

Impact of Soft Errors in a Brake-by-Wire System

Daniel Skarin, Martin Sanfridson, and Johan Karlsson

Abstract—This paper presents an experimental evaluation of the impact of soft errors in a brake-by-wire system for road vehicles. Our goal is to assess the risk that soft errors will cause the brake system to produce undetected dangerous outputs. To this end, we injected single bit-flips in the CPU registers and the main memory of a MPC565 microcontroller running a brake controller program. For each injected bit-flip, we recorded the outputs to the brake actuator for more than one thousand control cycles. The results show that 24% (769 of 3149) of the bit-flips injected in the CPU registers resulted in undetected erroneous outputs. Of these, 24% (188 of 769) resulted in dangerous outputs. The corresponding numbers for bit-flips injected in the main memory are 35% (985 of 2831), and 8.1% (80 of 985), respectively. Despite all injections were made into “live” registers or memory elements, 40% of the register bit-flips and 51% of the memory bit-flips had no impact at all on the output.

Index Terms—Brake-by-wire systems, error detection, fault injection, soft errors

I. INTRODUCTION

TO avoid accidents and prevent injuries caused by road vehicles, the automotive industry is developing new advanced active safety systems for collision avoidance and collision mitigation. Such systems use advanced sensors to detect situations where there is a high risk of a collision. If such a situation occurs, the system automatically brakes or steers the vehicle to avoid a collision or to reduce the forces of a potential impact.

As an active safety system can take control of the vehicle, a false or erroneous activation of the system can potentially cause an accident. Hence, active safety system must be highly reliable, fault-tolerant and fail-safe.

Because of the potentially severe consequences of an erroneous activation of an active safety system, and the large number of road vehicles in use, the probability for an

erroneous activation of an active safety system must be kept very low. It is therefore important to consider a wide range of failure mechanisms when designing the fault-tolerance features of an active safety system. In particular, it is important to consider the impact of soft errors, as they are expected to be a major cause of failures in future electronic circuits [1].

In this paper, we present results from an investigation of the impact of soft errors in an electronic braking system. This study constitutes the initial phase in a project where we intend to develop application-level software techniques for dealing with soft errors in active safety systems.

Our aim in this paper is to assess the risk that soft errors will cause the brake controller to produce dangerous outputs and to identify the parts of the controller that are the most sensitive to soft errors. We do so by injecting single bit-flip errors into CPU registers and the main memory of a MPC565 microcontroller running the brake controller program while monitoring the behavior of the system.

A secondary goal of our investigation is to study the impact of program-level error masking. Soft errors can be masked at the logic or microarchitectural level before becoming visible to a program [2], [3]. Soft errors that propagate and become visible to the program can further be masked by the program itself. To ensure that all the injected errors affected data that was indeed used by the program, we employed an automated pre-injection analysis technique [4] to avoid injections into “dead” registers. (A register is considered to be dead at a time, t , if it at that time holds data which will never be read again. Hence, soft errors that occur in a dead register cannot propagate.) Our pre-injection analysis technique ensures that the errors are injected in registers and memory elements immediately before they are read by the brake controller program.

II. BRAKE-BY-WIRE SYSTEM

A. Brake-by-Wire Controller

We used a prototype brake-by-wire controller, shown in Fig. 1, developed by Volvo Technology. The brake controller internally consists of three controllers: a base controller, a reset controller, and an ABS controller. At any time instance, the system selects one of these controllers to provide the output to the brake actuator. The selection of the controller is based on the current wheel speed and brake pedal position. The reset controller is selected when the brake pedal is not

This research has been conducted within the project CEDES, which is funded by the Swedish industry and government joint research program IVSS – Intelligent Vehicle Safety Systems.

Daniel Skarin is with the Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (+46 31 772 00 00; fax +46 31 772 36 63; e-mail: skarin@ce.chalmers.se).

Martin Sanfridson is with Volvo Technology, SE-405 08 Göteborg, Sweden (e-mail: martin.sanfridson@volvo.com).

Johan Karlsson is with the Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (e-mail: johan@ce.chalmers.se).

pressed. The base controller is used when the brake pedal is pressed and the wheel speed is low. Otherwise, the ABS controller is used. The brake command produced is sent to a brake actuator that applies the desired brake force to a wheel.

The ABS controller, shown in Fig. 2, uses a PI-controller whose output consists of the sum of two terms, one which is proportional to the current control error (the P part) and one proportional to the control error integrated over time (the I part). An anti-windup function [5] is used to limit the output of the integrator when the actuator output is saturated.

B. Experimental Setup

The experimental setup consisted of a brake system emulator and an error injection system, see Fig. 3. The brake system emulator contained two computer nodes. One of the nodes executed the brake controller program, while the other executed an environment simulation model that included models of a wheel, a brake pedal, a wheel speed sensor and a brake actuator. The two nodes exchanged sensor readings and actuator commands via a CAN-bus. The nodes were single board computers from PHYTEC [6] based on the MPC565 microcontroller from Freescale. The brake system emulator executes the brake controller at reduced speed to allow time for error injection and to collect the internal state of the controller program and the state of the environment model after each invocation of the control loop. The execution speed of the brake system emulator is approximately one thousand times slower than in a real system, which executes the control loop at 500 Hz.

The brake controller and the environment models were developed using Simulink and Matlab from MathWorks, Inc [7]. To generate C code from the Simulink models, we used the TargetLink code generator from dSPACE GmbH [8]. Code for communication and scheduling was added to the generated code to create a working system. Both nodes had their programs stored in the internal FLASH memory of the MPC565 microcontroller.

We injected errors into the brake controller node via the Nexus port [9] of the MPC565 microcontroller, using GOOFI [10], a fault/error injection tool developed in our research

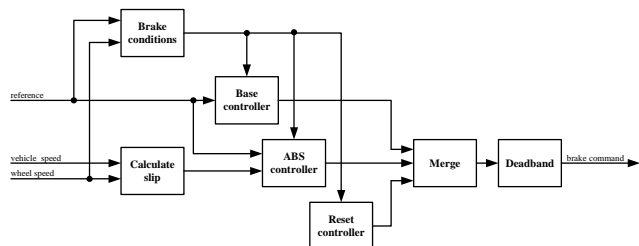


Fig. 1. Brake controller overview.

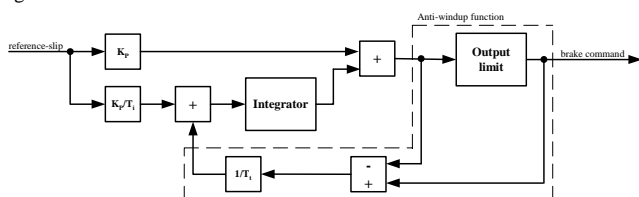


Fig. 2. ABS controller overview.

group. GOOFI accessed the Nexus port using a commercial debug environment from iSYSTEM [11] consisting of an iC3000 emulator and winIDEA, an integrated software development environment.

Single bit-flip errors were injected into the data area in the main memory, and into all CPU-registers used by brake controller program. (We did not inject errors in the program area since the program was stored in FLASH memory.) After an error was injected, GOOFI used the Nexus port to collect the state of the brake controller for each invocation of the control loop.

In order to uniquely identify the injected errors, we defined them in terms of a time-location pair. The location was a bit in a register or a memory element, while the time corresponded to a certain execution of an instruction, which was identified by the instruction's memory location and an invocation count. The time-locations pairs were selected using OFFSET [4], a pre-injection analysis tool. OFFSET selects the time of the injection so that errors are only injected in a register or a memory element immediately before it is read. This avoids injecting errors into "dead" registers, as explained in the introduction. For each injected error, we executed the control program for about 1900 control cycles. The errors corresponded to randomly selected time-location pairs between the control cycles 500 and 750 corresponding to 1.0s and 1.5s of real-time execution. (Time-locations pairs corresponding to a bit in a dead register were excluded from the random selection by the OFFSET tool.)

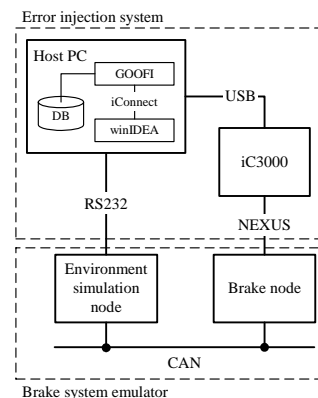


Fig. 3. Experimental platform.

III. RESULTS

This section presents the results from the fault injection experiments. We cover the following aspects of the results: *error impact classification, detections by hardware exceptions, latency for erroneous outputs, critical failures, stack pointer errors, and program-level error masking.*

A. Error Impact Classification

A classification of the error impact is shown in Table I. Outcomes of the experiments have been classified in four groups depending on how the brake controller was affected by the injected error:

TABLE I
FAULT INJECTION RESULTS

Error injected in	No. of experiments	Non-effective	Detected by HW exception	Undetected	System hang
CPU registers	100.00% 3149	39.70% 1250	32.99% 1039	24.42% 769	2.89% 91
Control loop	2562	1037	831	671	23
Scheduler	205	58	37	78	32
Communication	235	88	110	12	25
C library functions	147	67	61	8	11
Main memory	100.00% 2831	51.47% 1457	10.67% 302	34.79% 985	3.07% 87
Control loop	2404	1249	257	859	39
Scheduler	112	9	1	80	22
Communication	179	126	26	21	6
C library functions	136	73	18	25	20
Total	100.00% 5980	45.27% 2707	22.42% 1341	29.33% 1754	2.98% 178

- *Non-effective* – Only correct outputs were observed.
- *Detected* – The error was detected by a hardware exception.
- *Undetected* – An erroneous brake command was produced.
- *System hang* – No output was produced for more than one second.

A majority of the errors injected in registers did not cause erroneous outputs, 40% were non-effective and 33% were detected by hardware exceptions. However, a significant percentage of the register errors, 24%, caused the brake controller to produce erroneous outputs. Among the errors injected in memory were 51% non-effective, compared to 40% of the register errors. Hardware exceptions were less efficient in detecting memory errors than register errors, only 11% were detected compared to 33% of the register errors. Memory errors also caused erroneous outputs more often than register errors, 35% compared to 24%.

TABLE II
DISTRIBUTION OF DETECTIONS AMONG HARDWARE EXCEPTIONS

Exception	Register errors	Memory errors	Total
DPI	0.19% 2	1.32% 4	0.45% 6
DTLBER	2.98% 31	0.00% 0	2.31% 31
SEE	0.96% 10	0.66% 2	0.89% 12
FPASE	26.37% 274	79.14% 239	38.26% 513
ALE	5.20% 54	0.00% 0	4.03% 54
MCE	64.29% 668	18.87% 57	54.06% 725
Total	100.00% 1039	100.00% 302	100.00% 1341

TABLE III
UNDETECTED OUTPUTS LEADING TO CRITICAL FAILURES

Error injected in module	Register errors	Memory errors	Total
Control loop	20.42% 157	2.54% 25	10.38% 182
Scheduler	3.64% 28	4.26% 42	3.99% 70
Communication	0.39% 3	1.32% 13	0.91% 16
Total	24.45% 188	8.12% 80	15.28% 268

B. Detections by Hardware Exceptions

The distribution of detections among hardware exceptions is shown in Table II. A detailed explanation of the hardware exceptions is given in [12]. In no case did the system produce an erroneous output when the error was detected by a hardware exception.

C. Latency for Erroneous Outputs

The error latency, expressed as the number of control cycles from the time an error was injected until an erroneous brake command was produced, is shown in Fig. 4. A majority of the erroneous outputs, 64 %, were produced in the same control cycle as the error was injected in.

D. Critical Failures

Fifteen percent of the errors causing erroneous brake commands to be produced resulted in a critical failure. A critical failure was defined as the wheel being locked or a loss of braking. The wheel was considered locked if the wheel speed was zero or close to zero for more than 0.03s. Similarly, if the wheel speed did not decrease during 0.03s, the error was considered to have caused a loss of braking. Experiments classified as critical failures are shown in Table III.

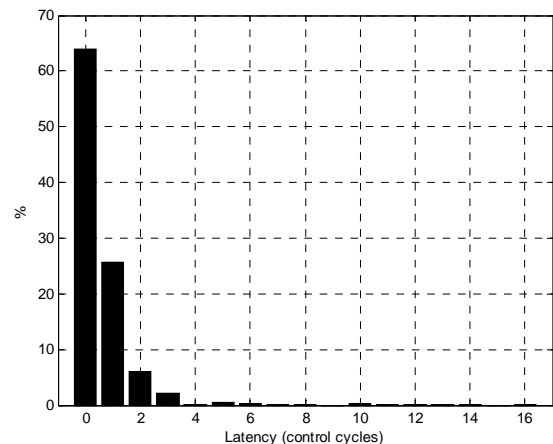


Fig. 4. Latency for erroneous outputs.

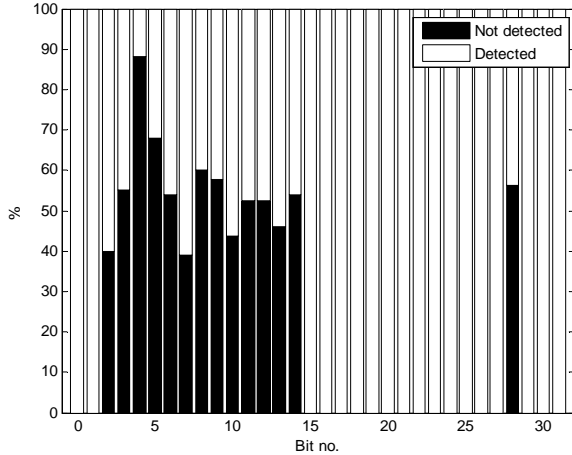


Fig. 5. Outcome of errors injected into the stack pointer.

A majority of the errors injected in the control loop resulting in a critical failure were injected into the stack pointer (133 of 182 errors). Errors resulting in an incorrect integrator state in the ABS controller were the second largest reason for critical failures (36 of 182 errors). None of these errors were injected directly into the register, or the memory location, holding the integrator state. An incorrect value, caused by an error, was used in calculations which resulted in an incorrect update of the integrator state, or in an erroneous brake command. An erroneous brake command that saturated the actuator caused the error to propagate to the integrator state via the anti-windup function. The creation or restoring of the stack frame was affected by 13 errors which resulted in critical failures.

A majority of the errors injected in the scheduler resulted in certain software modules not being executed (51 of 70 errors). Eleven errors were injected into a system clock, used by the integrator in the ABS controller, and eight errors were injected into the stack pointer. The creation or restoring of the stack frame was affected by a majority of the errors injected in the communication module (13 of 16 errors). The stack pointer was affected by three errors. All of these errors were classified as critical failures.

E. Stack Pointer Errors

Fig. 5 shows the outcome of errors injected into the stack pointer. These errors were either detected by hardware exceptions or produced erroneous outputs. Fig. 6 shows the different hardware exceptions that detected an erroneous stack pointer. The least significant bit is numbered bit 0.

F. System Hangs

About 3% of the injected errors resulted in a system hang. A majority of these errors caused a jump to the end of the brake controller program, or resulted in a communication failure. The environment simulator model crashed as a result of some of the injected errors. This caused the brake controller program to enter an endless loop waiting for sensor values. These experiments were not representative of a real system.

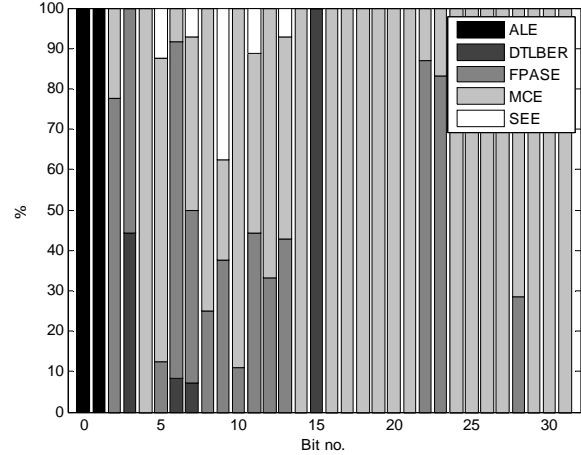


Fig. 6. Hardware exceptions detecting errors injected into the stack pointer.

G. Program-Level Error Masking

We performed an analysis of the error propagation to better understand the reasons for the large number of non-effective errors. The result of this analysis for errors injected in the control loop is shown in Fig. 7.

Instructions were divided into five instruction groups: memory load, memory store, arithmetic and logic, conditional branch, and miscellaneous instructions. An error was assigned to one of these instruction groups depending on at which instruction an error was injected at. This is shown by the percentage listed in each state. The execution of this instruction resulted in an erroneous value. The register, or the memory element, holding this erroneous value was determined using information from a disassembly of the brake controller program. A program trace, created by recording executed instructions during a fault-free experiment, was used together with the disassembly to determine the next instruction that used the erroneous value. This instruction was assigned to an instruction group, and a transition was made to this group from the previous instruction group. The analysis stopped when a conditional jump, or a return from the current function, was encountered.

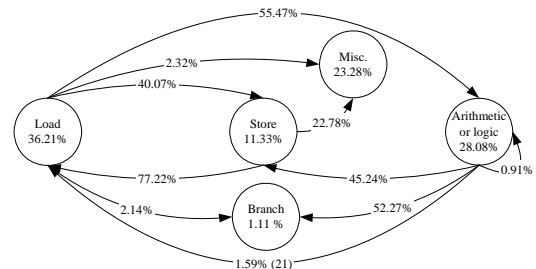


Fig. 7. Propagation of non-effective errors injected in the control loop.

IV. DISCUSSION

A. Program-Level Error Masking

Almost half of the injected errors, 45%, were non-effective even though the errors were injected in registers and memory elements containing live data. This is a measure of the

program-level error masking of the brake controller program.

There are several factors that can affect the error masking in a program. Fig. 7 shows that many non-effective errors affected branches while they propagated. If a branch is the only instruction affected by an error, the branch condition can mask the error. Even if the error results in an incorrect execution path, the program behavior may still be unaffected [13]. Another factor that can affect program-level error masking is dead and transitively dead instructions [14]. Dead instructions are instructions that produce results that are not used, e.g. a useless load, and transitively dead instructions generate results that are only used by dead instructions, e.g. calculation of an address used for a useless load. Errors were selected to avoid dead registers and memory elements but the pre-analysis does not consider dead or transitively dead instructions.

B. Errors Detected by Hardware Exceptions

Hardware exceptions were capable of detecting many of the injected errors, especially register errors (33% of register errors were detected compared to 11% of memory errors). A majority of the detected errors triggered a machine check exception (MCE). The MCE, or in some cases a data protection error exception (DTLBER), is caused by accessing invalid memory. Table II shows that a MCE was triggered more often for register errors than for memory errors, 64% compared to 19%. Registers are used for addressing memory and a register error is therefore more likely to result in an access to an invalid memory element than an error injected in memory.

The second most common exception triggered by errors was a floating point assist exception (FPASE). The MPC565 depends on an additional software routine to fully implement IEEE floating-point numbers. A FPASE is triggered when the floating point unit needs the assistance of the software routine to deliver a result compliant with the IEEE standard. An error that causes a denormalized value in a register, or memory location, can trigger a FPASE when the erroneous value is used by an instruction. The brake controller uses floating point variables for most of the calculations and an error in one of these variables can result in a FPASE caused by a denormalized value.

C. Stack Pointer Errors

Errors injected in the stack pointer were not always detected by hardware exceptions, erroneous outputs were instead produced. The outcome of an error in the stack pointer depends on the bit position affected by the error, see Fig. 5. Errors injected in certain bits were always detected by exceptions while errors injected in other bits could result in erroneous outputs being produced.

Hardware exceptions will detect an erroneous stack pointer that points to invalid memory. In our system, errors injected in bits 15 to 27, or bits 29 to 31, of the stack pointer resulted in accesses to invalid memory and were therefore detected. Errors injected in bit 0 or 1 resulted in memory accesses that

were not word-aligned, therefore detected by an alignment exception (ALE).

An erroneous stack pointer that points to valid memory was not always detected by hardware exceptions. Whether an erroneous stack pointer was detected or caused erroneous outputs to be produced depended on the time instant when the error was injected.

D. Undetected Outputs

An erroneous brake command was produced as a result of 29% of the injected errors. Fifteen percent of these resulted in outputs leading to a critical failure. The following reasons for the critical failures were identified from the experiments:

- Errors affecting the integrator state were not always compensated for by the control algorithm itself.
- Errors resulting in a saturated output propagated to the integrator state via the anti-windup function.
- Errors in the stack pointer, or errors affecting the stack frame, had a large impact on the system.

A PI-controller, with input signal $e(k)$ and actuator output $u(k)$, can be described with the pseudo-code shown below.

$$\begin{aligned} v(k) &= K_p * e(k) + I(k) \\ u(k) &= \text{sat}(v(k), \text{max}, \text{min}) \\ I(k+1) &= I(k) + (K_p * h / T_i) * e(k) + (h / T_r) * (u(k) - v(k)) \end{aligned}$$

The control signal, $v(k)$, is the sum of the P part, the input signal scaled with a control parameter K_p , and the I part, the integrator state. The function sat ensures that the actuator output is within the working range of the actuator. The integrator state is updated by adding the P part scaled with a control parameter T_i and sample time, h , to the previous integrator state. If the produced output results in a saturated actuator output (i.e. $v(k)$ is outside a valid range for the actuator), the anti-windup function tries to recalculate the integral to produce an actuator output at the saturation limit in a future control cycle. This is accomplished by feeding the difference between the control signal and the actuator output, the term $(u(k) - v(k))$, back to the integrator through a low-pass filter with a time constant T_r . Errors affecting the P part of the output, resulting in a saturated actuator output, will therefore propagate to the integrator state via the anti-windup function.

An incorrect controller state may not be visible at the output until several control cycles after an error has been injected (see Fig. 7). An incorrect internal controller state can therefore not always be identified immediately by only observing the controller output.

A memory with error-correcting codes (ECC) can reduce the number of failures in the system, but, as shown in Table I, 24% of the errors injected in registers resulted in an incorrect brake command. Register errors were also more likely to result in outputs causing a critical failure than memory errors. A memory with ECC is therefore not sufficient, it is necessary to also protect registers, or to use application-level techniques for dealing with errors.

V. RELATED WORK

Several fault injection techniques have been presented in the literature (an overview is presented in e.g. [15]) and fault injection has been used in many studies to investigate how computers are affected by errors.

Cunha et al. investigated if a controller for an inverted pendulum system could compensate for, not only external disturbances affecting the inverted pendulum, but also errors in the controller itself [16]. The authors showed that the feedback used in many control systems allowed the controller to compensate for errors inside the controller. It was also shown that a single erroneous output is not significant, but a sequence of erroneous outputs is.

The effects of transient data errors in a controller were modeled by Gäfvert et al. in [17]. The use of artificial limits on signals combined with an anti-windup scheme is suggested to reduce the effects of data errors.

The impact of soft errors in an engine control system was evaluated by Vinter et al. in [18]. The control algorithm was capable of compensating for many of the effects of a soft error without mechanisms for error detection and recovery. However, some of the errors resulted in severe system failures. The state variable of the controller was identified as a particularly vulnerable part of the system.

The effects of transient faults on a high performance microprocessor were explored by Wang et al. in [2]. The authors investigated the propagation of errors through different abstraction levels and found that 85% of the errors were masked before becoming visible to software. For errors that propagated to the architectural layer, and thus became visible to software, around half of them were masked by software.

Results on program-level error masking is also presented in [3] by Saggese et al. The authors studied the impact of soft errors in a microprocessor for embedded systems. Their results showed that about 53% of the errors that propagated to the application level were masked by software.

The results presented in [16]-[18] are consistent with the results in our study, a majority of errors affecting a control system is compensated for by the controller itself. Also, the most sensitive parts of a controller are its state variables. The robustness of our brake controller might be improved by using another anti-windup scheme. The results for program-level error masking presented in [2] and [3] are similar to the result obtained in our study, about half of the injected errors did not have an effect on our system.

VI. SUMMARY AND FUTURE WORK

We have experimentally evaluated the impact of soft errors in a brake-by-wire system. Single bit-flips have been injected in registers and the main memory of a microcontroller running a prototype brake controller. The fault injection experiments showed that a single bit-flip could have very severe effects. Of the injected errors, 29% resulted in undetected erroneous brake commands being produced. About 15% of these

(corresponding to 4.5% of all injected errors) resulted in outputs causing the wheel to lock or a loss of braking. The anti-windup function and integrator state were identified as particularly vulnerable parts of the brake controller.

A pre-injection analysis was used to avoid injecting errors in dead registers or memory elements. Despite this, 45% of the injected errors did not have an effect on the brake controller output. The percentage can be seen as a measure of the program-level error masking of the brake controller program.

We will extend the brake controller with software-implemented mechanisms for error detection and recovery in our future work. The effectiveness of these mechanisms will be evaluated using fault injection experiments. The impact of soft errors in advanced active safety systems will also be investigated.

REFERENCES

- [1] R.C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Trans. Device and Materials Reliability*, vol.5, no.3, pp. 305- 316, Sept. 2005
- [2] N. Wang, J. Quek, T.M. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proc. International Conference on Dependable Systems and Networks (DSN 2004)*, pp. 61-72, 2004
- [3] G.P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer, "Microprocessor sensitivity to failures: control vs. execution and combinatorial vs. sequential logic," in *Proc. International Conference on Dependable Systems and Networks (DSN 2005)*, pp. 760-769, 2005
- [4] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *Proc. Fifth European Dependable Computing Conference*, Budapest, Hungary, 2005
- [5] C. Edwards and I. Postlethwaite, "Anti-windup and bumpless transfer schemes," *Automatica*, vol. 34, no. 2, pp. 199-210, Feb. 1998
- [6] PHYTEC, <http://www.phytec.com/products/sbc/PowerPC/phyCORE-MPC565.html>, January 2, 2006
- [7] MathWorks, Inc, <http://www.mathworks.com>, March 23, 2007
- [8] dSPACE GmbH, <http://www.dsapce.de>, March 23, 2007
- [9] IEEE-ISTO, The Nexus 5001 Forum™ Standard for a Global Embedded Processor Debug Interface, 2003
- [10] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: Generic Object-Oriented Fault Injection Tool," in *Proc. International Conference on Dependable Systems and Networks (DSN 2003)*, Göteborg, Sweden, 2001, pp. 83-88
- [11] iSYSTEM, <http://www.isystem.com>, December 28, 2006
- [12] Freescale semiconductor, "MPC565 reference manual," revision 2.2, 2005
- [13] N. Wang, M. Fertig, and S. Patel, "Y-Branches: When you come to a fork in the road, take it," in *Proc. International Conference on Parallel Architectures and Compilation Techniques*, pp. 56-66, 2003
- [14] J.A. Butts and G. Sohi., "Dynamic dead-instruction detection and elimination," in *Proc. 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 199-211
- [15] M. Hsueh, T. Tsai, and R.K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75-82, Apr. 1997
- [16] J.C. Cunha, R. Maia, M.Z. Rela, and J.G. Silva, "A study of failure models in feedback control systems," in *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, pp.314-323, 2001
- [17] M. Gäfvert, B. Wittenmark, and Ö. Askerdal, "On the effect of transient data-errors in controller implementations," in *Proc. Of the 2003 American Control Conference*, no.pp. 3411- 3416, June 2003
- [18] J. Vinter, J. Aidemark, P. Folkesson, and J. Karlsson, "Reducing critical failures for control algorithms using executable assertions and best effort recovery," in *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, Göteborg, Sweden, 2001