SIListra Compiler: Building Reliable Systems with Unreliable Hardware

Martin Süßkraut*, André Schmitt*, Ute Schiffel†, Marc Brünink* and Christof Fetzer*
*Faculty of Computer Science
Technische Universität Dresden
Dresden, Germany
suesskraut@se.inf.tu-dresden.de, andre@se.inf.tu-dresden.de, marc@se.inf.tu-dresden.de, christof@se.in.tu-dresden.de
†School of Computer Science
Reykjavík University
Reykjavík, Iceland
utes@ru.is

Abstract—On one hand, hardware is expected to become more unreliable in future. On the other hand, more and more applications become safety critical, i.e., rely on the correct function of the underlying hardware. To close this gap we propose to add error detection capabilities to the application with our SIListra Compiler. Our compiler automatically transforms C applications based on the well-known principle of Coded Processing. Our evaluation demonstrates that applications compiled with our SIListra Compiler reliably detect hardware errors as requested by various safety standards (such as ISO 26262).

Keywords—safety; coded processing; failure detection; hardware reliability; arithmetic codes;

I. INTRODUCTION

In the future, decreasing feature sizes of integrated circuits will lead to less reliable hardware [1]. Current hardware-based solutions to detect hardware errors are expensive and usually an order of magnitude slower than commodity hardware [2]. Thus, due to economic pressure, more and more critical systems need to be based on unreliable commodity hardware. Unfortunately, commodity hardware not only exhibits fail-stop behavior but also more difficult to detect and to mask silent data corruptions (SDCs), i.e., commodity systems will also generate erroneous output instead of crashing.

Double- or triple-modular redundancy can detect transient errors with a high probability. However, such redundancy requires a more complicated and expensive hardware setup compared to our solution. Furthermore, common mode failures cannot be detected by double- or triple-modular redundancy.

Our vision is to enable the use of cost effective unreliable commodity hardware in safety critical systems. To achieve this, we extend the limited failure detection capabilities of commodity hardware with the help of software. In addition to our more sophisticated error detection, system architects can apply well known toleration approaches to mask SDCs. Our approach works well with retries, fail-over, and graceful degradation.

In this paper we introduce our SIListra Compiler. Applications compiled with our SIListra Compiler turn SDCs into easier to handle stop failures at runtime – without the need for custom hardware. We call applications compiled with our SIListra Compiler protected applications.

The upcoming automotive safety standard (ISO 26262) explicitly mentions the approach that our SIListra Compiler implements (Coded Processing [3]) as one technique to achieve the highest safety level (ASIL D) [4]. In addition to the automotive industry, system architects working in any other safety critical application area, for instance, railway, aerospace, and medical devices, can profit from using our SIListra Compiler. Any application where the risk of erroneous executions is unacceptable high can apply our SIListra Compiler and the principles of Coded Processing in software to reduce the risk introduced by SDCs to an acceptable level.

In summary the advantages of our SIListra Compiler are:

Hardware safety: Applications compiled with the SIListra Compiler detect influences of hardware errors that interfere with the application’s correct execution with a very high likelihood.

Failure propagation safety: Protected applications detect errors of 3rd-party software that influences the correct execution of the protected application. For instance, protected applications detect if the OS or another process erroneously changes their data segment or their code segment.

Simple Integration into Development Work Flow: Our SIListra Compiler applies the principles of Coded Processing as outlined in [5], [6]. Currently, our SIListra Compiler receives C code as input and generates C code as output. In that way it can be transparently inserted into existing tool chains. The SIListra Compiler additionally can directly generate x86_64 binaries. For the future we plan to support more hardware platforms directly.

Support for Distributed Systems: In a distributed protected application a node can check for each message it receives whether the message was correctly computed and not erroneously modified during transmission (see Figure 1).
Adaptable Scope of Protection: Depending on the requirements of a system, the architect can choose to apply our approach to the whole application or only to some components. This is useful in mixed-mode environments where safety critical components run along non-safety critical components. In the following, when we refer to applying our approach to applications, we implicitly include the possibility to applying our approach only to some but not all components of an application.

Adaptable Safety: Our SIListra Compiler supports three different levels of protection with increasing safety, i.e., the probability that a SDC remains undetected shrinks with each level. As expected, our measurements show that higher safety levels consume more CPU cycles. However, the amount of CPU cycles increases linearly but the probability of detecting errors increases exponentially [7].

In the rest of this paper, we describe the application and components of our SIListra Compiler and briefly introduce Coded Processing.

II. SIListra Compiler

Our SIListra Compiler transforms a given application by applying the principles of Coded Processing [3]. We call the input of the SIListra Compiler original application and the output protected application. The two goals of the transformation are:

Correctness: If no error influences the correct execution of the protected application and the original application, then both applications should produce the same outputs. This also means, if the original application contains a software bug, the protected application contains this bug too.

Safety: Any erroneous change of the data computation, the data flow or the control flow during the execution of a protected application is detected with high probability. Currently, we focus on hardware failures and erroneous behavior of 3rd-party components and exclude malicious manipulations by attackers.

Figure 2 shows one possible architecture for running protected applications. The protected application and the expected check values are the products of our SIListra Compiler. At runtime the protected application continuously produces check values. These check values have the following properties:

- They are independent from the concrete input processed by the protected application. Hence, the SIListra Compiler can pre-calculate the sequence of check values that will be produced by the protected application independent from a concrete execution of the protected application.
- Any computation, data flow, and control flow of the protected application influences the check values sent. Hence, with high likelihood any erroneous manipulation of the protected application’s execution, for example due to an hardware error, also influences the produced check values.
- The protected application will produce exactly the pre-calculated sequence of check values if the application is executed correctly, that is, without errors. Otherwise, the protected application will send check values that
Our SIListra Compiler can be easily inserted into an existing build process. It just needs to be placed in front of the existing C compiler.

Figure 4. Our SIListra Compiler can be easily inserted into an existing build process. It just needs to be placed in front of the existing C compiler.

differ from the pre-calculated sequence with a high likelihood.

The watchdog monitors whether the protected application is executed correctly. Therefore, the watchdog compares the check values received from the protected application with the pre-calculated (expected) check values. For additional safety, the output data can also carry check values. We call this data protected data. In a distributed setting, the destination node, which receives the protected output data can verify that the output is a result of a correct execution of the protected application.

Our SIListra Compiler consists of two major components as depicted in Figure 3:

- **Encoded Operations**: We provide protected versions of all operations that an application might use. We call these encoded operations [8].
- **Transformer**: The transformer operates on intermediate code. It substitutes all operations in the original unprotected application with their corresponding encoded operation. Additionally, it pre-calculates the expected check values and inserts code to send out the check values to the watchdog at runtime [5], [6].

Additionally, the SIListra Compiler contains components to translate C-code into the intermediate code and vice versa.

The SIListra Compiler is a C-to-C transformer. Thus, the developer can put it in front of an existing C compiler as shown in Figure 4.

### III. Principles of Coded Processing

Coded Processing approaches add redundancy to any value that is part of an application’s state. This redundancy enables us to detect erroneous executions. The SIListra Compiler adds redundancy to all data words. The redundancy added divides the domain of possible data words into valid data words and into invalid data words. Valid data words are only a small subset of all possible data words. A hardware error hitting a valid data word (e.g., a bit-flip in memory, cache or the register file of the CPU) produces an invalid data word with high likelihood.

For adding the redundancy, our SIListra Compiler uses arithmetic codes. An application protected by our SIListra Compiler solely processes protected data, i.e., all inputs are valid code words of an arithmetic code and all computations use and produce protected data. Thus, we have to use only operations that preserve the code in the error-free case.

Figure 5 shows the relation between valid data words and all possible data words. Correctly executed arithmetic operations preserve the arithmetic code. Thus, given valid data words as input, the output of a correctly executed arithmetic operation is also a valid data word. A faulty arithmetic operation or an operation called with invalid data words produces a result that is an invalid data word with high probability [9]. Furthermore, arithmetic codes also detect errors that modify data during storage or transport because such errors most likely transform a valid data word into an invalid data word.

### IV. Evaluation

We focused our evaluation of the SIListra compiler on safety. The most important aspect is the rate of undetected SDCs, i.e., the share of all hardware failures that a protected application does not detect. To measure the rate of undetected SDCs we used our error injection tool EIS [10] and a set of typically safety critical applications (such as a collision avoidance software).

We evaluated four configurations:

- **Unprotected**: In this configuration we didn’t apply our SIListra Compiler. Hence, we run the original applications.
- **Level 1, 2, and 3**: In the remaining three configurations we applied our SIListra Compiler. As mentioned above, our SIListra Compiler supports three safety levels at compile time. In each configuration we applied a different safety level.

In all configurations we ran the same applications with the same input. We repeatedly executed the applications under the control of EIS. EIS simulates hardware errors at application runtime. We used the same EIS settings for all four configurations. Whenever an output of a run diverted from the output of the application without error injection, we counted this as a SDC.

Figure 6 shows the results of our measurements. Nearly a quarter of all injected errors led to SDCs for the unprotected applications. With each additional safety level we reduce the rate of undetected SDCs by an order of magnitude. The ISO 26262 standard requires a detection coverage of 60% to 99% depending on the required safety level of an application.

The detection coverage is the amount of hardware errors that are detected. Thus, we conclude applications protected by our SIListra Compiler effectively detect hardware errors according to the ISO 26262.
Figure 6. Comparison of the detection coverage of unprotected applications and protected applications with different safety levels. The lower the rate of undetected SDC errors, the higher is the detection coverage.

V. Conclusion

Our SIListra Compiler is an effective approach to protect a C application against undetected execution failures. Because our SIListra Compiler is a C-to-C compiler it can easily be integrated into existing build processes.

The additional code inserted by the transformer reduces the throughput of the protected applications compared to the original application. In our measurements we demonstrated that the completely protected applications have an average throughput in the range of 50% and 70% of the throughput of the original application [7].

For the future, we plan to further optimize our SIListra Compiler for more speed and more safety of the protected applications. Furthermore, we plan to evaluate its usage in the settings of embedded systems.

Acknowledgement

We thank the DFG (Deutsche Forschungsgemeinschaft), the German Federal Ministry of Economics and Technology and EXIST for supporting the SIListra project.

References


