Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online
CONFERENCE PAPER ARCHIVE

IFAC-PapersOnLine 49-17 (2016) 248–253

Flight Control Software Failure Mitigation: Design Optimization for Software-implemented Fault Detectors

Andrey Morozov [*] **Klaus Janschek** [*]

[*] *Technische Universität Dresden, Institute of Automation, 01062 Dresden, Germany (e-mail: {andrey.morozov, klaus.janschek}@tu-dresden.de)*

Abstract: Failures of avionic and aerospace control hardware, caused by negative environmental impacts like increasing heat or cosmic radiation, can lead to silent data corruption and undetected incorrect system outputs. Traditionally, redundant and specifically protected hardware is used, which is expensive and available only on restricted markets. The application of software-implemented fault detectors like SWIFT, SWIFT ECF, or Software Encoded Processing is a promising alternative solution that offers the opportunity to use cost effective, but less reliable hardware. However, this entails generation of extra source code, resulting in a considerable computational overhead and, as a consequence, leads to performance degradations. This article introduces an approach that aims minimizing the negative performance impact while maintaining the required system reliability level. It is shown that selective and balanced application of the software-implemented fault detectors solely to the most critical parts of the control software is an efficient system design solution. The presented approach uses a combination of two methods for reliability and performance analysis. Both methods are used for the quantitative exploration of different strategies of selective protection and allow finding a balance between system performance and reliability. The article demonstrates the application of the introduced approach using embedded flight control software of an UAV.

Keywords: Flight control software, UAV, error propagation, reliability, performance, optimization, Markov models, model-based design.

1. INTRODUCTION

1.1 Motivation

Usually, software in safety-critical systems, assumes a fault free execution through the executing hardware. However, decreasing feature sizes of integrated circuits (e.g. CPU and memory) and increasing system complexity lead to less reliable hardware (Borkar (2005)). Hardware failures can cause hidden data corruption and may result in undetected incorrect system outputs (silent data corruption). In avionic and aerospace applications, an undetected erroneous output can cause hazardous issues. A failure of the Russian space mission "Phobos Grunt" in February 2012 is an example, see Oberg (2012). According to the official report (Roskosmos (2012)), this failure has happened because of an SRAM fault, caused by "a local influence of heavy charged particles" (cosmic radiation).

The example in Fig. 1 illustrates a hardware failure in focus. A negative environmental impact corrupts a part of memory of computing hardware. This results in a single or several bit flips that change the application state, e.g. the value of a critical variable as it shown in Fig. 1. Later, during an execution of the software function *f2*, this erroneous value is read and propagated further to the system output. An error in the output is considered as a system failure that may lead to various unintended consequences. Similar hardware failures leading to data corruption can happen not only in memory, but in a CPU, a bus or other computing hardware parts.

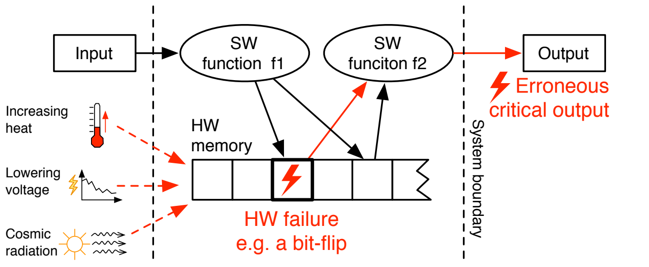

Fig. 1. An example of a hardware failure: A negative environmental impact corrupts a part of system's memory, this changes the value of a stored variable and cause a data error that propagates to a critical system output.

This article is organized as follows. The remaining part of this section presents the relevant state of the art and the basic concept of the introduced method. Section 2 demonstrates a case-study UAV platform. Section 3 introduces the design optimization method itself. Concluding, Section 4 describes the method evaluation and results.

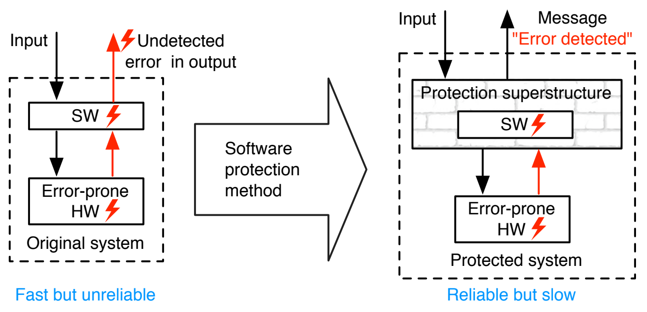

Fig. 2. A general principle of software-based system protection using a sofware-implemented fault detector. A data error caused by a hardware fault is detected on a software level. This results in an error-message instead of an undetected error.

1.2 Traditional Hardware-implemented Solutions

Traditionally, specific custom hardware protects systems from the situations like the one that is shown in Fig. 1. For example: heat and radiation protected hardware (hardened chips, bipolar integrated circuits, magnetoresistive RAM, shielding etc.), or hardware redundancy (dual or triple module redundancy). However, these solutions have several serious disadvantages. First of all, such custom systems are expensive and their markets are restricted. Second, custom hardware is usually an order of magnitude slower than up-to-date commodity hardware (Barnaby (2005)). Third, custom hardware solutions age over time. This results in appearance of new unexpected errors in critical components (Borkar (2005)). Fourth, custom hardware leads to critical dependencies on a single supplier, which can be unacceptable for long-running systems.

1.3 Software-implemented Hardware Fault Detectors

An alternative solution is the application of software-implemented hardware fault detectors (SFDs) like SWIFT (Reis et al. (2005)), SWIFT ECF (Reis et al. (2007)), or Software Encoded Processing (AN-, ANB-, ANBD-codes) (Schiffel et al. (2010)). These techniques can not prevent hardware faults but they can detect them early enough and prevent data errors in system. The principle of an SFD application is shown in Fig. 2.

The SFDs offer the opportunity for using cost effective but less reliable hardware, while maintaining the required level of system reliability. Nowadays, independent R&D projects based on cost effective platforms like CubeSats become more and more popular in the aerospace domain. SFDs do not require any special hardware. They are cheaper, more flexible, and have sufficiently high error detection rates (Schiffel et al. (2010)) in comparison with traditional hardware solutions. Moreover, SFDs can be applied automatically. This results in shorter development cycles and the minimization of programmer's errors.

1.4 Introduced Design Optimization Method

However, existing software-implemented solutions also have a strong drawback that limits their utilization - a considerable computational overhead. Application of SFDs

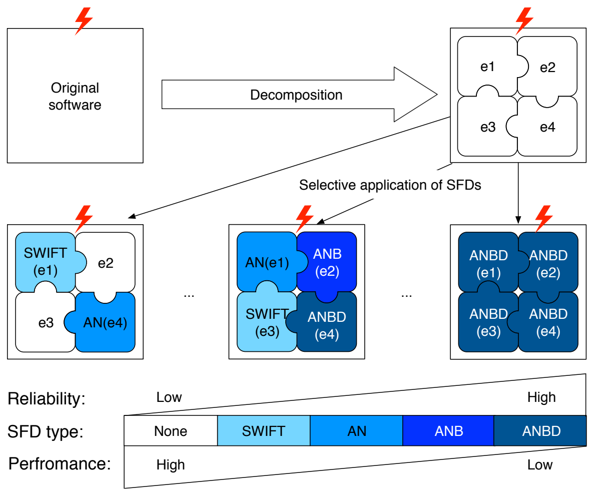

Fig. 3. System decomposition and selective application of different combinations of SFD result in different variants of selectively protected software with different performance and reliability characteristics.

entails generation of extra source code ("protection superstructure" in Fig. 2). This considerably increases the system execution time and leads to higher memory consumption and as a consequence leads to performance degradations, which are critical for control algorithms. This article is focused on balancing the performance degradation of SFDs versus increase of system reliability. Particularly, it is aimed to answer the following question:

How to minimize the negative performance impact of software-implemented hardware fault detectors while maintaining the required system reliability level?

We claim that it makes sense to apply SFDs selectively, only to the most critical parts of the system. This will maximize the achieved error-detection rate, while minimizing the performance overhead.

In Nakka et al. (2007), the authors also address reliability and performance optimization. As opposed to SFDs' application, this article describes reliability improvement using processor-level selective replication. However, the key idea is also to replicate only critical parts of the code. This research has shown the following quantitative results: *"With about 59% less overhead than full duplication, selective replication detects 97% of the data errors and 87% of the instruction errors that were covered by full duplication".*

In this article, we introduce a method for identification of a suitable places for SFD application through a probabilistic quantitative analysis of the original system. Fig. 3 and Fig.4 demonstrate the general idea of the proposed method.

First, we decompose the system into separate elements in order to apply SFDs selectively (see Fig. 3). In general, depending on software size, complexity, design and programming paradigms it can be done on different abstract levels: components, functions, basic blocks of code etc. In our case study, software functions play the role of the elements. This is reasoned by our software design approach with a UML Activity Diagram: Each action block models a functions of the control software, see Fig. 5 c).