2012 Ninth European Dependable Computing Conference

Eliminating Single Points of Failure in Software-Based Redundancy

Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann
Wolfgang Schröder-Preikschat
Chair of Distributed Systems and Operating Systems
Friedrich–Alexander University Erlangen–Nuremberg
Erlangen, Germany
{ulbrich, hoffmann, kapitza, lohmann, wosch}@cs.fau.de

Reiner Schmid
System Architecture and Platforms
Corporate Technology, Siemens AG
Munich, Germany
reiner.schmid@siemens.com

***Abstract*—In the domain of safety-critical embedded and cyber-physical systems, software-based redundancy is generally understood as an effective and cheap approach to improve reliability. Especially redundant execution in terms of triple modular redundancy is a well-known solution.**

However, triple modular redundancy (TMR) leaves unprotected single points of failure (SPOFs), such as the voter, which have to be carefully considered in all safety considerations.

We present *Combined Redundancy* (*CoRed*), a holistic approach that hardens safety-critical parts of a system against soft-errors, while effectively eliminating the vulnerability caused by SPOFs. *CoRed* leverages redundant execution in combination with encoded processing to tackle the unprotected voting and data distribution. Its implementation does not require specific knowledge about the application and can be easily integrated into existing projects. We evaluated *CoRed* in a realistic setting using a quadrotor helicopter and provide experimental evidence for soft-error resistance and comparable low resource demand. In our experimental comparison plain TMR left more than seven percent of failures undetected, whereas *CoRed* was able to eliminate all silent data corruptions while inducing an overhead of just seven percent.

***Index Terms*—Fault-tolerance, Soft errors, Domain-specific architectures, Software and System Safety, Frameworks, Reliability.**

I. Introduction

With the ongoing reduction of structure sizes, future hardware designs for embedded systems will exhibit more performance and parallelism on the price of being less and less reliable. For safety-critical systems, the handling of soft errors will become one of the major challenges. Soft errors occur randomly during execution and are caused by hardware faults, which are not easily detectable because their impact is only temporary. These errors are severe, as common software execution does not check for or even assume the presence of these events. The most salient error of this kind is caused by elementary particles striking an electronic control unit (ECU) leading to selected bit-flips in memory, data caches, processor registers or even the bus system and arithmetic logic unit. Soft errors, though in general of only rare occurrence, are nevertheless considered to occur frequently enough to be considered for SIL3 or SIL4 categorised safety functions [1], [2], [3].

To provide sufficient error detection and recovery, established solutions employ extensive hardware redundancy or specifically hardened hardware components [4], [5], [6], [7] – both of which are too costly to be employed in commodity products, like cars. Furthermore, there is a strong trend in this industry towards the integration of multiple applications – critical and noncritical – on a single ECU [8], which renders coarse-grained hardware-based approaches impractical.

Software-based redundancy techniques can selectively increase the reliability of such multi-application systems running on commodity hardware. Especially redundant execution in terms of TMR is a well-known solution [9].

However, TMR still leaves unprotected single points of failure, which have to be observed in all safety considerations. These include the sampling of input data, the necessary majority voting, and the distribution of output values. Although these parts are usually considered as relatively small and short in terms of execution time, their shape and number is highly application specific. Moreover, a recent study convincingly shows that the generally assumed random error distribution does not pass the reality check for commodity hardware [10]. All this makes the risk analysis of these single points of failure for an actual application difficult and resource-consuming – and the results questionable.

II. Our Contribution

We present *Combined Redundancy* (*CoRed*), a software-based redundancy approach to provide resilience against soft errors at the application level. *CoRed* combines the well-known techniques of TMR [11], data encoding [12] and control-flow encoding [13] into a holistic approach that omits the classical single points of failure, while inducing an additional overhead of just 7 percent compared to plain TMR. The key contributions of this paper are:

- A design approach for high-reliability voters, which eliminate the major SPOF in TMR systems.
- A design approach for the elimination of input and output vulnerabilities, which eliminates the remaining SPOFs, while keeping replica determinism.

This work was partly supported by Siemens AG under the CoSa grant, the Bavarian Ministry of State for Economics, Traffic and Technology under the (EU EFRE funds) grant no. 0704/883 25, and the German Research Foundation (DFG) under grants no. LO 1719/1-1 and KA 3171/2-1.

978-0-7695-4671-1/12 $26.00 © 2012 IEEE
DOI 10.1109/EDCC.2012.21

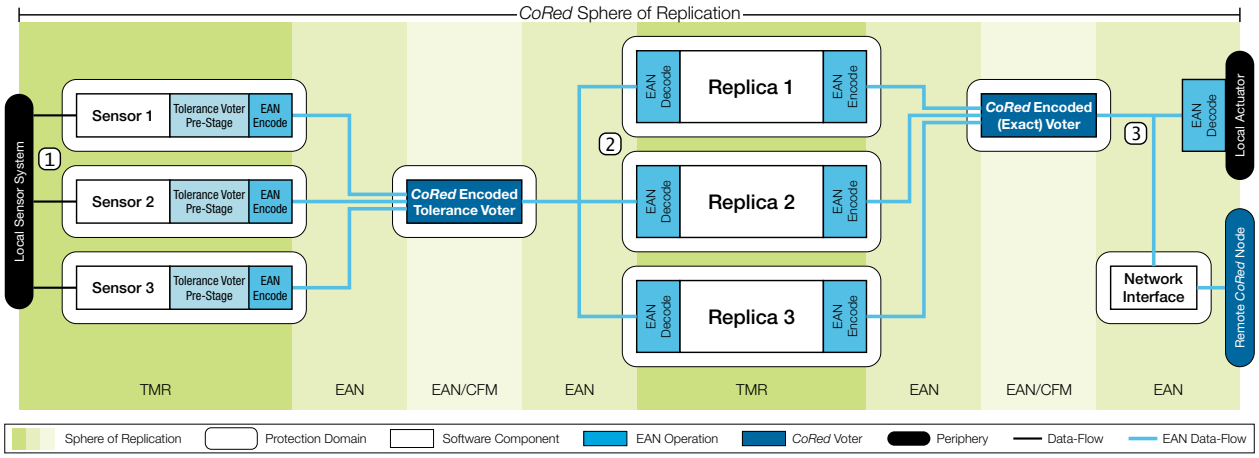

Fig. 1. Overview of the *Combined Redundancy* (*CoRed*) approach. *CoRed* combines, as the name implies, different fault-detection measures to expand the narrow sphere of replication of common TMR and eliminates the remaining single points of failure by leveraging encoded data flows and high-reliability voters (*CoRed Encoded Voters*). This example shows how *CoRed* covers the entire chain of execution from reading the *inputs* ① of a sensor system, to the *processing* ② of a safety-critical controller, and up to distributing the *outputs* ③. By keeping the data EAN-encoded, the approach is applicable even across node boundaries.

- The *CoRed* system, which implements these facilities in an easy to use C++ library.

We evaluate our approach with the flight-control task of a quadrotor helicopter. Our experimental results show that *CoRed* is able to eliminate all failures, whereas the single points of failure in plain TMR leave more than seven percent undetected – leading to fatal silent data corruptions. *CoRed* facilitates easy composability, real-time analysis and schedulability of TMR-based dependability. It can be applied selectively to critical applications in mixed-criticality scenarios.

III. The CoRed Approach

To describe the mechanics of fault detection schemes, Reinhardt and Mukherjee [14] proposed the concept sphere of replication (SOR). It identifies a logical domain protected by a fault detection scheme ensuring that any fault occurring within the sphere and propagating to its boundary will be detected.

The sphere of replication of common TMR concepts excludes the critical voting procedure and the entire data flow outside the replicated execution. Some implementations generate checksums across complex result data to simplify and shorten the voting procedure. This can slightly decrease the probability of errors within this single point of failure. However, any fault can still disrupt the critical voting procedure. The situation is even more serious seeing that the voting directly affects crucial data, leading to various uncovered fault scenarios. First, exact majority voters usually can detect and tolerate data errors that arise within the time span of the replicated execution until shortly before the actual comparison operation begins. However, after data are voted there is still the liability to data corruption, which is then undetected. Secondly, voting on checksums further increases this liability, as an alteration of the data will not be detected if it happens after checksum creation. The voter finds a quorum on the checksums, although the actual data are silently corrupted. Targeting distributed embedded systems, dominantly used in fields of automotive, medical, and avionic industry, this period of time is often extended resulting in an increased time span where the system is vulnerable. In the worst-case scenario this leads to false decisions: Silently corrupted data (SDC) propagate to the actuators.

A. System Model

In order to tackle this challenge *CoRed* makes the following assumptions regarding the safety-critical application and the necessary runtime environment:

- The runtime environment (i.e., the operating system) and underlying hardware enable strict fault-containment. More exactly, we require isolated execution in both a spatial and temporal manner. The former can be achieved using memory protection mechanisms provided by the hardware (i.e., a memory protection unit), the latter is generally realised by incorporating a real-time operating system.
- The safety-critical application resembles a deterministic state machine and must not have interdependencies with other uncovered applications and therefore show run-to-completion semantics. Furthermore, the application has to possess a dedicated and well-defined input and output interface, which will be linked to the specific *CoRed*-implementation artefacts. The prevailing pattern in safety-critical control applications are tasks consisting of three major building blocks: input data acquisition, data processing and output data propagation.

Figure 1 depicts a simple example of a safety-critical application consisting of the following blocks: *Input* ①, acquiring all necessary data by sampling a sensor system, *processing* ② as the primary application logic, and *output* ③ interacting with the environment, more specifically an actuator element. From this starting point, the *CoRed* approach applies discriminative techniques to gradually increase the reliability of the safety-critical application, ultimately yielding to a holistic input to output protection.

B. Holistic Protection Approach

Before going into detail, we first briefly discuss the overall approach (Figure 1). For each part of the processing chain *CoRed* uses tailored measures for ensuring reliability. The basic SOR is implemented by TMR, as used for the sensor data acquisition ① and the computation ② in this example.

In addition, *CoRed* employs data-flow encoding (EAN) to extend the SOR beyond the TMR boundaries: Inputs and outputs are encoded and decoded respectively within the replicas' protection domain, subsequently ensuring the data integrity.

Still, the voting, inevitable in TMR systems, tears gaps in the SOR. *CoRed*'s *Encoded (Exact) Voter* can determine a quorum on encoded results. However, data-flow encoding is insufficient and leaves the control-flow unprotected. To tackle this issue, *CoRed* introduced control-flow monitoring (CFM) in addition.

Finally, the voter passes its decision to the output where it is sent to the actuator. A convenient side effect is that the data can remain encoded, extending the sphere of replication even further. For instance, by transmitting the encoded values to a distributed actuator ECU or to seamlessly connect the outputs to the inputs of another *CoRed* block. In this way, even complex applications and systems can be composed.

The tolerance-based voting at the input side represents an exception. To omit the performance penalties of the encoded operations, it consists of two parts: The *Pre-Stages* that reside within the replicas, mutually determine the input distances and variants based on a tolerance range – hence, compute the costly part. Subsequently, the *Encoded Tolerance Voter* determines a quorum among the encoded variants as usual.

The remainder of this section will detail the techniques employed by *CoRed* step-by-step:

C. Basic Protection

Applying the *CoRed* approach should not require in-depth knowledge of the application to be safeguarded or the underlying system platform (runtime environment and hardware). We therefore employ the well-known and proven concept of TMR [11] as the basis of the *CoRed* approach, as it efficiently detects and masks transient faults of replicated instances. Here, TMR is especially suited, as it can be easily applied and does not require further knowledge of the safety-critical application itself.

The processing is threefold in terms of its state and code (optional) and mapped to the replica tasks, which reside in dedicated protection domains of the runtime environment. The redundant execution is thereby spanning the initial sphere of replication.

One of the advantages of implementing the replication on the coarse-grained software component level is, that it decreases the bandwidth required for output comparison and input replication. That in turn potentially simplifies the voting and replication logic [15].

D. Eliminating input and output vulnerabilities

The basic TMR approach protects only the replica execution itself, while the propagation of data across the SOR-boundaries and the voting procedure is still susceptible to transient faults. The corruption of output data within the voting procedure or on transmission level to the actuator elements can still lead to a silent data corruption. Even worse, corrupted input data will lead to a silent data corruption in every case, as the replicas will work with flawed data and produce apparently correct results. Data crossing the boundaries have to be protected to prevent the formation of single points of failure.

To overcome this weakness and extend the protection across the SOR-boundaries, we combined the basic TMR approach with an arithmetic encoding of the data propagation – thereby giving the name *Combined Redundancy* (*CoRed*).

To be more precise, we use an extension of an AN-Code, which is based on the VCP design presented by Forin et al. [12], specifically tailored to our purposes. It uses a combination of per value signatures and a time stamp to detect data and sequence faults.

To get a feel for this EAN, we exemplify the basics in the following. An arithmetic code can detect data manipulation and, at the same time, preserve arithmetic operations on encoded data. The result of an encoded arithmetic operation applied to encoded operands is again valid encoded data.

The basic AN-Code is the simplest form of an arithmetic code, formed by multiplying the operands by a constant A:

$$X' = X * A \tag{1}$$

A division by A can then restore the original value of AN-encoded data. If the remainder of the division does not equal zero, the value is an invalid code word, which exposes a data corruption. The multiplication factor A has to be chosen carefully to minimize the residual error probability and achieve an adequate Hamming distance. Most AN-Code implementations therefore suggest a large prime number [16].

A bare AN-Code can efficiently detect bit manipulations of encoded values. However, it cannot safely indicate addressing errors – erroneously pointing to another valid code word – nor can it reveal outdated or out-of-sequence data as it is not aware of periods.

Therefore the *Extended AN Code* used in the *CoRed* approach features a unique signature B_X per value to detect addressing errors and in addition a timestamp D to reveal outdated data.

$$X' = X * A + B_X + D \tag{2}$$

As dynamic timestamp D, a cycle counter can be used with the range $0..D_{max}$. The constant value of B_X can then be chosen arbitrarily with the constraint $B_X + D_{max} < A$. Furthermore the minimum distance between two signatures has to be greater than D_{max}.

Finally, to put EAN into use within arbitrary calculations, all arithmetic operations must be adapted. The result of an operation $X' \odot Y'$ generates an encoded value Z' that also includes the specific signature B_z. Applying the inverse

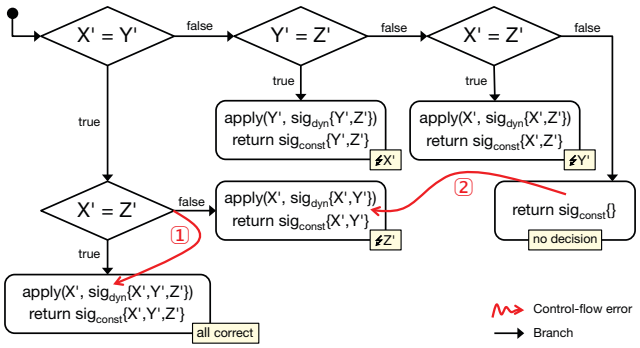

Fig. 2. Basic structure of the *CoRed Encoded Voter*: To ensure the data integrity, the variants remain encoded throughout the entire voting procedure. The control flow is safeguarded by scope signatures, which are recomputed dynamically at runtime. Applied to the winner, any subsequent block can verify the correctness even in the presence of an error (e.g., ① or ②).

transformation, an encoded subtraction operation results in the following equation:

$$\begin{aligned} \overbrace{\frac{1}{A}(Z' - B_z - D)}^{Z} &= \overbrace{\frac{1}{A}(X' - B_x - D)}^{X} - \overbrace{\frac{1}{A}(Y' - B_y - D)}^{Y} \\ Z' &= X' - Y' + \underbrace{B_y - B_x + B_z}_{const.} + D \end{aligned} \tag{3}$$

The encoded operands X' and Y' are subtracted and the signature value is adapted by a correcting term $B_y - B_x + B_z$. Since all signatures B are static values, this correcting term can be determined at compile time for each encoded operation.

By using the EAN, *CoRed* is able to detect faults in encoded values, a feature used to protect data flows going to and coming from the redundant execution. Each replica, for example, encodes its results before passing them to the voter. This is an enabler for validity checks of the data within the voting procedure and the extension of the sphere of replication beyond the initial boundaries.

E. High-reliability Voters

Up to this point, EAN is used to safeguard the data exchange between the redundant execution of software components and the inputs and outputs. The voting procedure, necessary to determine the majority from the three replicas is still unprotected and poses a major single point of failure. Triplicated execution of the voter cannot be employed since the results of the redundant voters would have to be voted again.

As described above, an arithmetic code allows arbitrary arithmetic operations on encoded values. Therefore, *CoRed* leverages EAN for protecting the arithmetic operations and operands used within the voter creating the *CoRed Encoded Voter* as shown in Figure 2. To preserve the EAN protection, it operates on encoded values throughout the entire voting procedure. Therefore, it takes the TMR results, subsequently called variants (X', Y', and Z'), as input, provides a voting result (equality set E) and determines the variant to use as the winner. In the following, we first present how to vote on encoded data. Subsequently, we detail the control-flow monitoring and the generalisation of the voting procedure to a tolerance voter.

1) Encoded data voting: For implementing the *Encoded Voter*, complex comparison operations as described by Forin et al. [12] could be applied to compare the encoded results of the replicas. For the specific purpose of finding a majority, we simplified the comparisons according to Equation 4. The equality of two encoded values can be easily determined by observing their difference:

$$\begin{aligned} \overbrace{(AX + B_X + D)}^{X'} - \overbrace{(AY + B_Y + D)}^{Y'} &= AX - AY + B_X - B_Y \\ &= B_X - B_Y \Leftrightarrow X = Y \end{aligned} \tag{4}$$

If equal, the differences of the particular results must match with the constant difference of the signatures. As the arithmetic code allows determining the equality on encoded values, it is not necessary to decode the result data to find a quorum.

2) Ensuring a correct control-flow and voting: As the comparison operator is able to operate on encoded values, the integrity of the data is ensured. Nevertheless, the voting algorithm itself can still suffer from transient faults, for example causing false branch decisions. Thus, the voter may return wrong results without the consumer being able to detect this. Consequently, it must be possible to not only determine the data integrity but also the correctness of the voting procedure itself. Further, it should be possible to protect the voting result[1] in the same way. For this, *CoRed* leverages EAN based program flow checks in terms of encoded scope signatures similar to the approach described by Schuette [13]. The basic design of the *Encoded Voter* is depicted in Figure 2. At first, its implementation equates a conventional voter and comes to a decision by a sequence of conditional branches. Each branch decision leads to a subsequent program block (scope) until a verdict is found. In addition, the *Encoded Voter* applies every EAN operation, which has led to the outcome, to the value of the winner before returning the voting result. This requires two additional measures for the monitoring of the control-flow: A static and a dynamic program flow signature.

$$\begin{aligned} \overbrace{\{X', Y', Z'\}}^{\text{equality set: } E} &\Leftrightarrow \overbrace{(B_X - B_Y) + (B_X - B_Z) + (B_Y - B_Z)}^{\text{static signature: } sig_{const}(E)} \\ \{X', Y'\} &\Leftrightarrow (B_X - B_Y) \\ \{X', Z'\} &\Leftrightarrow (B_X - B_Z) \\ \{Y', Z'\} &\Leftrightarrow (B_Y - B_Z) \\ \{\} &\Leftrightarrow B_{nodecision} \end{aligned} \tag{5}$$

The static program flow signatures are derived from the signatures of the variants. According to Equation 4, the difference between two equal variants is constant. As a result, the voter's decision and the corresponding control flow can

[1]Decoding is impossible without prior notice about the winner's identity.

be uniquely[2] mapped to a constant signature, as shown in Equation 5. Here, we utilize the fact that there is only one valid path to each possible decision. The static program flow signature then serves as a return value of the voting decision. Consequently, succeeding blocks (components) are able to relate the voting outcome and select the winner accordingly.

At this point, the static program flow signature is not sufficient to detect all possible errors, as the successor cannot validate the correctness of the decision – decoding a wrong candidate still works flawlessly. Therefore, the signature of each voting decision is computed dynamically at runtime, resulting in a dynamic signature $sig_{dyn}(E)$, and applied to the chosen variant. According to Equation 4, the dynamic signature is the difference of the encoded values and corresponds to the expected static signature in fault-free operation, as for example:

$$sig_{dyn}(\{X', Y'\}) = X' - Y' \tag{6}$$

To put it simple, the control-flow is recomputed at runtime and applied to the voting result in terms of EAN signature operations. All calculations are based on EAN signatures and values assuring fault detection throughout the entire path. Together with the voting result, a succeeding block can validate the correctness of the voting decision and the value itself, by inversely applying the constant program flow signature.

As a result, the *CoRed Encoded Voter* ensures data integrity at all times and further eliminates, in contrast to common checksum-based voting, control-flow errors as a SPOF.

To illustrate the error handling, we added two examples of a faulty control flow in Figure 2. In example ①, a bit flip leads to an improper branch decision: Although X' is not equal to Z', the true branch is taken. Subsequently, the voter assumes all variants as correct, applies the respective dynamic signature to X' and returns *all correct* – the defective Z' slipped through. Still, the subsequent block detects the error as the decoding of X' will fail because of $sig_{const} \neq sig_{dyn}$. The situation is similar for other types of control-flow anomalies as in example ②. In this case, an incorrect jump leads right in the middle of another scope. Although the dynamic signature is not even computed, the error is detected. Again, the subsequent decoding of X' with the static signature of $\{X', Y'\}$ will fail.

3) Tolerance Voting: Usually a *CoRed* block will not get its input data already encoded. Thus, until now the extension of the sphere of replication towards the input data acquisition is an unresolved issue.

To gain fault tolerance at the input side, a system has to provide redundant input data in terms of either multiple hardware sensors or temporal redundant sampling. Unfortunately, this introduction of diverse input data leads to the loss of the replica determinism [17]. Because of the varying hardware or diverging sampling times, sensor data will differ in specific ranges and the replicas' results are likely to differ – even in the absence of a fault. A simple majority voter cannot determine a quorum in this case.

[2]As a consequence, the selection of the static signatures has to observe the uniqueness of all decision paths.

Fig. 3. The *I4Copter* quadrotor helicopter; specifically designed with a redundant set of sensors.

A common solution is the integration of tolerance-based voting at the output side. On a correct execution, the distance of the results is within a bounded region and a threshold can be defined as comparison tolerance [18]. Such a tolerance voter first examines the particular distances of the replica results x, y and z:

$$\begin{aligned} d(x,y) &= |y-x| = \delta_1 \\ d(x,z) &= |z-x| = \delta_2 \\ d(y,z) &= |y-z| = \delta_3 \end{aligned} \tag{7}$$

Given the results of this evaluation, the voter then compares the distances δ_i to the application-specific comparison tolerance. However, finding appropriate tolerance values for the replicas' outputs is a difficult task especially if they arise out of complex control algorithms.

We therefore integrated a tolerance-based voter right after the sensor data acquisition to recover replica determinism at an early stage.

CoRed directly supports tolerance voting by providing a generic *Encoded Tolerance Voter* and facilities specifying the necessary tolerance information. Each sensor provides an EAN encoded value to the *Encoded Tolerance Voter* as shown in Figure 1 (input side). The tolerance voter itself is split into two parts. A TMR-based execution of the tolerance based voting, resulting in three equal variants, and a succeeding EAN-based majority voter choosing a quorum, as described in the previous section. Here, we follow the same design approach by executing complex calculations redundantly, that is the tolerance based voting, and concentrating the ultimate decision to a small single point of failure, which is resolved by the exact *Encoded Voter*.

In summary, *CoRed* is able to eliminate the voting procedures (whether exact or tolerance) as a single point of failure and facilitates input to output protection of safety-critical applications.

IV. Implementation

CoRed focuses on an application-oriented implementation approach, which can easily be applied to existing projects. For the sake of general applicability, *CoRed* is implemented using C++ template programming. This way only a common C++ compiler is necessary, which should leverage the application of our approach also in industry settings where tool chains have to be verified in operation or even accredited by regulatory authorities. All non-application specific parts of the implementation, such as EAN and voting, are provided as

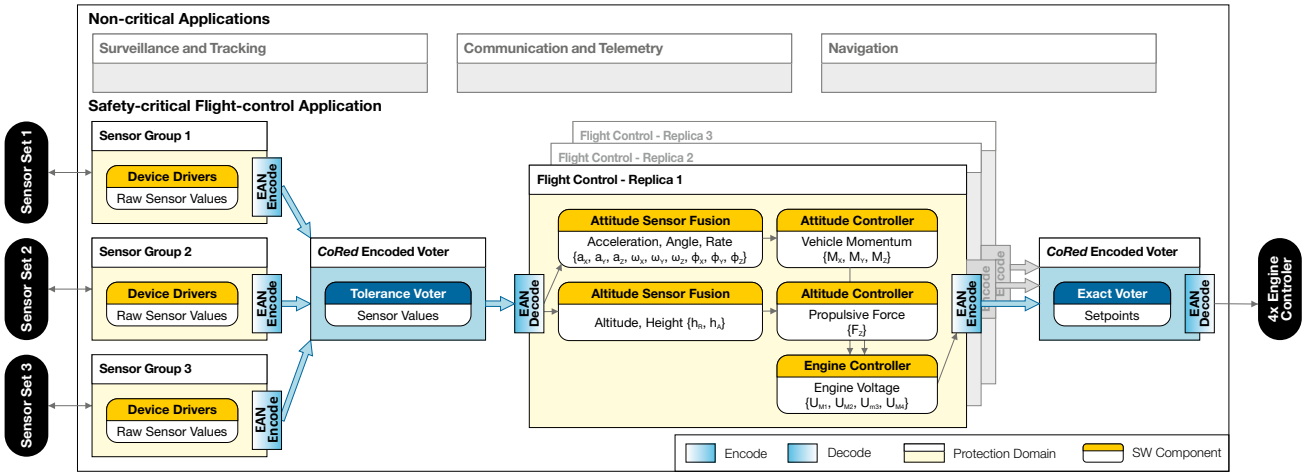

Fig. 4. Overview of the *I4Copter* software system. We selectively applied the *CoRed* approach to the safety-critical flight-control application without altering the non-critical applications such as Surveillance and Tracking. The *CoRed* extension involved the separation of the hardware-dependent device drivers and the identification of sensor tolerance parameters. Data encoding, voting, decoding, and replica generation is largely automated. An adaption of the complex sensor fusion and control algorithms is not necessary, as the replicas can process decoded data.

hardware-independent libraries that support automatic replica and interface generation. In contrast to existing compiler-based solutions [19], we implemented EAN as a platform independent library, which does not require proprietary tools.

Depending on the underlying operating system, *CoRed* can harness existing fault isolation mechanisms, as for example memory protection and deadline monitoring. We integrated the *CoRed* framework both into CiAO OS, our academic aspect-oriented AUTOSAR-OS implementation [20], and PxROS-HR,[3] an industry-strength commercial RTOS platform used in SIL-4 applications. By leveraging the requested facilities of the underlying operating system, replicas exposed by the watchdog or the trap handling routines, are terminated, reinitialized and restarted. Non-critical tasks that erroneously break their isolation domain (spatially, when accessing illegal memory areas, temporally when violating their deadline) are caught by the runtime monitor and restarted. Thereby, the system can always recover to an error-free state and remain fail-safe.

Having the means to restore the internal state of a replica, we further implemented an optimized variant of the TMR approach in terms of a Pair-and-Spare (PaS) [15] concept. Because of the relatively low probability of a transient fault, in most cases the first and second replica will produce identical results. In that case an additional third result is not required. Benefiting from this fact, the overall response time can be shortened in the average; the slack time available for low-priority tasks is increased. Initially, only two of the replicas are executed and voted. Only if a fault is detected by an absence of quorum, the third replica is also executed to ultimately find a majority and identify the erroneous replica. At the beginning of each cycle, the internal state of the spare replica must be updated using the state of one of the pairs.

[3]http://www.hightec-rt.com

A. Target System

To illustrate the application of the *CoRed* approach to a realistic real-time system, we choose the *I4Copter* quadrotor helicopter (see Figure 3) [21] as a target system: Quadrotor helicopters are simple in mechanical design and rely on four fixed pitch propellers, pair-wise spinning in the opposite direction, and a simple gearless drive. The flight attitude[4] is solely controlled by varying the rotation speed of the engines, which requires a challenging control software to reliably control its inherently unstable flight characteristics. The *I4Copter* has been especially designed and developed to resemble embedded real-time systems arising in real-world industrial settings.

The *I4Copter* is equipped with an Infineon TriCore micro-controller commonly used in automotive ECUs and a custom made sensor periphery board featuring a wide range of sensors (12 in total). With regard to the intended safety-critical mission scenario, we have extended the original system by a redundant sensor setting with three gyroscopes per axis and customised engine actuator controllers.

The control software of the *I4Copter* currently consists of approximately 26,000 physical lines of C++ source code and comprises a total of 13 mostly periodical tasks. In addition to the flight control, the system also provides less critical auxiliary and user-defined functions such as waypoint navigation or surveillance. Thereby, the *I4Copter* is a realistic example for a mixed-criticality multi-application real-time system.

The *I4Copter* software is structured in components, each implementing a dedicated functional aspect of the system and interacting only by means of data flow and messaging. The overall architecture is shown in Figure 4: The *Navigation* and *Surveillance* components serve as an example for auxiliary applications and do not require a highly dependable setting as they either do not influence on the attitude of the flight at all or can be overridden in case of faulty behaviour.

[4]Angle of the aircraft in regard to a reference point

The same applies to the communication subsystem, as sending telemetry data to the base station is optional at all and the steering data transmission is implemented dual channel.

For demonstrating the feasibility of our approach we selectively hardened the flight-control application of the *I4Copter*, where soft errors cannot be tolerated and might have disastrous consequences. The flight-control application consists of the three well-known stages input, processing, and output: The device drivers sample and preprocess the redundant sensor sets and provide the input data. The flight control, consisting of signal processing, fusion, and control, is the actual core; it is responsible for attitude and altitude control as well as rudimentary collision avoidance. The controllers compute the thrust levels of the four engines as an output. Finally, these setpoints are sent to the four dedicated engine controllers, which actuate the engines, via an SPI communication bus.

B. CoRed Protected Flight Control

To apply *CoRed* to the flight-control application, the developer first has to encapsulate its implementation into a self-contained class that inherits from generic *CoRed* wrapper classes. These classes then automatically replicate the implementation (by multiple instantiation of a C++ template) while keeping a generic interface to the input and output components. All application-specific information, these are data types and sinks, must be passed to this interface. During compilation, the templates generate the necessary interfaces and replicas accordingly. The individual replicas are mapped to separate tasks and isolation domains of the runtime environment, thereby implementing the basic TMR.

The next step is to apply *CoRed*'s data encoding facilities to harden input and output data. The basic element provided by the library is the `Encoded` template data type. It offers seamless encoding, numeracy and decoding with EAN protected variables by overloading the necessary arithmetic operators. The developer only has to declare the data to be safeguarded as `Encoded` data and assign unique signatures. The generic implementation of the voting procedure can then decide based on a quorum of the encoded result data of each replica.

The SOR could even be extended up to the actuator elements by transferring the encoded results directly to the remote engine controllers. This would make it possible to detect faults affecting the data in transit (if the protocol is unprotected) or on the actuator side. As the EAN library does not exhibit platform-specific parts, we could easily adapt the implementation of the ATmega–based engine controllers to directly deal with the encoded data.

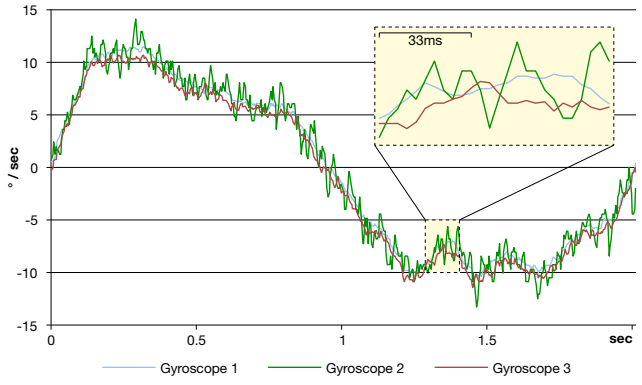

Fig. 5. Divergence in redundant sensor data; three diverse gyroscope sensors measuring the same axis.

TABLE I
EXPERIMENTAL RESULT CATEGORIES

	Result	Description
	Fail Silent	No errors have been detected, the application terminated normally and produced correct results.
Effective Faults	**Masked**	An error has been detected and masked by a redundant result.
	Hardware Detected	An error has been detected by a hardware error detection mechanism or the runtime-system temporal or spatial isolation.
	EAN-Code	An encoded data value was altered, lost or incorrectly accessed and the subsequent validation failed.
	Control-Flow Monitoring (CFM)	An unexpected control flow was detected by the voter's scope signature checks.
	Silent Data Corruption (SDC)	No errors have been detected, but the application produced wrong results compared to the reference.

To tackle faults at the input side, we integrated three redundant gyroscope sensors into the sensor system of the *I4Copter*. As can be seen in Figure 5, the particular measured values of the angular speed slightly differ within a bounded region.

The comparison tolerance required by the encoded tolerance voter can be easily determined applying the noise margins and deviations in production, which can be found in the respective data sheets, as well as the inertia of the quadrotor.

V. EVALUATION

The occurrence of soft-errors is very rare under normal operating conditions. Accordingly, we have to artificially induce errors to test the functionality of our fault-detection and fault-tolerance techniques (see for more details on testing [22]).

A. Fault-injection technique

There are various possibilities to inject faults into a target application [23]. Hardware-based fault injection tools utilize special physical phenomena, such as heavy-ion radiation or electromagnetic interference, causing spurious currents inside the target chip. This technique actually generates real transient faults, however it cannot be used to inject reproducible faults at specific locations at a specific time.

Software-based fault-injection approaches, on the other hand, allow repeatable experiments by adding fault injecting code to the target software of the system under test (SUT). These modifications, however, can influence the original behaviour of the system in other unwanted ways as a side effect (probe effect).

To gain reproducible experiments with minimal probe effect, our evaluation of the *CoRed* approach utilizes the *On-Chip*

TABLE II
ABSOLUTE NUMBER OF EXPERIMENTS

	Execution						Voter				
Type:	**Unprotected**		**Plain TMR**		**CoRed TMR**		**Plain**		**CoRed Encoded**		
Register:	Data	Address	Data	Address	Data	Address	Data	Address	Data	Address	*Sum*
# Fail-Silent	25,063	19,219	25,141	19,319	26,492	20,320	29,207	24,372	76,251	68,847	333,976
# Effective	2,073	7,917	2,144	7,966	2,273	8,444	3,049	7,884	9,253	16,657	67,617
Masked	0	0	1,527	1,668	1,619	1,996	1,583	882	859	369	10,503
Detected	442	5,926	546	6,255	654	6,448	832	6,413	8,394	16,288	52,198
SDC	1,631	1,991	71	0	0	0	634	589	0	0	4,916
# Total	27,136	27,136	27,285	27,285	28,764	28,764	32,256	32,256	85,504	85,504	401,592

Debug Support (OCDS) provided by the TriCore processor. This allows the injection of arbitrary fault patterns and the observation of the resulting system behaviour without the need to modify the actual program code. Faults are injected into the SUT by a hardware debugger controlled by a fault-injection script on the host platform.

This script first executes an injection free golden run to obtain the processing results of an unaffected execution of the target system as a reference. A specific fault pattern is then injected successively at each bit position of each data and address register at each instruction of an execution path. The script ultimately observes the possible effects of the injection that are outlined in Table I.

B. Fault model

A fault injection campaign processes a fault list containing the fault *pattern*, *location* (memory address or register) and injection *trigger* (time or instruction). This fault list can be generated randomly, which mimics the nature of transient faults very well. However, a large proportion of these faults behave fail-silent, for example affecting unused memory. The potential fault space, that is all possible combinations of fault pattern, location and trigger, is tremendous making a fault injection campaign extremely time consuming.

We therefore made some restrictions to the fault list regarding these three components, similar to other approaches [24]. The purpose of the *CoRed* approach is to encounter transient hardware faults. Therefore, the fault pattern assumed in this approach is the single-bit flip, which is inserted at the assembly level between two consecutive instructions. This single-bit flip fault model is similar to and representative for faults occurring in real systems [25] and is frequently used in existing fault injection techniques [22], [26]. We concentrated on single-bit flips as these amount to over 95 % of the soft-error rate [27].

Regarding the location, faults are injected only into CPU registers, as the impact is equivalent to faults in other parts of the system such as the buses or the arithmetic unit. A fault that changes the operand of an add instruction is equivalent to a fault in the cell containing the operand as well as a fault in the arithmetic unit itself [24]. As the employed TriCore processor features a load-store architecture and separated data and address registers, all memory accesses are processed indirectly via CPU registers[5]. On the one hand, faults affecting memory cells can be seen as equivalent to faults occurring in data registers. On the other hand, faults in address registers correspond to data access errors as for example, due to address miscalculation or mutation during execution (e.g., logic or buses).

We further excluded those parts of the execution that are entirely encapsulated within a SOR, as failures striking inside will either propagate to the sphere's boundaries or trigger the runtime-system's isolation mechanisms. Therefore, we focused on the critical boundaries as well as the entire voting procedure, to further decrease the number of possible fault triggers.

C. Experimental Results

To evaluate the effectiveness of the *CoRed* approach we applied it to the attitude flight control of the *I4Copter* being the safety critical part of the system. The *CoRed* experiments are compared with an equivalent campaign of the unprotected execution and voting procedure, respectively.

We applied extensive fault-injection campaigns according to the aforementioned testing rules and stressed our implementation with 401,592 systematic experiments as shown in Table II. In general, a high number of 333,976 faults is still fail-silent, at which their percentage certainly depends on the register utilization. Of vested interest is the set of 67,617 effective faults that do lead to substantial failures.

The overall experimental results are depicted in Figure 6 by two bar charts: On the left, the redundant execution and on the right, the voter campaign. In each case, the charts show the relative distribution among the failure classes described in Table I. For each campaign, two dedicated bars show the results for injections into address registers (A) and data registers (D) – a fact owed to the TriCore's architecture. The results are quoted below by (D: % | A: %).

1) Redundant Execution: The left-hand chart in Figure 6 checks the unprotected and the plain TMR execution against the *CoRed* TMR type. At first, the unprotected execution certainly is susceptible to soft errors and does show a high number of dangerous silent data corruption (SDC) (D: 78.7 % | A: 25.1 %). The remaining fraction of failures can be detected (D: 21.3 % | A: 74.9 %) by the runtime environment, even though the execution is unprotected. Noticeable, the number of detected failures is much higher when affecting

[5]With the exception of DMA transfers from peripherals to the memory.

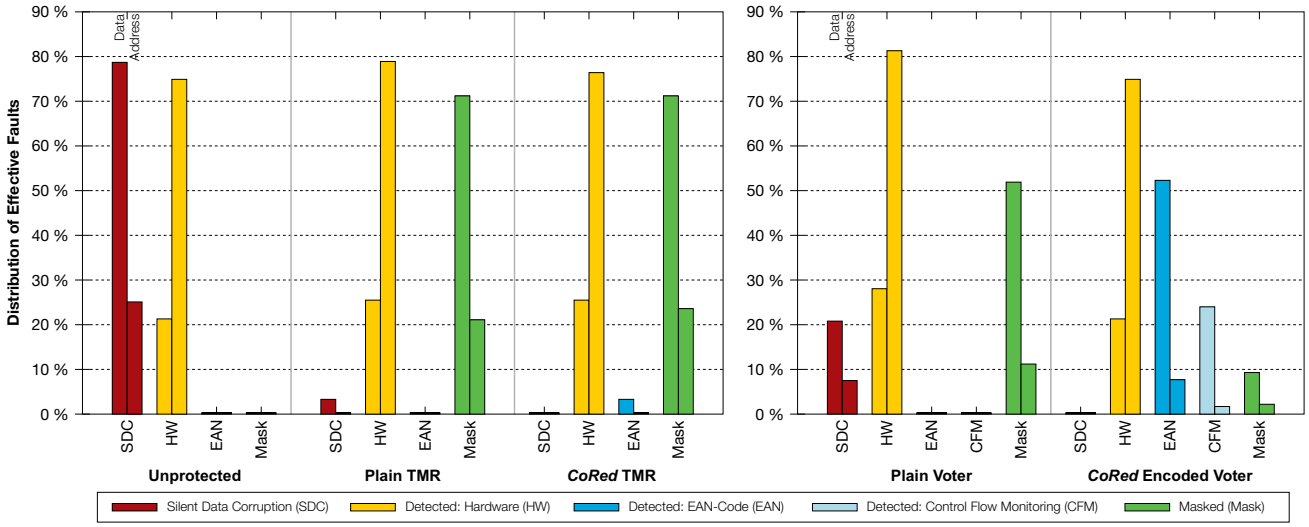

Fig. 6. Overall experimental results of the *I4Copter* flight-control scenario. Each of the charts compares the unprotected and *CoRed* enabled version of the execution (left) and voter (right).

the address registers in general. Here, the high detection rate is due to the employment of hardware memory protection, which effectively detects illegal memory accesses.

Employing TMR is a proven instrument to eliminate the fatal SDCs. The effectiveness of a redundant execution can be seen in direct comparison to the unprotected version, the number of SDC nearly changed over to masked failures (D: 71.2 % | A: 23.6 %). However, a small fragment of SDCs (D: 3.3 % | A: 0 %) still remains due to data manipulations on the boundaries of the replicated execution. This is before the replicas read the input data and after the results are passed to the voter. Although being only a small share, this renders the plain TMR unimmunized to SDCs. In principle, this is a general problem when passing data between protected and unprotected domains.

By contrast, *CoRed* is able to eliminate this incidents and thereby all remaining SDCs by encoding the susceptible data values. The thereby prevented failures appear in terms of detected by EAN-Code (D: 3.3 % | A: 0 %) in the plot for the *CoRed* TMR (data registers). The remaining numbers equal the plain version except for minor deviations due to the different code size.

2) Voting: Finally, the right-hand chart in Figure 6 is presenting the voting procedure results. Interestingly, the plain voter is able to mask a significant amount of failures (D: 51.9 % | A: 11.2 %) and even seems to outperform the *CoRed* voter. The ability to mask failures derives from the redundant input data, the mutation of a single replica result does not affect the constitution of a quorum. The higher masking rate is related to the different voter implementations and the resulting code size — a circumstance that applies to the TMR experiments as well. Although the voter is protected by the TMR extensions, it is nonetheless susceptible to fatal SDCs (D: 20.8 % | A: 7.5 %) and therefore a serious SPOF.

Again, the unique design of the *CoRed* voter is able to detect all effective faults and to eliminate this SPOF completely. The surprisingly low percentage of masked failures (D: 9.3 % | A: 2.2 %) is the outcome of the employed EAN (D: 52.3 % | A: 7.7 %), which is, in addition to the data distribution, able to detect failures before the voting logic would mask them. In contrast to the *CoRed* TMR experiments, the EAN is no longer sufficient. Here, the additional measures for monitoring the control flow (D: 24.0 % | A: 1.7 %) employed in the *CoRed* voter comes into play. In case of an unexpected control flow, the voting result is discarded and the voting is repeated. Interestingly, a high percentage of control flow violations seem to be detected within data registers. The reason is that the scope signatures are naturally stored in data registers and their mutation therefore is leading to an effective and detected fault.

In conclusion, we are able to eliminate all SDCs and the voter as a SPOF, respectively.

D. Runtime overhead

The overhead induced by our approach is certainly closely related to the ratio of SPOF to be protected (i.e., data-flows and voting) and the actual size of the application. Thus, the absolute results of the evaluation are case-specific. Nevertheless, we assume the results are representative for real-world applications.

We evaluated the overhead of our approach by a worst-case execution time (WCET) analysis of the basic building blocks and response time measurements of the realised TMR approaches. To determine the WCET we used the Absint[6] aiT WCET analyser.

Figure 7 depicts the resulting overhead of the various *CoRed* variants compared to a plain TMR execution. The bar chart plots a detailed break down of the overhead, which is necessary to implement a redundant execution, these are interfacing the replicas and voting in general. First of all, it is particularly

[6]AbsInt homepage - http://www.absint.de

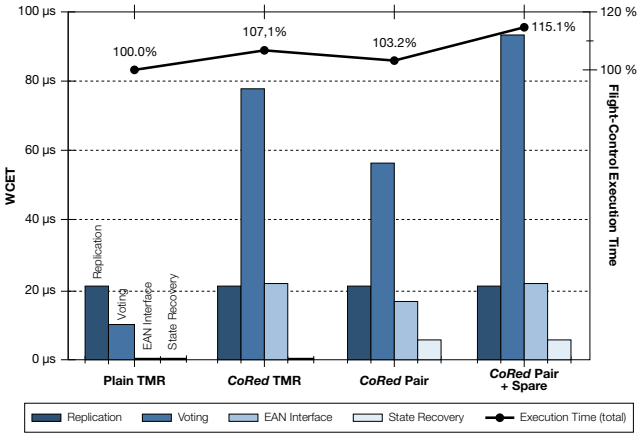

Fig. 7. Runtime overhead induced by the *CoRed* approach. The bars show the absolute overhead of the TMR-specific parts. The plot depicts the relative execution-time overhead added to the redundant flight-control application in comparison to the plain TMR setting.

noticeable that the overhead for the *CoRed* voter ($77.6\,\mu s$) is a multiple of the plain voter ($10.2\,\mu s$) – we merged the numbers for tolerance and exact voting. The reason for this major increment is owed to the generally more complex EAN operations as well as the control flow monitoring. The EAN encoding and decoding ($21.9\,\mu s$) at the SOR boundaries accounts for the remaining overhead.

As soft errors are of rare occurrence, we implemented the Pair-and-Spare (PaS) version to reduce the number of replicas running in the faultless case. The nascent slack time can be harnessed by background execution or less critical tasks. In general, this requires an additional state recovery ($6\,\mu s$) to keep the third replica state up-to-date. In exchange, the overhead for voting and EAN operations is reduced accordingly (minus $25.9\,\mu s$). In case of the absence of a quorum, the third replica is executed and the voting is repeated resulting in an additional overhead of $20.5\,\mu s$ compared to the nonoptimistic execution.

However, rating the *CoRed* approach on the basis of its building blocks (voter and interface) ignores reality, since this would disregard the costs for protecting the overall application. To put the overhead in perspective, we therefore compared a conventional TMR with the *CoRed* setting of the flight-control application. For this, we used the mean value of 256 runtime measurements. To avoid caching effects, the caches of the microprocessor were disabled during the test runs. The results are shown in Figure 7 by the line plot. In relative numbers, the overhead is low due to *CoRed*'s expedient use of the costly protection measures and its tailored design. In our example the overhead is just 7.1 percent compared to a plain TMR schema. In the PaS setting, the overhead even drops to 3.2 percent (15.1 % in case of an error). We consider this as a more than acceptable price – given that we thereby eliminate all silent data corruptions and the voter as a single point of failure!

E. I4Copter Results

The actual CPU utilisation of the unprotected *I4Copter* system is at approximately 41 percent, whereas the mission-critical flight control consumes about eight percent. At this point a plain triplication of the entire system is not only superfluous but also impossible as the CPU would be instantly overloaded. Thus, selective hardening is mandatory or a more powerful and likely more expensive hardware has to be used.

CoRed enables us to protect those events and tasks related to the distinct mission-critical application, while leaving the remaining uncritical applications unprotected. The three replicas can be shifted within the controller period ($9\,ms$) and the remaining slack time can be used to simplify the overall schedulability or to maximise the temporal distance between each other.

We therefore successfully protected the flight-control component, ensuring a controllable and stable attitude of the *I4Copter*, while keeping the rest of the system and the CPU utilisation below 60 percent.

VI. Related Work

Many hardware-centric approaches enable tolerating soft errors and focus on double or triple hardware redundancy ([4], [5], [6], [7]). These approaches have their origin in the aerospace domain and are extremely costly making their application too expensive for the targeted class of systems. Furthermore, due to the additionally required hardware components, they are not suited for an application like the quadcopter that has stringent weight demands. Consequently, we further focused on software-based approaches that are comparatively inexpensive and do not demand for additional hardware.

Over the last few years, there has been major interest in developing software-based fault handling methods for control flow ([28], [29], [30]) or memory protection [31] All these approaches typically focus on a single specific technique or a family of techniques that have their individual strength and weaknesses and their application is proposed on a rather coarse-grained level (i.e., the whole system).

Rebaudengo et al. [32] proposed a source-to-source code transformation that generates fault-detection code to the source language. Every variable in the source code is duplicated, steadily updated on every write operation and checked for consistency on every read operation.

EDDI, proposed by Oh et al. [28], implements a low-level detection technique by duplicating all instructions except branches. Validation code, pasted in before all store and control instructions, ensures the correctness of all values to be written to memory. Another approach presented by Oh et al. [33] is a pure software control-flow checking scheme (CFCSS) wherein the compiler generates signatures for every branch decision, which can be validated by an error checking code. ED4I [34] incorporates data diversity in duplicated execution of functionally equivalent programs.

Reis et al. [35] proposed SWIFT. SWIFT refines EDDI and uses a software only signature-based control-flow checking scheme. SWIFT does not include the memory subsystem in the SoR, as this part of the hardware is nowadays well-protected by hardware-based parity checks.

Chang et al. [36] proposed a SWIFT-like technique called

SWIFT-R using triplicated instructions and a majority voter, and TRUMP a recovery technique using AN-codes.

Homomorphic encryption techniques, allowing arbitrary mathematical operations on encrypted data, are designed to mainly address security issues, rather than safety aspects. Fully homomorphic codes, as for example presented by Gentry [37], [38], rely on costly operations and do not add safety related redundancy – for instance, in terms of outdated data detection.

Forin [12] presented such a safety related homomorphic encoding in the vital coded monoprocessor (VCP). In principle, it encodes all the variables of a program for fault-detection. The VCP consists of three types of codes, intimately mixed and able to detect various classes of errors, providing an enhanced form of arithmetic code. It is used to detect computing errors, in which a signature reveals addressing errors and a timestamp assures the data being up-to-date. The VCP incorporates special hardware for encoding input data and error checking.

Fetzer et al. [19] adopted this approach proposing *software encoded processing* which does not require special hardware, but also incorporates an additional tool chain calculating valid signatures and modifying the original code.

In comparison to the discussed related work *CoRed* enables selective and application specific soft error tolerance and combines encoding of data (based on the homomorphic code by Forin) and redundant execution in an effective way. At the implementation level a large tool chain such as an enhanced compiler is avoided.

More in the direction of *CoRed* is the architectural approach proposed by Afonso et al. [39], which enhances an embedded real-time system with fault tolerance on thread level. Based on a middleware using aspect-oriented programming (AOP) several fault-tolerant configurations have been integrated. Contrary to our approach, the data acquisition and output propagation is not covered.

Another approach for real-time system fault-tolerance focuses on temporal redundancy on task level [40]. Its aim is to mask transient errors by triple time redundant execution and majority voting of the results of all critical tasks. The replica tasks generate a checksum over all necessary output data, which is then compared to each other. Using fixed priority scheduling to control temporal error masking, the system is able to check whether a re-execution of a replica can meet all real-time constraints. Furthermore hardware inherent error detection mechanisms are employed. Fault injection experiments demonstrate the error detection and masking capability of this approach. The experiments also evidence a certain amount of undetected errors. Amongst others, these errors are caused by errors affecting the input variables and errors affecting the output after the checksum generation. These errors can be detected by our *CoRed* approach.

VII. Discussion

In our experience, the *CoRed* approach can be easily applied to a broad range of real-time systems. Beyond the assumptions regarding the design of a safety-critical application we stated in Section III, the composability of *CoRed* facilitates the implementation of even more complex tasks. Interdependencies, for example, can be split up to subtasks connected by the *CoRed* mechanisms. This kind of break down is a well-known design concept in real-time systems anyway. Mapping the *CoRed* artefacts and replicas to real-time operating system resources and setting up a schedule is therefore straightforward.

Eliminating the single points of failure in software-based TMR solutions might seem to be exaggerative as they usually are considered to be small and short in terms of execution time. Nevertheless, we consider the seven percent silent data corruptions in our experiments that are not detected by the plain TMR are worth the effort. Moreover, a primary advantage of *CoRed* is that it significantly simplifies the safety considerations: *single points of failure that have been eliminated do not have to be considered by a risk analysis* – a fact that is even more worthwhile as the generally assumed random error distribution does not pass the reality check for commodity hardware: Nightingale et al. [10] showed that soft errors tend to spatially dense; we assume that this also holds for the temporal distribution. Hence, it could be beneficial to schedule the replica executions with maximum distance – the easy composability of *CoRed* tasks provides just that.

While *CoRed* has very low technical requirements (it is only based on C++ template programming, which is certainly an advantage once the certification of a system is necessary), this also limits the support to automatically generate replica interfaces. Lifting the system description to a model-based approach could speed-up the design and the analysis as well as the implementation process.

VIII. Outlook

Most of our effort has focused on the protection on application level. Virtually all real-time operating systems designed for safety-critical applications do offer memory and timing protection. Nevertheless, the operating system itself is susceptible to transient faults. Certainly, it is possible to realise a basic system without an operating system. The impact of the operating system on the overall reliability is depending on its design. To solve these issues, the operating system and its vital services like scheduling need to be hardened against soft-errors. We are currently investigating *CoRed* as one building block to achieve this as part of the DanceOS project[7].

IX. Conclusion

In this paper we presented *CoRed*, a holistic approach that selectively hardens safety-critical parts of a system against soft-errors. Specifically, *CoRed* features an input-to-output protection by interweaving two software fault-tolerance schemes: redundant execution for the basic computation and *Extended AN Code* at the input and output side. To complete the coverage and ultimately eliminate all remaining SPOFs, *CoRed* employs *Encoded Voter*s featuring control-flow monitoring.

To apply *CoRed* no specific knowledge about the application and the hardware is demanded. Its implementation is based on C++ template programming and can be easily adopted and

[7]DanceOS (http://www.danceos.org)

integrated in existing tool chains and projects. As the approach is acting on software module level, it facilitates the real-time design of the system as the framework and replica modules can be scheduled in a user-defined way. In contrast to related approaches, *CoRed* does consider the input data acquisition and the output data distribution and even allows for extending the fault-detection mechanism to the communication with external actuator components. We successfully evaluated the approach by hardening the mission-critical flight control of the *I4Copter*. In our experimental comparison to plain TMR, *CoRed* induced an overhead of seven percent. However, TMR left more than seven percent of all failures undetected, whereas *CoRed* was able to eliminate *all* SDCs.

REFERENCES

[1] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2002, pp. 389–398.

[2] R. Mariani, P. Fuhrmann, and B. Vittorelli, "Fault-robust microcontrollers for automotive applications," in *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, 0-0 2006, p. 6 pp.

[3] International Electrotechnical Commission, *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, Dec. 1998.

[4] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, pp. 160–174, February 1988.

[5] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb, "Ibm's s/390 g5 microprocessor design," *IEEE Micro*, vol. 19, pp. 12–23, March 1999.

[6] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *Proceedings of the IEEE Aerospace Applications Conference*, vol. 1. IEEE, Feb. 1996, pp. 293 –307 vol.1.

[7] Y. C. B. Yeh, "Design considerations in boeing 777 fly-by-wire computers," in *The 3rd IEEE International Symposium on High-Assurance Systems Engineering (HASE '98)*, 1998.

[8] M. Broy, "Challenges in automotive software engineering," in *28th Int. Conf. on Software Engineering (ICSE '06)*. New York, NY, USA: ACM, 2006, pp. 33–42.

[9] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Secaucus, NJ, USA: Springer, 2006.

[10] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs," in *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2011 (EuroSys '11)*. ACM, Apr. 2011, pp. 343–356.

[11] J. von Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components," in *Automata Studies*, C. Shannon and J. McCarthy, Eds. Princeton University Press, 1956, pp. 43–98.

[12] Forin, "Vital coded microprocessor principles and application for various transit systems," *IFA-GCCT*, pp. 79–84, 1989.

[13] M. A. Schuette and J. P. Shen, "Processor control flow monitoring using signatured instruction streams," *IEEE Trans. Comput.*, vol. 36, pp. 264–276, March 1987.

[14] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," in *In Proceedings of the 27th Annual International Symposium on Computer Architecture*. ACM Press, 2000, pp. 25–36.

[15] s. Mukherjee, *Architecture design for soft errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008.

[16] P. Ozello, "The coded microprocessor certification," in *International Conference on Computer Safety, Reliability and Security (SAFECOMP '92)*, 1992, pp. 185–190.

[17] S. Poledna, "Replica determinism in distributed real-time systems: a brief survey," *Real-Time Systems Journal*, vol. 6, no. 3, pp. 289–316, 1994.

[18] P. Lorczak, A. Caglayan, and D. Eckhardt, "A theoretical investigation of generalized voters," in *Digest of Papers 19th IEEE Symp. Fault-Tolerant Computing Systems*. Los Alamitos, Calif.: IEEE Computer Society Press, 1989, pp. 444–451.

[19] C. Fetzer, U. Schiffel, and M. Suesskraut, "AN-Encoding compiler: Building safety-critical systems with commodity hardware," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, vol. 5775. Springer Berlin / Heidelberg, 2009, pp. 283–296.

[20] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk, "CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems," in *2009 USENIX ATC*. USENIX, Jun. 2009, pp. 215–228.

[21] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat, "I4Copter: An adaptable and modular quadrotor platform," in *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*. New York, NY, USA: ACM, 2011, pp. 380–396.

[22] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Transactions on Computers*, vol. 44, pp. 248–260, 1995.

[23] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, pp. 75–82, 1997.

[24] M. Rebaudengo and M. S. Reorda, "Evaluating the fault tolerance capabilities of embedded systems via bdm," in *VTS '99: Proceedings of the 1999 17TH IEEE VLSI Test Symposium*. Washington, DC, USA: IEEE Computer Society, 1999, p. 452.

[25] P. K. Lala, *Fault tolerant and fault testable hardware design*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.

[26] T. A. Delong, B. W. Johnson, and J. A. P. Iii, "A fault injection technique for vhdl behavioral-level models," *IEEE Design and Test of Computers*, vol. 13, pp. 24–33, 1996.

[27] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced srams," in *Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International*, Dec. 2003, pp. 21.4.1 – 21.4.4.

[28] N. Oh, N. Shirvani, and E. P.P. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Reliability, IEEE Transactions on*, vol. 51, pp. 63–75, 2002.

[29] J. Ohlsson and M. Rimen, "Implicit signature checking," *Fault-Tolerant Computing, International Symposium on*, vol. 0, p. 0218, 1995.

[30] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," *IEEE International On-Line Testing Symposium*, vol. 0, p. 137, 2003.

[31] P. P. Shirvani, N. Saxena, S. M. Ieee, E. J. Mccluskey, and L. F. Ieee, "Software-implemented edac protection against seus," *Reliability, IEEE Transactions on*, vol. 49, pp. 273–284, 2000.

[32] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," *Source Code Analysis and Manipulation, IEEE International Workshop on*, vol. 0, p. 0035, 2001.

[33] N. Oh, P. Shirvani, and E. McCluskey, "Control-flow checking by software signatures," *Reliability, IEEE Transactions on*, vol. 51, no. 1, pp. 111 –122, Mar. 2002.

[34] N. Oh, S. Mitra, and E. McCluskey, "Ed4i: Error detection by diverse data and duplicated instructions," *Computers, IEEE Transactions on*, vol. 51, no. 2, pp. 180–199, 2002.

[35] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *In Proceedings of the 3rd International Symposium on Code Generation and Optimization*, 2005, pp. 243–254.

[36] G. A. Reis, J. Chang, and D. I. August, "Automatic instruction-level software-only recovery," *IEEE Micro*, vol. 27, no. 1, pp. 36–47, 2007.

[37] C. Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[38] ——, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, pp. 97–105, Mar. 2010.

[39] F. Afonso, C. Silva, N. Brito, S. Montenegro, and A. Tavares, "Aspect-oriented fault tolerance for real-time embedded systems," in *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software (ACP4IS '08)*, 2008, pp. 1–8.

[40] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Experimental evaluation of time-redundant execution for a brake-by-wire application," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN '02)*, 2002, pp. 210–218.