Microprocessors and Microsystems 40 (2016) 102–112

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Reliability of data processing and fault compensation in unreliable arithmetic processors

Peter Raab [a,*], Stefan Krämer [b], Jürgen Mottok [b]

[a] Faculty of Mechanical Engineering and Automotive Technology, Hochschule Coburg, Friedrich-Streib-Straße 2, 96450 Coburg, Germany
[b] Faculty of Electronics and Information Technology, OTH-Regensburg, Seybothstr. 2, D-93953 Regensburg, Germany

ARTICLE INFO

Keywords:
Data flow
Error probability
Fault compensation
Software-implemented hardware fault tolerance (SIHFT)
Stochastic simulation
Simulated fault injection

ABSTRACT

In logical circuits, like arithmetic operations in a processor system, arbitrary faults become a more tremendous aspect in future. Modern manufacturing processes lead to less reliability and higher vulnerability of software execution to soft-errors. The correctness of certain results is important especially for safety–critical applications whose reliability depends on the fault-free execution of each single instruction and the dependencies between them. The more complex a software is the more unreliable the outcome is. But, there is a contrary effect. If the probability for multiple faults increases, there is also the chance that two faults compensate each other and the result is correct again. This paper presents the basic ideas for such a reliability evaluation of a software's data flow with arbitrary soft-errors and the effect of fault compensation. Further, this evaluation provides a possibility to compare different implementations of a data flow with respect to the reliability. This is shown by the comparison of two different error codes as alternatives for coded data processing.

1. Introduction

The complexity and functionality of electronic control units have more and more increased in several sectors of industry the last years. In addition, the requirements of these systems have become more demanding in terms of safety, reliability and availability. In contrast to this progress, industry demands a decrease in costs for electronics, while at the same time remaining competitive. The use of inexpensive commodity hardware is the result. However, the development of present microcontrollers follows the trend of decreasing feature size in silicon. This leads to less reliability and arbitrary hardware faults are more likely [1]. But despite unreliable hardware, fault tolerance is a requirement of safety–critical applications [2]. This can often be realized by *Software-Implemented Hardware Fault Tolerance* (=SIHFT) in many ways [3]. One simple possibility of hardening the data against soft-errors (*SEU* = Single Event Upset [5]) is the duplication (=data redundancy) and the multiple computation of data (=time redundancy) [3,4,6]. But, only transient faults can be detected by pure data redundancy. Permanent faults in the CPU (e.g. stuck-at fault in the adder hardware) will generate the same erroneous result. The consequence is the use of redundant hardware or of diverse data so that different units of the CPU are used [6]. An example of diverse data is coded data processing, which is considered as an important aspect for software-based hardware fault detection in recent applications. However, diverse data processing does not detect all faults. There is still a residual probability for non-detection. This residual error probability is a crucial metric for the evaluation of error codes. But in contrast to error detecting codes used in transmission and storage systems [21,24,25], where sufficient error models are available for determining this probability, there are no comparable models for arithmetic operations in a processor system [13]. But by means of such error models, the analytical evaluation of faulty outcomes in arithmetic instructions is then possible opposed to state-of-the-art experimental methods like fault injection. Indeed, the higher complexity of software as a set of interdependent arithmetic operations results in further effects of fault compensation. This effect of fault compensation is evaluated by fault injection in a stochastic simulation framework based on the Monte Carlo method. There a detailed microcontroller model is the basis for the analysis of fault compensation.

The structure of this paper is as follows:

Section 2 summarizes the related works in the domain of software-based hardware fault detection and coded data processing. Further, Section 3 repeats the necessary background of coded processing and reliability evaluation for better understanding. In

* Corresponding author.
E-mail addresses: peter-j.raab@hs-coburg.de (P. Raab), stefan.kraemer@oth-regensburg.de (S. Krämer), juergen.mottok@oth-regensburg.de (J. Mottok).

http://dx.doi.org/10.1016/j.micpro.2015.07.014
0141-9331/© 2015 Elsevier B.V. All rights reserved.

Section 4, we introduce the reliability evaluation of a given data flow and investigate linear codes as an alternative for coded processing based on the previously defined evaluation. The detailed analysis of fault compensation by a simulation approach is depicted in Section 5. The paper proceeds with a discussion of the results in Section 6 and ends with a conclusion for further works in Section 7.

2. Related works

In literature, they report a lot of pure software-based fault-tolerant approaches which are diverging in the effectivity of fault detection.

The approach of *coded processing* refers to special error detecting or error correcting techniques [29]. But this approach is not limited only to circuits. There are pure software methods available that protect the results of operations in an arithmetic unit by means of error detection codes, as well. The input data are encoded before being processed in an arithmetic unit and the output data are decoded again for verification at the end (Fig. 1). With this view, coded processing is related to channel coding as a part of the coding theory.

To be applicable for arithmetic operations, the used error code must preserve the result of the operation as a valid code word. In the past, a lot of codes with this property were described that can be used for arithmetic processors [7–12]. The most important code which is commonly used for coded processing is the so-called *arithmetic code* (AN-code) whose code words are the product of the constant generator A and the information word (Eq. (1)).

$$\mathbb{C}_{AN} := \{A \cdot X | A, X \in \mathbb{Z}\} \tag{1}$$

AN-codes are based on ordinary algebra and preserve the code word with respect to the addition of two code words. This means that the sum of two code words is still a multiple of A and thus it is an element out of the set of code words.

$$C_1 + C_2 = A \cdot X_1 + A \cdot X_2 = A \cdot (X_1 + X_2) \tag{2}$$

But the product of two code words does not match to the coded product of the originals and further corrections would be required.

$$C_1 \cdot C_2 = A \cdot X_1 \cdot A \cdot X_2 = A^2 \cdot (X_1 \cdot X_2) \neq A \cdot (X_1 \cdot X_2) \tag{3}$$

In 1989, Forin made use of AN-codes for coded processing in a real application the first time [9]. He defined coded operations (including additional corrective actions) for most arithmetic operations and he extended signatures to detect operation, operator and operand errors, as well. Furthermore in [13], Ozello discussed the probability of undetection of coded processing. He distinguished between the case where each instruction is verified and the second case when the verification is done after m operations. The latter case is more important for real applications, because the verification of the coded result is usually done at certain points within a task [14]. A possible fault E_1 during the first operation propagates the deviation in the code word with $C_1' = C_1 + E_1$ to the following operation and the result remains faulty also after the second operation ($C_2' = C_2 + E_2$). However, the operation itself or other faulty variables can introduce new faults and influence the final error word. He described this series of deviations by a polynomial $E_g = \{E_1, E_2, \dots, E_m\}$ with m is the number of operations until the verification of the result is done. His evaluation is independent of the underlying error model of the processor. But with the assumption that the elements of E_g are equally distributed and not zero, he demonstrated that the probability of non-detected faulty code words is $1/A$. This simplification is questionable for real operations and it does not consider the concrete realization of the underlying hardware. Additionally to the effect of the transition from a valid to an invalid code word, there is the effect that consecutive instructions with new error words compensate each other. For example, the sum of two faulty coded variables with deviations E_1 and E_2 results in a valid code word, if the sum of both errors is a multiple of A again:

$$(E_1 + E_2) mod\ A = 0 \tag{4}$$

Ozello further remarked that the longer the software the greater the probability to have a polynomial identical to zero which means no deviation in the result. But only programs with more than 10,000 lines show this effect.

The ED^4I (=Error Detection by Diverse Data and Duplicated Instructions) approach presented by Oh et al. [6] is basically a standard method of duplication. A program is executed twice (=instruction duplication) and the data of the copied program are diversely represented. These diverse data are generated by the multiplication of the original data with the so-called diversity factor. For verification, the coded result is compared with the original data at the end. Furthermore, they defined a diversity metric to evaluate several diversity factors with respect to the data integrity and the fault detection probability of different hardware functions (e.g. adder or bus line signals). The diversity factor k determines how diverse the copied program is compared to the original program. They also evaluated several optimal values of k for different hardware functions (e.g. $k = -2$ for an adder). Basically, this approach is a simple example for coded data processing where they used coded data instead of the original. In addition, they defined mathematic models to evaluate and compare different diversity factors with respect to the data integrity and detection probability.

Moreover, Benso et al. [15] introduced a reliability-weight for each variable in a program which is a function of the variable's life time and the dependencies to other variables. The life period of a variable is the time between the first write (initialization) of a variable till it is read the last time before it is written again. The life time is then the sum of all life periods and the longer this time is the higher the probability of being corrupted. Variables with a high reliability-weight are usually more critical for the reliability of the

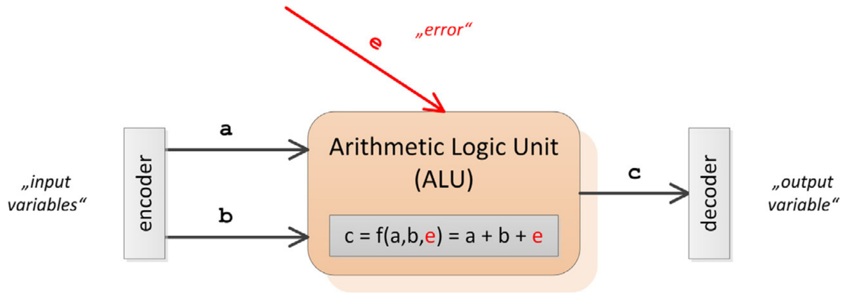

Fig. 1. Simplified processor model: The arithmetic unit in a processor represents a channel with respect to arbitrary (transient or permanent) faults which change the value of a result during the execution of an operation.