Software Mechanisms for Tolerating Soft Errors in an Automotive Brake-Controller

Daniel Skarin and Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{*skarin, johan*}*@chalmers.se*

Abstract

This paper describes the design and evaluation of two software implemented error detection and system recovery mechanisms that protect a prototype brake-by-wire controller from soft errors. We used an evaluation driven design process to develop the software mechanisms, which are specifically designed to prevent soft errors from causing critical failures in the brake controller. The design process involves 1) identifying vulnerable parts of the brake controller, 2) designing and verifying software mechanisms for error detection and recovery, and 3) performing an extensive evaluation of the proposed mechanisms. Results from error injection experiments in the last step show that our simple software mechanisms, combined with hardware exceptions for error detection, can effectively reduce the number of critical failures caused by soft errors in the brake controller.

1. Introduction

It is well known that *soft errors* caused by ionizing particles, such as cosmic high energy neutrons and alpha particles, pose an important reliability problem in modern microprocessors. They are in fact considered to be a dominating source of physical failures in modern commercial circuits used in ground-based applications [1]. Current high-end microprocessors manufactured in 45 nm and 65 nm technologies are therefore provided with extensive circuit-level and microarchitectural-level mechanisms that mitigate the effects of soft errors.

While these mechanisms can mask and recover from a vast majority of the soft errors, they rarely achieve 100% error coverage. For example, proton radiation tests of the IBM Power6 processor show that it achieves a soft error rate (SER) derating factor of 500 times [2], which means that in average one soft error in five hundred reaches registers in the processor's instruction set architecture (ISA).

Future microprocessors may achieve even higher SER derating factors, but it is unlikely that they will provide 100% error coverage, since this would be very costly both in terms of design effort and area overhead. Moreover, the soft error rate per chip is expected to increase significantly for each new technology generation [3], mainly because there will be more transistors on each chip.

For these reasons, we can expect that soft errors will remain an important reliability problem for a wide range of future computer applications. They have for a long time been a major reliability concern in space and aeronautical applications due to the high flux of ionizing particles present in these environments. In ground-based applications, soft errors are today mostly a concern for applications that use high-end microprocessor manufactured in technologies with feature sizes of 90 nm and below. However, since processors and microcontrollers targeting the embedded systems market are likely to be manufactured in such technologies in the future, we believe that soft errors will become an important concern also for ground-based embedded safety-critical applications.

One way to deal with soft errors that propagate to ISA-registers is to use a classical TMR (triple modular redundancy) system, i.e., to run programs in three processors and vote on the results produced by each processor. This is obviously a costly solution in terms of hardware and can therefore only be used in applications that require extreme levels of data integrity, availability, or safety. Another possibility is to run the programs in two processors (or two processor cores) and compare the results to detect errors, and then use software implemented recovery to restore correct operation. This solution may be an interesting

alternative for some applications, but would still be too expensive for many other. A third less costly alternative is to rely on hardware exceptions for error detection, and combine these with software implemented error detection and recovery techniques. This is an attractive solution for highly cost-sensitive applications, such as electronic systems in cars and other road vehicles. However, a drawback of this approach is that it must be very carefully validated since its effectiveness (error coverage) may vary strongly depending on the nature of the program and its activation patterns.

In this paper, we describe results from an experimental evaluation of a prototype brake controller program that uses the above mentioned low-cost approach to handle soft errors. The brake controller has been specifically designed to be included in a future brake-by-wire system with no hydro-mechanical backup. As the brake controller is the only unit that can send brake commands to the brake actuator in such a system, it must be more reliable and thereby achieve a higher degree of error tolerance than conventional ABS-controllers, which can be shut down without serious consequences.

We have conducted error injection experiments with two versions of the brake controller, one with software implemented error handling mechanisms and one without such mechanisms. These mechanisms were designed and constructed using an evaluation driven design process which we describe in Section 2. The error handling mechanisms are described in Section 3, while Section 4 presents the results of the error injection experiments. Section 5 concludes the paper.

2. Design Process

The software implemented error handling mechanisms were developed using an evaluation driven design process that involved three major steps. First, we conducted error injection experiments with a basic version of the brake controller that was not provided with any software error handling mechanisms. Second, based on an analysis of the results from the first step, a set of software implemented error detection and recovery mechanisms were designed with the aim of enhancing the error coverage of the brake-controller system. Third, we implemented a new version of the controller program containing the software mechanisms developed in the previous step, and then conducted an extensive error injection campaign to evaluate the effectiveness of these mechanisms. Both versions of the brake controller programs implement a PI-controller that optimizes the tire slip to achieve maximum brake performance.

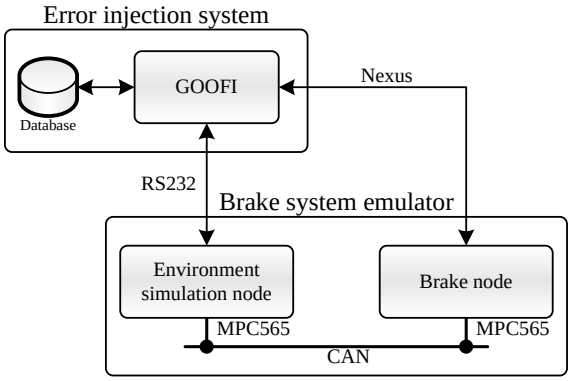

Figure 1. Experimental set-up.

For the error injection experiments, we used a set-up consisting of a brake system emulator and an error injection system, see Figure 1. The brake system emulator consists of two computer nodes, which are based on Freescale's MPC565 microcontroller. One node executes the brake controller program and the other node executes an environment simulator that simulates the dynamics of a wheel. We used an extended version of GOOFI [4], a generic fault/error injection tool developed in our research group, to randomly inject single bit-flip errors into CPU-registers and memory locations used by the brake controller program.

GOOFI injects errors into the brake node via the processor's debug port, which is also used to observe and log program variables and the state of the brake node. Values sent and received by the environment simulation node are logged by GOOFI via a serial connection. The set-up allows us to inject errors with full controllability and repeatability. To ensure that errors only were injected into "live" data, GOOFI was configured to inject them immediately before the target register or the target memory location was read by a machine instruction. This procedure avoided injection of errors into data that were not used by the brake controller program.

The experiments conducted in the first step of the design process had three major objectives: i) to identify errors that were not detected by the processor's hardware exceptions, ii) to identify those undetected errors that led to catastrophic control failures, and iii) to determine in what program variables these errors had been injected. To achieve these objectives, we had to observe and log a large number of program variables, which caused a significant time overhead in conducting the experiments. It took about 15 minutes to inject one error and monitor the behavior of the brake system. From these experiments, we noted that

a large proportion of the critical failures were caused by errors in either the stack pointer or the integrator state of the brake controller [5].

As already mentioned, the aim of the second step was to design software implemented error detection and system recovery mechanisms that could enhance the error coverage of the brake-controller system. More specifically, the aim of these mechanisms was to minimize the occurrence of critical control failures. Thus, for the purpose of this work, we define the error coverage as *the probability that a single bit error affecting live data in a CPU-register or a memory location does **not** cause a critical control failure*.

It is well known that closed-loop control systems, such as a brake controller, often compensate for incorrect outputs caused by soft errors even if no error detection and recovery mechanisms are activated [6]. Thus, for closed-loop control systems it makes sense to distinguish between *benign failures* and *critical failures*. Benign failures have none or only a minor impact on the controlled object, and hence there is no need for the control system to detect or compensate for errors that cause such failures. Critical failures, on the other hand, result in a functional failure of the control system. A control system that exhibits only benign failures is known to be fail-bounded [7]. Depending on the application, a fail-stop or fail silent behavior may be considered either as a benign failure or as a critical failure. According to Papadopoulos et. al [8] critical failures for a wheel braking function can be classified into three major categories: *loss of braking*, *unintended braking*, and *permanently locked wheel*.

In general, software mechanisms for error detection can be designed using application independent techniques such as duplication and comparison, or by using application-specific techniques that rely on knowledge about the application. For the brake controller, we designed two simple software checks, one for the stack pointer and one for the controller's integrator state [9]. The stack pointer check relies on duplication and comparison, while the check of the integrator state relies on knowledge of the maximum change of the integrator state between two samples. In addition, we implemented two methods for system recovery. For errors affecting the integrator state, we rollback the integrator state to a previous state (fast recovery). For errors in the stack pointer, we need to restart the program (slow recovery). These mechanisms are further described in Section 3.

The second step of the design process also involved the use of model checking to verify that the software error handling mechanisms worked as intended. A model checker such as SPIN [10] can verify or provide counterexamples of safety and liveness properties of formal models of concurrent processes, which makes it suitable for verification of error handling mechanisms. We created models of the error handling mechanisms which were verified by SPIN. This allowed us to verify the correctness of the error handling mechanisms before we implemented them in the brake controller program.

In the third step of the design process, we made an extensive evaluation of the error coverage of both versions of the brake-controller system in order to assess the effectiveness of the software mechanisms. To this end, we injected single bit-flip errors exhaustively in all bits in all CPU-registers as well as in all bits in the data area of the main memory, which were used by the program during one control loop iteration. We injected errors in this manner for three carefully selected control loops for each program version, see [11].

In these experiments, we only needed to observe and log variables in the brake controller program when an error was injected and at the end of each run. Thus, these experiments were far less time-consuming than those conducted during the first step of the design process, where we logged a large number of program variables for every iteration of the control loop. The error injection experiments conducted in the third step were about 60 times faster than the ones conducted during the first step.

3. Software Mechanisms for Error Detection and Recovery

This section describes the software error handling mechanisms in more detail. The development of the mechanisms involved i) finding suitable techniques for error detection and recovery and ii) verifying the correctness of the proposed mechanisms.

Results from our initial error injection experiments showed that a large number of the critical failures were caused by errors affecting either the stack pointer or the integrator state of the controller. As previously mentioned, the brake controller is designed to control the tire slip, i.e., the difference between a wheel's linear velocity and the vehicle's velocity, using a PI-controller.

The output from such a controller is the sum of two terms, one proportional to the current control error, the P-part, and one proportional to the control error integrated over time, the I-part. Soft errors affecting the P and the I-part can, as mentioned earlier, automatically be compensated for by the controller. However, errors that cause large deviations in the integrator state

from its correct value may not be compensated for by the brake controller fast enough. In our experiments, we saw that the actuator command could be saturated for several seconds, which resulted in a critical brake controller failure.

To cope with such errors, we designed a rate limit to detect errors affecting the integrator. The update of the integrator state can be described as:

$$I(k+1) = I(k) + h \times input(k),$$

where *I(k)* denotes the integrator state, *h* the sample time, and *input(k)* the integrator input (*k* denotes the current sampling point.) The inputs to the brake controller are bounded by factors such as the operational range of A/D-converters and physical limits, and the sampling time is constant. Therefore, we could calculate the maximum amount that the integrator state can change during a control loop iteration, and use that upper bound as a rate limit for error detection. We also designed a check to detect specific floating point values that were not handled correctly by the integrator. These values, which include infinity and not a number [12], are detected using a simple bit test on the integrator state.

The integrator state is rolled back to a previous state when an error is detected. This recovery is non-perfect and can therefore cause the integrator state to deviate from that of a fault-free integrator. Such a deviation is however compensated for by the control algorithm in subsequent samples.

Errors in the stack pointer are detected using duplication and comparison. A copy of the stack pointer is stored in a CPU-register immediately before each function call. When the execution returns from the called function, the values of the stack pointer and the copy are compared. If the two values should differ, we reinitialize the MPC565 microcontroller and all program variables, and restart the brake controller program.

The correctness of the error detection and recovery mechanisms was verified using model checking. We used SPIN to verify the following property expressed in Linear Temporal Logic (LTL) [10]:

$$\Box(error \Rightarrow \Diamond correct\ recovery).$$

The LTL formula states that whenever an error occurs, the system should eventually recover from the error. For errors affecting the integrator, we defined correct recovery as having an integrator state which contains a non-faulty value. For errors affecting the stack pointer, we assumed that errors will be recovered by the restart of the brake controller program, and only required that a faulty stack pointer was detected.

Model checking helped us to identify a flaw in an early design of the error detection and recovery mechanisms for the integrator, where an error was not recovered correctly. The mechanisms for the stack pointer and the integrator state were modeled separately, along with an error injector that changed program variables to faulty values. For errors affecting the integrator, SPIN found a counterexample where the mechanisms failed to recover the integrator state to a non-faulty value. It should be noted that the models define abstractions of the software mechanisms, and the above mentioned property may not hold for the implementation of the mechanisms.

4. Results

Results from the extensive evaluation conducted during the third step of the design process show that the software implemented mechanisms reduced the number of critical failures from 1.2% to less than 0.05%. For the basic version of the brake controller (the one without software error handling mechanisms) we observed approx. 1000 errors (of 88.000) that led to a critical behavior of the brake controller, such as locking the brake or not applying brake force for an extensive time. For the extended version of the brake controller we observed only 39 errors (of 93.000) that led to a critical failure. Almost all of these critical failures were caused by errors that were injected into the program counter (36 out of 39 errors.)

As much as 56% of the errors caused the extended version of the brake controller to produce incorrect outputs that had a negligible (non-critical) impact on the braking performance. That is, for these errors the controller exhibited a fail-bounded behavior. Despite that we injected all errors into registers holding "live" data (i.e., data used by the program), as much as 36% of the errors were masked by the program and did not in any way affect the produced brake commands.

5. Conclusions

This paper describes the design of two simple software implemented mechanisms that have been designed specifically to prevent soft errors from causing critical failures in a prototype brake controller. Using error injection experiments, we have demonstrated that the software implemented error detection and recovery mechanisms, combined with hardware exceptions for error detection, effectively can enforce fail-bounded semantics for a brake controller with respect to soft errors that manifest as single bit-flip errors in CPU-registers and the data area of the main memory.

While these results provide strong evidence that simple software implemented error detection and system recovery mechanisms can help to achieve a very high error coverage with respect to soft errors in control applications, they are not fully conclusive. First of all, it is well known that error coverage achieved by simple checking mechanisms is very sensitive to implementation details. Hence, small changes in the program or its activation patterns may change the error coverage significantly.

Also, our evaluation methodology has two obvious drawbacks. One is that we use single bit-flips errors evenly distributed over all "live" data in memory and CPU-registers to assess the error coverage; in reality many soft errors will manifest as multi-bit errors in CPU-registers, and they may not be evenly distributed. The second drawback is that our methodology does not consider out-of-specification behaviors of the microcontroller as we cannot inject errors in internal registers of the microarchitecture. Such behaviors can arise when soft errors affect the processor's internal control logic. Assessing the accuracy of our error injection methodology is a topic of future research, which would require the use of accurate microarchitectural simulations or particle radiation experiments.

References

[1] R. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sep. 2005.

[2] J. Kellington, R. McBeth, P. Sanda, and R. Kalla, "IBM®POWER6™Processor Soft Error Tolerance Analysis Using Proton Irradiation," *IEEE Workshop on Silicon Errors in Logic (SELSE 3)*, Apr. 2007.

[3] S. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[4] 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 2001)*, Jun. 2001, pp. 83–88.

[5] D. Skarin, M. Sanfridson, and J. Karlsson, "Impact of soft errors in a brake-by-wire system," in *IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE 3)*, 2007.

[6] J. Cunha, R. Maia, M. Rela, and J. Silva, "A Study of failure models in feedback control systems," in *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, Jul. 2001, pp. 314–323.

[7] J. G. Silva, P. Prata, M. Rela, and H. Madeira, "Practical issues in the use of abft and a new failure model," in *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, 1998, pp. 26–35.

[8] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner, "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure," *Reliability Engineering and System Safety*, vol. 71, no. 3, pp. 229–247, 2001.

[9] D. Skarin and J. Karlsson, "Software Implemented Detection and Recovery of Soft Errors in a Brake-by-Wire System," in *Seventh European Dependable Computing Conference (EDCC-7)*, 2008, pp. 145–154.

[10] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[11] D. Skarin and J. Karlsson, "Evaluation of low-cost detection and recovery of soft errors in an ABS controller," in *IEEE Workshop on Silicon Errors in Logic – System Effects (SELSE 5)*, 2009.

[12] "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–58, 29 2008.