Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery

Jonny Vinter, Joakim Aidemark, Peter Folkesson, Johan Karlsson
Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
+46 31 772 1667, +46 31 772 3663 fax
{vinter, aidemark, peterf, johan}@ce.chalmers.se

Abstract

Systems that use f+1 computer nodes to tolerate f node failures ordinarily require that the computer nodes have strong failure semantics, i.e. a node should either produce correct results, or no results at all. We show that this requirement can be relaxed for control applications, as control algorithms inherently compensate for a class of value failures. Value failures occur when an error escapes the error detection mechanisms in the computer node and an erroneous value is sent to the actuators of the control system. Fault injection experiments show that 89% of the value failures caused by bit-flips in a CPU had no or minor impact on the controlled object. However, the experiments also show that 11% of the value failures had severe consequences. These failures were caused by bit-flips affecting the state variables of the control algorithm. Another set of fault injection experiments show that the percentage of the value failures with severe consequences was reduced to 3% when the state variables were protected with executable assertions and best effort recovery mechanisms.

1. Introduction

The primary task for many embedded computer systems is to execute control algorithms. Embedded control systems are used in safety-critical applications such as fly-by-wire systems, jet-engine controllers, electronic throttles and active suspension. Applications requiring extreme levels of dependability, such as fly-by-wire often use massive redundancy and majority voting to achieve fault tolerance, see e.g. [1]. Today replication is usually introduced at the computer node level. Thus, classical TMR (triple modular redundancy) [2] requires three nodes to tolerate one node failure. In the general case, *2f+1* nodes are required to tolerate *f* faulty nodes. Some systems are designed to tolerate Byzantine faults, which requires as much as *3f+1* nodes to tolerate *f* node failures [3] [4].

An advantage of using massive redundancy is that the failure semantics[1] of the computer nodes can be weak. Weak failure semantics implies that a node can exhibit a wide range of failure classes, including value failures and timing failures. Massive redundancy is, however, considered too expensive for a wide range of embedded control applications. A more cost-effective approach is to use nodes with strong failure semantics. Examples of strong failure semantics are when a node exhibits only fail-stop or omission failures [5], i.e. the node should either produce correct results, or no result at all.

Systems using computer nodes with strong failure semantics need only *f+1* nodes to tolerate *f* faulty nodes. An example of such a system is the classical duplex system, which uses two computer nodes to tolerate one node failure. Strong failure semantics makes the task of identifying a faulty node simple in a duplex system. Duplex systems are commonly used in safety-critical or mission critical systems such as jet engine controllers, satellites, and satellite launchers. They are also preferred in the cost-sensitive automotive industry. Using computer nodes with strong failure semantics also simplifies the design of control systems that must ensure safe shutdown.

In order to achieve strong failure semantics, a computer node must be equipped with internal error detection mechanisms. Duplication and comparison can be used to achieve strong failure semantics for random hardware faults, but this is an expensive solution since each node then consists of two computers (with weak failure semantics) and extra logic for comparing results. Thus, we need *2(f+1)* computers to tolerate *f* computer failures.

[1] The concept of failure semantics was introduced in [5].

In cost-sensitive applications, strong failure semantics are achieved by combining low-cost hardware and software error detection mechanisms. These include hardware mechanisms such as error correcting codes, memory management units, hardware exceptions, control-flow checking and watchdog timers. Examples of software error detection mechanisms are executable assertions, software implemented exceptions, time redundant execution of tasks, and acceptance tests. Some of these mechanisms have the capability to detect errors caused by software faults, which is not possible with hardware duplication and comparison.

Unfortunately, most of these techniques also have rather low coverage for value errors, i.e. errors that affect the result, but not the control flow or the timing of a computation. Time redundant execution of tasks is one way to improve the coverage of such errors, but as demonstrated in [6] the coverage may still be less than 100%.

In this paper, we address the problem of dealing with value failures in control applications. A value failure occurs when an error escapes the error detection mechanisms in a computer node and an erroneous result is sent to the actuators of the control system. We consider value failures caused by transient bit-flips occurring in the central processing unit (CPU) of a computer node using a single CPU. Particles such as heavy-ions, alpha particles and high-energetic neutrons are known to be causing bit-flips in VLSI-circuits in aerospace applications [7]. Recent research indicates that such errors also can occur at ground level, although with a much lower probability than in space or the upper atmosphere [8] [9].

In an experimental study of an embedded engine controller, we demonstrate that a vast majority of the value failures caused by bit-flips in the CPU had no or little impact on the engine. The reason for this is that control applications in general are inherently robust with respect to value failures, provided that their impact on the control algorithm is similar to that of external disturbances affecting the controlled object.

However, the experiments also show that bit-flips affecting the state variables of the control algorithm could cause value failures with unacceptable consequences, such as permanently locking the engine's throttle at full speed. By adding executable assertions and best effort recovery mechanisms to the control program, we managed to significantly reduce the probability of value failures. In particular, failures that locked the throttle at full speed were not observed when the target system was running the modified control program.

The engine control algorithm was taken from a design library supplied with Simulink [10], a toolbox for MATLAB, which is widely used for design of control algorithms. The control program was executed on the Thor CPU [11], which has been specifically designed for use in critical embedded space applications. Bit-flips were injected into the Thor CPU using a new fault injection tool called GOOFI. We have previously evaluated the hardware implemented error detection mechanisms in Thor [12]. In this paper we use software techniques to handle errors that escape the hardware mechanisms.

The remainder of the paper is organized as follows. Section 2 describes the engine control algorithm. Section 3 describes the experimental set up. The results are presented in Section 4. Finally, the conclusions are presented in Section 5.

2. The engine controller

The engine controller investigated in this study was developed with the MATLAB toolbox Simulink, which is a software package for modeling, simulating and analyzing dynamical systems. Simulink provides a graphical user interface for building block diagrams of models using click-and-drag mouse operations and generating the corresponding software code that implements the model. The Ada code used for our model was generated with the product Real-Time Workshop Ada Coder, which is an extension for Simulink.

The controller is a proportional-integral (PI) controller used for controlling the speed of an engine. The target system for the fault injection experiments executed only the code generated for the PI controller block in Figure 1.

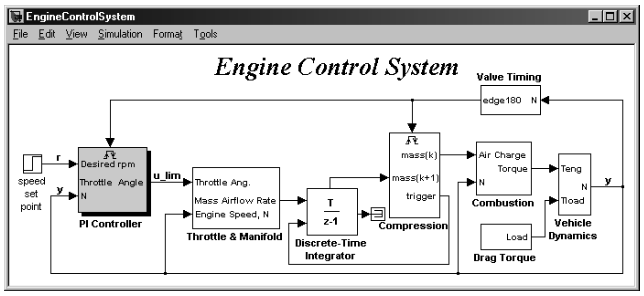

Figure 1. Simulink model of the engine control system.

The code generated for the rest of the engine control system in Figure 1 was used to simulate the controlled object (i.e. the engine) and was executed on a Unix workstation and was not subjected to fault injection. The workstation hosted the board containing the target system and was also executing the fault injection tool.

The PI controller is shown in detail in Figure 2. It controls the speed of the engine by adjusting the opening angle of the engine's throttle, which lies between 0.0 and 70.0 degrees. The control error signal *e* is calculated as the difference between the reference value *r* (speed wanted) and the actual engine speed *y* by:

$$e(k) = r(k) - y(k) \qquad (1)$$

where *k* is the sample number. The PI controller consists of an integrating part and a proportional part. The integrating part integrates the error signal *e* (multiplied with the integral gain K_i) as:

$$x(k) = x(k-1) + T \cdot K_i \cdot e(k) \qquad (2)$$

where *T* is the sample interval. In addition, the proportional part directly scales the control error signal *e* by the proportional gain K_p, and the desired throttle angle is the sum of the integral and the proportional parts, as:

$$u(k) = K_p \cdot e(k) + x(k). \qquad (3)$$

The output signal *u(k)* can assume values outside the interval 0.0 to 70.0 degrees. The limit output function in Figure 2, assures that the output signal *u_lim* lies within this interval. There is also an anti-windup function that cuts off the integration if the input *y* from the engine is not responding to the output *u_lim* from the controller when the signal is limited (i.e. at 0.0 or 70.0 degrees).

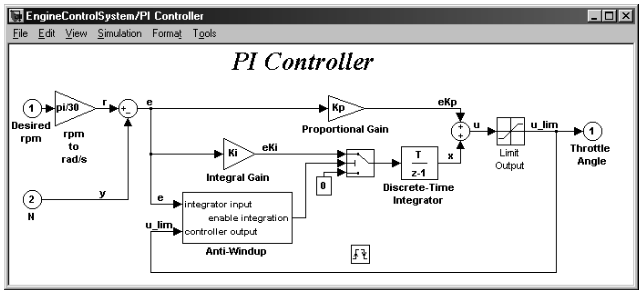

Figure 2. PI controller block from Figure 1, executing on the target system.

This results in the control error signal *e* not being equal to zero. An example could be a full throttle angle of 70 degrees (upper limit) but a heavy load holding the engine speed down resulting in $e > 0$ that erroneously increase the value of the state *x* (above the upper limit) according to equation (2). In this case, the integration will be stopped until *u_lim* is back within the defined limits.

A simplified algorithm of the PI controller workload can be expressed as:

```
x : float                   -- state of the controller

function PI_Controller(r, y : float)
  Kp, T: constant float    -- constants
  e, u, u_lim, Ki : float -- controller variables
begin
  e = r - y                 -- calculate control error
  u = e * Kp + x            -- calculate output signal
  u_lim = limit_output(u) -- range check of u
  if anti_windup_activated then
    Ki = 0.0                -- disable integration
  else
    Ki = integral_gain      -- enable integration
  end if
  x = x + T * e * Ki        -- integrate, update x
  return u_lim
end
```

Algorithm I. The PI controller algorithm.

In our experiments, a sequence of 650 iterations of the PI controller algorithm was executed. This corresponds to a total time interval of 10 seconds with a sample interval of 15.4 milliseconds. As shown in Figure 3, the reference speed *r* was kept constant at 2000 rpm for the first half of the 10 second interval and was then changed momentarily to 3000 rpm. The figure also shows the actual engine speed *y*.

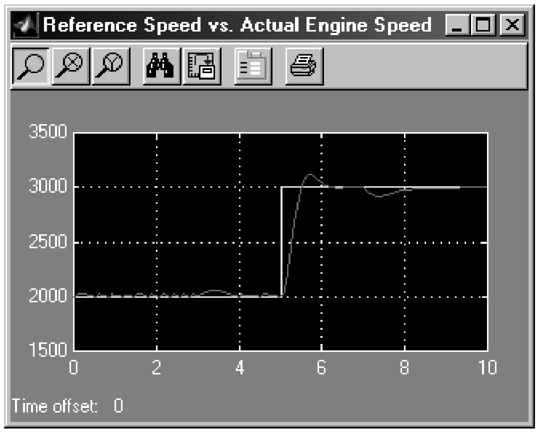

Figure 3. Reference speed *r* (white) vs. actual engine speed *y* (grey).

Figure 4 shows how the engine load varies during the observed time interval. The variations of the engine load cause the differences between the reference speed and the actual engine speed at time $3 < t < 4$ and $7 < t < 8$ as shown in Figure 3. The variations in engine load may occur when the engine is used to move a vehicle at a desired speed in hilly terrain. The output of the PI control algorithm under fault-free conditions is shown in Figure 5.

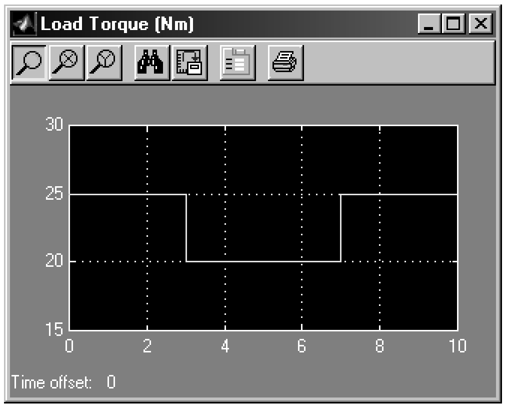

Figure 4. Engine load.

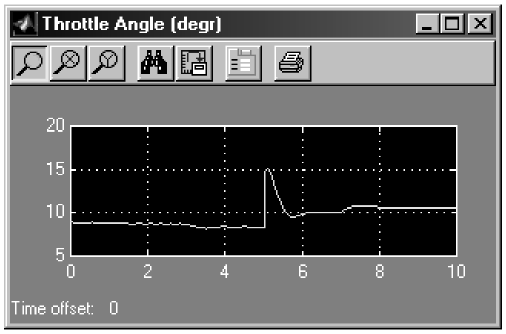

Figure 5. Fault-free output *u_lim* from the PI controller.

3. Experimental Setup

The experiments were conducted using a new fault injection tool called GOOFI (Generic Object-Oriented Fault Injection tool) [13]. The GOOFI tool was executed on a UNIX workstation hosting a processor board based on the Thor microprocessor [11], which was configured as the target system for the fault injection experiments (see Figure 6).

3.1. The Thor microprocessor

The Thor microprocessor executed the code generated for the PI controller algorithm described in Section 2. Thor is a 32-bit CPU with a four-stage pipeline and a 128 byte data cache located within the pipeline. Several error detection mechanisms, see Table 1, and support for Ada tasking are included in the processor. Thor also features advanced scan-chain logic that allows read access to more than 3000 of the almost 4500 internal state elements of the CPU and write access to more than 2700 internal state elements.

So far, Thor has been used in space applications, where low weight, small volume and low power consumption are important factors to be considered in addition to high dependability. The processor has been used for attitude-control of the ODIN satellite and will be used in on board instruments on the comet explorer mission ROSETTA and the Mars Express mission.

Table 1. Error detection mechanisms of Thor.

Error Detection Mechanism	Description
BUS ERROR	Bus time-out on external memory access
ADDRESS ERROR	Access to non-existing or protected memory
INSTRUCTION ERROR	Attempt to execute a privileged instruction in user mode or an illegal instruction
JUMP ERROR	Attempt to jump, call or return to a target address outside memory address space
CONSTRAINT ERROR	A run-time assertion failed
ACCESS CHECK	Attempt to follow a null pointer
STORAGE ERROR	Attempt to access memory outside the task's stack in user mode
OVERFLOW CHECK	Overflow of signed integer and float arithmetic operations
UNDERFLOW CHECK	Underflow or denormalized result of float arithmetic operations
DIVISION CHECK	Divide by zero for integer division. Divide by ±0 for float division
ILLEGAL OPERATION	Illegal operation for float and double arithmetic instructions involving 0 and
DATA ERROR	Uncorrectable error in data read from memory
CONTROL FLOW ERROR	A control flow error (wrong sequence of instructions) occurred
MASTER/SLAVE COMPARATOR ERROR	An error was detected when comparing the results from two Thor processors (not used in this study)

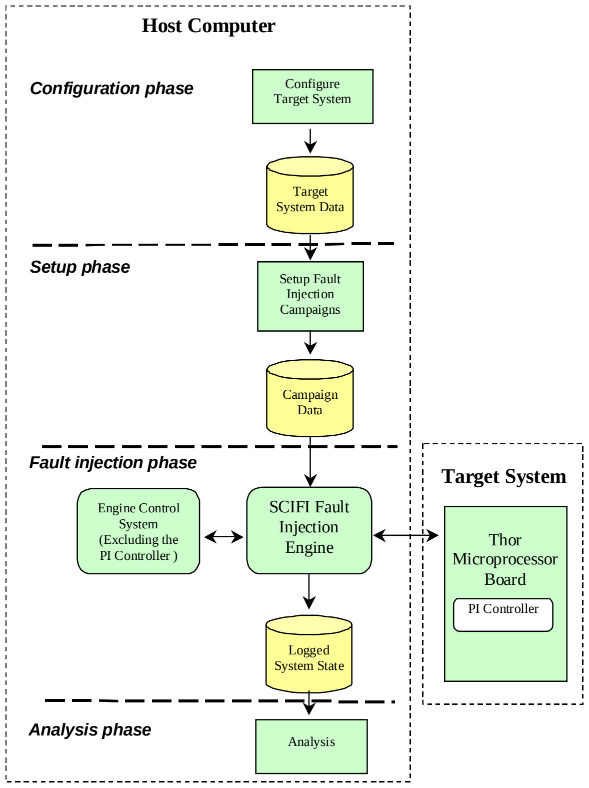

Figure 6. Overview of the experimental setup.

3.2. The GOOFI tool

The GOOFI tool can perform fault injection campaigns using different fault injection techniques on different target systems. A major objective of the tool is to provide a user-friendly fault injection environment with a graphical user interface and an underlying generic architecture that assists the user when adapting the tool for new target systems and new fault injection techniques.

In addition, the tool is highly portable between different host platforms since the tool was implemented using the Java programming language and all data is saved in a SQL compatible database. GOOFI was designed using object orientation to facilitate maintainability and portability.

3.3. Campaign configuration

As shown in Figure 6, conducting fault injection campaigns using GOOFI involves four phases: the *configuration*, *set-up*, *fault injection* and *analysis* phase.

3.3.1. Configuration phase. The configuration phase involves adapting the GOOFI tool to the chosen fault injection technique and target system. The current version of GOOFI supports pre-runtime Software Implemented Fault Injection (SWIFI) and Scan-Chain Implemented Fault Injection (SCIFI). In SCIFI, faults are injected via boundary scan-chains or internal scan-chains in a VLSI circuit [12]. This allows faults to be injected into the pins and many of the internal state elements of the target circuit. The scan-chains are also used to observe the internal state of the circuit before and after a fault is injected. In this study, the GOOFI tool was configured to use the SCIFI technique on the Thor microprocessor.

3.3.2. Set-up phase. In the set-up phase, the user selects a target system and a workload, and chooses the fault injection locations from a hierarchical list of possible locations presented in a window. The user must also select the fault models to use, the points in time when faults should be injected, and the total number of faults in the fault injection campaign.

2250 fault locations of the 3000 state elements of Thor accessible via the scan-chain logic were chosen for fault injection. The fault injection locations were selected randomly using uniform sampling among the 2250 state elements.

Using the GOOFI tool with the SCIFI technique requires break-points to be set according to the points in time when faults should be injected. The break-points are set via the scan-chains in the fault injection phase and allow the Thor processor to be halted for fault injection when a machine instruction is to be executed. The points in time for fault injection are selected by analyzing the workload code. In this study, the points in time for fault injection were selected randomly using a uniform sampling distribution among the points in time each of the instructions of the workload begin their execution.

The fault model used was single bit-flip faults, which model the effects of transients occurring in the CPU.

The *termination conditions* for the experiments are also selected in the set-up phase. A fault injection experiment is terminated by a debug event (generated via the scan chains) i.e., an error has been detected or the execution of the workload ends, whichever comes first. The workload may consist of a program that either terminates by itself or is executed as an infinite loop. For an infinite loop, such as the PI controller used in this study, the user must specify the maximum number of iterations that should be executed before the fault injection experiment is terminated. 650 loop iterations were used in this study (see Section 2). In each loop iteration, data may be exchanged with a user provided environment simulator program emulating the target system environment. In our case the environment simulator was the Simulink generated model of the engine shown in Figure 1 (excluding the PI controller). The environment simulator was executed on the same host computer as the GOOFI tool.

Information about the memory locations holding input and output data within the target system as well as the points in time the data exchange occurs, e.g. when each loop iteration finishes, must also be selected by the user.

All set-up data is stored in a database for use during the fault injection phase.

3.3.3. Fault injection phase. In the fault injection phase, the GOOFI tool starts by reading the campaign information from the database. The target system is initialised and the workload and initial input data is downloaded to the system. Then a reference execution of the workload is made, logging the fault-free system state to the GOOFI database. After this, each fault injection experiment begins by reinitialising the target system and downloading the workload and initial input data.

For each fault injection experiment, a breakpoint is set via the scan-chains at the instruction to be executed when a fault should be injected. When the break-point condition has been fulfilled, the fault is injected by reading the scan-chains, inverting the bits corresponding to the fault location, and then writing back the altered scan-chain data.

After injecting a fault, the execution starts from where the target system was halted and continues until the termination condition occurs. The system state is then logged to the database. Finally, the target system is reinitialized and a new fault injection experiment begins.

GOOFI can be operated in either *normal* or *detail* mode. In normal mode, the system state is logged only when the termination condition is fulfilled. In detail mode the system state is logged as frequently as the target system allows, in this case before the execution of each machine instruction, which increases the time-overhead. The detail mode operation is used to produce an execution trace, allowing the error propagation to be analyzed in detail. The logged system state includes the contents of all the locations in the target system that are observable as well as the workload input and output values, together with information about when and where any faults were injected.

3.3.4. Analysis phase. The final phase involved when conducting fault injection campaigns using GOOFI is the analysis phase. In this phase, the data in the database is analyzed in order to obtain various dependability measures. Currently, there is no support for automatic generation of software that analyses the logged data. The user must write tailor made scripts or programs that query the database for the required information. However, this is typically done once for each new target system.

4. Results

In this section, we first describe the error and failure classification scheme used in the presentation of the results. We then present the results of a fault injection campaign evaluating the PI controller algorithm described in Section 2 (Algorithm I), which is followed by a description of a modified algorithm aimed at reducing critical failures (Algorithm II). Finally, the results of fault injection experiments using Algorithm II are presented and compared with the results for Algorithm I.

4.1. Error and failure classification

The errors are classified into either effective or non-effective errors in the results from the fault injection experiments.

4.1.1. Effective errors. Effective errors are errors which were either detected by the error detection mechanisms of the Thor processor (see Table 1) or errors causing undetected wrong results (value failures) to be produced by the PI controller:

Detected errors: Errors detected by the error detection mechanisms in Thor. These errors are further classified into errors detected by each of the various mechanisms or *other errors*. Other errors are errors that were detected, but the GOOFI analysis software could not determine which mechanism that detected the error.

Undetected wrong results: The controller produced an incorrect result, i.e. a value failure. These failures are classified into severe or minor value failures depending on their impact on the controlled object

Severe: The value failure has a severe impact on the controlled object. These failures are either:

***Permanent*:** The output from the controller is either at maximum value (70.0 degrees) or minimum value (0.0 degrees) from the time the value failure first appears until the end of the observed time interval, see Figure 7. Note that the observed time interval is limited to 10 seconds (650 iterations of the control algorithm) and that the output may converge towards the fault-free output sequence later.

Semi-permanent: The output from the controller differs strongly (more than 0.1 degrees) from the fault-free output during more than one iteration, but the output starts to converge towards the fault-free output sequence within the observed time interval, see Figure 8:

Minor: The value failure has a minor impact on the controlled object. These failures are classified into:

Transient: The output from the controller differs strongly (more than 0.1 degrees) from the fault-free output during one iteration and then rapidly starts to converge towards the fault-free output, see Figure 9.

Insignificant: The output from the controller is almost identical to the fault-free output. We define an insignificant error to have a deviation from the fault-free output less than 0.1 degrees.

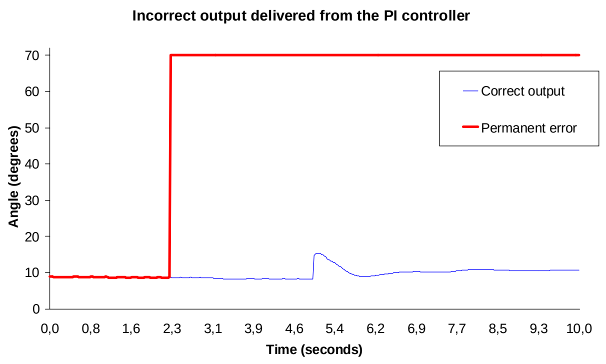

Figure 7. Severe undetected wrong result (permanent).

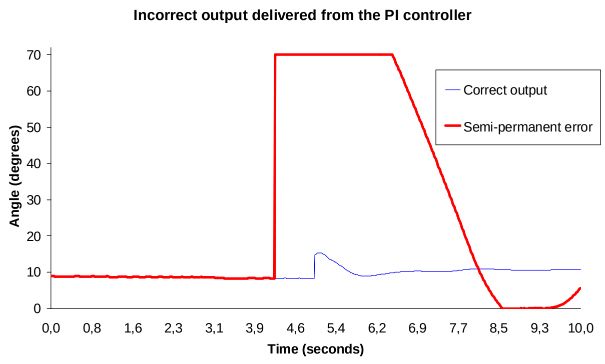

Figure 8. Severe undetected wrong result (semi-permanent).

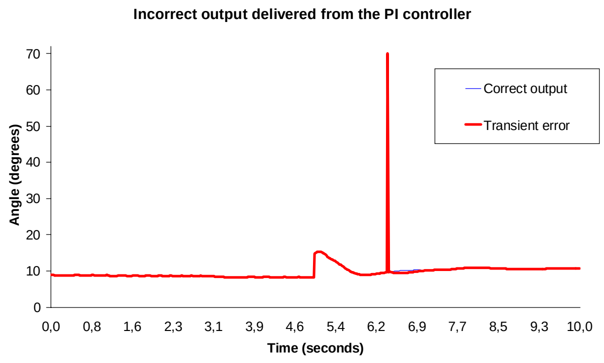

Figure 9. Minor undetected wrong result (transient).

4.1.2. Non-effective errors. The non-effective errors are errors, which could not be identified as either Detected errors or Undetected wrong results. These errors are classified into latent or overwritten errors:

Latent errors: The fault injection experiments where differences between the correct system state logged after the reference execution and the system state logged after the fault injection experiment terminated could be observed, but which could not be identified as either Detected errors or Undetected wrong results.

Overwritten errors: The fault injection experiments for which no difference between the correct system state logged after the reference execution and the system state logged after the fault injection experiment terminated could be observed.

4.2. Results for Algorithm I

Table 2 shows the results obtained for the PI controller implemented according to Algorithm I. The first two columns present the results separately for faults injected into the data cache of Thor (denoted "Cache" in the table) and into all other parts of the CPU (denoted "Registers"). The last column presents the total results for all faults injected into the CPU. The percentage of errors obtained for each of the categories presented in Section 3 and their corresponding 95% confidence intervals are given in the table together with the observed number of errors (#). The results show that most of the undetected wrong results caused by bit-flips in the CPU were minor value failures having no or minor impact on the controlled object (4.48% of all faults injected). However, 0.54% of the faults injected had severe impact on the controlled object (severe undetected wrong results) corresponding to 10.73% of all the value failures produced.

The results also show that faults injected into the data cache of Thor produced a higher percentage of undetected wrong results (6.06%) than faults injected into the registers (0.91%). A detailed investigation revealed that most of the severe undetected wrong results were caused by faults injected into the cache lines where the global variable *x* representing the state, see Algorithm I, is stored. Since *x* represents the state, any errors in *x* will propagate to the next iteration of the algorithm and cause a permanent or semi-permanent value failure, see Figure 7 and 8.

Table 2. Results for Algorithm I.

Part of CPU fault injected (no. of state elements)	**Cache (1824)**			**Registers (426)**			**Total (2250)**		
Type of Errors and Wrong Results	**%**	**(95 % conf)**	**#**	**%**	**(95 % conf)**	**#**	**%**	**(95 % conf)**	**#**
Latent Errors	0,05%	(± 0,05%)	4	59,99%	(± 2,22%)	1126	12,16%	(± 0,66%)	1130
Overwritten Errors	72,28%	(± 1,02%)	5358	19,98%	(± 1,81%)	375	61,71%	(± 0,99%)	5733
Total (Non Effective Errors)	**72,33%**	**(± 1,02%)**	**5362**	**79,97%**	**(± 1,81%)**	**1501**	**73,88%**	**(± 0,89%)**	**6863**
Address Error	16,84%	(± 0,85%)	1248	4,37%	(± 0,92%)	82	14,32%	(± 0,71%)	1330
Data Error	0,00%	(± 0,00%)	0	0,37%	(± 0,28%)	7	0,08%	(± 0,06%)	7
Instruction Error	1,50%	(± 0,28%)	111	2,08%	(± 0,65%)	39	1,61%	(± 0,26%)	150
Jump Error	0,07%	(± 0,06%)	5	0,05%	(± 0,10%)	1	0,06%	(± 0,05%)	6
Constraint Check	0,01%	(± 0,03%)	1	0,00%	(± 0,00%)	0	0,01%	(± 0,02%)	1
Access Check	0,01%	(± 0,03%)	1	0,27%	(± 0,23%)	5	0,06%	(± 0,05%)	6
Storage Error	0,13%	(± 0,08%)	10	9,80%	(± 1,35%)	184	2,09%	(± 0,29%)	194
Overflow	0,00%	(± 0,00%)	0	0,11%	(± 0,15%)	2	0,02%	(± 0,03%)	2
Illegal Operation	2,29%	(± 0,34%)	170	0,80%	(± 0,40%)	15	1,99%	(± 0,28%)	185
Control Flow Errors	0,71%	(± 0,19%)	53	1,07%	(± 0,46%)	20	0,79%	(± 0,18%)	73
Other Errors	0,01%	(± 0,03%)	1	0,32%	(± 0,26%)	6	0,08%	(± 0,06%)	7
Undetected Wrong Results (Severe)	0,66%	(± 0,18%)	49	0,05%	(± 0,10%)	1	0,54%	(± 0,15%)	50
Undetected Wrong Results (Minor)	5,40%	(± 0,51%)	400	0,85%	(± 0,42%)	16	4,48%	(± 0,42%)	416
Total (Effective Errors)	**27,67%**	**(± 1,02%)**	**2051**	**20,03%**	**(± 1,81%)**	**376**	**26,12%**	**(± 0,89%)**	**2427**
Total (Faults Injected)	**100,00%**		**7413**	**100,00%**		**1877**	**100,00%**		**9290**
Total (Undetected Wrong Results)	**6,06%**	**(± 0,54%)**	**449**	**0,91%**	**(± 0,43%)**	**17**	**5,02%**	**(± 0,44%)**	**466**
Coverage	**93,94%**	**(± 0,54%)**		**99,09%**	**(± 0,43%)**		**94,98%**	**(± 0,44%)**	

4.3. Adding executable assertions and best effort recovery

The results for Algorithm I show that errors in the state variables stored in the data cache caused severe value failures. One way to avoid single bit-flips affecting the sensitive data stored in the cache is to use a parity protected cache. Since parity protected caches may not be available in commercial off-the-shelf (COTS) microprocessors and the cost for using custom-designed microprocessors with parity protected caches may be unacceptable, a cost-effective software-based solution for reducing the amount of severe value failures is presented. The solution is based on protecting the state variables and output signals with executable assertions and best effort recovery mechanisms.

Three approaches where adopted to make the control algorithm more robust with respect to severe value failures:

1. The state variable and output were protected by executable assertions[2] to detect errors using the physical constraints of the controlled object. The constraints used in this study are based on the physical limitations of the engine throttle. Assertions are made on the state variable *x* and the limited output signal *u_lim* just before a back-up of the variables is made, thereby reducing the probability of error propagation.
2. When an incorrect state is detected by an executable assertion during one iteration of the control algorithm, a recovery is made by using the state backed-up during the previous iteration in the calculations instead. Note that this is not a true recovery, since the input to the controller may differ from the input used in the previous iteration. This may result in the output being slightly different from the fault-free output, thus creating a minor value failure. We therefore call this a *best effort recovery*.
3. When an incorrect output is detected by an executable assertion, recovery is made by delivering the output produced in the previous iteration instead. The state variable is also set to the state of the previous iteration that corresponds to the delivered output. This approach is also a best effort recovery since the effects on the output will be similar to those of approach 2.

A modified algorithm with executable assertions and best effort recovery mechanisms is shown in Algorithm II.

[2] An executable assertion is a software implemented check verifying that a variable fulfills limitations given by a specification.

Changes from the original Algorithm I are in bold:

```
x : float                   -- state of the controller
x_old, u_old : float        -- two back-up states

function PI_Controller(r, y : float)
  Kp, T : constant float  -- constants
  e, u, u_lim: float      -- controller variables
begin
  e = r - y               -- calculate control error
  if not in_range(x)      -- x out of limits?
    x = x_old             -- ERROR! recover state x
  else
    x_old = x             -- save state x
  end if
  u = e * Kp + x          -- calculate output signal
  u_lim = limit_output(u) -- range check of u
  if anti_windup_activated then
    Ki = 0.0              -- disable integration
  else
    Ki = integral_gain    -- enable integration
  end if
  x = x + T * e * Ki      -- integrate, update x
  if not in_range(u_lim)  -- u_lim out of limits?
    u_lim = u_old         -- ERROR! get last output
    x = x_old             -- and corresponding state
  end if
  u_old = u_lim           -- save output
  return u_lim
end
```

Algorithm II. The PI controller algorithm with executable assertions and best effort recovery mechanisms.

A general approach for making a control algorithm with an arbitrary number of states variables and an arbitrary number of output signals more robust can be summarized as:

1. Before making a back-up of any state $x_i(k)$, $0 < i < totalNrOfStates$, an assertion is made validating the correctness of $x_i(k)$. If an erroneous value is detected, then a best effort recovery is made by executing $x_i(k) = x_i(k - 1)$, $0 < i < totalNrOfStates$, otherwise the state is backed-up by executing $x_i(k - 1) = x_i(k)$, $0 < i < totalNrOfStates$.
2. Before returning the output results, $u_j(k)$, $0 < j < totalNrOfOutputs$, an assertion is made validating the correctness of $u_j(k)$, $0 < j < totalNrOfOutputs$. If an incorrect output result is detected, a best effort recovery is made by executing $u_j(k) = u_j(k - 1)$, $0 < j < totalNrOfOutputs$ and $x_i(k) = x_i(k - 1)$, $0 < i < totalNrOfStates$.
3. Back-up the output signals by executing $u_j(k - 1) = u_j(k)$, $0 < j < totalNrOfOutputs$.
4. Return the output signals $u_j(k)$, $0 < j < totalNrOfOutputs$.

4.4. Results for Algorithm II

Table 3 shows the results obtained for the modified PI controller Algorithm II. Most of the undetected wrong results had no or minor impact on the controlled object, similar to what is observed for Algorithm I (5.06% of all faults injected lead to minor undetected wrong results).

Table 3. Results for Algorithm II.

Part of CPU fault injected (no. of state elements)	Cache (1824)			Registers (426)			Total (2250)		
Type of Errors and Wrong Results	**%**	**(95 % conf)**	**#**	**%**	**(95 % conf)**	**#**	**%**	**(95 % conf)**	**#**
Latent Errors	21,04%	(± 1,85%)	391	59,14%	(± 4,25%)	304	29,30%	(± 1,83%)	695
Overwritten Errors	50,38%	(± 2,27%)	936	18,87%	(± 3,39%)	97	43,55%	(± 2,00%)	1033
Total (Non Effective Errors)	**71,42%**	**(± 2,05%)**	**1327**	**78,02%**	**(± 3,58%)**	**401**	**72,85%**	**(± 1,79%)**	**1728**
Address Error	16,15%	(± 1,67%)	300	5,45%	(± 1,96%)	28	13,83%	(± 1,39%)	328
Data Error	0,00%	(± 0,00%)	0	0,58%	(± 0,66%)	3	0,13%	(± 0,14%)	3
Instruction Error	2,69%	(± 0,74%)	50	2,14%	(± 1,25%)	11	2,57%	(± 0,64%)	61
Jump Error	0,05%	(± 0,11%)	1	0,00%	(± 0,00%)	0	0,04%	(± 0,08%)	1
Constraint Check	0,00%	(± 0,00%)	0	0,00%	(± 0,00%)	0	0,00%	(± 0,00%)	0
Access Check	1,45%	(± 0,54%)	27	0,19%	(± 0,38%)	1	1,18%	(± 0,43%)	28
Storage Error	0,43%	(± 0,30%)	8	10,12%	(± 2,61%)	52	2,53%	(± 0,63%)	60
Overflow	0,00%	(± 0,00%)	0	0,00%	(± 0,00%)	0	0,00%	(± 0,00%)	0
Illegal Operation	0,54%	(± 0,33%)	10	1,17%	(± 0,93%)	6	0,67%	(± 0,33%)	16
Control Flow Errors	0,70%	(± 0,38%)	13	0,97%	(± 0,85%)	5	0,76%	(± 0,35%)	18
Other Errors	0,05%	(± 0,11%)	1	0,78%	(± 0,76%)	4	0,21%	(± 0,18%)	5
Undetected Wrong Results (Severe)	0,22%	(± 0,21%)	4	0,00%	(± 0,00%)	0	0,17%	(± 0,17%)	4
Undetected Wrong Results (Minor)	6,30%	(± 1,10%)	117	0,58%	(± 0,66%)	3	5,06%	(± 0,88%)	120
Total (Effective Errors)	**28,58%**	**(± 2,05%)**	**531**	**21,98%**	**(± 3,58%)**	**113**	**27,15%**	**(± 1,79%)**	**644**
Total (Faults Injected)	**100,00%**		**1858**	**100,00%**		**514**	**100,00%**		**2372**
Total (Undetected Wrong Results)	**6,51%**	**(± 1,12%)**	**121**	**0,58%**	**(± 0,66%)**	**3**	**5,23%**	**(± 0,90%)**	**124**
Coverage	**93,49%**	**(± 1,12%)**		**99,42%**	**(± 0,66%)**		**94,77%**	**(± 0,90%)**	

The results also show that 0.17% of the faults injected had severe impact on the controlled object (caused severe undetected wrong results), which corresponds to 3.23% of all value failures produced using the Algorithm II. Thus, the percentage of severe undetected wrong results are significantly reduced for Algorithm II compared to Algorithm I, where 10.7% of the value failures were classified as severe.

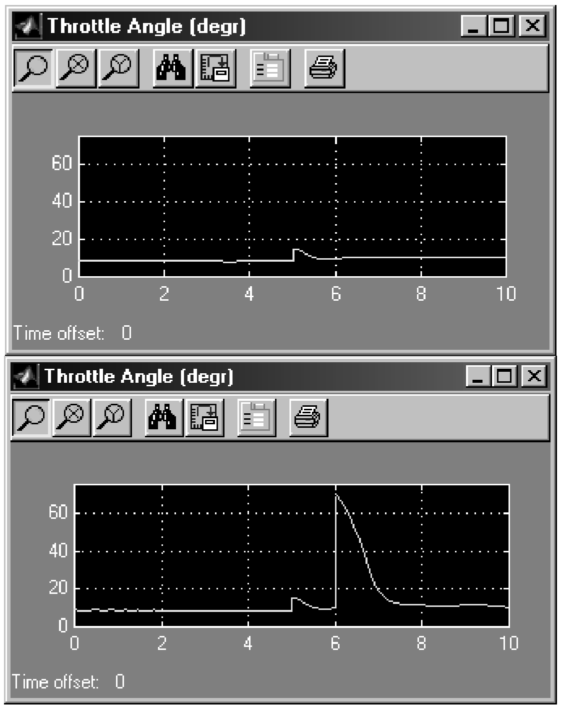

Figure 10. Fault-free output vs. undetected wrong result not detected by the assertions.

A detailed investigation of the severe undetected wrong results shows that no permanent value failures were observed for Algorithm II. However, since 0.17% of all errors still lead to severe undetected wrong results which were semi-permanent, additional research focusing on more sophisticated assertions capable of detecting the remaining errors is required. One explanation for the undetected wrong results produced by Algorithm II is shown in Figure 10. The figure shows the controller output when the state variable x changes from a correct value of approximately 10 degrees to an incorrect value of 69 degrees at time $t = 6$.

The assertions will not detect such errors since x is within the valid range (i.e. between 0.0 and 70.0 in our case). After approximately 1 second, the output stabilizes, but the output sequence will be classified as a severe undetected wrong result that is semi-permanent.

4.5. Comparison of results for Algorithm I and II

Table 4 shows a detailed comparison of the undetected wrong results (value failures) obtained for Algorithm I and II. The table shows that the percentage of severe value failures (permanent and semi-permanent) are reduced for Algorithm II while the percentage of minor value failures (transient and insignificant) increased. The reason for this is that the best effort recovery mechanisms of Algorithm II managed to detect many potential severe value failures resulting in minor value failures being produced instead. Thus, the total percentage of undetected wrong results is almost equal for Algorithm I and II (5.02% vs. 5.23%).

The percentage of permanent value failures is 0.12% for Algorithm I while no permanent value failures at all are observed for Algorithm II. The percentage of semi-permanent value failures decreased from 0.42% for Algorithm I to 0.17% for Algorithm II. This corresponds to a total reduction of the percentage of severe value failures, from 0.54% for Algorithm I to 0.17% for Algorithm II. Although, the results for Algorithm II are based on only 2372 injected faults vs. 9290 injected faults for Algorithm I, the corresponding 95% confidence intervals indicate that the total percentage of severe value failures is lower for Algorithm II.

Table 4. Comparison of results for Algorithm I and II.

	Results for Algorithm I			*Results for Algorithm II*		
	%	(95 % conf)	#	%	(95 % conf)	#
Total (Non Effective Errors)	**73,87%**	**(± 0,89%)**	**6863**	**72,85%**	**(± 1,79%)**	**1728**
Total (Detected Errors)	**21,11%**	**(± 0,83%)**	**1961**	**21,92%**	**(± 1,67%)**	**520**
Undetected Wrong Results (Permanent)	0,12%	(± 0,07%)	11	0,00%	(± 0,00%)	0
Undetected Wrong Results (Semi-Permanent)	0,42%	(± 0,13%)	39	0,17%	(± 0,17%)	4
Undetected Wrong Results (Transient)	0,94%	(± 0,20%)	87	1,56%	(± 0,50%)	37
Undetected Wrong Results (Insignificant)	3,54%	(± 0,38%)	329	3,50%	(± 0,74%)	83
Total (Undetected Wrong Results)	**5,02%**	**(± 0,44%)**	**466**	**5,23%**	**(± 0,90%)**	**124**
Total (Effective Errors)	**26,12%**	**(± 0,89%)**	**2427**	**27,15%**	**(± 1,79%)**	**644**
Total (Faults Injected)	**100,00%**		**9290**	**100,00%**		**2372**

5. Conclusions

We have demonstrated that bit-flips inside a central processing unit executing an engine control program can cause critical failures, such as permanently locking the engine's throttle at full speed. These failures were caused by errors that escaped several hardware implemented error detection mechanisms included in the CPU. Our fault injection experiments showed that 11% of the undetected errors leading to value failures seriously affected the control of the engine. These critical failures were caused by errors affecting the state variable of the control algorithm. By using software assertions and a best effort recovery mechanism, we managed to reduce the percentage of the critical failures to 3%. The control algorithm used in our experiments was a simple PI controller. Thus, we have demonstrated that software assertions in combination with best effort recovery can be very effective in reducing the number of critical failures for simple control algorithms. In our future research we will investigate the use of software assertions and best effort recovery techniques for multiple input and multiple output control algorithms such as jet-engine controllers.

Acknowledgements

We would like to thank Martin Fabian of the Dept. of Control Engineering, Chalmers University of Technology and Stefan Asserhäll and Thorbjörn Hult at Saab Ericsson Space for their many valuable suggestions and technical assistance. This work was supported by the Swedish Agency for Innovation (VINNOVA), the Swedish Network for Real-Time Research and Graduate Education (ARTES), The Swedish Foundation for Strategic Research (SSF) and Saab Ericsson Space.

References

[1] Y.C. Yeh, "Dependability of the 777 Primary Flight Control System", *in 5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, pp. 3-17, (Urbana Champaign, IL, USA) Sep. 1995.

[2] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components", *Automata Studies*, *Annals of Mathematical Studies*, Princeton University Press, No. 34, pp. 43-98, 1956.

[3] L. Lamport, P. M. Melliar-Smith, "Byzantine Clock Synchronization", in *Proc. Third ACM Symp. Principles of Distributed Computing*, August 1984, pp. 68-84.

[4] R. M. Kieckhafer, C. J. Walter, A. M. Finn, P. M. Thambidurai, "The MAFT Architecture for Distributed Fault Tolerance", *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 398-404, 1988.

[5] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", *Communications of ACM*, Vol. 34, No. 2, 1991, pp. 56-78.

[6] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", in *5th IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-5)*, pp. 267-287, (Urbana Champaign, IL, USA) Sep. 1995.

[7] G. C. Messenger, "Collection of Charge on Junction Nodes From Ion Tracks". IEEE Trans. on Nuclear Science, Vol. NS-29, No. 6, Dec. 1982, pp. 2024-2031.

[8] K. Johansson, P. Dyreklev, B. Granbom, M. C. Calvet, S. Fourtine, O. Feuillatre, "In-Flight and Ground Testing of Single Event Upset Sensitivity in Static RAMs", IEEE Transaction on Nuclear Science, vol.45 no.3 June 1998 pp. 1628-1632.

[9] E. Normand, "Single Event Upset at Ground Level", IEEE Transaction on Nuclear Science, vol. 43, no.6, December 1996, pp. 2742-2750.

[10] The MathWorks, Inc. "Using Simulink Version 3, Dynamic System Simulation for MATLAB" January 1999.

[11] Saab Ericsson Space AB, "Microprocessor Thor", Product Information, September 1993.

[12] P. Folkesson, S. Svensson, J. Karlsson, "A Comparison of Simulation Based and Scan Chain Implemented Fault Injection", *in Proc. 28th Int. Symp. on Fault-Tolerant Computing (FTCS-28)*, pp. 284-293, (Munich, Germany) June 1998.

[13] J. Aidemark, J. Vinter, P. Folkesson, J. Karlsson, "GOOFI: Generic Object-Oriented Fault injection tool", *Proceedings International Conference on Dependable Systems and Networks*, DSN 2001, Gothenburg, Sweden, July 2001.