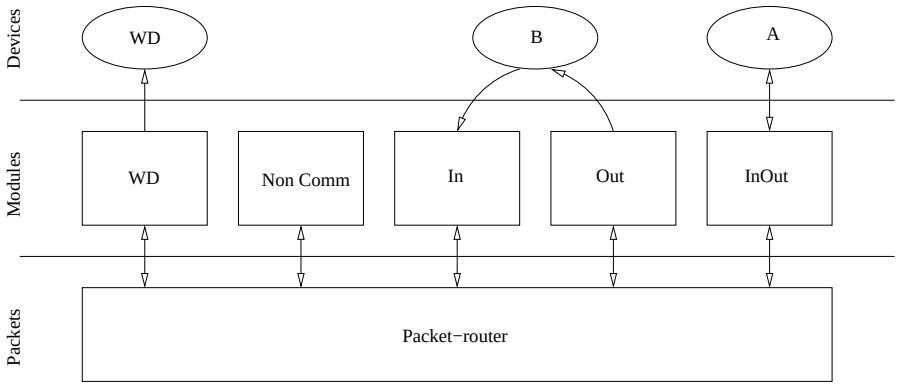

Figure 5.1: Logical connections in the application.

transport information contains the module-id of the source and destination modules and the size of the message. The generic information contains a message-id and an argument.

The message-id defines the type of the message, while the argument is an option argument to the message. The message-ids are subdivided in two fields, where the first field contains the id of the originating module, and the second field is a module specific identifier. The payload of the message follows the header, and contains message-specific information. A packet (PDU) has the following structure:

$$PDU ::= (source; destination; size; message-id; argument; \langle payload \rangle)$$
$$size ::= |\langle payload \rangle|$$
$$message-id ::= (module-id; identifier)$$

Instead of accessing the fields directly, a range of accessor-functions are implemented, which are used to interface to the PDU's. When accessing messages there are basically two use cases: Either a message is being constructed, or single fields of the message are being extracted. Therefore, the functions which extract fields from a message are implemented on a single field-basis, while only a single function exists for constructing a message (i.e. it is assumed that all information needed to construct the message is available).

Inter-Module Communication

In order for two modules to communicate, the messages passed must make sense to both ends, but is does not necessarily have to make sense to third parties. However, from a practical point of view, it is sometimes convenient to define messages which are generally understood – this suggests that messages are partitioned in two major categories: Private messages and public messages. Public messages are implemented by using a special value for the module-id in the message-id field. The public messages currently implemented, are related to the watchdog algorithm and acquisition of housekeeping data.

module-id	identifier	description
In	0x01	PDU_MSG_INPUT_DATA: Used to send data-samples from module *In* to modules *Out.*
none	0x02	PDU_MSG_GEN_ALIVE_REQ: Alive request-message send from the *WD*-module to each module, in order to determine if they are alive. Each alive-request contains an argument, which is expected to be returned in a PDU_MSG_GEN_ALIVE_RES_ACK-message.
none	0x03	PDU_MSG_GEN_ALIVE_RES_ACK: Alive response-message send in response to a PDU_MSG_GEN_ALIVE_REQ. The response carries the same argument as the originating request.
none	0x10	PDU_MSG_GEN_HK: A generic housekeeping message. Send by producers of housekeeping data (i.e. the *In* and *InOut* -modules) to the *non comm*-module.

Figure 5.2 shows the messages passed between modules and the dataflow between modules and devices. Devices are symbolized as circles, while modules are symbolized as boxes. The lines and arrows show the directions of communication. Even though the watchdog-related communication consists of two messages, one in each direction, it is symbolized as a single line with two arrows, to minimize the clutter.

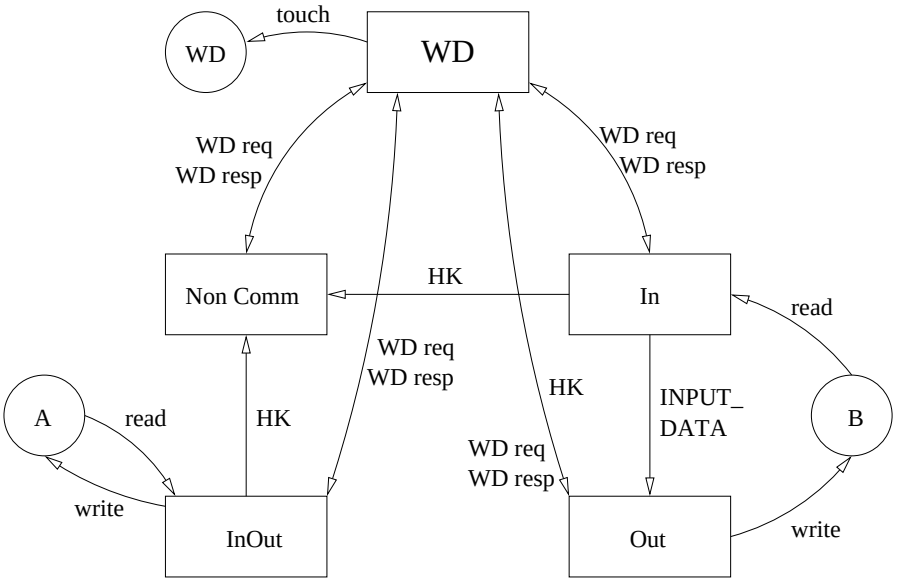

Figure 5.2: Message- and data- flow in the application.

5.2.6 Details

In this section, the implementation of the modules and devices will be discussed in more details. Particularly the allocation of threads, and some of the implementational details, will be pointed out.

Watchdog Algorithm

The watchdog algorithm consists of a supervisor part and a number of subordinate parts. The supervisor part is implemented in the watchdog module, while the subordinate parts are implemented in each module being under watchdog supervision. The watchdog algorithm revolves around the notion of a round. A round corresponds to a single watchdog-cycle, in which all supervised modules are queried, and the watchdog device is reset (provided that all modules answered the query correctly).

The supervisor part of the algorithm is implemented as a separate thread in the watchdog module. Each round is initiated by the supervisor, which sends out alive-requests to each supervised module. Each alive-request contains a nonce, which the modules must return in an alive-response message. The alive-response messages are received by the watchdog inbox-thread, which stores the values in the global array `ack` (the value of a nonce is stored with the modules id as the index). The modules are expected to answer the alive-request messages within a specific time-frame, and when this time has elapsed, the global `ack` array is inspected. If all the values in the array contains the correct nonce, it is concluded that all modules have successfully answered the alive-request, and the watchdog device is therefore reset. As the global array is shared between multiple threads, it is of course protected by a mutex.

The subordinate parts of the watchdog algorithm is implemented in each supervised module. The objectives of the subordinates is to determine if all its threads are alive and if so, to answer the alive-requests. Some modules (*Non Comm* and *Out*) are implemented as single threads, which makes the implementation particularly simple: When an alive-request is received, just answer it right away.

The multithreaded modules *In* and *InOut* are implemented with two threads: An inbox-thread and a worker thread. The worker-threads are implemented as infinite loops with a work-sleep structure. As the periods of the worker-threads are several times higher than the period of the watchdog algorithm, the liveness of the worker-threads can be determined, by having the workers incrementing a shared counter on each iteration. On reception of an alive-request, the value of the counter is read and reset. If the value read is greater than 0, the worker is taken to be alive, and an alive-response is returned.

As the watchdog-module itself is under watchdog supervision, it too contains a subordinate part, but it does not check the liveness of its worker-thread, as it is assumed to be alive. This can be assumed because the alive-request originates from the worker-thread, and therefore it can be concluded that it must have been alive recently. Also, as it is the worker-thread which resets watchdog-device, it must be alive to do so..

Housekeeping

As previously mentioned, the *Non Comm* module receives housekeeping-data from the two input-modules, and stores this in a database. The housekeeping-data messages are sent with message-id `PDU_MSG_GEN_HK`, and single integer-value as payload. The housekeeping-data differs from much of the other global data, in the application in that the data might be left in the database for extended period of time, without being accesses. This makes the data vulnerable to bitflips, but unlike most of the other global data, bitflips in the housekeeping data does not affect the application per se.

In and Out Modules

The *In*- and *Out*- modules communicate privately, and share the handling of the *A*-device. The worker-thread of the *In* module reads the *B*-device, and sends the sampled data to the *Out*-module as `PDU_MSG_INPUT_DATA`-messages. The sampled data is also sent to the *Non Comm* -module as housekeeping data, using the `PDU_MSG_GEN_HK`-messages. The *Out*-module simply outputs the received samples to the *B*-device.

InOut Module

As mentioned above, the *InOut*-module implements a regulation loop, which tries to prevent changes to the values sampled from device *A*. This is done by calculating the difference between successive readings, and applying a negative proportional force on the virtual object. The force is clamped to limit its absolute value.

5.2.7 Injector

As mentioned in the introduction to this chapter, the experiments assume that the application has already been hardened by placing static objects (code and data) in non-volatile storage (i.e. FLASH). The injection targets are therefore only the parts of the application, which are stored in volatile storage (i.e. RAM). The volatile data is divided into variables allocated on the program stacks and global variables allocated in the data section. The variables allocated in the data section are further divided into ”os-data” and ”regular globals”.

The os-data are global data, which is used within the operating system. For instance, mutexes and thread-objects are required to be allocated by the application, on behalf of the operating system. Because these objects are used directly by the scheduler, they are categorized as os-data. The regular globals (or just ”globals”) are used to represent the global state of the program. This gives cause to the three subtargets: stacks, globals and os-data.

Injector Targets

The structure of the application means that it is not readily possible to define a single address-range in which to inject faults. This is caused by the application having several subtargets for injection, and by subtargets not being continuous.

The solution is to make the application cooperate in the task: Each module, subject to fault injection, exports a function which is used to declare the ranges of is subtargets. The function is called with a bitpattern containing a set of subtargets, and for each selected subtarget, the regions of the corresponding variables are added to the cumulative injection target.

In order to test the application, the injector supports a special dummy injection target. This target is an area unused by the application, and therefore injections into this target only has secondary effects (i.e. the timing effects caused by the active injector).

Accuracy of eCos Timers

The injector and other periodic tasks are implemented as infinite work-sleep -loops, which are based on the function `cyg_thread_delay` to implement the sleep-periods. As eCos uses a 100 Hz timer to time drive its scheduler, it is expected that this is also the granularity of time-based events (i.e. 1 tick). A small program written during development of the injector to examine the accuracy of the `cyg_thread_delay`-function. The program implements two threads: One with a high priority and one with a low priotity. The low priority thread increments a global variable continuously, while the high-priority thread implements a work-sleep-loop.

The sleep-periods in the high-priority thread are varied, and just before and after each sleep-period, the value of the global counter is read. The difference of the counts are used as a measure of the length of the period, and the measurements show, that for sleeps longer than 5 ticks, the jitter is about 1 tick.

Implementation

On injector startup, the size of the accumulated target is calculated, and used to calculate the proper injection rate. The injector is implemented as a high-priority thread with a general sleep-work pattern. The calculated injection rate is used to generate exponentially distributed periods of sleep between injections and, as in the failsafe injector, the faults are uniformly distributed over the size of the accumulated target.

Unlike the failsafe injector, the application-injector injects faults in 12 bits per address (per byte), because it simulates the use of error-correcting storage (see chapter 6). The 4 upper bits are injected on the address $+512\ K$. Each injection is printed on the console with the address, and memory contents before and after. The injector counts the number injections in the global variable `injector_count`.

5.2.8 Detectors

Like the failsafe injector, the application contains a number of detectors. The operating system provides means of handling processor exceptions through the function `cyg_exception_set_handler`. Using this function, handlers for the ”data-abort” and ”prefetch-abort” exceptions are installed. There is no need to install a handler for the ”undefined instruction”- exception as the application runs under the supervision of the GDB-stub, which provides detection automatically.

During the development of the detectors, the capabilities of the GDB-stub was investigated, in order to determine to which extend it could be used for detection. However, it was found that it would only provide detection for undefined instructions, as data-aborts was silently ignored, and prefetch-aborts could cause a loss of the debug connection.

As mentioned in section 5.1.1, the operating system is configured with the option ”Decode Exception types in kernel” enabled, which is disabled per default. This option is necessary if different handlers for different exception are needed (as it is in this case). The default setting will only allow a single shared handler for all exceptions.

Application-level Detectors

In addition to the hardware-based exceptions described above, the application implements a set of detectors on the application-level, which monitor the state of the application.

As message-passing is an integral part of the application, this is an obvious area in which to implement detectors. On the module-level, each module implements detectors which check the type of incoming messages, to make sure they are known.

The packet-router implements detectors which check the incoming messages for valid source, destination and valid size, and that messages can be successfully placed on the proper queue (i.e. that it is not full). The packetrouter checks the the id of modules when they read messages off their queue; no checks are done on the actual messages at this point. The scrubber of the EDAC-system (see section 6.1.5), checks the validity of handles passed by the application.

The *WD*-device implements two timer: A ’normal’ watchdog timer and a time limit timer. While expiry of the time limit is not a failure per se, expiry of both timers are treated as failure detections.

Halting an Experiment

As in the failsafe-experiment, the activation of a detector is supposed to end the experiment. As the application is running under supervision of a debugger, the termination of the application can be implemented using a breakpoint, which also has the effect of returning control to the debugger, thus allowing it the opportunity to perform any post-experiment activities.

The eCos environment supports ”compiled-in” break-points, which are break-points inserted into the ’C’-code. Before terminating the application, a detector should issue a message

stating why the experiment has ended, i.e. which detectors have been activated, and why (if known). This functionality is implemented in the function `exit_break_point`, which is used by all detectors.

5.2.9 Configuration Interface

As described above, there are various aspects of the application, which can be changed to provide different scenarios for experimentation. These parameters are implemented as global integer variables, which are available to the debugger. The following parameters are defined:

Parameter injector_mode : As the name implies, the `injector_mode` controls the mode of the injector. The `injector_mode` encodes the following subparameters as bitfields:

Injector parameters	**Values**
mode	*enable* *disable*
model	*constant rate* *exponential stochastic*
target	*stacks* *globals* *os-data* *dummy*

Parameter injector_rate : The interpretation of this parameter depends on the chosen injections model. For exponential stochastic simulation, this value specifies the average time between injections per K of injection target. For constant rate simulation, this parameter specifies the time between injections. Both values are measured in $\frac{1}{100}$ seconds.

Parameter seed : This value is used to seed the PRNG. Its use is similar to the use of the seed primitive in the failsafe injector.

Parameter edac_mode : This parameter specifies the mode of operation of the EDAC (Error Correction And Detection) system (see chapter 6). The following modes are supported:

EDAC mode	**Values**
function mode	*enable* *disable*
process mode	*enable* *disable*

Parameter wd_maxage : This parameter specifies the time limit of the application. The value is measured in $\frac{1}{100}$ seconds.

5.3 Harness

The application is uploaded to the target using GNU debugger over a serial connection. The upload and execution of the application can be done interactively by manually issuing the commands to connect to the target, upload the executable file, and execute the program, but in order to carry out a large number of experiments, this procedure has to be automated. This is accomplished by a harness, which controls the debugger by sending it the proper commands and logging its output.

5.3.1 Commandline Parameters

The harness is controlled by a number of commandline parameters. The functionality of the harness is similar to that of the failsafe harness, but it is slightly simpler.

The harness supports the following command-line parameters:

Parameter	Description
-c	Specifies the number of experiments to perform. Defaults to 10.
-l	Specifies the basename of the logfile. Defaults to no logfile.
-s	Specifies the name of the scriptfile. This parameter is mandatory.
–	(dash-dash) Specifies the end of the parameters to the harness, and the start of the parameters to the scriptfile. This parameter is mandatory.

The options for controlling the number of experiments **-c**) and the basename of the logfile **-l**), are similar to their counterparts in the failsafe harness. The use of the scriptfile parameter is different from the failsafe harness, and is explained in the next section.

5.3.2 Commanding GDB

The debugger in question is a variant of the basic GDB, known as ”Redhat Insight”. The difference between Insight and GDB is that Insight is extended with a Tcl/Tk GUI. However, using the commandline option `-nw` causes the debugger to run in commandline mode (i.e. reverting to ”normal” GDB mode).

During the construction of the harness, the capabilities and functionality of the debugger was examined. While the debugger has support for execution of scripts, using a commandline

option, this approach is not usable as it suppresses the console output from the debugged program. Instead it was attempted to use pipes to pipe in the commands, and pipe out the output of the debugger. This approach has proved usable and is the method used for controlling the debugger.

The commands for the debugger is basically a number of lines of text, with a single command on each line. There are two major ways in which the commands can be represented: Either they can be hardcoded into the harness, or they can be read from a file.

The advantage of storing the commands in a file, is that they can be changed without changing the harness, which is why this solution was selected. However, some commands need to be modified or parameterized by the harness, which means that the contents of the command-file cannot be used as input directly. Instead, the command-file is used as a template, from which the final commands are extrapolated.

5.3.3 Controlling GDB

The extrapolation of the template is performed by our friend `vsprintf`, a member of the hard-working `printf`-family. The actual extrapolation is done by reading the contents of the scriptfile into memory, and calling `vsprintf` with the contents as the format-string, and a pointer to the part of the `argv`-array following the `'--'`-parameter. [2] Before doing the actual extrapolation, a basic substitution is performed: The `argv` array is searched for the string `<seed>`, and if found, it is substituted with the current microsecond time.

When the commands have been extrapolated, the clientside is reset using the modified RTS-signal (as in the failsafe harness). Finally, the harness starts the debugger as a separate process, which is controlled through pipes attached to stdin, stdout and stderr. The child-process (GDB) is started in the standard (UNIX) fork-exec manner.

5.3.4 The Scriptfile

The scriptfile, which act as a template for the GDB-commands, is called `run.gdb`, and is located in directory of the harness executive. The commands in the scriptfile can be divided into commands for setup, commands for program loading and execution, and commands for cleanup.

The setup-commands configure which communications settings (device and speed) to use, and which program-file to read symbols from. The commands for program loading and execution, load the program-file into the target and configure the program for the current experiment. When the program has been loaded and configured, it is started, after which the application gains control over the target. The debugger awaits to regain control over the target, which happens when the application stops with an exception or a breakpoint. Then the

[2] This use of `vsprintf` is a slight hack, as the argument-pointer really should be a `va_list`, not a `char **`, as the `argv` is. However, as long as only the `%s` format is used, this usage is safe (at least on the x86 platform).

cleanup-commands are executed – the injection-count is read from the target, and the debugger quits.

Note that the program file (i.e. the file containing the application) is loaded in two steps: First it is loaded into the debugger (using the `file`-command), and next it is loaded into the target (using the `load`-command).

5.4 Discussion

This section describes some problems identified during the development of the application.

An important question is whether realistic simulations can be performed with a timer granularity of 100 Hz. The problem is that all time-based activities are based on the same 100 Hz timer, which might cause aliasing, i.e. that injections can only occur at certain points in the execution of the program. A solution to this is to use a separate high-speed timer for the injections. This should be feasible, as the eCos AT91 HAL (hardware abstraction layer) does not make use of all three timer/counters, and eCos does provide support for user-handled interrupts.

Some of the issues mentioned in the discussion of the failsafe injector apply to the application injector as well: Timestamping of events would be of benefit, as it would provide more accurate measurements of runtime. Unlike the experiments on the failsafe software, the communication in the application experiments has run quite stable, with only a couple of instances of lost communication. This can be explained by the GDB implementing a protocol, providing error-detection.

Chapter 6

Correctors

While chapters 4 and 5 have described how to simulate the injection of bitflips, this chapter describes a scheme for reversing the effects of bitflips, based on the commonly used Hamming (12,8) code. Following the description of the Hamming code, the results of the experiments performed on the application are presented, and a model for fault injections is proposed. Last, the implementation is discussed, and the testing of the application is described.

6.1 Implementing

In this section the implementation and usage of the EDAC system is discussed. First the implementation of the Hamming code is described, then the practical implementation of EDAC-primitives is explained, and lastly the application of these primitives is discussed.

6.1.1 Hamming Code

As described in sections 3.5 and 3.6, the Hamming codes can be implemented using the $\underline{\underline{A}}$-matrix only. Recall from equation 3.8 and 3.9, that the calculation of $\underline{d} \otimes \underline{\underline{A}}$ is used in both encoding and decoding. Defining the function $p(\underline{d}) = \underline{d} \otimes \underline{\underline{A}}$, note that $p : \{2^8\} \rightarrow \{2^4\}$. Because of the low number of combinations in the input and output domains, the function is an obvious candidate for tabulation. However, the function must of course be implemented before it can be tabulated.

This suggests that the algorithm is implemented in two versions: A version based on its arithmetic definition, and an optimized version, based on a table created by the arithmetic version.

Because of details in the use of the code, the implementations handle the data and parity parts of the codewords separately.

Both implementations implement two primitives: An encoder, which calculates the parity; and a decoder, which calculates the syndrome. Recall from section 3.6, that the value of

the syndrome is equal to the row in $\underline{\underline{H}}^T$, whose corresponding bit is in error. Thus, it is trivial to create a table, which translates syndromes into bitmasks which point out the bits in error. However, because of the separation of the data and parity information, two tables are generated: One which corrects errors in the data part, and one which corrects errors in the parity part.

The Arithmetic Implementation

If $\underline{\underline{A}}_i$ denotes the i'th column of $\underline{\underline{A}}$, then the i'th bit of output of p is:

$$\underline{r}_i = \bigoplus_{n=1}^{8} \left(\underline{d}_n \otimes \underline{\underline{A}}_{i,n} \right)$$

but if using bitwise exclusive-or on an 8-bit quantity, this can be translated into:

$$\begin{aligned} \underline{t}_i &= \underline{d} \otimes \underline{\underline{A}}_i \\ \underline{r}_i &= \bigoplus_{n=1}^{8} \underline{t}_{i,n} \end{aligned}$$

This way, the calculation of the parity is simply implemented as:

```
hamming_parity(data)
{
   for i = 1...4
   {
      temp = data ⊗ A_i;
      res_i = ⊕_{n=1}^{8} temp_n;
   }
   return res;
}
```

The syndrome is calculated by recalculating the parity, and xor'ing it with the received parity, thus:

```
hamming_syndrome(data, par)
{
   temp = hamming_parity(data);
   return temp ⊕ par;
}
```

The Optimized Implementation

The optimized implementation is based on a table created using the arithmetic version. The table consists of the parity of the 256 possible values of *data*. The table is populated simply as:

$$\begin{array}{l} for\ i = 0...255 \\ \quad hamming\_lut[i] = hamming\_parity(i); \end{array}$$

and the parity and syndrome operations simply become:

$$\begin{array}{l} hamming\_parity_{opt}\ (\underline{data}) \\ \{ \\ \quad return\ hamming\_lut[\underline{data}]; \\ \} \end{array}$$

$$\begin{array}{l} hamming\_syndrome_{opt}\ (\underline{data}, \underline{par}) \\ \{ \\ \quad \underline{temp} = hamming\_lut[\underline{data}]; \\ \quad return\ \underline{temp} \oplus \underline{par}; \\ \} \end{array}$$

6.1.2 Applying EDAC

The Hamming code is used to implement an EDAC system, which protects the vulnerable storage of the application. The implementation of the code separates the data from the parity bits, and the reason for this is a desire to make the EDAC system as transparent as possible. If the parity-bits were not separated from the data-bits, the implementation would be limited to work on byte entities only, as consecutive data-bytes would be interspaced with the corresponding parity bits. Instead, the separation allows sequences of data-bytes, whose parity bits are stored elsewhere.

Now that the data and parity parts are separated, an obvious question is where the parity-bits should be stored. One approach would be to explicitly allocate a buffer for the parity-bits, for each block of data bytes to be protected. But a weak point in this approach is that it requires the user to allocate the parity-buffers. Instead it has been chosen to allocate the upper half of the ram for parity-bits. This way, every byte in the lower half of the memory has a corresponding byte in the upper half, which contains its parity-bits. The advantage of this approach is that the user does not have to allocate space for the the parity bits, and that the address of the parity-bits is trivial to calculate from the address of the data-bits. The major disadvantage is, that not only is half the memory of the parity-space wasted because only 4 bits per byte are actually used, but this scheme also allocates parity-space for the parts of the program, which is assumed to be stored in non-volatile storage (i.e. code and static data).

6.1.3 EDAC Helper

The implementation of the Hamming code used in the application consists of two primitives: `edac_encode` and `edac_decode`. The arguments for both primitives is a buffer, in the form

of a pointer and a count of bytes. The encode-primitive calculates and updates the parity-space associated to the buffer, while the decode-primitive calculates the syndromes and the buffer and its associated parity-space, and corrects any errors found. Note that both primitives are assumed to always succeed, as the encoding primitive cannot fail, and decoding primitive cannot detect errors it cannot correct. Therefore they do not return any status-information. The primitives are implemented as functions containing the appropriate loops, inlined with the algorithms described above.

As the `edac_encode` and `edac_decode` are called quite often, the program tends to spend quite some time executing these functions. This raises an interesting question:

When `edac_encode` and `edac_decode` are busy protecting the memory used by other functions, who protects the memory used by `edac_encode` and `edac_decode` (i.e. stack)? The only practical solution is to have `edac_encode` and `edac_decode` protecting their own memory. As `edac_encode` and `edac_decode` are assumed to execute for short periods of time, the scheme for protecting their own memory is slightly simplified, in that it will only correct errors in the data part, and leaves errors in the parity part uncorrected.

The implementation of the Hamming code is a highly optimized version, written in assembler. The reason for writing it in assembler is, that an analysis of the code generated by the C-compiler, revealed that its use of registers, and the translation of the loops was not optimal.

The EDAC system supports two different types of protection of storage: Manual and automatic. Manual protection requires the user to select at which points in the execution of the program a given variable is encoded and decoded, while automatic protection leaves this job for a scrubber-process. The manual methods described in the following should be quite easy to automate in a compiler, thus relieving this burden from the user.

6.1.4 Manual Protection

In principle, the use of manual protection can be formulated in two simple rules: After an assignment to a variable, its contents should be protected, and before its use, its contents should be validated. However, for practical reasons, the rule for validation is slightly relaxed, as variables are assumed to stay valid for short periods of time. In general, it is assumed that a variable can be invalid if control has passed to an other function, between the protection and the use of the variable. Variables are considered valid across simple arithmetic operations, that is: If a variable is assigned to, and some arithmetic operations are performed before its use, it is still considered valid. Also, variables are considered valid across validation of other variables – otherwise it would be impossible to perform operations which require more than one argument on validated variables.

The distinction between *protect* and *encode*, and *validate* and *decode*, is used to indicate that encoding and decoding signifies low-level operations, while protecting and validating signifies higher-level operations. The interface for the higher-level operations is implemented as three flavors of macros:

Macro-names	Usage
`EDAC_PROTECT_STACK_opt2(regc,argc)` `EDAC_VALIDATE_STACK_opt2(regc,argc)`	Protects implicit contents on the stack
`EDAC_PROTECT(var)` `EDAC_VALIDATE(var)`	Protects named variable, compiler calculates size.
`EDAC_PROTECT2(var,size)` `EDAC_VALIDATE2(var,size)`	Protects named variable, user supplies size.

As can be seen, each flavor consists of two macros: one which protects and one which validates. The stack flavor is used to protect implicit variables on the stack, and takes two arguments: The number of stored registers on the stack, and the number of function-arguments on the stack. The two other flavors are used to protect explicit variables. The macros obey the global variable `edac_mode`, which is used to enable and disable the EDAC-system.

Protecting the Stack

The general discipline for protection of the stack is that, upon entry to a function, the stack should be protected, and upon exit (or other use of the protected values), the values should be validated.

On ARM systems, the first four words worth[1] of function arguments are passed in the first four registers (`a1` through `a4`), while subsequent arguments are passed on the stack ([17] defines the ARM-specific details of intrinsic types of common programming languages). This means that for functions which take many arguments, it is necessary to protect the arguments passed on the stack. Besides its use for argument passing, the stack is also used for saving the contents of registers during execution of functions. Figure 6.1 shows the layout of the stack after a hypothetical function has executed its prologue. The function takes six words worth of arguments, of which two words are passed on the stack; and saves four registers on the stack.

The rounded boxes show where the frame-pointer (the `fp`-register) and the stack-pointer (the `sp`-register) point to, after the prologue. Note the use of the frame-pointers: The saved frame-pointers form a linked list, from which the stack-contents of the previous functions in the call-chain be obtained. The stack-contents saved on behalf of a currently executing function is pointed out by the frame-pointer (the `fp`-register). The parts of the stacks which need protection, are marked by the `argc` and `regc` values, which are passed the macros.

[1] Note that some multi-word types are passable in registers. This applies to `double` and small `struct` -arguments.

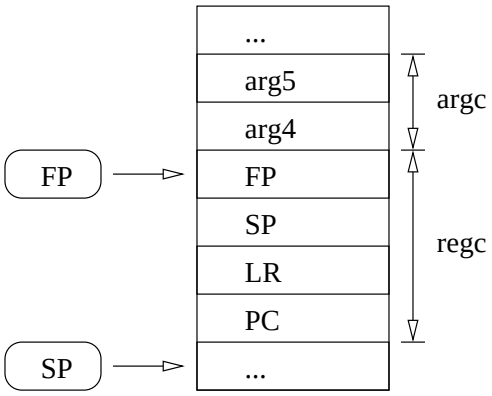

Figure 6.1: A typical layout of the stack for a function taking many arguments.

In order to efficiently access the contents of `fp`, the macros are implemented as inline-assembler, and the following structure of the macros is used:

```
1   asm("sub %%0,%%fp,%0  "
2       "mov %%r1,%1      "
3       "bl edac_function "
4           :: "I"(4*(regc-1)), "I"(4*(regc+argc))
5           : "r0","r1","r2","r3","lr");
```

The calculation `4*(regc-1)`, is the amount which should be subtracted from `fp`, to get the address of the first saved register (i.e. the address of `PC` in the above example). The calculation `4*(regc+argc)` is the number of bytes which should be handled by the function (`edac_function` being replaced by `edac_encode` and `edac_decode`). The argument-class `"I"` specifies that the argument must be an integer, which can be used as an immediate value, which really ensures that the `sub` and `mov` -instructions are valid.

Note that the use of manual protection of the stack, requires that the user supplies the `argc` and `regc` -values. While the `argc`-value can be deducted from the 'C' source-code, the `regc`-value is determined by the code generated by the compiler.

The most practical way to obtain this value is to insert the macros at the proper locations in the code, but with dummy values for the `argc`-value. Then compile the code and examine the generated assembler to obtain the correct value; insert the found values, and recompile. After the recompile, it should be checked that values used for `regc` are still valid.

Even though this is the most practical way to use stack-protection, it really is not very convenient, but as mentioned above, these methods are better suited for implementation directly in the compiler.

Protecting Explicit Variables

Recall, that the arguments for the EDAC-functions are a pointer and a byte-count. The macros for protecting explicit variables come in two variants: One which is just a wrapper around the two EDAC-functions, and one in which the compiler automatically calculates the size of the protected variable. The automatic variant is most commonly used, but in some circumstances, it cannot determine the size of the variable, and then it is necessary that the user provides the size.

6.1.5 Automatic Protection

The automatic protection is build around a scrubber-process, which validates the protected variables on a regular basis. The scrubber-interface has three primitives:

Function	Usage
`edac_scrub(ptr,size)`	Registers a memory-region for scrubbing. Returns a handle to the region.
`edac_lock(handle,ptr,size)`	Locks the region identified by handle, and validates the subregion identified by `ptr` and `size`.
`edac_unlock(handle,ptr,size)`	Protects the subregion identified by `ptr` and`size`, and unlocks the region identified by handle.

The modules which use the scrubber must register their regions during module-initialization, and for each registered region, a handle is returned. Because the scrubber-process accesses the registered regions, accesses from the other processes must be serialized using the lock and unlock primitives. As the names apply, the lock and unlock primitives marshal the control to the regions, but they also perform validation and protection of the subregion. This is based on the observation, that when the user locks a region, it is with the intension of accessing at least a part of it. Therefore, the lock and unlock primitives also provide the handling of a subregion. This way, the user only has to make one call to gain, and one call to release access to a subregion.

The scrubber process is implemented as a loop, which iterate over the registered regions. For each region, the region is locked, validated and unlocked.

6.1.6 Examples of Use

In this section, some examples of the use of the implemented primitives is given. A commonly used construct in the application, is the following loop, which retrieves a message from the packet-router, and dispatches control to a function which carry out some action. The following piece of code is taken from the *Non Comm*-module:

```
1   static void inbox(cyg_addrword_t pow_data)
2   {
3     pdu_t packet;
4     for(;;)
5     {
6       EDAC_PROTECT_STACK_opt2(5,0);
7
8       getMsg(MODULE_THIS, &packet);
9       switch(pdu_get_message(&packet))
10      {
11        ...
12        case PDU_MSG_GEN_ALIVE_REQ:
13          handle_alive_message(pdu_get_argument(&packet));
14          break;
15
16        case PDU_MSG_GEN_HK:
17          handle_hk_data(pdu_get_source(&packet),
18                         pdu_get_payload(&packet));
19          break;
20        ...
21      }
22    EDAC_VALIDATE_STACK_opt2(5,0);
23    }
24  }
```

Note the use of the stack-protection: As the function never returns, the stack is protected and validated in the start and end of the loop. As the function only take a single argument, there are no arguments on the stack to protect, why the `argc`-value is zero. Note that as the function does not return, it is actually not necessary to protect the saved registers in this function, so it is only done for completeness here.

The function `pdu_set_all` is special in the way that it takes ”many” (7) arguments, some of which (3) are passed on the stack.

```
1   void pdu_set_all(pdu_t *pdu,
2            unsigned char src,
3            unsigned char dest,
4            unsigned short int message,
5            unsigned short int argument,
6            unsigned short int payloadsize,
7            const void *pp)
8   {
9     const unsigned char *payload=pp;
10
```

```
11      EDAC_PROTECT_STACK_opt2(4,3);
12      EDAC_PROTECT(payload);
13
14      pdu->header.source=src;
15      pdu->header.destination=dest;
16      ...
17      EDAC_VALIDATE_STACK_opt2(4,3);
18    }
```

The four saved registers and the three stack-passed arguments are protected. Note that the arguments are actually considered valid at this point, as the argument passing can be seen as a type of assignment, in which temporary variables are assigned the values being passed.

The last example demonstrates the use of the scrubber in the *Non Comm*-module. The *Non Comm*-module allocates a global array, which contains housekeeping information received from the other modules. This array is accessed infrequently, and therefore is a candidate for automatic protection, using the scrubber. During module initialization, the array is registered with the scrubber. The handle for the region is stored in the global variable `hk_store_handle`. As the handle itself is an injection target, it too must be protected.

```
1   void noncomm_init(void)
2   {
3     ...
4     hk_store_handle=edac_scrub(hk_store, sizeof(hk_store));
5     EDAC_PROTECT(hk_store_handle);
6     ...
7   }
```

The global array is used to implement a circular fifo, in which the housekeeping-data is stored. The function `handle_hk_data` does the actual storage:

```
1   static void handle_hk_data(unsigned char source, void *buffer)
2   {
3     pdu_contents_housekeeping_t *payload;
4
5     EDAC_PROTECT_STACK_opt2(4,0);
6
7     payload=(pdu_contents_housekeeping_t*)buffer;
8     EDAC_PROTECT(payload);
9     EDAC_PROTECT(*payload);
10
11    EDAC_VALIDATE(hk_store_next);
12    EDAC_VALIDATE(hk_store_handle);
13    edac_lock(hk_store_handle, &hk_store[hk_store_next],
```

```
14                                sizeof(hk_store[hk_store_next]));
15     {
16       EDAC_VALIDATE(hk_store_next);
17       EDAC_VALIDATE(source);
18       EDAC_VALIDATE(payload);
19       EDAC_VALIDATE(*payload);
20       hk_store[hk_store_next].source=source;
21       hk_store[hk_store_next].datum=payload->sample;
22     }
23     EDAC_VALIDATE(hk_store_handle);
24     edac_unlock(hk_store_handle, &hk_store[hk_store_next],
25                                  sizeof(hk_store[hk_store_next]));
26
27     EDAC_VALIDATE(hk_store_next);
28     hk_store_next=(hk_store_next+1) % HK_STORE_SIZE;
29     EDAC_PROTECT(hk_store_next);
30
31     EDAC_VALIDATE_STACK_opt2(4,0);
32   }
```

Note that the subregion only contains the record being accessed. Also, note the revalidation of `hk_store_next` in line 15 and 25, and that the handle is being revalidated in line 22. The reason for this, is that variables are considered invalid across a `edac_lock` operation, as it includes a semaphore-locking, which is potentially lengthy.

6.2 Results

A large number of experiments have been carried out on the application. One of the main objectives of the experiments is to examine the effects of the EDAC-system. Therefore, a number of series of experiments are performed, which are used to compare the performance of the application with and without the use of the EDAC-system.

Recall from section 5.2.7, that the application defines four subtargets for injection, namely: Stacks, globals, os-data and a dummy target. Experiments were carried out on each target, and the combined target of stacks and globals. Except for the dummy-target, experiments were performed with rates of 1000 (corresponding to $0.1\frac{\frac{injection}{s}}{K}$) and 5000 (corresponding to $0.02\frac{\frac{injection}{s}}{K}$). Experiments on the globals-, stacks-, and combined- targets were performed both with and without use of the EDAC-system. Note from section 5.2.3, that the watchdog-device implements a time limit which, per default, limits the runtime of an experiment to 60 minutes. However, it was found that this limit was too low for the experiments on the globals target, with rate 5000 and EDAC turned on. Therefore, an extra series of experiments was performed, but with the time-limit raised to 200 minutes. In the following this variation is

called HTL (meaning ”High Time Limit”). For each series, the number of experiments, the average number of injections before failure and the average number of successful watchdog resets are calculated. The number of successful watchdog resets is an indirect measure of runtime, because the watchdog is reset every 2 seconds. Except for the ’globals’ targets with EDAC, the number of successful watchdog resets is proportional to the rate, for each pair of target and EDAC-mode. For instance: The number of successful WD resets for ’stack, rate 5000, edac’ is about five times higher than for ’stack, rate 1000, edac’, etc. A total of about 1250 experiments were performed (some of which have been discarded due to various problems), which is summarized in the following table:

Target Figure	Rate 1000 no EDAC	Rate 1000 EDAC	Rate 1000 rel	Rate 5000 no EDAC	Rate 5000 EDAC	Rate 5000 rel
Stack						
# experiments	101	99		50	49	
⟨**# injections** ⟩	156	176	1.12	165	215	1.30
⟨**# succ. WD resets**⟩	19.0	21.8	1.27	95.9	124	1.29
#inj to #WD ratios	8.2	8.1		1.7	1.7	
Global						
# experiments	100	97		49	34	
⟨**# injections**⟩	684	4575	6.69	676	2612	3.86
⟨**# succ. WD resets**⟩	97.5	658	6.75	480	1843	3.84
#inj to #WD ratios	7.0	7.0		1.4	1.4	
Global, HTL						
# experiments					50	
⟨**# injections**⟩					3662	5.42
⟨**# succ. WD resets**⟩					2585	5.39
#inj to #WD ratios					1.4	
Stack & Global						
# experiments	99	95		98	99	
⟨**# injections**⟩	204	270	1.32	238	313	1.32
⟨**# succ. WD resets**⟩	13.8	18.3	1.33	76.2	101	1.33
#inj to #WD ratios	14.8	14.8		3.1	3.1	
Operating System						
# experiments	99			97		
⟨**# injections**⟩	8.81			9.31		
⟨**# succ. WD resets**⟩	28.2			150		
#inj to #WD ratios	0.31			0.062		
Dummy						
# experiments		12				
⟨**# injections**⟩		22206				
⟨**# succ. WD resets**⟩		1681				

Note that the ratios of ⟨**# injections**⟩ to ⟨**# succ. WD resets**⟩ are proportional to the injection rates and the size of the aggregate targets. This means that experiments on the same targets with constant rates have similar ratios. Furthermore, as the ratios are also proportional to size of the aggregate target, the ratios of **#inj to #WD ratios** are proportional to their rates, for same target. For instance, for the 'stacks'-target, the ratios for rate 1000 (8.1) is about a factor 5 higher, than the ratios for rate 5000 (1.7) , corresponding to a factor 5 in rate. Likewise, for the other targets. The fact that the ratios are equal for experiments on targets with same aggregate size, and their ratios are equal to the ratios of the injection rates pro-

vides evidence that the injections are performed with the correct rates (i.e. that the stochastic simulation is working as expected).

The failure-causes for each series of experiments have been analyzed, which is summarized in the following tables:

Target	Rate 1000					Rate 5000				
	no EDAC		EDAC			no EDAC		EDAC		
Figure	#	%	#	%	±%	#	%	#	%	±%
Stack										
Data Abort	31	30.1	35	35.1	5.0	13	26.0	18	36.7	10.7
Pref. Abort	11	10.9	5	5.1	-5.8	8	16.0	3	6.1	-9.9
Undef. Instr.	37	36.6	39	39.4	2.8	17	34.0	17	34.7	0.7
WD t.o.	12	11.9	12	12.1	0.2	4	8.0	7	14.3	6.3
Unkn. Mess.	3	2.9				2	4.0			
PDU large						1	2.0			
Mailb. full	6	5.9	8	8.1	2.2	3	6.0	4	8.2	2.2
Copy failed	1	1.0				1	2.0			
Recv failed						1	2.0			

Target	Rate 1000					Rate 5000				
	no EDAC		EDAC			no EDAC		EDAC		
Figure	#	%	#	%	±%	#	%	#	%	±%
Global										
Data Abort	13	13.0	29	29.9	16.9	7	14.3	7	20.6	6.3
Pref. Abort	1	1.0								
Undef. Instr.			12	13.4						
WD t.o.	48	48.0	36	37.1	-10.9	24	49.0	9	26.5	-22.5
Time-limit			9	9.3				14	41.2	
Bad scr. handle	6	6.0				2	4.1			
Unkn. Mess.	20	20.0	11	11.3	-8.7	11	22.4	4	11.8	-10.6
Mailb. full	12	12.0				5	10.2			

Target	Rate 5000 EDAC		
Figure	#	%	±%
Global, HTL			
Data Abort	17	34	19.7
Undef. Instr.	4	8	
WD t.o.	9	18	-31.0
Time-limit	9	18	
Unkn. Mess.	11	22	-0.4

Target / Figure	Rate 1000 no EDAC #	Rate 1000 no EDAC %	Rate 1000 EDAC #	Rate 1000 EDAC %	Rate 1000 ±%	Rate 5000 no EDAC #	Rate 5000 no EDAC %	Rate 5000 EDAC #	Rate 5000 EDAC %	Rate 5000 ±%
Stack & Global										
Data Abort	32	32.3	36	37.9	5.6	37	37.8	31	31.3	-6.5
Pref. Abort	6	6.1	8	8.4	2.3	6	6.1	7	7.1	1.0
Undef. Instr.	33	33.3	29	30.5	-2.8	28	28.6	37	37.4	8.8
WD t.o.	17	17.2	18	18.9	1.7	12	12.2	20	20.2	8.0
Bad scr. handle	1	1.0								
Unkn. Mess.	3	3.0				9	9.2	1	1.0	-8.2
Mailb. full	6	6.1	4	4.2	-1.9	4	4.1	3	3.0	-1.1
Copy failed	1	1.0				2	2.0			

Target / Figure	Rate 1000 no EDAC #	Rate 1000 no EDAC %	Rate 5000 EDAC #	Rate 5000 EDAC %	±%
Operating system					
Data Abort	34	34.3	36	37.1	2.8
Pref. Abort	3	3.0	2	2.1	-0.9
Undef. Instr.	17	17.2	20	20.6	3.4
WD t.o.	29	29.3	31	32.0	2.7
Mailb. full	16	16.2	8	8.2	-8.0

Target / Figure	Rate 5000 EDAC #	Rate 5000 EDAC %
Dummy		
Time-limit	12	100

The experiments performed on the dummy target show that for every invocation, it runs until it reaches its timelimit.

Figure 6.2 shows the number of successful injections and watchdog resets of the experiments on the stack. As can be seen, the effects of EDAC is rather limited: The increase in all figures is in the range 12 to 30 percent. The most notable change in the causes of failure (figure 6.3), is that EDAC seems to cause a decrease in the number of prefetch aborts at the expense of a similar increase in the number of data aborts. This effect seems to be larger for rate 5000. An explanation for this could be that the protection of the stacks is successful at protecting code-pointers (i.e. return addresses for function), thus preventing run-away execution.

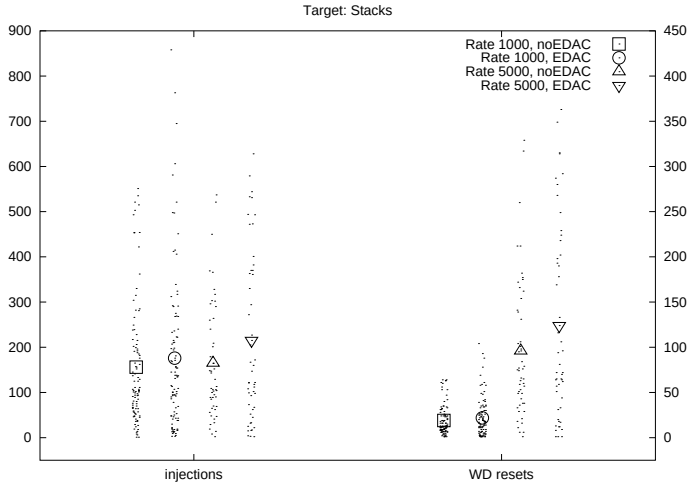

Figure 6.2: The number of injections and successful watchdog resets before failure for target 'Stacks'.

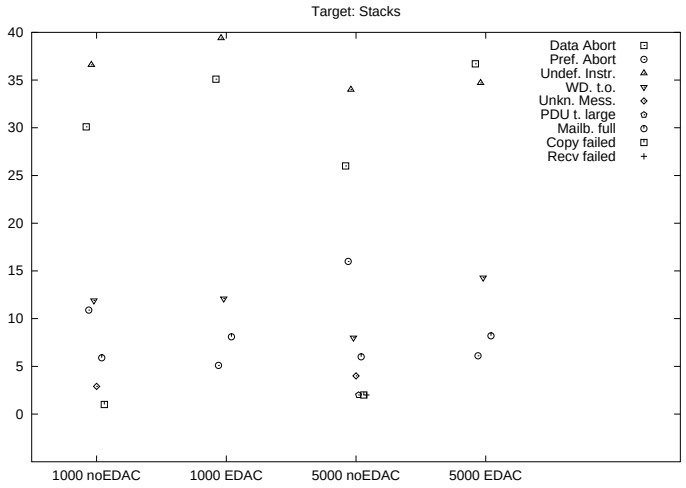

Figure 6.3: Frequency of failure-causes for target 'Stacks'.

As can be seen in figure 6.4, the effect of EDAC for rate 1000 is an increase in runtime and number of injection of a factor 6.7. For rate 5000 with a time limit of 60 minutes, the increase is about 3.8, but figure 6.5 shows that the time limit account more than 41 percent of the failure causes, which clearly limits the average runtime. For the high-limit experiments, the use of EDAC increases the runtime and the number of injections with a factor of about 5.4. This factor is smaller than for rate 1000, but the failure-causes reveal that for rate 5000 with raised time limit, there is still twice as many experiments, reaching the time limit, than for rate 1000.

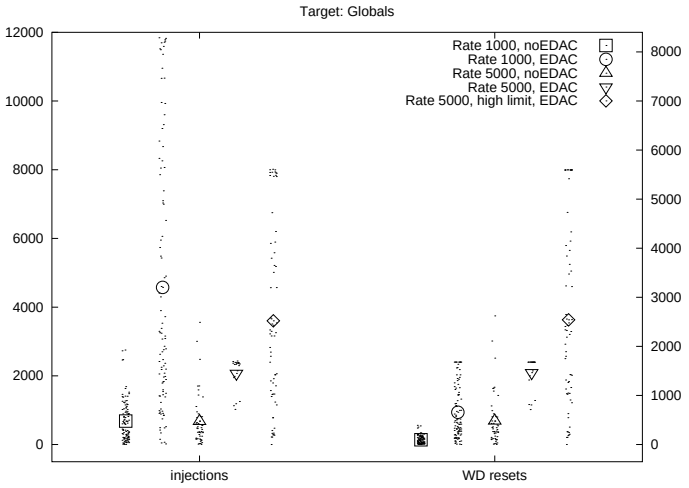

Figure 6.4: The number of injections and successful watchdog resets before failure for target 'Globals'.

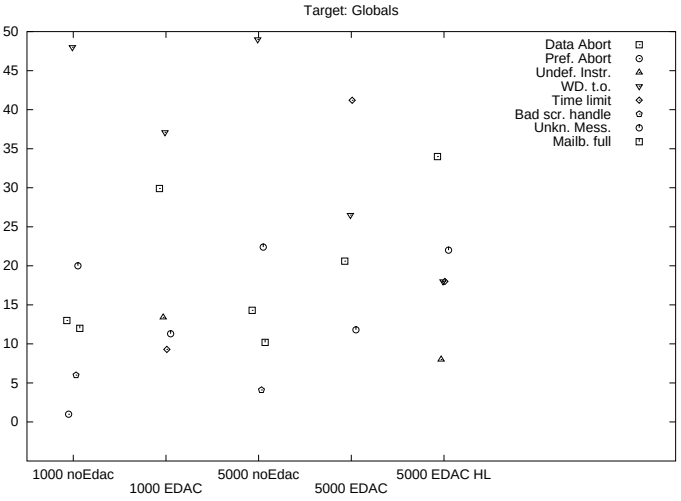

Figure 6.5: Frequency of failure-causes for target 'Globals'.

The runtime and number of injections in the experiments on the combined ’stacks and globals’-target are about 20 percent lower than the corresponding figures for the stacks - experiments. This is not surprising, as the injection-rates in the stack-subtargets are the same as for the corresponding stack-experiment, but in addition, faults are also injected into the global-subtargets.

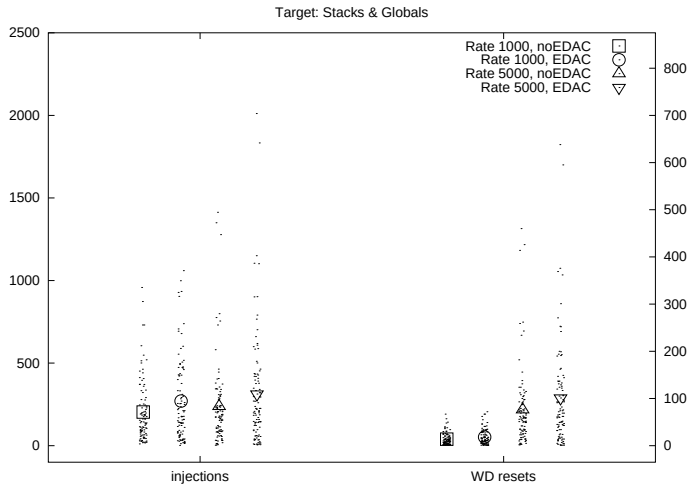

Figure 6.6: The number of injections and successful watchdog resets before failure for target ’Stacks & Globals’.

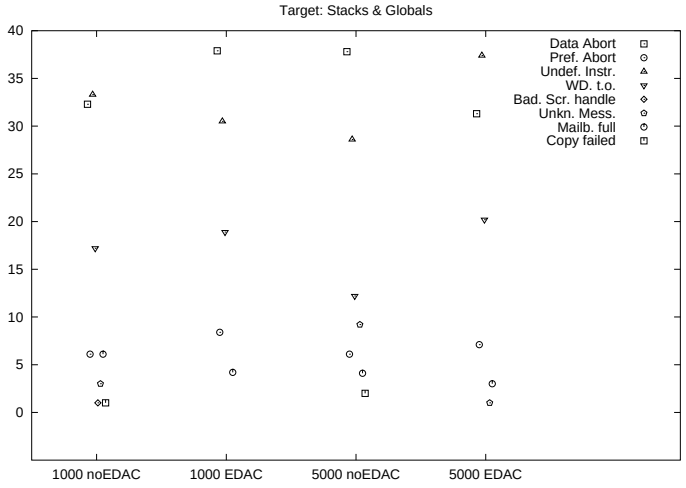

Figure 6.7: Frequency of failure-causes for target ’Stacks & Globals’.

As the os-data are opaque data to the application, they are not protected by EDAC, and the experiments on the os-data are used only to investigate the effects of the injections. As can be seen from figure 6.8, the os-data are highly sensitive to faults. This is due to the fact that the os-data are used internally by for instance the scheduler.

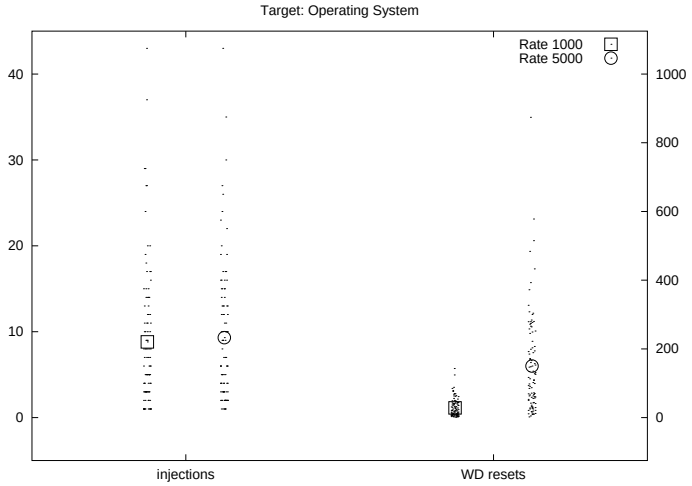

Figure 6.8: The number of injections and successful watchdog resets before failure for target ’Operating System’.

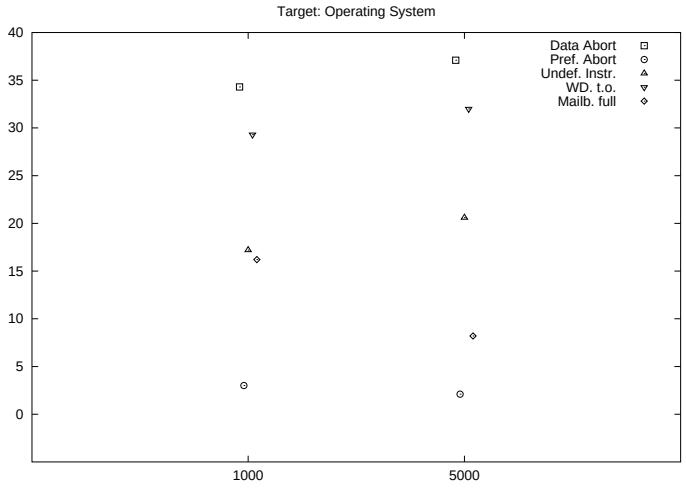

Figure 6.9: Frequency of failure-causes for target ’Operating System’.

6.3 Discussion

As with the failsafe experiments, it is noted that the injection rates are many times higher than what can be expected in LEO. Again, the question is whether the results gathered are representative of lower rates. However, the experiments are very time consuming, and the performed experiments already represent a significant amount of CPU-time. With an aggregate fault rate of 10 faults per day, an experiment on the 'stacks and globals'-target has an estimated runtime of more than 90 days.

While allocating half the RAM for parity bits provides a systematic and fast mapping, it is also quite wasteful. Not only does it allocate parity-bits for the assumed static parts of the program, but it also only utilizes half the allocated space, as only four of every 8 allocated bits are used. A solution to the latter problem could be to use a (16,8) code, for instance the code used in [40].

An other question related to the EDAC is how to correct bit-errors: A general principle of scrubbing is that all single bit-errors should be corrected before they grow into multi-bit errors. This is usually ensured by a scrubber process, but if the bit-error rate is sufficiently low, it might be acceptable not to employ a scrubber, but only to correct errors on a demand basis (i.e. faulty codeword are corrected when they are loaded by the processor). The Hamming code provides means for directly correction of errors in both the data and parity part of the codeword, but codes such as [40] can only correct errors on the data-part (the parity can, of course, be regenerated from the data-part). However, even though the chosen code does provide means for directly correction of errors in both parts of the codeword, it may make sense not to correct errors in the parity part, if the temporal scope of the codeword is limited (as it is the case for the protection of the stack in the encode/decode-primitives).

The manual part of the EDAC-system is somewhat complicated to use, as it requires the user to supply information intrinsic to the compilation of the program. Also, this part of the EDAC-system requires quite disciplined use, as performing validation on data which has not been protected, may cause corruptions, as the data-parts do not correspond to the the parity-parts.

A weakness in the scrubber-system is that the synchronization of the regions is based on mutexes, on which the scrubber might sleep. This means that if a process locks its region for an extended period, the scrubber might sleep on the regions mutex, thereby causing the suspension of scrubbing on all regions. However, it does not cause starvation of other processes, i.e. processes will still be able to lock their respective regions. A simple way to solve this problem is to use a timed lock operation, when acquiring the region – this way the scrubber can simply skip any locked regions.

6.4 Testing

As with the failsafe injector, the application injector logging provides means for testing its operation, as injector address and contents before and after are printed.

As with the failsafe injector, the application also builds on an existing code-base, as the packet-router is based on the packet-router used in the DTUSat application. However, it is streamlined and reorganized slightly and modified to add support for injection and EDAC. The initial operation of the message-passing was largely tested by printing the contents of the received messages, but some of the algorithms provide means for checking themselves. For instance, if the *Out*-module cannot receive messages from the *In*-module, the *A*-device will block, and if the watchdog does not receive all the expected messages and thereby resets the *WD*-device, it will halt the application. Also, a number of experiments have been performed on the dummy target, which shows that the application is capable of running until it reaches its timelimit.

Testing the Tabulated Hamming Implementation

The implemented Hamming code has the ability to correct single-bit errors. As each code-word is 12 bits, for each value of the data part, there is 13 code-words which can correctly be decoded (one code-word without errors, and 12 code-word with one bit-error each). This gives a total of $2^8 \cdot 13 = 3328$ code-words which can be decoded. Exhaustive search of this combination-space is trivial, so the implementation is tested as follows:

For each value of the data-part, the parity is calculated. Then the syndrome of data-parity tuple is calculated, and it is ensured that it is zero. Then, for each 4 bits of the calculated parity, the bits are flipped in succession, thereby creating a corrupted code-word. For each bit flipped, the syndrome is calculated, and it is ensured that it is nonzero. Then the syndrome is translated into error-bitmasks, and it is ensured that they indicates no errors in the data part, and an error in the parity. The error-bitmask is then applied to the corrupted code-word, and then it is assured that the code-word has been corrected. Then the same procedure is applied to the data-part. For each calculation of the parity and the syndrome, both implementations are used, and it is ensured that the results are equal.

Chapter 7

Conclusion

7.1 What Has Been Done

This report is the results of a masters thesis in software-based fault-tolerance. The report documents the development of fault-injectors, which simulate the effects of radiation on space-borne computers and their storage. In accordance with the nature of the faults, the fault-injectors are implemented as simulated Poisson-processes.

As the injectors fit tightly to their injection-targets, two injectors are implemented because experiments are run on two targets. The hardware used to run the injectors and targets are prototype versions of the DTUSat onboard computers, running either the flight-version of the DTUSat boot-software (failsafe software); or an application running on the embedded real-time operating system *eCos*. Both types of experiments investigate the failure modes and rates of their target software, but the experiments on the application also serve to test an implementation of an error detecting and correcting code (EDAC).

There are two principal ways in which injection experiments can be performed: Either a single fault is injected, after which the system is examined to determine if it has been affected; or faults are injected continuously (subject to a stochastic model) until the system fails. All experiments are performed using the test-until-destruction -approach.

The EDAC system comprises two parts: An automatic part and a manual part. The automatic part is implemented as a classic scrubber process, which automatically detects and corrects fault in the storage, while the manual part is implemented with cooperation from the user. It is the intension that the manual part can be replaced with an automatic method, implemented either directly in the compiler, or as a standalone tool.

In order to automate the experiments, a harness is produced for each target, which allows batches of experiments to be run automatically. The harnesses store the output from the experiments in logfiles, which are then processed and refined offline.

7.2 Limitations

In this section, a number of weak points in the implementations are pointed out.

The injectors are implemented in software, which has both advantages and disadvantages. The main advantage is that the actual injection code is relatively simple to implement, and that no new hardware is needed. The disadvantages is that even though the stochastic simulation is carefully implemented, the granularity of the simulation is somewhat limited by the hardware (for instance by limited resolution in the hardware-timers). Also, the injectors are limited in their scope, in that they do not inject faults into implicit storage such as CPU- and device-registers.

In order to complete the experiments in a reasonable time, the injectors are run with fault-rates that are higher than what can be expected in space. However, analysis of the results indicate that the failure modes and rates are comparable over a wide range of injection rates.

7.3 Results

The experiments on the failsafe software show that, on average, it takes about 20 injections for the software to be affected (i.e. the first transient failure), and about 100 injections for the software to fail completely (i.e. destruction).

While the failsafe software contains only a single injection-target, the application is divided into four subtargets: These cover the program stacks (*stacks*), global variables (*globals*), both stacks and globals (*stacks and globals*), and operating system data (*os-data*). The experiments on the *globals*-target show that when using the EDAC system, the application runs 5-6 times longer before experiencing problems, than without the EDAC system. However, the increases in runtime are only 12 to 30 percent for the *stacks*- and *stacks and globals*- targets. The experiments on the *os-data*-target were performed only to test its sensitivity, and were not performed using the EDAC system, as this would require changes to the operating system. As can be expected, the operating system is highly sensitive to injected faults.

7.4 Conclusion

Based on the results obtained from the experiments, it can be concluded that EDAC-protection of the program stacks is infeasible in its current implementation.

The increase in runtime is a mere 30 % at best, and while it can be argued that a small increase is better than no increase, it must be noted that protection of the stacks is quite complicated to use in its current implementation. It is therefore judged that protection of the stacks is not a competitive alternative to hardware-based methods (e.g. using error-correcting RAM and radiation-hardened devices). However, is it also found that protection of global storage was effective, yielding a significant increase in runtime.

This suggests a division of the storage in two parts – one part which is software-hardened, and one which is hardware-hardened. The software-hardened part could be used to store data, which is not critical to the reliability of the system, while the other could be used for the critical parts. The non system-critical data is typically scientific data, which are obtained during the mission and stored for later download (to the Earth). This implies a store-retrieve-delete use-pattern, in which data usually are accessed quite infrequently. The system-critical data are usually those attributed to the operating system and the application software.

There is a trend for using an ever increasing amount of storage for scientific data, and since hardware hardened storage typically has an increased cost, higher power-consumption and lower capacity, compared to non-hardened storage, there are economically and design-based reasons for avoiding their use.

7.5 Future Work

There is a number of areas which could be further investigated, for instance: It would be interesting to perform experiments with rates similar to what is expected in space, and a more elaborate analysis of the failure modes might also be of interest.

Bibliography

[1] Pless, Vera
Introduction to the Theory of Error-correcting Codes
John Wiley & Sons Inc, 1998

[2] Blanquart, Jean-Paul; Coppola, Paolo
RAMS Related Software Requirements and Design Constraints
PASCON, 2000

[3] ESTEC
Study of GNSS-2/Galileo System Software Certification ESTEC, 2000

[4] Gonthier, Georges; Lévy, Jean-Jacques
Software Robustness Engineering: Robustness Methodology Survey
Terma, 2002

[5] Pedersen, Jan S.; Palm, Steen U.
Software Robustness Engineering: Design and Coding Constraints
Terma, 2002

[6] Press, William H.
Numerical Recipes in C, 2nd. ed.
University of Cambridge 1992

[7] Jørsboe, Ole G.
Sandsynlighedsregning
Matematisk Institut, DTU 1992

[8] Hatton, Les
Safer C
McGraw-Hill 1995

[9] Kernighan, Brian W.; Ritchie, Dennis M.
The C Programming Language, 2nd. ed.
Prentice Hall 1988

[10] Hall, Douglas V.
Microcomputers and Interfaces 2nd. ed.
McGraw-Hill 1992

[11] Manna, Zohar; Pnueli, Amir
Temporal Verification of Reactive Systems: Safety
Springer-Verlag 1995

[12] European Space Agency
ESA Software Engineering Standards, issue 2
ESA Publications Division 1992

[13] Holdt, David
FSTERM Manual
IMM/CSE, DTU 2003

[14] Holdt, David
DTUSat boot-software, implementation details
IMM/CSE, DTU 2003

[15] Atmel
AT91 ARM®Thumb®Microcontrollers
Atmel Corporation 2001

[16] ARM 7TDMI Data Sheet
Advanced RISC Machines Ltd (ARM), 1995

[17] Earnshaw, Richard
Procedure Call Standard for the ARM®Architecture
ARM Limited 2003

[18] Writing Interrupt Handlers
http://www.arm.com/support/faqdev/1456.html

[19] Interrupt Generation Using the AT91 Timer/Counter
Atmel, http://www.atmel.com

[20] Interrupt Management: Auto-vectoring and Prioritization
Atmel, http://www.atmel.com

[21] Extensions to the C Language Family
http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc_4.html

[22] Red Hat
eCos™Reference Manual
Red Hat Inc. 1998, 1999, 2000

[23] TTY_IOCTL
http://homepages.cwi.nl/~ aeb/linux/man2html/man4/tty_ioctl.4.html

[24] Gerth, Rob
Concise Promela Reference
Eindhoven University
http://spinroot.com/spin/Man/Quick.html

[25] Basic Spin Manual
http://spinroot.com/spin/Man/Manual.html

[26] Spin Version 3.3: Language Reference
http://spinroot.com/spin/Man/Promela.html

[27] Spin Run-Time Options
http://spinroot.com/spin/Man/Spin.html

[28] Pan Verification Options
http://spinroot.com/spin/Man/Pan.html

[29] UPPALL – Environment for Validation and Verification of Real-Time Systems
http://www.docs.uu.se/uppall/

[30] Petterson, Paul; Lasen, Kim G.
UPPALL2k
http://www.docs.uu.se/uppall/

[31] Petterson, Paul; Lasen, Kim G.
Timed and Hybrid System
http://www.docs.uu.se/uppall/

[32] Petterson, Paul; Lasen, Kim G.; Yi, Wang
UPPALL in a Nutshell
http://www.docs.uu.se/uppall/

[33] Arora, Anish; Kulkarni, Sandeep, S.
Component based Design of Multitolerance
The Ohio State University

[34] Arora, Anish; Kulkarni, Sandeep, S.
Detectors and Correctors: A Theory of Fault-Tolerance Components
The Ohio State University

[35] Arora, Anish; Gouda, Mohammed
Closure and Convergence: A Foundation of Fault-Tolerance Computing
The Ohio State University

[36] Matsumoto, M.; Nishimura, T.
Mersenne Twister:
A 623-dimensionally equidistributed uniform pseudo-random number generator
ACM Transactions on Modeling and Computer Simulation, vol. 8, issue. 1

[37] Dijkstra, Edsger W.
Self-stabilizing Systems in Spite of Distributed Control
Communications of the ACM, vol. 17, number 11, 1974

[38] LaBel, K.A. et al.
Anatomy of an In-flight Anomaly:
Investigation of Proton-Induced SEE Test Results for Stacked IBM DRAMs
IEEE Transactions on nuclear Science, vol. 45, number 6, dec. 1998

[39] Swift, Gary M.; Guertin, Steven M.
In-flight Observations of Multiple-Bit Upset in DRAMs
IEEE Transactions on nuclear Science, vol. 47, number 6, dec. 2000

[40] Hodgart, M.S.; Tiggeler, H.A.B.
A (16,8) Error Correcting Code (T=2) for Critical Memory Applications
Surrey Space Center, University of Surrey
HTTP://www.estec.esa.nl/wsmwww/core/edac8cyclic.pdf

[41] Benso, Alfredo; Carlo, Stefano Di; Natale, Giorgio Di; Prinetto, Paolo
SEU Effect Analysis in a Open-Source Router
via a Distributed Fault Injection Environment
IEEE, 2001

[42] Velazco, R.; Corominas, A.; Ferreyra, P.
Injecting Bit Flip Faults by Means of a Purely Software Approach:
a Case Studied
Proc. of the 17th IEEE int. Symposium on Defect
and Fault Tolerance in VLSI Systems (DFT'02)

[43] Cambell, Athur B.
SEU Flight Data from the CRRES MEP
IEEE Transactions on Nuclear Science, vol. 38, number 6, dec 1991

[44] Underwood, C.I.; Ward, J.W.; Dyer, C.S.; Sims, A.J.
Observations of Single-Event Upsets in Non-Hardened High-Density SRAMs in Sun-Synchronous Orbit
IEEE Transactions on Nuclear Science, vol. 39, number 6, dec 1992

[45] Harboe-Sørensen, R.; Daly, E.J.; Underwood, C.I.; Ward, J; Adams, L.
The Behavior of Measured SEU at Low Altitude
During Periods of High Solar Activity
IEEE Transactions on Nuclear Science, vol. 37, number 6, dec 1990

[46] Harboe-Sørensen, R.; Daly, E.J.; Adams, L.; Underwood, C.I.; Müller, R.
Observation and Prediction of SEU in Hitachi SRAMs
in Low Altitude Polar Orbits
IEEE Transactions on Nuclear Science, vol. 40, number 6, dec 1993

Appendix A

Modeling in SPIN

Recall from section 3.1, that the abstract models form a basis for analyzing and proving properties of fault tolerant programs. As an alternative to manual proofs, this section presents a method for proving the same properties, by using a verification tool for concurrent systems, called SPIN.

A.1 Promela

In the following, a brief overview of the features and usage of SPIN is given. It is by no means meant to provide a detailed description, as it is given in [24], [25], and [26].

SPIN is a tool for analyzing the logical consistency for concurrent systems. The system to be analyzed is described in `promela`, a meta-language for process description. Programs in promela are described using variables, processes and channels. The use of variables is somewhat similar to that of 'C', in that the notation for basic arithmetics in `promela` is derived from 'C', and the intrinsic types are equivalent. Unlike 'C', `promela` directly supports the notion of processes, but does not support functions. In many cases however, functions can be modeled using processes and channels. Just like 'C', `promela` uses the notion of variable-scopes (i.e. local and global), but as functions are not supported, local and global are relative to processes, not functions. Channels are used to provide structured communication between processes, similar to that of CSP (Communicating Sequential Processes). However, as channels are not used in the following examples, they will not be discussed further. SPIN uses the 'C' preprocessor to provide support for C-style macros, which are used extensively in the following examples.

Guarded Statements

A boolean expression, used as a statement, is interpreted as a guard in `promela`. For instance, the construct `(a>b) -> d=e;` comprises two statements; the first (`a>b`) is used as a guard,

which only allows the second (`d=e`) to be executed if the first is true. Note the use of `->`: In `promela`, `;` and `->` are interchangeable as statement-separators, but in the following `->` is used to separate guards from statements, as it emphasizes the nature of the construct.

Do-Constructs

One of the main control-structures in `promela` is the `do`-construct, which has the general structure:

```
1  do
2  :: alternative_1
3  :: ...
4  :: alternative_n
5  od
```

The `do`-construct is executed as an infinite loop, in which an executable alternative is selected for execution on each iteration. Alternatives which are non-executable include guarded alternatives with false guards, and channel operations which are blocked. If more than one alternative is executable, a nondeterministic choice is simulated. Alternatives may contain a `break` construct, which causes the `do`-construct to terminate.

If-Constructs

Another important control-structure is the `if`-construct. The structure of the `if`-construct is similar to that of the `do`-construct:

```
1  if
2  :: alternative_1
3  :: ...
4  :: alternative_n
5  :: else alternative_e
6  fi
```

Like the `do`-construct, a number of alternatives is given, but the `if`-construct also offers the optional `else`-construct, which is executable when none of the others are.

Atomic Operations

Sometimes it is necessary or beneficial to be able to specify that compound operations should be treated as atomic. For instance, certain protocols rely on the ability to perform atomic increment or swap -operations, but in other cases, atomic operations can be used to optimize the `promela`-program, as it reduces the number of states in program.

Starting Processes

When a promela program starts, the special init process is implicitly started. The init-process may instantiate other processes from process-types, by using the run-statement. A process has a function-like structure, which supports parameters to be passed from the creator.

An other special process is the ”never-claim”, identified by the never-keyword. The never-claim is a process which is expected never to terminate, and doing so raises an error (i.e. the never-claim has been violated). The never-claim is used to express and verify properties of the program, such as invariants and temporal dependencies.

A.2 Examples

In the following examples, it will be demonstrated how to translate some of the proofs given in [35] and [33] into promela-programs in a fairly systematic way. It is, however, not possible to translate every construct in every example in a one-to-one fashion, as some of the examples use high-level primitives, which are not directly representable in promela.

A.2.1 Example: Two-phase

The two-phase program, due to [35], implements a distributed voting process, in which a number of processes cooperate in making a decision. The idea is the following:

The processes should reach a decision, either *commit* or *abort* based on votes cast by the individual processes. All processes should reach the same decision, and the *commit* decision should only be reached if all processes vote *yes*. If all processes vote *yes*, then all processes reach a *commit* decision. The faults modeled are crash-restart, i.e. that processes spontaneously stops and restarts.

The protocol consists of two phases: First the vote are cast, and then a decision is made. Each process has three variables, which define the state of the process: The fault-state of the process is determined by its up-variable – when $up = true$ the process is running, and when $up = false$ the process is halted. The phase of decision is controlled by ph – when $ph = 0$ the process has not yet cast its vote; when $ph = 1$ the process has cast its vote, but has not yet made its decision; and when $ph = 2$ the process has made its decision. The vote and the decision is represented by d: In phase 1 it holds the vote of the process, and in phase 2 it holds the decision.

The system consists of a number of processes. A central process c acts as a coordinator, in that it effectively controls the phases of all processes.

When process c is in phase 0, it casts its vote, and advances to phase 1. In phase 1, it examines the states of the other processes: If they are all found to be running (i.e. $up = true$), and in phase 1, and has cast a *yes*-vote (i.e. $d = true$), process c advances to phase 2, with the decision *commit* (i.e. $d = true$). However, if the examination shows that there is one or more

processes, which either have failed (i.e. $up = false$), or have advanced to a higher phase with the decision *abort*, process c advances to phase 2, with the decision *abort* (i.e. $d = false$).

When in phase 0, the other processes cast their vote, provided process c is running. If process c is found not to be running, the other processes will advance to phase 2, with the decision *abort*.

In [35], the following pseudo-code for the program is given:

program	$Two-phase$	
constant	$X: set\ of\ ID;$	
	$c: X;$	
var	$ph: array\ X\ of\ 0..2;$	
	$up: array\ X\ of\ boolean;$	
	$d: array\ X\ of\ boolean;$	
process	$j: X;$	
parameter	$l: X;$	
begin	$j = c \wedge up.j \wedge ph.j = 0$	$\rightarrow ph.j, d.j := 1, ?$
$\|\|$	$j = c \wedge up.j \wedge ph.j = 1 \wedge (\forall l \in X : up.l = 1 \wedge d.l)$	$\rightarrow ph.j, d.j := 2, true$
$\|\|$	$j = c \wedge up.j \wedge ph.j = 1 \wedge$	
	$(\exists l \in X : \neg up.l = 1 \vee (ph.l \geq 1 \wedge \neg d.l))$	$\rightarrow ph.j, d.j := 2, false$
$\|\|$	$j \neq c \wedge up.j \wedge ph.j = 0 \wedge (up.c \wedge ph.c = 1)$	$\rightarrow ph.j, d.j := 1, ?$
$\|\|$	$j \neq c \wedge up.j \wedge ph.j = 0 \wedge \neg up.c$	$\rightarrow ph.j, d.j := 2, false$
$\|\|$	$j \neq c \wedge up.j \wedge ph.j < ph.k \wedge (up.k \wedge ph.k = 2)$	$\rightarrow ph.j, d.j := 2, d.k$
end		
faults	F	
	$\{true$	$\rightarrow up.j = \neg up.j\}$

The following invariant is shown for the program:

$$
\begin{array}{lll}
S = & ph.c = 0 & \Rightarrow (\forall j : ph.j = 0 \vee (ph.j = 2 \wedge \neg d.j)) \\
\wedge & ph.c = 1 & \Rightarrow (\forall j : ph.j \neq 2 \vee \neg d.j) \\
\wedge & ph.c = 2 \wedge d.c & \Rightarrow (\forall j : ph.j \neq 0 \wedge d.j) \\
\wedge & ph.c = 2 \wedge \neg d.c & \Rightarrow (\forall j : ph.j \neq 2 \vee \neg d.j)
\end{array}
$$

The program is now expressed in `promela`, with the purpose of proving the invariant.

As `promela` does not support the implication ($\Rightarrow$), existence ($\exists$) and forall ($\forall$) -operators, these are defined using macros. The implication-operator is simply defined from its definition:

```
#define IMPLIES(a,b) ((a)->(b):true)
```

Note that The Ternary Infix Operator in `promela` uses `->` instead of `?`, as `?` is used for channel operations. The notation `a->b:c` is therefore equivalent to `a?b:c` in 'C'.

The quantifiers are defined with the assumption that the domain they work upon is the integers in the range 0..2. The argument for the macros is a predicate, which is evaluated for each element in the domain, thus:

```
#define FORALL(p) (p(0) && p(1) && p(2))
#define EXIST(p)  (p(0) || p(1) || p(2))
```

The boolean expression used in the program are quite complex, so they are build in steps, using macro-expansion. First, the subexpressions used in the quantifiers are defined, and then they are used in the quantifiers, thus:

```
#define UP_PH1_YES(l)   (( up[(l)]) &&  (ph[(l)]==1) && ( d[(l)]))
#define DOWN_PH12_NO(l) ((!up[(l)]) || ((ph[(l)]>=1) && (!d[(l)])))

byte ph[N] = 0;
bool up[N] = true;
bool  d[N];
bool vote[N] = 1 ;

proctype process(byte j)
{
      byte k;
      k=0;

end:  do
      :: atomic{ (j==c && up[j] && ph[j]==0) ->
            ph[j]=1 ; d[j]=vote[j] }

      :: atomic{ (j==c && up[j] && ph[j]==1 && FORALL(UP_PH1_YES) )    ->
            ph[j]=2; d[j]=true }

      :: atomic{ (j==c && up[j] && ph[j]==1 && EXIST(DOWN_PH12_NO) )   ->
            ph[j]=2; d[j]=false }

      :: atomic{ (j!=c && up[j] && ph[j]==0 && (up[c] && ph[c]==1 ))   ->
            ph[j]=1 ; d[j]=vote[j] }

      :: atomic{ (j!=c && up[j] && ph[j]==0 && !up[c]) ->
            ph[j]=2 ; d[j]=false }

      :: atomic{ (j!=c && up[j] && ph[j]<ph[k] && (up[k] && ph[k]==2)) ->
            ph[j]=2 ; d[j]=d[k] }
      od
}
```

The constant `N` is the number of processes (3), and the `vote` array is used to hold the value of the votes being cast (corresponding to the question-marks in the abstract program). As can be seen, the program uses guarded statements, and each compound statement is made atomic in accordance with the abstract description.

The faults are implemented as separate processes, in a one-to-one correspondence, and are implemented such that they cause a random number of faults, and then terminate. This is achieved by using the fact that a random execution is simulated when more than one alternative is executable. This way, one of the alternatives is to cause a single fault, while the other is to stop causing faults.

```
proctype fail(byte j)
{
end:  do
```

```
      :: (ph[j]!=2) -> up[j] = !up[j]
      :: break
      od
}
```

The processes are started from the `init`-process, simply by six run-statements (in parallel):

```
init
{
      atomic
      {
            run process(0);
            run process(1);
            run process(2);

            run fail(0);
            run fail(1);
            run fail(2);
      }
}
```

Like the expressions in the main processes, the invariant is quite complicated, so it too is build in steps. First, S is divided into subexpressions corresponding to the conjuncts in S:

```
#define invariant (S1 && S2 && S3 && S4)
```

Then each Sn is defined, and for each, another subexpression is defined, thus:

```
#define S1  (IMPLIES(ph[c]==0, FORALL(PH0_PH2_NO)))
#define S2  (IMPLIES(ph[c]==1, FORALL(PHN2_NO)))
#define S3  (IMPLIES(((ph[c]==2) &&  d[c]), FORALL(PHN0_YES)))
#define S4  (IMPLIES(((ph[c]==2) && !d[c]), FORALL(PHN2_NO)))

#define PH0_PH2_NO(j)   ((ph[(j)]==0) || ((ph[(j)]==2) && !d[(j)]))
#define PHN2_NO(j)      ((ph[(j)]!=2) || (!d[(j)]))
#define PHN0_YES(j)     ((ph[(j)]!=0) &&  (d[(j)]))
#define PHN2_NO(j)      ((ph[(j)]!=2) || (!d[(j)]))
```

Then, using the compound expression for the invariant, the never-claim can be defined. The never-claim used is an idiom for an invariant, taken from the SPIN documentation. Note that since our requirement is formulated as a negation, i.e. something that should never happen, the invariant is negated in the never-claim. The comment is an LTL-formula (linear-time temporal logic) specifying the same requirement as the never-claim (invariance). The textual interpretation of this is ”always not invariant”.

```
never /* []!(invariant) */
{
      do
      :: (invariant)
      :: (! (invariant)) -> break
      od
}
```

A.2.2 Example: Token-ring

This example shows the use of the incremental approach, in which layers of functionality are added sequentially. The program implements a distributed token-based mutual-exclusion algorithm. The processes are logically placed in an array, which is treated as a ring. Each process has a flag, which is accessible to all processes, and the idea is that each process looks at the flag of the previous process, and assigns this value to its own flag. The first process looks at the last process' flag, and assigns the negated value of its flag to its own flag. The copying of the neighboring values creates a ripple-effect, in which alternating waves zeroes and ones ripple through the flags. At any moment there should be exactly one pair of neighboring processes which has different flag-values, and this marks the position of the token (i.e. which process owns the critical section). In the specification, due to [33], j is the number of the process, and $+_2$ denotes addition modulo 2 (i.e. exclusive-or):

$TR1::$	$j \neq 0$	$\wedge$	$x.j \neq x.(j-1)$	$\rightarrow$	$x.j := x.(j-1)$
$TR2::$	$j = 0$	$\wedge$	$x.j \neq (x.N +_2 1)$	$\rightarrow$	$x.j := x.N +_2 1$

If the values of the flags are interpreted as a bitstring, the value of this string can be expressed with a regular expression. If X denotes the bitstring, then the following is invariant:

$$S_{TR} = X \in \left(\bigcup l : 0 \leq l \leq N+1 : \left(0^l 1^{(N+1-l)} \cup 1^l 0^{(N+1-l)} \right) \right)$$

While the regular expressions provide concise descriptions, there is also a significant gap between what can be expressed with regular expressions, and what can be expressed as never-claims in a `promela`-program. However, with a few modifications the above expression can be expressed in `promela`. Instead of looking for runs of zeroes and ones, one can look for neighboring flags which are not equal, of which there should be at most one pair. The following function is used to calculate the difference between the flags of two neighboring processes.

$$f(j) \equiv \begin{array}{l} \textbf{if } x.j = x.(j+1) \textbf{ then } 1 \\ \textbf{else } 0 \end{array}$$

Determining the number of different neighboring flags is now simply a matter of calculating $\sum_{j=0}^{N-1} f(j)$. As there should be exactly one pair of neighboring processes with different flags, the invariant becomes $S'_{TR} = \left(\sum_{j=0}^{N-1} f(j) = 1 \right)$.

The implementation is now hardened: A new symbol, $\perp$, is introduced, which is used to symbolize that the state of $x.j$ has been corrupted, and the specification is augmented to take advantage of this. The addition simply makes sure that a process never copies a corrupt value. As long as just a single non-corrupt value exists, this will propagate to the corrupt values, and the system is able to recover.

$PTR1::$	$x.(j-1) \neq \bot$	$\wedge$	$TR1$
$PTR2::$	$x.N \neq \bot$	$\wedge$	$TR2$
Faults	$true \rightarrow x.j := \bot$		

The invariant for the augmented program is:

$$S_{PTR} = X \in \quad (\bigcup l : 0 \leq l \leq N+1 : \quad ((0 \cup \bot)^l (1 \cup \bot)^{(N+1-l)} \cup (1 \cup \bot)^l (0 \cup \bot)^{(N+1-l)})) \quad \wedge |j : x.j = \bot| \leq 1$$

Again, the invariant is expressed as a regular expression over the bitstring X, and as can be seen, the additions did not make the invariant any simpler – quite the contrary. This regular expression cannot readily be expressed in `promela`, not even by using clever transformations. However, instead of focusing on a regular expression, which captures the relationships between the internal variables, is it more beneficial to focus on the actual purpose of the algorithm: Mutual exclusion. An easy way checking if the mutual exclusion property has been violated, is to implement a global counter, which counts how many processes are in the critical region: Whenever a process enters the region, the counter is incremented, and when a process leaves the region, the counter is decremented. This can be expressed in `promela` directly.

The implementation is now hardened even more. If all processes fail (i.e. $\forall j : x.j = \bot$), the algorithm deadlocks, as all processes refuse to copy any values. The next addition addresses this problem by introducing another symbol, $\top$, which is used to indicate that $x.N$ has incurred a failure. The $\top$ symbol is propagated in reverse direction, and when $x.0 = \top$ it is reset to $x.0 = 0$. This value then ripples forward.

$FTR1::$	$x.(j-1) \neq \top$	$\wedge$ $PTR1$	
$FTR2::$	$x.N \neq \top$	$\wedge$ $PTR2$	
$FTR3::$	$x.N = \bot$		$\rightarrow x.N := \top$
$FTR4::$	$j \neq N \wedge x.j = \bot \wedge x.(j+1) = \top$		$\rightarrow x.j := \top$
$FTR5::$	$x.0 = \top$		$\rightarrow x.0 := 0$

Again, the invariant can be expressed as a regular expression (which is not representable in `promela`):

$$S_{FTR} = X \in (\bigcup l,m : 0 \leq l,m,l+m \leq N+1 : \quad ((1 \cup \bot)^l (0 \cup \bot)^m (\bot \cup \top)^{(N+1-l-m)} \cup (0 \cup \bot)^l (1 \cup \bot)^m (\bot \cup \top)^{(N+1-l-m)}))$$

Next, the program is described in `promela`. The implementation corresponds to the description of FTR.

First some constants and variables are defined. The constant `ASZ` is the size of the array (`x`), and `N` is the last (highest numbered) process. Further, the $\top$ and $\bot$ symbols are defined as constants, and a macro for modulo 2 arithmetics is defined. The variable `csc` is the critical section counter, which counts the number of processes in the CS.

```
#define ASZ 4
#define N ASZ-1

#define BOTTOM 2
#define TOP 3

#define MOD2(a) ((a)%2)

byte x[ASZ]=0;

int csc;
```

The processes are defined as a process-type, called `tokenring`. The operations on `x` directly correspond to the operations in the abstract program. The `csc`-variable is incremented upon entry and decremented upon exit of the critical section.

```
proctype tokenring(int j)
{
     byte px;

end: do
     :: atomic { ((j!=0) && (x[j-1]!=BOTTOM) &&
                            (x[j-1]!=TOP)    &&
                            (x[j]!=x[j-1])   &&
                            (x[j-1]+x[j]==1)) ->
                                    px=x[j-1]; }
          /* critical section */
          csc=csc+1;
          csc=csc-1;
          x[j]=px

     :: atomic { ((j!=0) && (x[j-1]!=BOTTOM) &&
                            (x[j-1]!=TOP)    &&
                            (x[j]!=x[j-1])   &&
                            (x[j-1]+x[j]!=1)) ->
                                    x[j]=x[j-1] }

     :: atomic { ((j==0) && (x[N]!=BOTTOM)       &&
                            (x[N]!=TOP)          &&
                            (x[j]!=MOD2(x[N]+1)) &&
                            (MOD2(x[N]+1)+x[j]==1)) ->
                                    px=MOD2(x[N]+1); }
          /* critical section */
          csc=csc+1;
          csc=csc-1;
          x[j]=px

     :: atomic { ((j==0) && (x[N]!=BOTTOM) &&
                            (x[N]!=TOP) &&
                            (x[j]!=MOD2(x[N]+1)) &&
                            (MOD2(x[N]+1)+x[j]!=1)) ->
                                   x[j]=MOD2(x[N]+1) }

     :: atomic { (x[N]==BOTTOM) -> x[N]=TOP }

     :: atomic { ((j!=N) && (x[j]==BOTTOM) &&
                            (x[j+1]==TOP)) ->
                                   x[j]=TOP }

     :: atomic { (x[0]==TOP) -> x[0]=0 }
```

```
     od
}
```

As in the previous example, the fault-actions are implemented as a separate process:

```
proctype fail(int j)
{
     do
     :: x[j]=BOTTOM
     :: break
     od
}
```

The `init`-process implements a loop, in which the processes are started.

```
init
{
     int c;

     atomic
     {
          do
          :: if
             :: (c<=N) ->
                    run tokenring(c);
                    run fail(c);
                    c=c+1
             :: else -> break
             fi
          od;
     }
}
```

The never-claim is identical to the previous example (expressing invariance). In effect, the invariant states that at most one process is in the critical section at any moment in time.

```
#define invariant S
#define S ((csc==0) || (csc==1))

never /* []!(invariant) */
{
     do
     :: (invariant)
     :: (! (invariant)) -> break
     od
}
```

A.3 Discussion

In the previous chapter, two examples from [35] and [33] are examined. It is demonstrated how it is possible to formulate and model the examples using a tool for automatic program analysis.

The main advantage of using automated tools is of course that it eliminates the need for manually proving properties, however the automated analysis is paid for in CPU and RAM

resources. It is not unusual that even a small simulation requires more than 100 MB of RAM, and it is not unrealistic to formulate problems which cannot be simulated on a 32-bit machine (due to the 'limited' address-space). SPIN does support advanced simulation-modes, which can be used to handle large problems, but these techniques have not been studied, as they are out of the scope of this report.

Another problem is the limited expressive power of `promela`, which sometimes makes it difficult to formulate the models. However, using a bit of creativity, it is possible to reformulate the models in such a way that they can be expressed in `promela`.

Appendix B

Results of Failsafe Experiments

B.1 Experiment 1: Rate 10

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
8	1	6	3	Data Abort	1085	98	0x01002810
3	4	3	191	Watchdog	3636	362	0x00000000
6	1	5	6	Data Abort	1954	182	0x01001938
14	4	8	7	Data Abort	2406	207	0x01007FE4
14	5	10	12	Comm Lost			
	1		44	Watchdog	1193	114	0x00000000
2	3	2	4	Pref. Abort	1170	98	0xA888E4B4
2	1	1	5	Pref. Abort	1499	140	0xEFEFEFEE
1	2	1	4	Pref. Abort	1121	112	0xE9E9E9E8
6	3	4	6	Watchdog	2248	235	0x00000000
17	11	15	39	Watchdog	2758	293	0x00000000
4	3	3	3	Data Abort	806	82	0x01000CDC
	1		5	Data Abort	1785	156	0x01002810
14	1	8	8	Data Abort	2394	226	0x01001938
	1		5	Comm Lost			
17	1	11	8	Data Abort	2483	225	0x01000CDC
10	1	7	5	Watchdog	1983	185	0x00000000
	1		1	Watchdog	359	31	0x00000000

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
2	1	1	7	Watchdog	2215	226	0x00000000
	1		4	Pref. Abort	774	72	0xE9E9E9E8
3	1	1	6	Watchdog	1853	171	0x00000000
	1		2	Watchdog	379	33	0x00000000
6	4	4	4	Data Abort	1177	126	0x01002D58
8	2	6	5	Pref. Abort	763	96	0x09000C08
	1		6	Watchdog	806	72	0x00000000
4	5	4	2	Data Abort	470	51	0x01002358
16	1	9	23	Pref. Abort	8155	792	0xE5D5002C
1	2	1	12	Data Abort	3374	307	0x0100120C
2	1	1	4	Pref. Abort	1108	121	0x41002518
3	1	1	3	Watchdog	1092	107	0x00000000
7	1	4	17	Data Abort	3186	304	0x01001ECC
2	3	2	2	Data Abort	430	40	0x01002358
	1		73	Watchdog	6849	625	0x00000000
8	1	4	5	Data Abort	1565	147	0x0100120C
1	2	1	3	Pref. Abort	750	80	0x05000C08
1	2	1	6	Comm Lost			
5	6	5	12	Comm Lost			
	1		11	Watchdog	1842	151	0x00000000
2	3	2	2	Pref. Abort	436	34	0xE3A04A00
2	3	2	30	Data Abort	10931	1005	0x01001EF0
	1		2	Data Abort	384	35	0x010011F8
4	5	4	5	Watchdog	1547	149	0x00000000
2	3	2	1	Watchdog	406	45	0x00000000
8	3	6	3	Undef. Instr.	1099	96	0x01003EEC
4	3	3	6	Data Abort	1911	174	0x01001938
5	2	4	7	Data Abort	2273	223	0x0100197C
	1		1	Pref. Abort	104	9	0x09001E54
4	1	3	4	Pref. Abort	1358	143	0x05001E54
4	2	3	3	Pref. Abort	798	83	0xE9E9E9E8
	1		35	Data Abort	1022	103	0x01002040
1	2	1	2	Pref. Abort	408	36	0x81000C08
28	1	13	34	Watchdog	5547	552	0x00000000
1	2	1	17	SWI	2681	257	0x0100A754

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
8	1	4	15	Data Abort	5524	528	0x01002810
	1		1	Data Abort	202	21	0x01002810
14	1	8	12	Pref. Abort	4197	396	0xEFEFEFEE
3	2	2	7	Comm Lost			
5	3	4	2	Watchdog	792	82	0x00000000
3	4	3	5	Watchdog	837	74	0x00000000
5	4	4	102	Comm Lost			
	1		3	Pref. Abort	764	63	0x41002518
	1		3	Data Abort	766	59	0x01002358
21	3	13	10	Data Abort	3552	342	0x01002358
2	3	2	5	SWI	348	31	0x00000C0C
8	1	5	8	SWI	1569	141	0x00002740
15	6	10	8	Watchdog	3146	295	0x00000000
	1		2	Watchdog	744	67	0x00000000
1	2	1	5	Data Abort	1471	152	0x01002358
	1		2	Pref. Abort	387	39	0xEFEFEFEE
3	4	3	5	Data Abort	1507	162	0x01001948
20	4	15	12	Comm Lost			
321	1	178	404	Watchdog	14859	1465	0x00000000
	1		7	Comm Lost			
1	2	1	1	Pref. Abort	35	2	0xE9E9E9E8
10	3	7	13	Data Abort	3500	342	0x01001A3C
3	4	3	30	SWI	5597	485	0x00022060
11	9	9	8	SWI	1697	143	0x0261BF54
7	1	5	34	Pref. Abort	11988	1161	0x21002518
14	5	10	10	Comm Lost			
3	1	1	3	Data Abort	926	88	0x01002810
1	2	1	2	Pref. Abort	413	38	0xEFEFEFEE
6	3	5	5	SWI	495	42	0x000025B4
6	5	5	3	Pref. Abort	864	77	0x8864B49E
	1		1	Pref. Abort	370	33	0x05001E54
13	3	9	24	Comm Lost			
4	1	1	7	Data Abort	2314	237	0x01002544
	1		4	Data Abort	415	42	0x01001DBC
42	1	25	23	Comm Lost			

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
2	1	1	4	Pref. Abort	1397	132	0xE1E1E1E0
2	1	1	6	SWI	724	66	0x022D0232
6	7	6	5	Comm Lost			
	1		3	Data Abort	710	89	0x0100197C
34	2	27	29	Comm Lost			
	1		6	Comm Lost			
1	2	1	3	Pref. Abort	773	71	0xA88864B4
3	4	3	2	Pref. Abort	458	41	0xEFEFEFEE

B.2 Experiment 2: Rate 100

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
352	12	339	15	Data Abort	14022	133	0x01001938
145	2	142	4	Watchdog	5060	49	0x00000000
54	55	54	39	SWI	1742	10	0x00002060
423	21	412	12	Data Abort	14497	127	0x01001DE0
275	17	266	10	Pref. Abort	9773	90	0xEFEFEFEE
240	36	230	48	Comm Lost			
251	65	241	11	Data Abort	9966	101	0x01002248
141	21	136	66	Comm Lost			
255	5	245	11	Watchdog	10502	100	0x00000000
83	1	79	5	Watchdog	3606	33	0x00000000
94	39	90	5	Watchdog	4266	44	0x00000000
327	2	308	20	Data Abort	15413	142	0x010025C4
33	25	32	2	Data Abort	1206	12	0x01002724
2	3	2	1	Pref. Abort	60	2	0x81001DE0
58	19	56	3	Pref. Abort	2191	19	0xE1A0C00C
2	3	2	1	Pref. Abort	57	4	0x810025B0
40	2	37	41	SWI	2171	25	0x00002728
21	22	21	22	Data Abort	1823	22	0x01001AD0
371	4	356	53	Comm Lost			
106	38	104	3	Pref. Abort	3344	36	0xE9E9E9E8
104	38	49	111	Pref. Abort	48749	469	0x09002518
	1		78	Data Abort	33006	337	0x01007FE4
43	44	43	1	Data Abort	1158	13	0x01002810
2	3	2	2	Pref. Abort	484	9	0x21000C08
98	12	94	42	Comm Lost			
460	31	440	21	Pref. Abort	18565	192	0xA88864B4
160	10	147	14	Data Abort	5257	52	0x01002724
256	8	246	315	Undef. Instr.	25345	235	0x00001630
8	3	7	39	Comm Lost			
45	18	42	5	Data Abort	2708	26	0x01002358
143	27	140	4	Watchdog	5006	52	0x00000000

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
273	14	256	20	Watchdog	14573	135	0x00000000
189	8	175	16	Pref. Abort	10535	103	0x41000C08
187	21	179	9	Data Abort	7384	80	0x0100258C
245	26	232	95	Data Abort	15318	169	0x01007FE4
260	14	240	21	Data Abort	7581	92	0x0100197C
388	3	368	22	Watchdog	18052	201	0x00000000
64	7	62	16	Watchdog	3291	31	0x00000000
183	35	179	491	Data Abort	27510	288	0x000015AC
18	15	17	4	Data Abort	1694	20	0x01000AD8
136	20	126	49	Comm Lost			
81	5	78	42	Comm Lost			
4	5	4	2	Data Abort	542	4	0x010011F8
282	27	271	13	Data Abort	11540	124	0x01001938
174	6	166	46	Comm Lost			
151	8	143	9	Data Abort	6807	69	0x01007FE4
77	16	74	144	Data Abort	8784	76	0x01002810
56	9	54	3	Pref. Abort	2139	23	0x41000F0C
372	12	353	58	Comm Lost			
46	2	44	4	Data Abort	2332	25	0x0100120C
4	5	4	684	Comm Lost			
738	26	708	68	Comm Lost			
18	19	18	1	Pref. Abort	445	3	0xA88864B4
42	8	38	5	Pref. Abort	2985	30	0x11002744
516	4	488	30	Watchdog	23737	244	0x00000000
118	50	115	5	Watchdog	4466	47	0x00000000
158	13	146	89	Pref. Abort	12392	126	0x81000C08
13	14	13	38	Comm Lost			
48	30	45	4	Pref. Abort	2339	30	0x21000F0C
29	7	28	3	Undef. Instr.	1534	15	0x0100A51C
274	12	266	351	SWI	22234	212	0x00000060
122	32	104	143	Pref. Abort	16230	153	0x21002518
339	9	320	21	Data Abort	16096	157	0x01000CDC
210	6	174	76	SWI	20642	185	0x00002740
264	13	250	44	Watchdog	23676	246	0x00000000
98	20	96	3	Data Abort	2754	21	0x0100120C

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
358	24	345	15	Pref. Abort	14162	132	0x09002518
296	12	284	50	Comm Lost			
152	4	144	47	SWI	7291	59	0x0000251C
224	109	217	46	Comm Lost			
133	9	129	43	Comm Lost			
32	6	29	4	Data Abort	1949	31	0x010025C4
49	39	48	3	Data Abort	2017	15	0x01001A3C
487	14	467	23	Data Abort	19217	209	0x0100258C
46	11	45	3	Pref. Abort	1926	23	0xE91BA830
17	18	17	318	Data Abort	14011	128	0x01002810
213	8	202	13	Watchdog	9882	113	0x00000000
143	73	138	7	Data Abort	5845	56	0x01001938
239	30	234	6	Watchdog	8118	76	0x00000000
144	19	137	8	Pref. Abort	6218	76	0xE9E9E9E8
83	10	78	6	Data Abort	4099	55	0x01007FE4
87	8	84	5	Pref. Abort	3710	35	0x09002518
	1		2	Data Abort	444	5	0x01000CDC
138	19	131	9	Data Abort	6517	64	0x01001948
322	41	311	651	Pref. Abort	25508	237	0xE9E9E9E8
115	2	108	22	Pref. Abort	11492	115	0xEFEFEFEE
374	22	355	36	Data Abort	23293	229	0x0100197C
109	12	104	6	Data Abort	4602	44	0x01002040
230	15	216	15	Data Abort	11043	133	0x0100279A
155	4	149	7	Watchdog	6472	75	0x00000000
92	4	88	43	Comm Lost			
167	49	162	7	Data Abort	6413	53	0x01000CDC
22	3	17	6	Data Abort	2556	19	0x01002810
478	8	455	24	Pref. Abort	20564	209	0xEFEFEFEE
54	5	50	37	Data Abort	16528	171	0x01001EF0
265	17	253	13	Watchdog	10723	112	0x00000000
91	49	90	2	Watchdog	3004	29	0x00000000

B.3 Experiment 3: Rate 1000

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
1303	11	1293	11	Watchdog	35742	38	0x00000000
11305	269	11253	53	Data Abort	270968	271	0x01002040
2157	2	2144	14	Pref. Abort	56617	56	0xE5853000
1557	331	1548	10	Data Abort	40994	45	0x0100279A
3632	186	3609	24	Undef. Instr.	96433	114	0x01008F10
926	77	919	8	Watchdog	25482	28	0x00000000
570	299	568	3	Data Abort	14490	15	0x01002040
1241	2	1234	8	Data Abort	32637	25	0x01001CDC
14992	98	14928	435	Comm Lost			
4441	155	4420	23	Data Abort	115482	124	0x01001ECC
4675	109	4644	32	Watchdog	124737	117	0x00000000
8288	245	8232	427	SWI	221114	229	0x022D0232
321	147	320	29	Pref. Abort	20079	26	0x11000C08
1236	340	1231	7	Watchdog	32142	25	0x00000000
548	165	544	314	Pref. Abort	148020	142	0x09002518
1925	318	1914	13	Data Abort	51088	45	0x0100197C
6061	579	6023	39	Data Abort	160366	184	0x01002040
2719	286	2712	8	Watchdog	68472	61	0x00000000
1479	76	1474	6	Data Abort	37515	38	0x01002E78
6536	828	6510	27	Pref. Abort	167319	158	0xE5853000
3498	79	3484	15	Data Abort	89597	92	0x01002810
104	105	104	2980	Data Abort	80521	82	0x01007FE4
3345	135	3141	582	Comm Lost			
575	133	572	374	Comm Lost			
6245	189	6220	396	Comm Lost			
797	226	792	6	Watchdog	21180	20	0x00000000
547	401	546	3943	Pref. Abort	116170	105	0xE9E9E9E8
1233	343	1228	377	SWI	32064	30	0x0269026E
4086	530	4068	390	SWI	105514	116	0x02000004
3694	86	3678	17	Pref. Abort	95109	82	0xE91BA830
547	401	546	13155	Comm Lost			

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
5973	119	5940	34	Watchdog	156650	160	0x00000000
3123	304	3108	16	Data Abort	81017	76	0x01001AD0
1201	35	1193	379	Comm Lost			
478	19	474	10758	Pref. Abort	286007	277	0x21002E60
4439	216	4408	32	Data Abort	119239	138	0x01001DC4
6775	261	6734	42	Watchdog	179763	181	0x00000000
1534	83	1526	379	SWI	40065	40	0x0285028A
1790	181	1779	12	Data Abort	47412	48	0x01001DB4
999	207	993	7	Watchdog	26832	24	0x00000000
4420	25	4396	395	Comm Lost			
962	184	958	5	Data Abort	23145	30	0x01002880
8800	289	8755	46	Data Abort	229320	222	0x01002040
3147	1	3127	391	Comm Lost			
816	163	808	379	Comm Lost			
9139	85	9095	517	Comm Lost			
4330	147	4314	17	Data Abort	110355	100	0x01001E4C
1858	8	1846	14	Data Abort	49877	51	0x0100197C
2119	42	2106	15	Watchdog	56960	58	0x00000000
695	25	691	5	Data Abort	18311	23	0x01001A3C
5144	155	5113	32	Pref. Abort	135950	152	0x11000C08
3527	78	3508	390	Comm Lost			
2677	9	2666	12	Watchdog	69085	70	0x00000000
8103	205	8070	34	Watchdog	207722	213	0x00000000
2493	184	2479	15	Data Abort	60074	71	0x01002544
4984	132	4956	29	Watchdog	131330	130	0x00000000
757	546	754	374	Comm Lost			
363	364	363	12679	Data Abort	337921	343	0x01002358
1303	11	1293	11	Watchdog	35742	38	0x00000000
149	146	148	3	Data Abort	4432	5	0x0100258C
2277	396	2268	4629	Comm Lost			
7190	120	7155	406	Comm Lost			
188	189	188	1	Pref. Abort	4524	2	0x41001E54
5674	223	5648	28	Data Abort	147090	140	0x0100197C
21081	99	20981	26528	Watchdog	1234208	1213	0x00000000
5488	395	5463	26	Data Abort	141759	154	0x01002458

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
734	443	732	3	Data Abort	18438	20	0x010020EC
438	59	432	7	Data Abort	12962	18	0x01001CDC
816	163	808	379	Comm Lost			
268	97	266	3	Pref. Abort	7252	8	0x41000F0C
964	153	960	6	Data Abort	25187	26	0x0100197C
1620	696	1613	8	Data Abort	41634	35	0x010025C4
75	76	75	1	Pref. Abort	1820	3	0x21000C08
494	495	494	132	Pref. Abort	68536	73	0x09001E54
1985	155	1973	1285	Pref. Abort	83379	85	0xA88864B4
5558	10	5539	20	Pref. Abort	140701	119	0x41001CA0
1108	131	1103	450	Data Abort	220050	222	0x0100197C
3768	107	3745	24	Data Abort	99720	95	0x01002358
1330	69	1324	7	Watchdog	34754	29	0x00000000
5136	369	5109	28	Pref. Abort	134147	130	0x410025A0
504	11	503	2	Data Abort	12525	8	0x01002724
1828	263	1821	2067	Data Abort	102969	108	0x0100258C
1778	68	1773	7	Data Abort	45124	33	0x01001A44
4805	72	4786	418	Comm Lost			
622	328	621	2	Pref. Abort	15331	16	0xEFEFEFEE
4104	13	4080	25	Watchdog	108569	132	0x00000000
4225	59	4199	27	Watchdog	112331	120	0x00000000
215	133	213	717	Watchdog	25958	34	0x00000000
1374	453	1365	11	Data Abort	37054	46	0x010011F8
953	77	948	7	Watchdog	25748	28	0x00000000
316	182	315	1729	Watchdog	52652	58	0x00000000
2268	527	2236	403	Comm Lost			
1219	111	1213	377	Comm Lost			
1398	81	1390	9	Data Abort	36784	34	0x01001DC4
594	119	590	5	Data Abort	15900	18	0x01001D08
6896	300	6856	41	Pref. Abort	321601	309	0xEFEFEFEE
4888	137	4862	27	Watchdog	128182	121	0x00000000

B.4 Experiment 4: Rate 6000

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
4350	617	4348	2225	Comm Lost			
31021	322	30997	25	Data Abort	753756	141	0x01002830
843	380	842	3843	Pref. Abort	112826	16	0x41001E54
9117	186	9108	10	Pref. Abort	222062	29	0xEBFFFFC8
12802	1005	12791	12	Data Abort	311777	59	0x0202071C
13693	12	13685	9788	Data Abort	570592	90	0x010011F8
8702	259	8700	3	Data Abort	209654	28	0x01001DC4
66011	291	65960	52	Pref. Abort	1604413	261	0xE1E1E1E0
71639	1684	71553	8534	Data Abort	1960432	322	0x01002248
22272	1161	22249	37693	Data Abort	1473424	233	0x01002544
2156	1143	2153	4	Data Abort	52977	7	0x01002810
12647	1355	12633	16	Data Abort	309614	64	0x01007FE4
7823	130	7816	8	Watchdog	191005	41	0x00000000
7737	665	7734	4	Data Abort	186906	28	0x01002FB4
7126	214	7122	37182	Comm Lost			
45186	2253	43775	1439	Data Abort	1688092	270	0x01001EF0
25458	2507	25439	20	Watchdog	619081	96	0x00000000
6085	272	6080	45738	Data Abort	1259818	195	0x01002358
2683	202	2680	4	Data Abort	65616	9	0x010025C4
16203	376	16188	16	Pref. Abort	395000	68	0xE9E9E9E8
139	140	139	2	Data Abort	3788	2	0x01000CDC
17777	929	17761	17	Pref. Abort	433325	81	0xEBFFFFC8
16246	2776	16236	11	Watchdog	394360	62	0x00000000
4123	555	4119	2227	Comm Lost			
198	199	198	1	Pref. Abort	4779	1	0xEFEFEFEE
27972	229	27948	25	Pref. Abort	681034	109	0x11002724
2186	1953	2185	2	Watchdog	53304	8	0x00000000
3839	132	3834	6	Data Abort	94170	18	0x01007FE4
14043	1340	14032	13	Watchdog	342332	48	0x00000000
4334	2377	3739	596	Watchdog	354293	58	0x00000000
4334	2377	3739	596	Watchdog	354293	58	0x00000000

continued on next page

continued from previous page

Last OK	first n-OK	good repl.	bad repl.	cause of failure	runtime	# faults	exception address
4734	729	4730	6	Data Abort	126381	22	0x01000CDC
13053	739	13046	9	Data Abort	316539	50	0x01000CDC
13694	615	13688	7	Watchdog	331484	64	0x00000000
13432	182	13423	10	Watchdog	326395	54	0x00000000
14990	1104	14972	19	Data Abort	360112	71	0x01002458
13947	566	13938	11	Data Abort	338997	49	0x010011F8
7223	991	7218	41075	Undef. Instr.	1175842	191	0x0100A51C
32681	688	32657	25	Data Abort	793697	119	0x01002040
22542	2776	22521	22349	Watchdog	1096266	205	0x00000000
56478	300	56426	53	Pref. Abort	1376214	239	0xE1A0C00C
17032	151	17015	14177	Data Abort	763543	139	0x0100197C
1260	1261	1260	1	Data Abort	30258	8	0x0100120C
5669	1842	5666	5	Data Abort	137706	19	0x010011F8
16901	209	16890	12	Pref. Abort	410062	78	0x00C66754
4729	729	4724	6	Data Abort	125981	22	0x01002040
905	679	904	2	Pref. Abort	22140	3	0x0500273C

Appendix C

Results of Application Experiments

C.1 Stack and Globals, Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
28	400	Data Abort	0x2044e7c
46	707	Data Abort	0x204e264
4	60	SIGTRAP	Cyg_Exception_Control:: deliver_exception
4	63	SIGTRAP	Cyg_Exception_Control:: deliver_exception
20	284	SIGTRAP	powtab.72
8	122	Data Abort	0x204c034
11	156	Data Abort	0x2044a44
22	299	SIGTRAP	vprintf
72	1060	Router, mailbox full	mailbox_noncomm
18	250	Prefetch Abort	0x2049ff8
5	76	Watchdog	
30	429	SIGTRAP	0x02001278
2	27	SIGTRAP	Cyg_Exception_Control:: deliver_exception
1	2	Data Abort	0x2049f94
17	236	Data Abort	0x2044e38

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
27	427	Watchdog	
11	176	SIGTRAP	Cyg_Exception_Control:: deliver_exception
11	137	Data Abort	0x204f284
49	739	Data Abort	0x2049fcc
1	25	Watchdog	
6	86	Data Abort	0x204c09c
33	474	SIGTRAP	0x02067c98
9	124	Data Abort	0x2067c34
1	36	Watchdog	
9	137	Data Abort	0x204e280
18	292	Watchdog	
32	474	Data Abort	0x204e23c
4	41	SIGTRAP	Cyg_Exception_Control:: deliver_exception
11	178	SIGTRAP	stack_main
2	13	Data Abort	0x204e23c
9	151	Data Abort	0x204c09c
12	169	Router, mailbox full	mailbox_noncomm
28	422	SIGTRAP	powtab.72
5	69	SIGTRAP	0x02001278
41	589	Data Abort	0x2046dd0
41	679	Watchdog	
34	506	SIGTRAP	Cyg_Scheduler_Implementation:: set_need_reschedule
34	470	Data Abort	0x2046e64
11	160	SIGTRAP	Cyg_Exception_Control:: deliver_exception
17	261	Router, mailbox full	mailbox_wd
5	81	SIGTRAP	0x02001278
62	928	Prefetch Abort	0x204e28c
5	58	Data Abort	0x2046e6c

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
4	58	SIGTRAP	powtab.72
4	82	Watchdog	
13	175	Prefetch Abort	0x204b08c
4	70	Watchdog	
42	602	Watchdog	
29	461	Data Abort	0x202b4c0
1	20	Watchdog	
8	116	Watchdog	
46	693	Data Abort	0x204d198
9	141	Prefetch Abort	0x2047388
31	498	SIGTRAP	Cyg_Counter::add_alarm
11	141	Data Abort	0x204e230
12	178	Prefetch Abort	0x204e29c
23	311	SIGTRAP	0x02001278
7	89	SIGTRAP	cyg_libc_main_stack
35	517	Data Abort	0x204f290
13	212	Watchdog	
23	333	Data Abort	0x202a3c8
15	195	Data Abort	0x204b03c
55	904	Data Abort	0x204d1c0
34	491	Prefetch Abort	0x204d198
17	255	SIGTRAP	0x02001278
67	934	Router, mailbox full	mailbox_wd
2	32	Data Abort	0x204f238
15	212	SIGTRAP	Cyg_Scheduler_Implementation:: set_need_reschedule
11	158	SIGTRAP	0x02001278
8	111	Watchdog	
1	30	Watchdog	
37	554	Data Abort	0x2049f98
8	93	Prefetch Abort	0x204e23c
23	332	Data Abort	0x2049fdc

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
3	31	Data Abort	0x202aec8
23	377	Data Abort	0x2049f98
1	37	Watchdog	
34	477	SIGTRAP	0x02001278
1	36	Watchdog	
4	41	Data Abort	0x2044e24
15	218	Prefetch Abort	0x204bfd8
16	225	Data Abort	0x204c05c
30	427	SIGTRAP	0x02001278
31	436	SIGTRAP	0x02001278
67	999	SIGTRAP	–
9	135	SIGTRAP	0x02001278
8	138	Watchdog	
1	18	Watchdog	
15	205	SIGTRAP	powtab.72
1	18	SIGTRAP	0x02001278
17	256	Data Abort	0x2044e98
11	143	Data Abort	0x2044e60
4	49	Data Abort	0x204b08c
18	254	Data Abort	0x2046e64
2	12	Data Abort	0x2044e60

C.2 Stack and Globals, Error-correction, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
5	11	Data Abort	0x204b09c
11	40	Watchdog	
92	269	SIGTRAP	0x00000314
18	60	Watchdog	
162	511	SIGTRAP	handle_IRQ_or_FIQ
301	902	SIGTRAP	0x00084314
193	588	Data Abort	0x204f200
1	9	Watchdog	
106	341	Data Abort	0x204d18c
144	464	SIGTRAP	0x00084314
190	600	SIGTRAP	–
3	7	SIGTRAP	__cygvar_discard_me__.348
136	397	SIGTRAP	0x00000314
137	419	SIGTRAP	0x02001278
376	1150	Data Abort	0x204f228
129	434	Prefetch Abort	0x204d1c0
51	140	Data Abort	0x2046e64
362	1102	Watchdog	
9	24	Data Abort	0x204d198
93	295	Watchdog	
120	348	Router, mailbox full	mailbox_wd
369	1105	Watchdog	
19	59	Data Abort	0x2046e6c
105	334	SIGTRAP	0x02067cbc
253	766	SIGTRAP	handle_IRQ_or_FIQ
1	5	Watchdog	
18	57	Data Abort	0x204f290
69	207	Data Abort	0x2049fa4
595	1833	SIGTRAP	0xfffffffe
47	146	SIGTRAP	0x02001278
32	95	Data Abort	0x204b03c

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
65	208	SIGTRAP	0x00084314
15	38	Data Abort	0x204d1dc
49	142	Data Abort	0x204b08c
25	61	Prefetch Abort	0x204b03c
242	702	SIGTRAP	0x02001278
122	391	Data Abort	0x204e264
14	36	SIGTRAP	powtab.72
149	453	SIGTRAP	0x00008880
1	6	Watchdog	
45	147	SIGTRAP	0x02001278
13	33	SIGTRAP	0x03055674
194	585	Data Abort	0x2049f98
66	226	Data Abort	0x204c034
16	47	Watchdog	
252	791	SIGTRAP	0x00000314
50	171	Data Abort	0x204d198
24	65	SIGTRAP	hk_store
98	296	SIGTRAP	0x00084314
43	144	Prefetch Abort	0x204b08c
93	270	Data Abort	0x2049f78
39	123	Data Abort	0x204d1dc
44	140	SIGTRAP	0x00084314
124	376	Watchdog	
638	2011	SIGTRAP	0x00084314
34	97	Data Abort	0x2049f98
7	18	Watchdog	
52	159	Watchdog	
76	226	Router, mailbox full	mailbox_wd
95	320	Data Abort	0x204d198
32	91	Watchdog	
271	901	Watchdog	
41	132	SIGTRAP	0x02001278

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
34	122	Watchdog	
2	10	SIGTRAP	0x00084314
164	501	Data Abort	0x2049fdc
126	408	Watchdog	
74	212	Unknown Message	in
62	187	Prefetch Abort	0x204b03c
4	16	Data Abort	0x2046ea4
59	184	SIGTRAP	powtab.72
137	420	SIGTRAP	0x02001278
131	433	Watchdog	
23	83	SIGTRAP	0x03050a10
78	222	Watchdog	
191	619	Data Abort	0x204c08c
82	260	SIGTRAP	Cyg_Scheduler_Implementation:: set_need_reschedule
45	145	Prefetch Abort	0x204c09c
5	16	SIGTRAP	powtab.72
14	56	Data Abort	0x204d1c4
61	199	Data Abort	0x204f260
10	39	Watchdog	
72	238	Prefetch Abort	0x2047388
200	660	SIGTRAP	0x00000314
74	222	Data Abort	0x204d198
199	609	SIGTRAP	0x00084314
133	428	SIGTRAP	0x00000314
117	378	Router, mailbox full	mailbox_noncomm
107	321	SIGTRAP	0x00084314
69	226	Prefetch Abort	0x204f2a0
91	297	SIGTRAP	0x00000314
147	437	Data Abort	0x2046e54
61	196	Data Abort	0x2049fc0
60	185	Data Abort	0x2046e6c

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
50	159	Data Abort	0x2044df0
1	8	Watchdog	
12	35	SIGTRAP	0x00000314
62	199	SIGTRAP	0x0302b59c
43	112	Data Abort	0x2046e64

C.3 Stack and Globals, No Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
25	371	Prefetch Abort	0x2044e88
10	139	SIGTRAP	0x00084314
9	146	Data Abort	0x204d13c
3	32	SIGTRAP	0x00000314
8	114	Data Abort	0x2049fdc
10	176	Watchdog	
1	21	Watchdog	
11	162	SIGTRAP	__cygvar_discard_me__.348
7	108	Router, mailbox full	mailbox_noncomm
12	176	Data Abort	0x204f260
3	34	Data Abort	0x204e264
3	42	Data Abort	0x204de58
1	26	Watchdog	
1	30	Watchdog	
23	322	Prefetch Abort	0x2044e88
3	45	SIGTRAP	0xfffffffc
3	47	Data Abort	0x204c034
10	138	Unknown Message	Non comm
10	131	SIGTRAP	powtab.72
3	64	Watchdog	
10	154	Data Abort	0x2046e94
5	92	Watchdog	
36	501	Unknown Message	in
26	396	Watchdog	
2	22	Bad handle	edac_lock
25	406	Data Abort	0x204f290
5	66	Data Abort	0x2044e74
9	124	SIGTRAP	0x02001278
13	198	Data Abort	0xfffb0
34	520	SIGTRAP	0x02001278
17	247	Data Abort	0x2046e3c

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
6	89	SIGTRAP	0x00000314
4	60	SIGTRAP	0x02001278
6	90	Watchdog	
48	731	Prefetch Abort	0x204e29c
8	144	Watchdog	
23	371	Watchdog	
2	17	Data Abort	0x204c05c
3	43	Data Abort	0x2049f98
9	138	Router, mailbox full	mailbox_noncomm
1	11	SIGTRAP	0x00000314
13	192	Data Abort	0x2046e64
4	86	SIGTRAP	0x02001278
13	185	Data Abort	0x204e23c
5	65	Data Abort	0x204e264
24	360	SIGTRAP	hk_store
4	47	Data Abort	0x204e2b8
11	186	Data Abort	0x204d1d4
14	219	Data Abort	0x204d1dc
15	239	Watchdog	
30	452	Router, mailbox full	mailbox_wd
7	81	Data Abort	0x2044e5c
19	279	Watchdog	
8	94	Data Abort	0x204b080
57	873	Data Abort	0x2044e74
4	53	Data Abort	0x2044e2c
13	185	SIGTRAP	interrupt_end
14	208	Router, copy failed	mailbox_in
2	22	SIGTRAP	vprintf
2	21	Data Abort	0x2046ea4
24	372	SIGTRAP	0x00008880
7	112	Data Abort	0x204c00c
14	194	SIGTRAP	0x02001278

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
15	229	Watchdog	
2	18	SIGTRAP	0x00084314
4	49	Data Abort	0x2049fc0
50	731	SIGTRAP	0x02001278
11	162	SIGTRAP	0x02001278
10	133	Prefetch Abort	0x204b03c
17	229	SIGTRAP	0x02001278
26	412	Router, mailbox full	mailbox_wd
7	98	SIGTRAP	__cygvar_discard_me__.348
38	514	Unknown Message	WD
14	200	SIGTRAP	0x00084314
31	463	SIGTRAP	powtab.72
38	548	Prefetch Abort	0x2046e94
5	96	Watchdog	
19	277	SIGTRAP	0x00084314
22	337	Data Abort	0x204c05c
8	105	SIGTRAP	–
7	115	SIGTRAP	powtab.72
4	71	Watchdog	
40	605	SIGTRAP	0x00084314
67	958	Data Abort	0x2046e3c
11	162	Prefetch Abort	0x204d1e8
9	110	SIGTRAP	0x02001278
13	188	Data Abort	0x204e270
14	186	Data Abort	0x2044e34
19	291	SIGTRAP	powtab.72
2	28	Data Abort	0x204c078
2	29	SIGTRAP	0x00084314
4	57	SIGTRAP	Cyg_Exception_Control:: deliver_exception
21	333	Router, mailbox full	mailbox_noncomm
7	106	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
30	441	Router, mailbox full	mailbox_noncomm
18	252	Data Abort	0x2049fd4
5	83	SIGTRAP	Cyg_Counter::add_alarm
18	246	SIGTRAP	0x00084314
4	73	Watchdog	

C.4 Stack and Globals, No Error-correction, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
73	220	Data Abort	0x2046e64
414	1350	Data Abort	0x204d1d4
108	343	SIGTRAP	stack_main
234	731	Watchdog	
88	278	Router, copy failed	mailbox_in
63	174	SIGTRAP	Cyg_Counter::add_alarm
61	211	Unknown Message	WD
33	97	SIGTRAP	0x0206c9d4
9	34	SIGTRAP	0x00000314
5	18	Data Abort	0x204d1b0
10	33	Unknown Message	out
13	31	Prefetch Abort	0x204f2a0
3	11	Data Abort	0x204c034
33	107	Router, mailbox full	mailbox_noncomm
91	290	Data Abort	0x2046e3c
104	355	Unknown Message	in
51	164	Router, mailbox full	mailbox_inout
426	1278	Data Abort	0x204c05c
14	34	SIGTRAP	0x02001278
115	373	Data Abort	0x204c08c
182	581	Data Abort	0x204e248
23	83	Data Abort	0x203cbb0
27	85	Data Abort	0x204e23c
3	6	Router, copy failed	mailbox_out
131	407	Data Abort	0x2049f98
25	88	Data Abort	0x204e23c
106	297	Unknown Message	in
41	142	SIGTRAP	0x00084314
14	42	Router, mailbox full	mailbox_out
30	90	SIGTRAP	0x00084314
93	344	Prefetch Abort	0x204e23c

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
35	112	Data Abort	0x2046e64
69	225	Data Abort	0x2046e64
56	207	Unknown Message	WD
156	464	Data Abort	0x204d194
93	284	Watchdog	
60	162	Data Abort	0x2046e3c
59	174	Data Abort	0x2046e54
60	189	Data Abort	0x204f260
52	186	SIGTRAP	0x02001278
72	204	SIGTRAP	0x02001278
2	4	Data Abort	0x204c09c
124	404	Unknown Message	in
13	28	SIGTRAP	0x02001278
53	172	SIGTRAP	exception_handler
50	153	Data Abort	0x2044e38
243	755	Data Abort	0x204c05c
34	87	Watchdog	
14	56	Watchdog	
11	31	Data Abort	0x204e258
76	234	SIGTRAP	–
259	776	SIGTRAP	0x02001278
59	199	Prefetch Abort	0x2044e38
460	1413	SIGTRAP	0x02001278
138	446	Prefetch Abort	0x204f2a0
31	94	SIGTRAP	–
75	244	Unknown Message	WD
34	88	Data Abort	0x2044e34
26	85	Data Abort	0x2049f9c
49	155	Watchdog	
56	172	SIGTRAP	0x00000150
35	104	Watchdog	
5	14	Unknown Message	inout

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
23	80	Prefetch Abort	0x204a014
42	115	SIGTRAP	powtab.72
58	180	SIGTRAP	0x00000150
35	114	Unknown Message	WD
78	242	Watchdog	
49	153	Data Abort	0x204e278
59	190	Watchdog	
51	175	Data Abort	0x2049fcc
5	13	SIGTRAP	0x02001278
79	242	SIGTRAP	0x00000150
17	60	Watchdog	
100	309	Data Abort	0x204c034
25	73	Data Abort	0x204c08c
71	233	Watchdog	
112	341	Watchdog	
51	160	Data Abort	0x204e264
65	217	SIGTRAP	0x00000150
124	379	SIGTRAP	0x02001278
112	346	Data Abort	0x204b064
74	231	SIGTRAP	0x00000150
73	205	Watchdog	
87	277	Data Abort	0x204cda4
262	799	SIGTRAP	0x02001278
97	305	SIGTRAP	Cyg_Exception_Control:: deliver_exception
98	311	SIGTRAP	0x00000150
58	180	Data Abort	0x204e258
68	206	Data Abort	0x204e278
2	3	SIGTRAP	0x00000150
62	234	Data Abort	0x204d1d4
17	52	SIGTRAP	0x02001278
42	122	Data Abort	0x204e29c

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
31	89	Data Abort	0x2044e38
18	58	Data Abort	0x204b064
118	327	Router, mailbox full	mailbox_noncomm
20	62	Prefetch Abort	0x2044e60

C.5 Dummy, Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
1681	22541	Time-limit	
1680	21869	Time-limit	
1681	22058	Time-limit	
1681	22048	Time-limit	
1681	22366	Time-limit	
1680	22113	Time-limit	
1680	22354	Time-limit	
1681	22358	Time-limit	
1681	22001	Time-limit	
1681	22265	Time-limit	
1680	22335	Time-limit	
1680	22161	Time-limit	

C.6 Operating System, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
11	4	Router, mailbox full	mailbox_wd
22	4	Data Abort	0x2046e44
3	1	Watchdog	
9	5	Prefetch Abort	0x204d184
50	17	Data Abort	0x204f1e8
20	11	Watchdog	
64	15	SIGTRAP	0x020aa62c
11	3	Data Abort	0x203cb60
54	17	Watchdog	
5	1	Data Abort	0x20649c4
6	1	Data Abort	0x203cbfc
85	29	Data Abort	0x203cbfc
25	8	Watchdog	
38	9	Watchdog	
16	7	Router, mailbox full	mailbox_noncomm
69	20	Watchdog	
7	2	Data Abort	0x2046e44
3	3	Data Abort	0x2049f9c
9	5	Data Abort	0x203cb70
16	6	Router, mailbox full	mailbox_wd
10	3	SIGTRAP	Cyg_Exception_Control:: deliver_exception
9	1	Router, mailbox full	mailbox_noncomm
76	24	Router, mailbox full	mailbox_noncomm
63	15	SIGTRAP	Cyg_Exception_Control:: deliver_exception
7	3	Data Abort	0x2046e44
27	9	Data Abort	0x203cb9c
1	1	SIGTRAP	0x02001278
22	8	Router, mailbox full	mailbox_wd
43	20	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
3	1	Data Abort	0x204c01c
16	6	Prefetch Abort	0x203cbb0
43	13	Watchdog	
12	5	Watchdog	
2	3	Data Abort	0x204c01c
143	37	Data Abort	0x204b028
2	3	Data Abort	0x204b030
8	2	Data Abort	0x2049fb4
7	5	Watchdog	
11	4	SIGTRAP	0x02001278
11	1	SIGTRAP	Cyg_Exception_Control:: deliver_exception
3	1	SIGTRAP	0x02001278
21	4	Data Abort	0x2046e24
8	2	SIGTRAP	0x02001278
13	3	Watchdog	
41	9	Data Abort	0x2046e44
13	5	Data Abort	0x203cb9c
47	14	Watchdog	
19	6	Router, mailbox full	mailbox_noncomm
2	1	Data Abort	0x204c03c
23	8	Watchdog	
31	15	Watchdog	
30	7	Router, mailbox full	mailbox_noncomm
12	3	Prefetch Abort	0x203cbb0
68	18	Data Abort	0x2046e44
19	5	Router, mailbox full	mailbox_noncomm
62	14	Data Abort	0x2046e60
32	12	Watchdog	
40	14	Data Abort	0x2046e44
15	6	Watchdog	
35	11	SIGTRAP	powtab.72

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
14	1	Data Abort	0x2044e3c
37	10	Watchdog	
41	19	SIGTRAP	0x02001278
5	3	Data Abort	0x204c01c
124	43	Watchdog	
20	8	Router, mailbox full	mailbox_wd
9	3	Router, mailbox full	mailbox_noncomm
5	5	Watchdog	
4	2	Data Abort	0x203cb5c
44	16	Data Abort	0x2046e44
21	8	Watchdog	
8	3	SIGTRAP	Cyg_Exception_Control:: deliver_exception
11	3	Data Abort	0x204cd94
88	27	Data Abort	0x2046e44
33	11	Watchdog	
28	5	SIGTRAP	–
46	13	Router, mailbox full	mailbox_wd
12	5	Watchdog	
37	10	Watchdog	
61	17	Data Abort	0x204c048
6	1	Data Abort	0x204b040
6	1	Router, mailbox full	mailbox_wd
36	10	Router, mailbox full	mailbox_noncomm
17	8	Watchdog	
70	27	Watchdog	
49	12	SIGTRAP	Cyg_Exception_Control:: deliver_exception
24	7	Watchdog	
12	5	Router, mailbox full	mailbox_noncomm
44	12	Data Abort	0x2053260
5	4	Router, mailbox full	mailbox_wd

continued on next page

continued from previous page

# Watchdog resets	**# Injections**	**Cause of failure**	**exception address**
5	2	Watchdog	
49	11	SIGTRAP	0x02001278
79	29	SIGTRAP	Cyg_Exception_Control:: deliver_exception
56	14	Watchdog	
31	5	Data Abort	0x2046e44
43	10	SIGTRAP	0x02001278
9	4	Data Abort	0x204c03c
5	1	Watchdog	
28	7	SIGTRAP	mt

C.7 Operating System, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
53	6	SIGTRAP	0x00000150
98	2	Data Abort	0x2046e44
66	5	Data Abort	0x20649c4
874	43	Data Abort	0x2046dd8
71	3	Data Abort	0x204c03c
9	1	Watchdog	
43	2	Data Abort	0x2046e24
44	3	Data Abort	0x2046e24
207	22	Data Abort	0x2044e2c
21	2	Data Abort	0x2046dd8
11	2	Data Abort	0x204d148
308	20	SIGTRAP	0x02001278
91	5	Data Abort	0x204f24c
148	9	Watchdog	
264	13	SIGTRAP	Cyg_Exception_Control:: deliver_exception
300	11	Watchdog	
189	10	SIGTRAP	0x02001278
108	10	Watchdog	
61	4	SIGTRAP	0x00000150
252	13	Data Abort	0x204b040
50	6	Watchdog	
258	16	Watchdog	
272	24	SIGTRAP	0x02001278
274	13	Watchdog	
17	5	Watchdog	
29	2	SIGTRAP	0x02001278
10	1	Watchdog	
280	17	Data Abort	0x203cb70
99	6	Watchdog	
41	3	Prefetch Abort	0x203cbb0

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
93	4	Router, mailbox full	mailbox_noncomm
85	4	Data Abort	0x2046e50
169	9	SIGTRAP	Cyg_Exception_Control:: deliver_exception
266	17	Watchdog	
87	4	Watchdog	
277	15	Watchdog	
19	2	Watchdog	
10	2	Watchdog	
57	6	Data Abort	0x203cbfc
12	2	Router, mailbox full	mailbox_wd
279	19	Watchdog	
147	14	Watchdog	
28	3	Prefetch Abort	0x203cbb0
42	3	Watchdog	
17	4	Data Abort	0x2046e44
222	16	Router, mailbox full	mailbox_noncomm
102	5	Data Abort	0x204c03c
110	11	Watchdog	
157	16	Watchdog	
153	12	SIGTRAP	Cyg_Exception_Control:: deliver_exception
192	16	Data Abort	0x2044e5c
372	15	SIGTRAP	0x00000150
113	8	Watchdog	
5	1	SIGTRAP	Cyg_Exception_Control:: deliver_exception
166	7	Watchdog	
27	2	Data Abort	0x204f24c
135	7	Router, mailbox full	mailbox_noncomm
515	35	Data Abort	0x203cb60
194	13	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
126	6	SIGTRAP	Cyg_Exception_Control:: deliver_exception
117	4	SIGTRAP	0x00000150
63	10	Data Abort	0x204ac38
25	2	Data Abort	0x203cb70
160	10	Watchdog	
284	12	SIGTRAP	0x02001278
13	1	Data Abort	0x204c048
304	17	Router, mailbox full	mailbox_wd
72	7	Data Abort	0x204e240
128	10	Data Abort	0x204f1e8
131	10	Data Abort	0x2046e50
69	6	Router, mailbox full	mailbox_noncomm
185	12	SIGTRAP	0x00000150
245	15	Watchdog	
199	9	Data Abort	0x203cb60
59	3	Watchdog	
203	14	Watchdog	
56	2	Watchdog	
67	2	Router, mailbox full	mailbox_wd
70	2	Router, mailbox full	mailbox_wd
29	2	Data Abort	0x2064a80
21	3	Data Abort	0x2046e44
134	5	Data Abort	0x203cb9c
172	13	SIGTRAP	Cyg_Exception_Control:: deliver_exception
393	26	Data Abort	0x203cb60
36	3	Data Abort	0x2046e24
249	15	Watchdog	
63	9	Data Abort	0x203cb9c
578	30	SIGTRAP	hk_store
433	19	SIGTRAP	Cyg_Exception_Control::

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
			deliver_exception
60	4	Data Abort	0x2046e24
327	23	SIGTRAP	0x00000150
2	1	Data Abort	0x204b054
484	27	Data Abort	0x2049ba0
279	12	Watchdog	
32	3	Watchdog	
31	2	SIGTRAP	Cyg_Exception_Control:: deliver_exception
25	1	Data Abort	0x2046e60

C.8 Globals, Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
126	908	Watchdog	
78	492	Data Abort	0x2046e6c
1681	11494	Time-limit	
453	3241	Data Abort	0x204c064
370	2699	SIGTRAP	0x0207f698
350	2471	Watchdog	
266	1836	Data Abort	0x2046e6c
1680	11712	Time-limit	
1420	9930	SIGTRAP	0x02001278
416	2786	Data Abort	0x2046e6c
334	2418	SIGTRAP	0x02001278
382	2629	Unknown Message	Non comm
796	5486	SIGTRAP	0x02001278
8	48	Data Abort	0x2046e6c
257	1791	Data Abort	0x2046e6c
1004	7032	Watchdog	
290	1965	Data Abort	0x204c064
1527	10667	Unknown Message	Non comm
272	1984	Watchdog	
1680	11824	Time-limit	
288	2054	Unknown Message	out
487	3255	Watchdog	
181	1228	SIGTRAP	0x0207e29c
1681	11694	Time-limit	
136	973	Unknown Message	inout
134	908	Watchdog	
1016	6996	Unknown Message	inout
9	67	Watchdog	
444	3087	Watchdog	
699	4902	Data Abort	0x2046e6c
1180	8330	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
1122	7855	Data Abort	0x2046e6c
1150	8048	SIGTRAP	0x02001278
312	2185	Watchdog	
109	748	Data Abort	0x2046e6c
259	1820	Data Abort	0x204c064
120	855	Data Abort	0x2046e6c
1158	8067	Data Abort	0x2046e6c
1680	11784	Time-limit	
927	6525	Unknown Message	WD
1680	11843	Time-limit	
831	5735	Data Abort	0x2046e6c
52	338	Data Abort	0x2046e6c
1420	9958	SIGTRAP	0x02001278
1304	9200	Watchdog	
1020	7098	Watchdog	
1331	9312	Data Abort	0x2046e6c
63	486	Watchdog	
519	3721	SIGTRAP	0x02001278
215	1493	Watchdog	
197	1419	Watchdog	
139	942	SIGTRAP	0x02001278
838	5952	Data Abort	0x2049bbc
1561	10660	Unknown Message	in
1567	10954	Watchdog	
1636	11362	Watchdog	
334	2290	Watchdog	
411	2862	Watchdog	
1680	11744	Time-limit	
74	520	Data Abort	0x2044e38
457	3296	Data Abort	0x2046e6c
57	421	Watchdog	
123	889	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
325	2251	SIGTRAP	0x02001278
518	3532	Watchdog	
228	1542	Watchdog	
415	2761	Data Abort	0x204c064
697	4856	Data Abort	0x204c064
192	1386	Data Abort	0x204c064
318	2195	Watchdog	
133	985	Data Abort	0x2046e6c
23	154	Unknown Message	in
598	4298	Watchdog	
503	3383	Watchdog	
437	3070	Watchdog	
1226	8674	Watchdog	
1680	11591	Time-limit	
1369	9601	Data Abort	0x204c064
344	2366	Unknown Message	Non comm
1	17	Watchdog	
1256	8835	Watchdog	
205	1410	Data Abort	0x2049fb4
141	1035	Data Abort	0x204f268
777	5431	Data Abort	0x204c064
1345	8259	Watchdog	
214	1449	Data Abort	0x204c064
1063	7387	Watchdog	
311	2077	Watchdog	
457	3158	Watchdog	
272	1892	Watchdog	
621	4408	Unknown Message	in
1681	11519	Time-limit	
575	3899	SIGTRAP	vfnprintf
676	4602	SIGTRAP	0x02001278
893	6062	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
233	1595	Unknown Message	WD
133	852	Data Abort	0x2046e6c

C.9 Globals, Error-correction, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
1679	2370	Time-limit	
1315	1947	Watchdog	
1399	1985	Watchdog	
1680	2391	Time-limit	
807	1188	Watchdog	
1679	2383	Time-limit	
1500	2147	Data Abort	0x204c058
1680	2357	Time-limit	
1679	2374	Time-limit	
897	1253	Data Abort	0x204c064
1680	2419	Time-limit	
816	1129	Unknown Message	in
1681	2332	Time-limit	
713	1024	Data Abort	0x204c064
1680	2337	Time-limit	
1680	2367	Time-limit	
1679	2433	Time-limit	
1680	2388	Time-limit	
1680	2296	Time-limit	
1679	2347	Time-limit	
630	781	Watchdog	
3475	5014	Data Abort	0x204c02c
3672	5215	Data Abort	0x2046e5c
4146	5893	Watchdog	
5596	7839	Time-limit	
1412	2061	Watchdog	
2337	3221	Data Abort	0x2046e5c
1131	1572	Unknown Message	inout
2323	3297	Watchdog	
4731	6748	Unknown Message	inout
183	283	Unknown Message	inout

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
1285	1917	Watchdog	
1044	1470	Data Abort	0x204c054
1425	2025	Watchdog	

C.10 Globals, Error-correction, High Limit, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
4333	6201	Data Abort	0x204c038
852	1152	Data Abort	0x204c054
1751	2398	SIGTRAP	0x02001278
1799	2570	Data Abort	0x2046e5c
145	219	Data Abort	0x2046e70
2431	3509	Unknown Message	in
548	778	Data Abort	0x204c038
196	286	Watchdog	
211	273	Unknown Message	Non comm
5593	7936	SIGTRAP	0x02001278
3535	5182	SIGTRAP	0x02001278
5596	7802	Time-limit	
2045	2827	Data Abort	0x2046e5c
2649	3684	Unknown Message	WD
2558	3660	Watchdog	
261	328	Unknown Message	inout
5595	7923	Time-limit	
4056	5856	Unknown Message	inout
3841	5424	Unknown Message	inout
1024	1458	Unknown Message	in
1265	1794	SIGTRAP	–
2479	3472	Data Abort	0x204d1b4
2244	3161	Data Abort	0x2046e5c
909	1238	Data Abort	0x204c054
1163	1708	Watchdog	
5595	7899	Time-limit	
5595	7999	Time-limit	
1316	1853	Data Abort	0x204c054
1038	1399	Data Abort	0x2046e5c
5597	7926	Time-limit	
5597	8014	Time-limit	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
5416	7808	Data Abort	0x204c054
3957	5582	Data Abort	0x2046e5c
5596	7996	Time-limit	
2304	3257	Watchdog	
3219	4568	Data Abort	0x2044e28
5591	7832	Time-limit	
3232	4565	Data Abort	0x2049fb4
1	2	Watchdog	
145	217	Data Abort	0x2049b7c
1049	1470	Watchdog	
247	333	Unknown Message	in
355	477	Data Abort	0x2046e2c
1426	2038	Unknown Message	inout
5595	7909	Time-limit	
2546	3541	Watchdog	
2406	3337	Unknown Message	inout
1892	2692	Unknown Message	inout
2189	3161	Watchdog	
260	407	Watchdog	

C.11 Globals, No Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
45	322	Watchdog	
41	305	Unknown Message	WD
45	346	Watchdog	
91	623	Watchdog	
81	587	Unknown Message	Non comm
38	246	Data Abort	0x2046e64
7	57	Watchdog	
135	954	Watchdog	
32	234	Data Abort	0x2046e64
25	161	Unknown Message	Non comm
204	1461	Router, mailbox full	mailbox_in
61	411	Unknown Message	in
102	711	Bad handle	edac_unlock
26	200	Watchdog	
121	784	Watchdog	
41	294	Watchdog	
180	1281	Bad handle	edac_lock
144	1018	Router, mailbox full	mailbox_noncomm
10	75	Watchdog	
51	369	Data Abort	0x2046e64
392	2730	Watchdog	
355	2471	Watchdog	
112	816	Router, mailbox full	mailbox_inout
168	1216	Data Abort	0x2046e64
379	2752	Watchdog	
143	971	Watchdog	
97	679	Router, mailbox full	mailbox_inout
22	160	Watchdog	
54	365	Unknown Message	out
30	220	Watchdog	
201	1394	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
2	9	Router, mailbox full	mailbox_out
27	197	Watchdog	
1	4	Bad handle	edac_lock
61	418	Data Abort	0x2044e34
15	98	Watchdog	
18	129	Watchdog	
29	194	Router, mailbox full	mailbox_wd
129	877	Unknown Message	Non comm
9	57	Data Abort	0x204f260
336	2462	Watchdog	
127	891	Watchdog	
86	644	Watchdog	
88	610	Watchdog	
197	1409	Watchdog	
55	381	Data Abort	0x2044e24
39	301	Watchdog	
194	1406	Watchdog	
30	206	Unknown Message	in
129	894	Watchdog	
146	1022	Watchdog	
95	669	Watchdog	
149	1042	Watchdog	
25	168	Data Abort	0x2046e3c
19	121	Bad handle	edac_unlock
234	1685	Watchdog	
136	921	Watchdog	
89	624	Router, mailbox full	mailbox_in
42	281	Router, mailbox full	mailbox_wd
21	144	Watchdog	
20	126	Unknown Message	Non comm
7	48	Unknown Message	Non comm
180	1228	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
26	183	Unknown Message	inout
163	1154	Data Abort	0x2049ff8
1	3	Unknown Message	Non comm
161	1112	Watchdog	
55	410	Router, mailbox full	mailbox_wd
118	817	Unknown Message	out
215	1483	Watchdog	
189	1259	Watchdog	
26	209	Watchdog	
26	160	Unknown Message	in
9	75	Watchdog	
238	1631	Watchdog	
151	1087	Unknown Message	in
25	182	Router, mailbox full	mailbox_out
22	134	Unknown Message	Non comm
15	100	Data Abort	0x204c05c
100	711	Data Abort	0x2046e3c
61	450	Unknown Message	inout
66	439	Unknown Message	inout
113	792	Watchdog	
12	82	Watchdog	
185	1269	Watchdog	
7	59	Watchdog	
39	267	Unknown Message	Non comm
165	1165	Bad handle	edac_unlock
122	899	Watchdog	
222	1526	Unknown Message	Non comm
83	604	Data Abort	0x2046e3c
207	1377	Unknown Message	WD
5	30	Data Abort	0x2046e64
103	736	Router, mailbox full	mailbox_out
125	939	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
160	1062	Router, mailbox full	mailbox_inout
171	1178	Watchdog	
100	678	Prefetch Abort	0x2046e74
44	318	Watchdog	
53	355	Bad handle	edac_lock

C.12 Globals, No Error-correction, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
431	622	Watchdog	
691	951	Watchdog	
1143	1703	Watchdog	
626	893	Data Abort	0x2046e64
266	373	Data Abort	0x2046e64
172	245	Watchdog	
237	345	Watchdog	
272	382	Unknown Message	out
399	555	Unknown Message	inout
258	365	Data Abort	0x204c05c
102	157	Data Abort	0x2044e5c
481	638	Watchdog	
360	517	Bad handle	edac_lock
1150	1592	Watchdog	
222	292	Watchdog	
122	163	Router, mailbox full	mailbox_in
67	97	Watchdog	
2	4	Bad handle	edac_lock
160	242	Watchdog	
1001	1387	Unknown Message	inout
113	169	Router, mailbox full	mailbox_out
346	480	Unknown Message	out
320	466	Watchdog	
1089	1441	Watchdog	
17	27	Watchdog	
476	686	Unknown Message	Non comm
106	144	Unknown Message	inout
343	491	Watchdog	
34	56	Watchdog	
246	346	Unknown Message	in
145	189	Watchdog	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
2110	3004	Unknown Message	out
119	169	Watchdog	
433	615	Data Abort	0x2046e2c
255	352	Unknown Message	in
1761	2481	Unknown Message	inout
570	819	Router, mailbox full	mailbox_noncomm
97	128	Watchdog	
306	442	Watchdog	
381	575	Watchdog	
788	1106	Watchdog	
566	784	Unknown Message	WD
264	358	Router, mailbox full	mailbox_inout
9	13	Watchdog	
1168	1706	Watchdog	
2621	3558	Data Abort	0x204c04c
25	34	Router, mailbox full	mailbox_inout
100	128	Data Abort	0x2046e2c
561	811	Watchdog	

C.13 Stacks, Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
25	197	Data Abort	0x2044e60
2	13	Data Abort	0x2044e60
12	95	SIGTRAP	0x032e2474
23	210	Data Abort	0x2046e64
12	107	Router, mailbox full	mailbox_wd
93	763	SIGTRAP	0x02069a10
19	156	Data Abort	0x2044e7c
34	269	Data Abort	0x2044e00
3	16	SIGTRAP	Cyg_Counter::add_alarm
36	291	SIGTRAP	cyg_libc_main_stack
7	48	Prefetch Abort	0x204c09c
4	22	SIGTRAP	powtab.72
1	17	Watchdog	
6	46	SIGTRAP	Cyg_Exception_Control:: deliver_exception
28	236	SIGTRAP	Cyg_Exception_Control:: deliver_exception
21	171	SIGTRAP	0x020ab5a0
2	5	SIGTRAP	Cyg_Exception_Control:: deliver_exception
88	695	SIGTRAP	0x02001278
1	19	Watchdog	
30	240	Data Abort	0x204cdac
16	115	Data Abort	0x2049f98
2	3	Data Abort	0x2049fcc
33	292	Watchdog	
1	11	Watchdog	
7	57	Router, mailbox full	mailbox_noncomm
12	94	SIGTRAP	powtab.72
41	339	Data Abort	0x204d644
30	223	Router, mailbox full	mailbox_noncomm

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
26	222	SIGTRAP	Cyg_Exception_Control:: deliver_exception
4	38	Prefetch Abort	0x2046e64
5	48	SIGTRAP	Cyg_Exception_Control:: deliver_exception
25	207	Router, mailbox full	mailbox_noncomm
69	581	Watchdog	
1	14	Watchdog	
7	50	Data Abort	0x204b03c
1	3	Data Abort	0x2049ff8
23	181	SIGTRAP	0x02001278
16	138	Data Abort	0x204b03c
5	38	SIGTRAP	Cyg_Exception_Control:: deliver_exception
9	69	Prefetch Abort	0x204d1c0
11	86	Data Abort	0x204d1c0
12	113	Watchdog	
14	101	Data Abort	0x204e228
26	215	Prefetch Abort	0x204f250
59	497	SIGTRAP	–
78	606	Data Abort	0x204b068
3	9	Data Abort	0x2044e34
51	406	SIGTRAP	0x02001278
42	317	SIGTRAP	powtab.72
14	109	Data Abort	0x204afec
20	170	Data Abort	0x2046e64
2	21	Router, mailbox full	mailbox_wd
2	9	Data Abort	0x204b03c
9	78	SIGTRAP	__cygvar_discard_me__.348
29	198	SIGTRAP	powtab.72
48	415	Data Abort	0x204e29c
13	92	SIGTRAP	powtab.72

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
14	102	SIGTRAP	__cygvar_discard_me__.348
40	324	Data Abort	0x20432b4
17	137	SIGTRAP	Cyg_Exception_Control:: deliver_exception
104	858	SIGTRAP	Cyg_Exception_Control:: deliver_exception
3	30	Watchdog	
58	498	SIGTRAP	Cyg_Exception_Control:: deliver_exception
1	14	Watchdog	
25	211	Router, mailbox full	mailbox_wd
19	138	Data Abort	0x2049f98
34	275	SIGTRAP	Cyg_Exception_Control:: deliver_exception
14	97	SIGTRAP	__cygvar_discard_me__.348
18	136	Data Abort	0x204c028
7	43	Data Abort	0x204c08c
12	90	SIGTRAP	Cyg_Exception_Control:: deliver_exception
27	233	SIGTRAP	Cyg_Exception_Control:: deliver_exception
11	93	Data Abort	0x2049f6c
23	194	SIGTRAP	0x02001278
1	18	Watchdog	
19	145	Data Abort	0x2049fc0
9	66	Prefetch Abort	0x204d1f4
18	154	Data Abort	0x204b03c
17	134	SIGTRAP	–
35	268	Data Abort	0x204b038
17	145	SIGTRAP	__cygvar_discard_me__.348
22	168	SIGTRAP	Cyg_Exception_Control:: deliver_exception

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
15	100	Router, mailbox full	mailbox_noncomm
49	412	Data Abort	0x202ab04
2	13	SIGTRAP	__cygvar_discard_me__.348
36	268	SIGTRAP	__cygvar_discard_me__.348
5	43	SIGTRAP	Cyg_Exception_Control:: deliver_exception
61	521	SIGTRAP	powtab.72
18	154	Data Abort	0x204d198
58	451	Data Abort	0x204b038
39	312	Data Abort	0x204c05c
2	19	SIGTRAP	Cyg_Exception_Control:: deliver_exception
6	41	Router, mailbox full	mailbox_wd
35	290	Data Abort	0x204f290
11	95	Data Abort	0x204c028
20	131	Data Abort	0x204b09c
19	157	SIGTRAP	0x02001278
5	53	Watchdog	
1	17	Watchdog	

C.14 Stacks, Error-correction, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
13	25	Watchdog	
72	121	Prefetch Abort	0x2046e3c
268	472	SIGTRAP	cyg_libc_main_stack
94	167	Data Abort	0x2044e38
12	16	Data Abort	0xfffac
314	531	SIGTRAP	–
218	361	SIGTRAP	cyg_libc_main_stack
106	160	Data Abort	0x204c05c
1	2	Watchdog	
292	493	Data Abort	0x204e230
1	4	Watchdog	
57	99	Data Abort	0x204b03c
280	533	SIGTRAP	powtab.72
1	3	Watchdog	
178	294	SIGTRAP	__cygvar_discard_me__.348
12	17	Router, mailbox full	mailbox_noncomm
249	473	Router, mailbox full	mailbox_wd
65	102	SIGTRAP	powtab.72
64	112	Data Abort	0x2046e3c
96	172	SIGTRAP	0x00000164
287	494	Watchdog	
51	73	SIGTRAP	0x02001278
198	363	Data Abort	0x204e280
31	52	Prefetch Abort	0x2049ff8
59	93	Router, mailbox full	mailbox_wd
315	544	Data Abort	0x204d198
17	45	Data Abort	0x204b064
363	628	Prefetch Abort	0x2044e38
9	22	Data Abort	0x2044e38
72	122	Data Abort	0x204b03c
8	15	SIGTRAP	Cyg_Counter::add_alarm

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
65	122	SIGTRAP	0x00000164
349	579	Data Abort	0x204d1dc
116	205	Data Abort	0x2046e3c
18	33	Watchdog	
81	147	Data Abort	0x204c05c
33	62	Watchdog	
224	382	SIGTRAP	0x02001278
44	67	SIGTRAP	0x00000164
62	98	SIGTRAP	0x00000164
64	108	Data Abort	0x2046ea4
169	272	Data Abort	0x204c034
193	330	Router, mailbox full	mailbox_noncomm
20	32	SIGTRAP	0x00000164
190	370	SIGTRAP	0x00000164
133	227	SIGTRAP	0x02001278
202	370	SIGTRAP	0x00000164
229	401	Data Abort	0x2044e38
70	118	Data Abort	0x2044e38

C.15 Stacks, No Error-correction, Rate 1000

# Watchdog resets	# Injections	Cause of failure	exception address
11	66	Unknown Message	in
61	493	Data Abort	0x204b078
19	191	SIGTRAP	–
9	53	SIGTRAP	Cyg_Exception_Control:: deliver_exception
4	24	Prefetch Abort	0x204d198
3	12	Prefetch Abort	0x204e28c
25	196	Watchdog	
7	57	Data Abort	0x204d1dc
1	1	Data Abort	0x204e23c
24	182	SIGTRAP	0x02001278
18	145	SIGTRAP	Cyg_Exception_Control:: deliver_exception
9	81	Watchdog	
16	125	Data Abort	0x204bfd0
60	503	Router, mailbox full	mailbox_wd
7	41	SIGTRAP	0x02001278
1	8	SIGTRAP	__cygvar_discard_me__.348
7	38	Data Abort	0x204b078
4	27	Data Abort	0x2049fd4
13	104	SIGTRAP	Cyg_Exception_Control:: deliver_exception
33	285	Watchdog	
12	91	SIGTRAP	__cygvar_discard_me__.348
34	304	Router, mailbox full	mailbox_wd
8	56	Data Abort	0x202b54c
12	99	Data Abort	0x2046dd8
7	52	SIGTRAP	0x02001278
1	1	Data Abort	0x204d1b0
2	14	SIGTRAP	Cyg_Exception_Control:: deliver_exception

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
63	535	SIGTRAP	0x02001278
12	104	SIGTRAP	0x02001278
15	129	SIGTRAP	0xfffffffc
8	67	SIGTRAP	Cyg_Exception_Control:: deliver_exception
31	249	Data Abort	0x204b09c
6	48	Router, copy failed	Data-abort at 0x204be5c
13	111	Router, mailbox full	mailbox_noncomm
12	102	Prefetch Abort	0x204b0b8
19	152	Data Abort	0x204d1d0
13	93	SIGTRAP	exception_handler
63	551	Data Abort	0x204f228
34	281	SIGTRAP	Cyg_Exception_Control:: deliver_exception
13	111	Unknown Message	inout
11	76	Prefetch Abort	0x204c4ec
6	57	SIGTRAP	__cygvar_discard_me__.348
32	239	SIGTRAP	__cygvar_discard_me__.348
19	165	Data Abort	0x2049fd4
35	282	Watchdog	
26	206	SIGTRAP	powtab.72
5	36	SIGTRAP	__cygvar_discard_me__.348
10	85	Router, mailbox full	mailbox_noncomm
64	515	SIGTRAP	powtab.72
7	48	Data Abort	0x2049fd4
8	65	Data Abort	0x2046e5c
8	59	Data Abort	0x2049fcc
7	47	Data Abort	0x204e29c
12	89	SIGTRAP	0x02001278
9	64	Data Abort	0x204b060
10	80	Prefetch Abort	0x204b09c
1	6	SIGTRAP	0x02001278

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
11	94	SIGTRAP	0x02001278
11	103	SIGTRAP	0x02001278
6	45	Unknown Message	WD
16	137	Watchdog	
24	202	SIGTRAP	Cyg_Exception_Control:: deliver_exception
53	454	Watchdog	
16	141	Prefetch Abort	0x204e2b8
35	315	Watchdog	
10	70	SIGTRAP	Cyg_Exception_Control:: deliver_exception
12	97	Prefetch Abort	0x204b08c
16	125	Data Abort	0x204d1d4
13	101	Data Abort	0x204b03c
53	454	Data Abort	0x204b03c
13	107	SIGTRAP	0x02001278
11	104	Watchdog	
56	453	Data Abort	0x2044e3c
12	94	Watchdog	
18	157	Data Abort	0x2046e54
9	75	Data Abort	0x204e230
23	188	Data Abort	0x204e23c
8	64	SIGTRAP	0x020696e8
13	107	Prefetch Abort	0x204a014
21	162	Data Abort	0x204c08c
28	217	SIGTRAP	powtab.72
12	101	Data Abort	0x2046ea4
32	266	SIGTRAP	0x02001278
64	521	Router, mailbox full	mailbox_wd
5	36	Prefetch Abort	0x204f260
29	228	Data Abort	0x2049fd0
24	192	SIGTRAP	0x02001278

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
2	18	Watchdog	
23	185	SIGTRAP	Cyg_Exception_Control:: deliver_exception
42	362	SIGTRAP	Cyg_Exception_Control:: deliver_exception
9	72	Data Abort	0x2049f98
6	46	Router, mailbox full	mailbox_noncomm
2	19	Watchdog	
3	29	Prefetch Abort	0x2046e94
24	199	SIGTRAP	0x02001278
40	330	Data Abort	0x204d194
25	215	Watchdog	
16	133	SIGTRAP	Cyg_Exception_Control:: deliver_exception
47	422	Prefetch Abort	0x2047388
20	176	SIGTRAP	0x02001278
31	238	Data Abort	0x204a014

C.16 Stacks, No Error-correction, Rate 5000

# Watchdog resets	# Injections	Cause of failure	exception address
75	148	Watchdog	
11	29	SIGTRAP	0x00000164
77	143	SIGTRAP	0x00000164
59	100	Unknown Message	out
53	89	SIGTRAP	0x00000164
131	209	SIGTRAP	0x020abe74
16	33	Data Abort	0x2044e74
76	129	Data Abort	0x2044e74
65	106	Watchdog	
29	47	Prefetch Abort	0x2044eb4
141	260	Unknown Message	WD
104	179	Data Abort	0x204f2a0
74	109	SIGTRAP	stack_inbox
166	303	Prefetch Abort	0x2049ff8
106	201	Watchdog	
43	70	SIGTRAP	0x00000164
84	152	Router, copy failed	mailbox_noncomm
38	62	SIGTRAP	0x00000164
317	521	SIGTRAP	0x02001278
57	104	SIGTRAP	0x02001278
212	369	Data Abort	0x204c024
172	296	Data Abort	0x2044e60
51	85	Data Abort	0x2044e74
36	65	Prefetch Abort	0x2046ea4
212	366	Data Abort	0x204d1d4
66	103	SIGTRAP	0x00000000
182	328	Data Abort	0x2044e60
177	302	Data Abort	0x2044e38
162	290	Router, mailbox full	mailbox_wd
329	537	Prefetch Abort	0x204b09c
33	56	Router, recv failed	

continued on next page

continued from previous page

# Watchdog resets	# Injections	Cause of failure	exception address
139	240	Data Abort	0x204d1c0
29	48	Prefetch Abort	0x204c05c
6	8	Data Abort	0x204e228
63	106	SIGTRAP	__cygvar_discard_me__.348
154	264	Router, mailbox full	mailbox_wd
21	41	Prefetch Abort	0x204bff8
175	317	Prefetch Abort	0x204f290
60	90	SIGTRAP	0x02001278
38	69	Data Abort	0x204e23c
18	19	Data Abort	0x204e260
49	89	SIGTRAP	__cygvar_discard_me__.348
95	147	SIGTRAP	0x02001278
260	450	Watchdog	
101	178	Router, mailbox full	mailbox_noncomm
98	164	SIGTRAP	0x0207e29c
49	95	SIGTRAP	exception_handler
1	2	Router, PDU too large	mailbox_in
6	14	SIGTRAP	0x02001278
79	113	Prefetch Abort	0x2044e38