Soft Error Handling for Embedded Systems using Compiler-OS Interaction

Michael Engel and Peter Marwedel

1 New Requirements for Fault Tolerance

The ongoing downscaling of semiconductor feature sizes in order to integrate more components on chip and to reduce the power and energy consumption of semiconductors also comes with a downside. Smaller feature sizes also lead to an increasing susceptibility to *soft errors*, which affect data stored and processed using semiconductor technology. The amount of disturbance required to cause soft errors, e.g. due to the effects of cosmic particles or electromagnetic radiation on the semiconductor circuit, has declined significantly over the last decades, thus increasing the probability of soft errors affecting a system's reliable operation.

2 Semantics of Errors

Traditionally, system hardening against the effects of soft errors was implemented using hardware solutions, such as error-correcting code circuits, redundant storage of information in separate memories, and redundant execution of code on additional functional units or processor cores. These protection approaches share the property that they protect all sorts of data or code execution, regardless of the requirement to

M. Engel (✉)
Department of Computer Science, Norwegian University of Science and Technology (NTNU), Trondheim, Norway
e-mail: michael.engel@ntnu.no

P. Marwedel
Department of Computer Science, TU Dortmund, Dortmund, Germany
e-mail: peter.marwedel@tu-dortmund.de

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_2

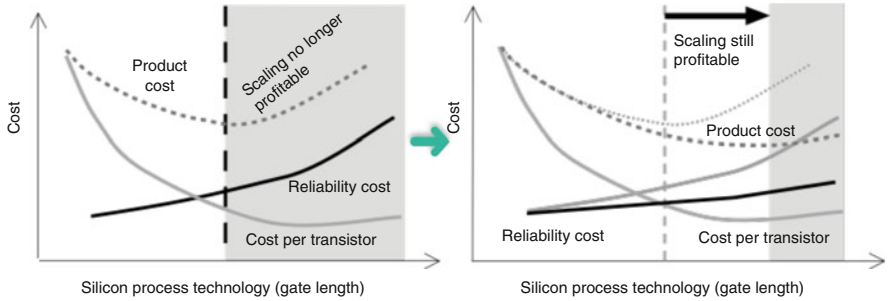


Fig. 1 Enabling profitable scaling using software-based fault tolerance [1]

actually enforce protection. In other terms, they do not possess knowledge about the *semantics* of data and code related to the reliable operation of a system.

As shown by Austin [1], reproduced on the left-hand side of Fig. 1, together with a rising probability of soft errors, this results in a significantly increasing overhead in hardware required to implement error protection. As technology progresses, at a certain point in time, the cost of this overhead will exceed the savings due to the utilization of more recent semiconductor technologies, resulting in diminishing returns that render the use of these advancements unattractive.

The fundamental idea applied by the FEHLER project is to reduce the amount of error handling required in a system by introducing semantic knowledge. We enable a system's software to dynamically decide at runtime whether an error that occurred is critical to the system's operation, e.g. it might result in a crash in the worst case, or is not critical, e.g. an error might only result in an insignificant disturbance of a system's output. In turn, this enables the system to handle only *critical errors* and ignore the others. This *flexible error handling* results in a significantly reduced hardware overhead for implementing fault tolerance, which leads to an increased profitability window for semiconductor scaling, as shown on the right-hand side of Fig. 1.

One important consideration when designing such a selective approach to fault tolerance is which components of a system actually have to be protected from errors. Inspired by the concept of the trusted computing base in information security, we introduced the *Reliable Computing Base* (RCB) [6] to indicate the hardware and software components of a system that are critical in ensuring that our flexible error handling approach is effective.

Accordingly, we define the RCB as follows:

> The Reliable Computing Base (RCB) is a subset of software and hardware components that ensures the reliable operation of software-based fault-tolerance methods. Hardware and software components that are not part of

(continued)

the RCB can be subject to uncorrected errors without affecting the program's expected results.

To design efficient fault-tolerant systems, it is essential to *minimize* the size of the reliable computing base. In the case of FEHLER, this implies that the number and size of hardware and software components required to ensure that upcoming *critical errors* will be corrected are to be reduced as far as possible.

Commonly, code-based annotations such as [13] are used to indicate sections of code to be protected against errors regardless of the data objects handled by that code. This implies an overhead in runtime—protection of the executed section of code applies to all its executions without considering its execution semantics—as well as in programmer effort, since error propagation analyses using control and data flow information would have to consider all data objects handled in annotated code sections. In order to increase the efficiency of this approach, additional manual annotations seem indispensable.

A more efficient approach from a software point of view is to identify the minimal amount of *data objects* that have to be protected against soft errors. Data flow analyses provided by FEHLER allow to determine the worst-case propagation of errors throughout a program's execution, thus determining the precise set of data objects requiring protection against errors. Additional savings at runtime are achieved by employing a microkernel system tailored to exclusively address error handling, leaving the remaining operating system functions to a traditional embedded kernel running on top of it. An analysis of the possible savings for a real-world embedded application is given in Sect. 7.

3 FEHLER System Overview and Semantic Annotations

Based on the observations described above, one central objective of the FEHLER system is to enable the provision of semantics describing the worst-case effects of errors on data objects.

Commonly, the hardware of a system only has very limited knowledge about the semantics of data that it processes.[1] More semantic information, such as the types of data objects, is available on the source code level. However, this information is commonly discarded by the compiler in later code generation and optimization stages when it is no longer required to ensure program correctness. Some of this information can already be utilized to provide error semantics. For example, pointer

[1]For example, a processor could distinguish between integer and floating point data due to the use of different registers and instructions to process these, but a distinction between pointer and numeric data is often not possible on machine code level.

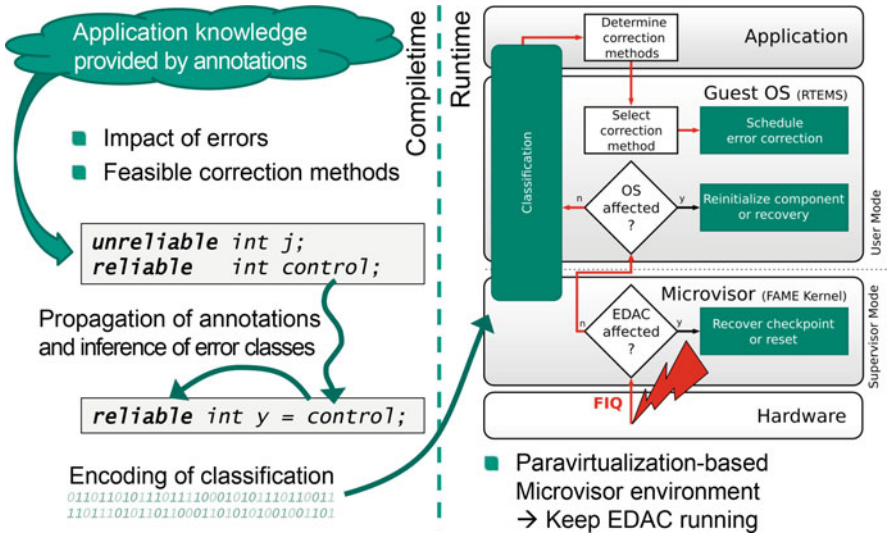


Fig. 2 Interaction of compile time and runtime components of FEHLER

data types and variables influencing the control flow (e.g. used to conditionally execution of code or control loops) are deemed essential in ensuring correct program execution. Accordingly, *static analyses* performed during compile time are able to extract this information.

However, additional information about the relevance of data with regard to the correct behavior of a system in its intended context, e.g. in embedded systems where an incorrect output controlling a peripheral might result in damaging effects, is not expressed *explicitly* in the code. Hence, we have to provide additional information in order to enable static analyses to derive more information about the worst-case criticality of a data object.

This additional semantic information allows the system to classify errors. Data objects which are deemed critical to a program's execution, i.e. may cause the program to crash, are annotated with a `reliable` type qualifier. All objects for which errors in data will result only in an insignificant deviation of the program's behavior in the worst case are provided with an `unreliable` type qualifier.

Classifying data objects into only these two classes is a rather coarse approach. However, as shown later, this minimalistic approach is effective and efficient for systems experiencing normal error rates, i.e. applications not exposed to radiation-rich environments, such as space and aviation systems. Approaches for improved QoS assessment are discussed in Sect. 10.

The interaction of compile time and runtime components of a FEHLER-based system is shown in Fig. 2. Here, the compile time component, realized as a compiler performing static analyses and transformations in addition to code generation, extracts semantic information on the criticality of data objects, analyzes the

program's control and data flow to determine possible error propagation paths and to generate appropriate type qualifiers, and encodes this information along with the generated program binary.

This information lies dormant as long as no error affects the system's operation. Since error detection is outside of the scope of FEHLER, the system is prepared to interface with a number of different error detection methods. In the example in Fig. 2, we assume that a simple hardware mechanism, such as memory parity checks, is employed. When an incorrect parity is detected during a memory access, a special interrupt is raised that informs the system of the error.

Here, our runtime component, the Fault-aware Microvisor Environment (FAME) [11] comes into play. FAME is intentionally restricted to only provide functionality that enables decisions about the necessity of error handling, relegating all other functionality typically found in system software to components outside of the microkernel. This reduced functionality is an additional contribution to RCB minimization. FAME provides a handler for the given error signalization method, which is able to determine the *address of the affected memory location*. As soon as the microkernel is able to ensure that itself is not affected, which can be ensured by RCB analysis and minimization, it determines whether the embedded OS kernel running on top or the application is affected. If this is the case, error handling is initiated. In case of an error affecting the application, FAME consults the semantic information provided by the compile time components and determines if error correction is required or if the error can be safely ignored. Further details of FAME are described in Sect. 6.

Like error detection, specific correction methods are not the focus of FEHLER. Instead, FEHLER is enabled to interface with different standard as well as application-specific correction methods. An example for a standard error correction would be the application of checkpointing and rollback. An application-specific method would be a function that corrects an affected macro block in a video decoder by interpolating its contents from neighboring blocks instead of redecoding the complete video frame, thus saving a considerable amount of compute time.

4 Timing Behavior

Figure 3 shows possible scheduling orders in case of a detected error. In an approach that neglects to use criticality information ("naive approach"), the detection of an error implies an immediate correction action in hardware or software. This potentially time-consuming recovery delays the execution of subsequent program instructions, which may result in a deadline miss.

The flexible approach enabled by FEHLER allows the system to react to an error in a number of different ways. Here, the classifications described above come into play. Whenever an error is detected, the system consults the classifications provided alongside the application ("C" in Fig. 3). This lookup can be performed quickly

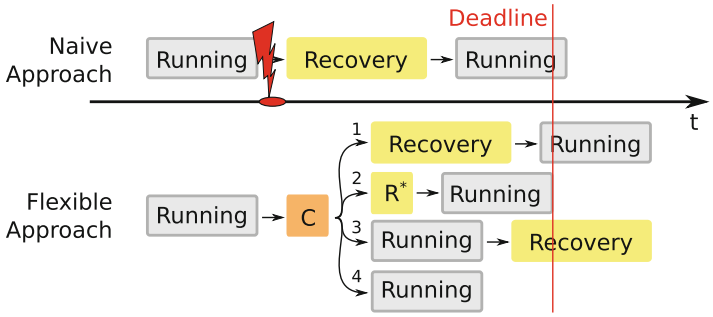


Fig. 3 Different scheduling possibilities for error handling

and provides information on how to handle the error at hand. Specifically, it can be determined *if*, *how*, and *when* the error needs to be corrected:

- *if:* Whether errors have to be handled or not depends mainly on the error impact. If an error has a high impact, error recovery will be mandatory. In contrast, if an error only has a low impact at all, e.g. the color of a single pixel in a frame buffer is disturbed, further handling can be omitted. Handling errors in the latter case will improve the quality of service at the cost of requiring additional compute time. Error impacts are deduced using static analysis methods as described below.
- *how:* Error handling depends on the available error correction methods, the error impact, and the available resources. In FEHLER, commonly a bit-precise correction method such as checkpoint-and-recovery as well as an "ignore" method (case 4 in Fig. 3) doing nothing is available. In addition, the programmer can provide application-specific correction methods, denoted by "R*". Such a method may be preferable, since it can be faster than the generic correction method provided.
- *when:* Error scheduling can decide when an error correction method has to be scheduled. In a multitasking system, often, the task with the highest priority is executed. Hence, if a high priority task is affected, error correction has to be scheduled immediately (cases 1 and 2). If a low priority task is affected, the high priority task can continue execution and the error handling will be delayed (case 3). In order to enable the mapping of errors to different tasks, a subscriber-based model can be employed [12].

Overall, this flexibility allows a system to improve its real-time behavior in case of errors. While this may not be acceptable for hard real-time systems, the behavior of soft real-time applications, such as media decoding, can be significantly improved. The example of an H.264 video decoder is used in Sect. 7 to evaluate the effectiveness and efficiency of FEHLER.

To enable the flexible handling of errors at runtime, the runtime system requires the provision of detailed, correct meta information about the data objects in the given

application. The static analyses employed to obtain this information are described in the following section.

5 Static Analyses

The correct determination of all data objects critical to a program's execution, which form part of the RCB, is crucial to ensure that errors threatening to result in a system crash are corrected before they affect its operation.

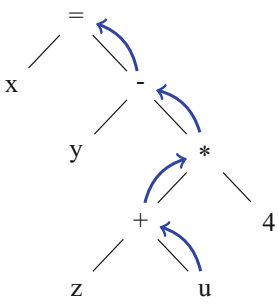
Our static analysis is based on the concept of subtyping [7]. In FEHLER, additional semantic information on the criticality of a data object to the application's stability is provided by extending the C language using *reliability type qualifiers*. These qualifiers enable a developer to indicate to the static analysis stages whether a data object is deemed critical to a program's execution (`reliable` classification) or if errors in data will result only in an insignificant deviation of the program's behavior in the worst case (using the `unreliable` type qualifier).

Accordingly, we have to ensure that reliable data objects must not be modified unpredictably to guarantee that the application will not crash. In contrast, the application can tolerate deviations from the expected values in unreliable data objects.

Rules for the use of our new type qualifiers applied by our static code analysis fall into two groups: *prohibit* and *propagation rules*. Prohibit rules ensure that operations on the annotated data objects are executed error-free, whereas propagation rules reflect the possible propagation of errors from an affected data object to others throughout the control and data flow.

Errors in certain data objects may result in a large deviation in the control flow or even an unintended termination of the application. Prohibit rules ensure that those data objects are annotated with the reliable type qualifier; accordingly, errors affecting that data are classified as fatal errors. Data objects serving as a reference to a memory address, i.e. pointers in C, are an important example for this. An error in a pointer that is used for reading will result in either a different address that is read, possibly resulting in the loading of a completely unrelated data object, or even an access to a non-existing memory location, resulting in a processor exception that terminates the application. Pointers used for writing data can result in correct data being written to an unintended memory location, resulting in unexpected error propagation that is especially hard to diagnose. Indexes for arrays behave in a similar way, resulting either in a write to a different array element or, due to the lack of bounds checking for array indexes in C, a write to an arbitrary memory location. Other critical data types include controlling expressions for loops and conditional statements, divisors, branch targets, and function symbols. For details, we refer the reader to the description in [15].

```
unreliable int u, x;
reliable int y, z;
```




```
...

x = y - (z + u) * 4;
```

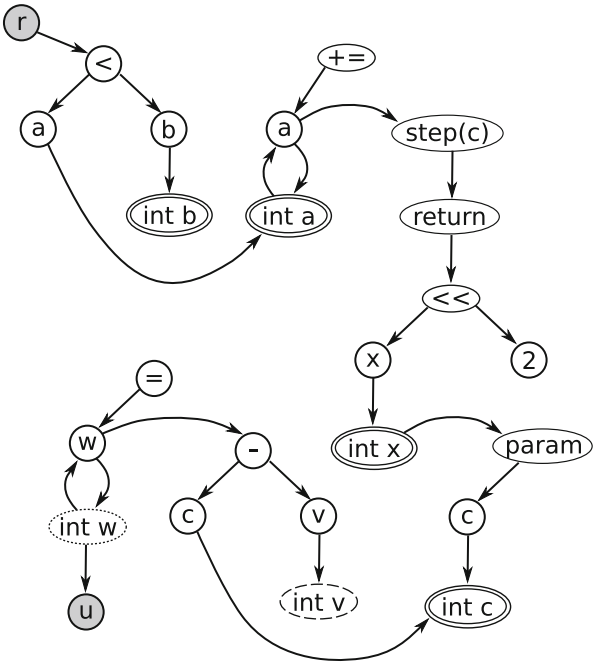
Listing 1 Data flow analysis of possible horizontal error propagation and related AST representation

The content of a data object annotated as unreliable may be affected by an uncorrected error. In turn, that error can propagate to other data objects whenever its content is copied or used in an arithmetic or logic expression, as shown in Listing 1. Here, the curved arrows indicate that an error can propagate from one subexpression to the following along the edges of the syntax tree. Accordingly, the content of a resulting data object cannot be considered reliable and thus has to be qualified as unreliable. The dependencies between type qualifiers of different data objects are modeled by the FEHLER propagation rules. In addition to calculations and assignments, other uses of data objects affected by error propagation are the copying of parameters to functions using call-by-value semantics and cast expressions.

```
int step(int x) {
    return x << 2;
}

void main(void) {
    int a, b, c;
    unreliable int w;
    int v;

    // Initializations
    // omitted for brevity
    while (a < b)
        a += step(c);

    w = c - v;
}
```

Listing 2 Code example and related type deduction graph

Propagation rules not only help in detecting erroneous data flow from unreliable to reliable data objects, but also reduce the overhead required by the programmer



to create annotations. Since it is unrealistic to expect that each and every data object in a complex program will be annotated manually, our static analyses help to deduce correct type qualifiers for unannotated data. This deduction is enabled by the construction of a *type deduction graph* (TDG), as shown in Listing 2. Here, the shaded special vertices marked ⓡ and ⓤ represent an enforcement of the type qualifiers reliable and unreliable by prohibit rules or explicitly stated annotations. The set of edges of the TDG then reflects the dependencies between the type qualifiers, data objects, operations, and assignments.

```
unreliable int u, pos, tmp;
reliable int r, a[10];
u = 10;
r = u;                // invalid assignment
pos = 0;
while (pos < r) {     // invalid condition
  tmp = r / u;        // invalid division
  a[pos++] = tmp;     // invalid memory access
}
```

Listing 3 Invalid assignments

Accordingly, the use of the TDG enables the compiler to flag invalid data propagation from unreliable to reliable data objects. An example containing a number of such invalid propagations is given in Listing 3.

Overall, the static analyses provided by the FEHLER compiler toolchain enable programmers to state reliability requirements that cannot be deduced from the program itself while ensuring that these manual annotations do not accidentally provide a way to propagate unreliable data to reliable data objects. During runtime, the annotations are then used to enable flexible error handling by allowing the operating system to ignore errors in data objects marked as unreliable, thus enabling a tradeoff between the obtained quality of service and the required error correction overhead, e.g. in terms of time or energy.

6 FEHLER Runtime System

Viewed from the top, as shown in Fig. 6, an application with integrated classification information is running on a virtualized guest OS. The guest OS is linked against the FAME Runtime Environment (FAMERE). FAMERE is responsible for the flexible error handling as well as the interfacing with the microvisor. The microvisor runs low-level error correction and ensures the feasibility of software-based error handling (Fig. 4).

The FAMERE runtime is based on our specialized microvisor component which has control over the hardware components relevant to error handling. The main purpose of the microvisor is to isolate critical system components from possible error propagation and schedule the error handling if required. Critical components in this context are resources required to keep error detection and correction running. Depending on the underlying hardware, the actual critical resources vary. If, for example, errors are signaled via interrupts, the interrupt controller will be an element of the critical resource set.

Since the microvisor itself can be affected by errors, it is considered to be a part of the RCB. The microvisor is incapable of protecting itself, since it implements the basic error handling routines. In order to ensure the effectiveness of error

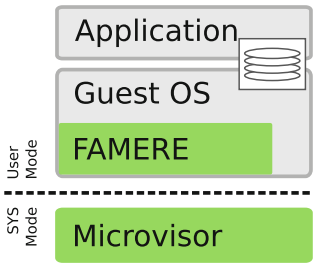


Fig. 4 The runtime software stack of FEHLER. The microvisor is only involved in case of an error, whereas all other resources are administered by the paravirtualized guest OS. The guest OS is extended by FAMERE, the system component responsible for evaluating compiler-provided information on the criticality of errors

handling, fault-free hardware components are required to execute the software-based fault-tolerance mechanisms. In turn, these hardware components also have to be considered part of the RCB. Reducing the code and data size of the microvisor itself is, thus, an optimization objective required to reduce the overall size of the RCB.

To shield the RCB from error propagation, our microvisor uses paravirtualization [16]. The microvisor is tailored to the needs of embedded systems and fault tolerance. To keep the virtualization overhead low, it supports only a single guest operating system. This removes the requirement to provide virtual CPUs and CPU multiplexing. In addition, caches and TLB entries need not be switched between different guest OS instances. An additional responsibility of the microvisor is the creation of full system checkpoints. These are used to restore a valid system state in case of a severe error affecting the FAMERE runtime. FAMERE is a library in the guest OS that combines compile and runtime information required to implement flexible error handling [12].

Error handling is the central task of FAMERE. Figure 5 gives a detailed view of the error-handling procedure at runtime (the right-hand side of Fig. 2). In order to enable a prioritization of error handling, tasks affected by an error in the OS running on top of the microvisor have to be identified. FAMERE determines affected tasks using a memory subscriber model [12] in which tasks explicitly subscribe to and unsubscribe from data objects prior resp. after their use. Accordingly, each data object is annotated with a set of tasks currently using the object, enabling FAMERE to assign a memory address to the set of tasks using the address at the current moment.

If there are higher prioritized tasks not affected by current error, further error handling will be delayed until all higher prioritized tasks finish execution. Error classification will then be performed when the error handling is scheduled again by the microvisor, thus minimizing the impact on system timing when an error occurs.

Together with classification information for data objects, our microvisor and the FAMERE library enable the FEHLER system to implement the envisioned flexible error-handling principles. By keeping the amount of functionality and the related code and data sizes of the microvisor low, the RCB size could be reduced significantly.

7 Use Case: A Fault-Tolerant QoS-Aware Soft Real-time Application

In order to assess the effectiveness and efficiency of the selective error correction approach enabled by FEHLER, we analyzed typical embedded applications in the presence of errors. Since microbenchmarks only tend to give a restricted view of the effects of errors, we used a real-world application to evaluate the possible reduction in overhead.

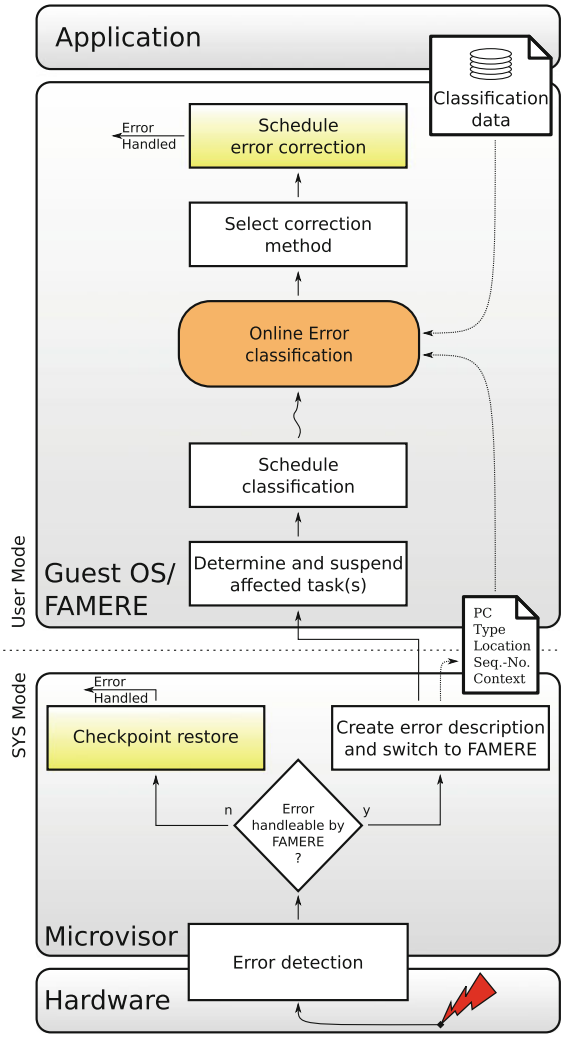


Fig. 5 Error handling in the runtime software stack of FEHLER. If an error is signaled (red flash symbol), the microvisor checks whether the fault affects the RCB. If the RCB is affected, the microvisor automatically restores the last system checkpoint. Otherwise, error handling is delegated by sending a message to FAMERE, which includes an error description containing information about the occurred error as well as the user space context

As mentioned above, the class of applications that we expect to benefit most from our flexible error-handling approach are soft real-time applications that are able to accept—or even make use of—varying levels of QoS in their output. Thus, we used a constrained baseline profile H.264 video decoder application comprising ca. 3500 lines of ANSI C code as a real-world benchmark to assess the effectiveness and efficiency of FEHLER [9].

The evaluation is performed on a simulated embedded system using Synopsys' CoMET cycle-accurate simulator as well as a physical platform based on a Marvell ARM926-based SoC. CoMET is configured to resemble the real system by simulating a 1.2 GHz ARM926 system with 64 MiB RAM, 16 MiB ROM, and 128 KiB

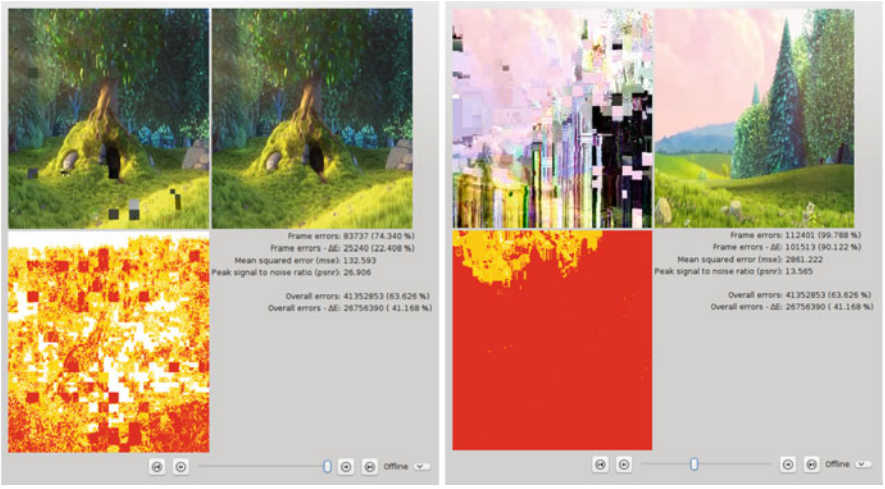


Fig. 6 Analysis of error impacts on the H.264 video decoder using different injection rates

reliable RAM (using ECC-based hardware error protection). All components are considered reliable, except the 64 MiB of RAM.

The H.264 video decoder is configured to create a checkpoint after every displayed frame. In each experiment, we decoded 600 frames in total at a rate of 10 frames per second and a resolution of 480×320 pixels.[2]

We were primarily interested in evaluation results showing the impact of the injected errors themselves on the achievable QoS of the decoded video, the possible reduction of the RCB size using flexible error handling as well as the impact of error handling on system timing.

To assess the impact on the QoS, we developed the quality assessment tool shown in Fig. 6 [8]. It receives video frames decoded by the target ARM system under the influence of errors using FEHLER's flexible error correction and compares these frames to the correctly decoded reference frames (indicated by the yellow and red squares in the lower left pictures–the more red, the larger the difference between the two frames is). For each frame, the tool then calculates several different metrics indicating the QoS, e.g. the peak signal-to-noise ratio (PSNR) and the ΔE color distance metrics. The left-hand side of Fig. 6 shows a moderate error injection rate resulting in some visible defects in the output, whereas the right-hand side shows an artificially high injection rate which renders the output unusable.

For evaluation, we injected uniformly distributed transient faults into RAM. For each memory access, error detection in hardware is simulated. If the processor

[2] Although resolution and frame rate seem rather low, this setup leads to a CPU utilization of more than 65%, since we decode H.264 in software only. However, higher resolutions and frame rates will be possible if more computing power is available.

Table 1 Peak Signal-to-Noise Ratio (PSNR) for different error injection rates [dB]

	$\lambda = 1 \cdot 10^{-16}$	$\lambda = 1 \cdot 10^{-15}$	$\lambda = 1 \cdot 10^{-14}$
All errors handled	36.19	36.15	n/a
Flexible error handling	36.19	36.18	29.01
Flexible + application-specific	36.20	36.12	28.95

accesses an erroneous word, an interrupt will be raised. The number of faults to be injected is determined by a Poisson distribution with a configurable parameter λ.[3]

Table 1 shows QoS results given as PSNR values for different injection rates and correction approaches. We compare a standard correction approach–correcting all errors irrespective of the worst-case outcome–with two approaches based on FEHLER, one which only uses generic error correction such as checkpoint-and-restore and one which, in addition, applies more efficient, application-specific error correction methods. It can be seen that for low error injection rates ($\lambda = 1 \cdot 10^{-16}$ and $1 \cdot 10^{-15}$), uncorrected errors result in a PSNR of about 36 dB, which is still a reasonable quality for lossy compressed media and is similar to the quality of VHS video. For the high error rate ($\lambda = 1 \cdot 10^{-14}$), however, the PSNR drops below 30 dB.[4]

It is important to notice that, although high injected error rates can lead to a significant degradation of the perceived QoS, the primary objective of the binary classification of error impacts employed by FEHLER is achieved—we were unable to provoke the system to crash no matter what the used error injection rate was.

Based on the configuration described above, we analyzed the fraction of memory that the compiler annotated as `unreliable`, implying no protection against errors is required. This fraction is a direct indicator of the reduction of the size of memory that has to be protected, i.e., the RAM memory component of the RCB. In traditional software-based error correction approaches, all of the RAM would be considered part of the RCB. Table 2 shows the results of this evaluation for different video resolutions. It can be observed that for low resolutions, the amount of data classified as `reliable` dominates the memory usage. However, the share of this type of memory is reduced when decoding videos with higher resolutions. For a 720p HD video, already 63% of the RAM used by the H.264 decoder can remain unprotected using FEHLER classifications.

The remaining interesting evaluation is the impact of flexible error handling on the soft real-time properties of the video decoder application. In the first two

[3]Not all injected faults are visible by the application, since faults are only detected when the corresponding memory cell is accessed.

[4]To control the amount of faults to inject, a Poisson distribution with configurable parameter λ is used. The time base used for the Poisson distribution is memory bus ticks. Faults are randomly injected and are equally distributed over the memory. Hence, the locations of the accesses have no influence on the fault distribution.

Table 2 Reduction of the amount of reliable memory required by the H.264 video decoder

Video resolution	Memory size of reliable data	Percentage	Memory size of unreliable data	Percentage
176 × 144	90 kB	55%	74 kB	45%
352 × 288	223 kB	43%	297 kB	57%
1280 × 720	1585 kB	37%	2700 kB	63%

Table 3 Average deadline misses for different error-handling configurations

Error rate		Naive error handling		Flexible error handling		Flexible + application -specific	
λ	$[s^{-1}]$	# Avg. Miss	Avg Missed by	# Avg. Miss	Avg Missed by	# Avg. Miss	Avg Missed by
$\lambda = 1 \cdot 10^{-16}$	0.14	0.00	0.00 ms	0.00	0.00 ms	0.00	0.00 ms
$\lambda = 1 \cdot 10^{-15}$	1.44	2.86	8.15 ms	0.52	7.93 ms	0.36	4.89 ms
$\lambda = 1 \cdot 10^{-14}$	35.84	–	–	1937.87	10,268.98 ms	1887.12	9346.16 ms

columns of Table 3, the observed average error rates (of detected faults) are given, ranging from several faults per minute to an artificially high rate of 36 per second.

We analyzed three different scenarios. In naive error handling, the system treats every error as an error which cannot be handled by FAMERE. Hence, a checkpoint is immediately restored. For this scenario, columns three and four in Table 3 show the average amount of missed deadlines and the average duration of a deadline miss, respectively. For the lowest error rate, no deadline misses occur since enough slack time is available for the recovery of checkpoints. If the error rate increases by an order of magnitude, deadline misses can be observed. On average, deadlines were missed by 8.15 ms. For the highest error rate, no run of the experiment terminated within a set limit of 2 h of simulation time, thus no results are given here.

The results for flexible error handling are shown in columns five and six. Here, only errors affecting reliable and live data are handled by checkpoint recovery. Errors affecting other data are ignored. Flexible error handling reduces the number of deadline misses significantly (81.75%). The time by which a deadline is missed is reduced as well (2.70%). For the artificially increased rate of 35.84 errors per second, however, significant deadline misses could be observed.

The final timing evaluation scenario augmented flexible error handling by including an application-specific error-handling method. For data objects with a special annotation, this method is able to transform a corrupted motion vector into a valid state. For these cases, a time-consuming rollback to a valid system state is not required, reducing the overhead for error correction. Accordingly, using this approach, deadline misses could be reduced by 87.37% for the second highest error rate.

To conclude the overview of our evaluation, we provide an overview of a possible use of application-specific error correction approaches for our H.264 video decoder.

	Impact on QoS	Urgency	Fault handling methods
High Impact	Program termination	fix immediately	rollback
	Corrupted frame input	until frame is displayed	redisplay last frame
	Disturbance in frame header	until frame is displayed	rollback, spare frame, redisplay last frame
Medium Impact	Disturbance in DCT coefficients	fix immediately	rollback, ignore
Low Impact	Disturbance in macro block	until frame is displayed	rollback, copy neighbor block, ignore,
	Disturbance in single pixel	until frame is displayed	rollback, copy neighbor pixel, ignore
		⋮	
	No Impact	none	ignore

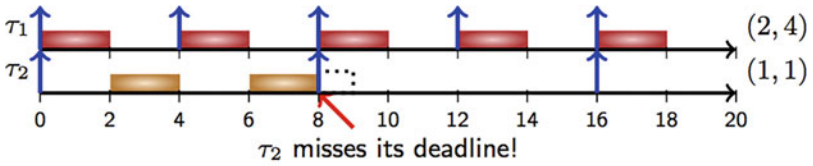
Fig. 7 Classification of error impacts for the H.264 video decoder

As shown in Fig. 7, errors can occur in different data structures, such as frame header or macro blocks. These can be handled by a number of efficient application-specific error correction approaches.

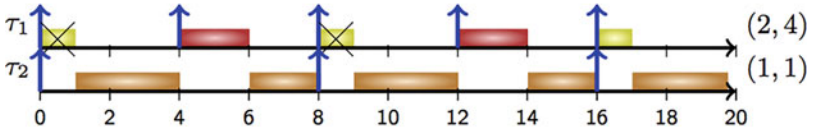
8 Use Case: Adaptive Error Handling in Control Applications

Control-based systems are the basis of a large number of applications for embedded real-time systems. The inherent safety margins and noise tolerance of control tasks allow that a limited number of errors might be tolerable and might only downgrade control performance; however, such limited errors might not lead to an unrecoverable system state. In control theory literature, techniques have been proposed to enable the stability of control applications even if some signal samples are delayed [14] or dropped [2]. Accordingly, we expect that our idea of flexible fault tolerance as described for the video decoder case will also be applicable to control applications.

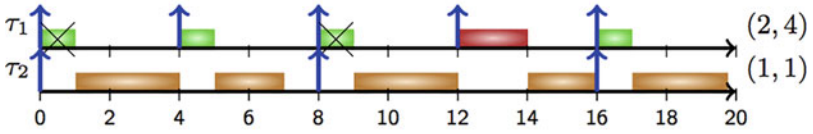
As described above, software-based fault-tolerance approaches such as redundant storage or code execution may lead to system overload due to execution time overhead. For control tasks, an adaptive deployment of related error correction is desired in order to meet both application requirements and system constraints.



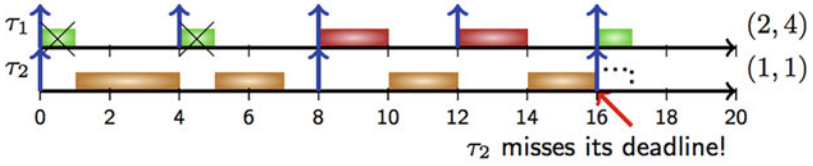
(a) τ_1 is fully protected with reliable versions.



(b) The reliable version of τ_1 is executed with a static distribution trivially.



(c) τ_1 is dynamically compensated with a reliable execution on the forth instance.



(d) τ_1 is not feasible if an additional fault occurring at $t = 4$.

Fig. 8 Different ways to deal with soft errors: red blocks represent reliable executions, green blocks represent executions with error detection, while yellow blocks represent unreliable version without any protection (deadlines are implicit in the schedules shown)

Thus, it has to be investigated how and when to compensate, or even ignore errors, given a choice of different techniques. In an initial case study, we observed that a control task can tolerate limited errors with acceptable performance loss [5].[5]

The general approach used to investigate the effectiveness of this approach is to model the fault tolerance of control applications as a (m, k)-constraint which requires at least m correct runs out of any k consecutive runs to be correct. We investigate how a given (m, k) constraint can be satisfied by adopting patterns of task instances with individual error detection and compensation capabilities. Figure 8

[5]This section is based on joint work with Kuan-Hsun Chen, Björn Bönninghoff and Jian-Jia Chen, TU Dortmund.

shows four different ways to handle soft errors. Some of the presented schedules are infeasible, since they lead to deadline misses.

A static approach to ensure this property is Static Pattern-Based Reliable Execution. In this approach, we enforce the (m_i , k_i) constraints by applying (m, k) static patterns to allocate the reliable executions for task τ_i. While the adopted pattern will affect the schedulability, stability, and flexibility, deciding the most suitable pattern is out of scope of this work.

Due to its inability to react dynamically to changes at runtime, it is obvious that this approach has to be overprovisioning. Thus, we introduce a runtime adaptive approach called Dynamic Compensation that enhances Static Pattern-Based Reliable Execution by recognizing the need to execute reliable instances dynamically instead of having a static schedule.

It is too pessimistic to allocate the reliable instances strictly due to the fact that soft errors randomly happen from time to time. To mitigate the pessimism, we propose an adaptive approach, called Dynamic Compensation, to decide the executing task version on-the-fly by enhancing Static Pattern-Based Reliable Execution and monitoring the erroneous instances with sporadic replenishment counters.

The idea is to execute the unreliable instances and exploit their successful executions to postpone the moment that the system will not be able to enforce an (m, k) constraint, in which the resulting distribution of execution instances still follows the string of static patterns in the worst case.

With Dynamic Compensation, we prepare a mode indicator Π for each task to distinguish the behaviors of dynamic compensation for different status of tasks, i.e., $\Pi \in \{\text{tolerant, safe}\}$. If a task τi cannot tolerate any error in the following instances, the mode indicator will be set to safe and the compensation will be activated for the robustness accordingly. If it can tolerate error in the next instance, the mode indicator will be set to tolerant and execute the unreliable version with fault detection.

Our investigation showed that in embedded systems used for control applications which are liable to both hard real-time constraints and fulfillment of operational objectives, the inherent robustness of control tasks can be exploited when applying error-handling methods to deal with transient soft errors induced by the environment. When expressing the resulting task requirement regarding correctness as a (m, k) constraint, scheduling strategies based on task versions with different types of error protection become applicable. We have introduced both static- and dynamic-pattern-based approaches, each combined with two different recovery schemes. These strategies drastically reduce utilization compared to full error protection while adhering to both robustness and hard real-time constraints. To ensure the latter for arbitrary task sets, a schedulability test is provided formally. From the evaluation results, we can conclude that the average system utilization can be reduced without any significant drawbacks and be used, e.g., to save energy. This benefit can be increased with further sophistication; however, finding feasible schedules also becomes harder.

For an in-depth discussion in the context of a follow-up investigation of this topic, we refer the reader to [17].

9 Application of FEHLER to Approximate Computing

Whereas the work described above concentrates on handling bit flips in memory, more recently, *approximate computing* approaches have been investigated to design energy-efficient systems that trade result precision for energy consumption.

One of the novel semiconductor technologies at the basis of approximate computing is Probabilistic CMOS [3] (PCMOS). Figure 9 shows the general layout of a ripple-carry adder based on PCMOS technology (PRCA). While traditional energy-conserving circuits use uniform voltage scaling (UVOS), PCMOS employs biased voltage scaling (BIVOS), which provides different single-bit full adder components with differing supply voltages that increase from the least to the most significant bit in multiple steps. As a consequence, the delay required to calculate a bit decreases from the LSB to the MSB; accordingly, the probability p_c of bit errors due to carry bits arriving too late is larger in the least significant bits. Using the PCMOS voltage scaling approach, we also employed a probabilistic Wallace-tree multiplier (PWTM) component and added a related energy model and instructions enabling the use of the probabilistic components to our ARMv4 architecture simulator.

We investigated whether FEHLER reliability annotations would also be applicable to determine which arithmetic operations of a program could be executed on PCMOS-based arithmetic components instead of a less energy-efficient traditional ALU without sacrificing the program's stability [10]. A first evaluation using floating point data objects showed that the use of PCMOS technology has the potential for significant energy conservation. Accordingly, we investigated the possible conservation potential for a real-world embedded application. FEHLER type qualifiers were used to indicate data which accepts precision deviation (`unreliable`). Accordingly, our compiler backend generated instructions using probabilistic arithmetic instructions operating on these data objects.

Table 4 shows that a significant fraction of arithmetic ARM machine instructions of our H.264 video decoder could be executed safely on probabilistic components.[6]

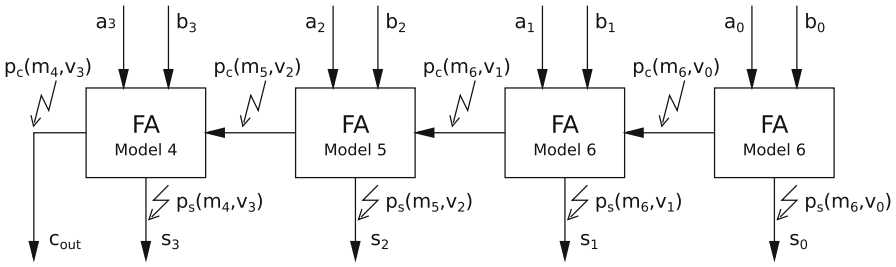


Fig. 9 Probabilistic ripple-carry adder

[6] `rsb` is the ARM reverse subtract instruction.

Table 4 Instructions executed using probabilistic components

Instruction type	Add	Sub	rsb	Mul	Overall
Executed using PRCA/PWTM	18.59%	18.60%	43.01%	76.27%	13.36%

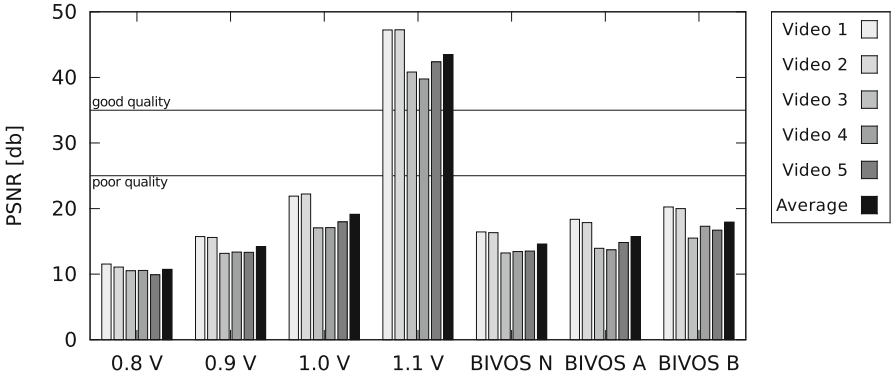


Fig. 10 PSNR (peak signal-to-noise ratio) values for different supply voltage configurations

Surprisingly, our results indicated that for our typical embedded H.264 decoder application, the use of PCMOS components did not result in energy conservation for the identical level of QoS compared to uniform voltage scaling.[7] This result contradicts the microbenchmarks described in [4]. Figure 10 shows the PSNR of the H.264 decoder output for different video clips decoded with circuits using four different UVOS (0.8 V–1.1 V) as well as three BIVOS schemes with similar energy consumption to the UVOS schemes. It can be observed that the PSNR of the BIVOS-decoded videos does not increase, which is a counterintuitive result at first.

A subsequent investigation of the differences between our H.264 decoder and the code used in the microbenchmarks gave insights into the observed effects. Whereas the microbenchmarks employed floating point numbers, our video decoder is a typical embedded application that employs integer and fixed-point numbers.

```
void enter(unreliable uchar *ptr, unreliable int q_delta) {
    unreliable int i = *ptr + ((q_delta + 32) >> 6);
    *ptr=Clip(i);
}
```

Listing 4 H.264 decoder clipping code

This difference in data representation is one of the reasons for the observed phenomenon. The H.264 specification requires a special behavior when copying 32 bit integer values into an eight bit value in the frame buffer. Here, a saturating clipping function (cf. Listing 4) is used. This function restricts the value to 255 if

[7] This only concerns the static and dynamic energy consumption of the PCMOS components. The additional static energy required by the traditional ALU has not been considered here.

the input is larger than that. Accordingly, the shift operation used has the ability to eliminate bit errors in the least significant bits, diminishing the gains of BIVOS scaling.

In contrast, floating point values are always normalized after arithmetic operations. This implies that the bits most relevant to a floating point number's value–sign, exponent, and the MSBs of the mantissa–are always the MSBs of the memory word. In this case, the BIVOS approach to construct arithmetic components that show larger error probabilities in the LSBs is beneficial.

Since it is unrealistic to assume that separate adders for different data widths and data types will be provided in future architectures, an analysis of the number of bits actually used in arithmetic operations is required. However, this implies further complications. One idea for future compiler-based analyses is an approach that combines bit-width analysis methods for arithmetic operations and code transformations to use bits with optimal supply voltage for the operation at hand. The effectiveness of this approach, however, requires further implementation and analysis work.

10 Summary and Outlook

The results of the FEHLER project have shown that for a large class of embedded applications, software-based fault tolerance is a feasible way to reduce the overhead of error handling. The results, as demonstrated using real-world applications, show that already the simple binary classification employed so far is able to avoid crashes due to soft errors while reducing the size of the reliable computing base, i.e. the amount and size of hard- and software components requiring protection from errors.

The technologies developed in the context of FEHLER suggest a number of ways to further improve on the ideas and design of the approach. One constraint of the current design is that the current version of reliability type qualifiers is too coarse-grained. Correcting only errors that affect `reliable` data objects will result in avoiding program crashes. However, a sufficiently high error rate affecting `unreliable` data might still result in a significant reduction of the QoS, rendering its output useless.

The existing static analysis in FEHLER is based on subtyping. Accordingly, to provide a more fine-grained classification of errors, additional error classes have to be introduced. These classes would have to be characterized according to a given total order, so that an error can be classified with the correct worst-case effect. If, for example, the impact of errors is measured in the degradation of a signal-to-noise ratio, a total order can be determined by the resulting amount of degradation.

However, for the overall assessment of a program's QoS, the resulting overall error visible in the output that accumulated throughout the data flow is relevant. Here, one can imagine setting an acceptable QoS limit for the output data and backtracking throughout the arithmetical operations in the program's data flow to

determine the *worst-case deviation* that an error in a given variable can cause in the output. Here, we intend to employ approaches related to numerical error propagation analysis.

We expect that approximate computing approaches will be able to directly benefit from these analyses. Since the approximations already trade precision for other non-functional properties, such as energy consumption, a Pareto optimization of the differing objectives could benefit from worst-case QoS deviation analyses. Here, our initial analysis of the use of binary classifiers for the PCMOS case has already given some interesting preliminary insights.

References

1. Austin, T., Bertacco, V., Mahlke, S., Cao, Y.: Reliable systems on unreliable fabrics. IEEE Des. Test Comput. **25**(4), 322–332 (2008)
2. Bund, T., Slomka, F.: Sensitivity analysis of dropped samples for performance-oriented controller design. In: 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, pp. 244–251 (2015)
3. Chakrapani, L.N., Akgul, B.E.S., Cheemalavagu, S., Korkmaz, P., Palem, K.V., Seshasayee, B.: Ultra-efficient (embedded) SoC architectures based on probabilistic CMOS (PCMOS) technology. In: Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06), pp. 1110–1115. European Design and Automation Association (2006)
4. Chakrapani, L.N., Muntimadugu, K.K., Lingamneni, A., George, J., Palem, K.V.: Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation. In: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08), pp. 187–196. ACM, New York (2008)
5. Chen, K.H., Bönninghoff, B., Chen, J.J., Marwedel, P.: Compensate or ignore? Meeting control robustness requirements through adaptive soft-error handling. In: Languages, Compilers, Tools and Theory for Embedded Systems (LCTES). ACM, Santa Barbara (2016)
6. Engel, M., Döbel, B.: The reliable computing base—a paradigm for software-based reliability. In: INFORMATIK 2012, pp. 480–493. Gesellschaft für Informatik e.V., Bonn (2012)
7. Foster, J.S., Fähndrich, M., Aiken, A.: A theory of type qualifiers. In: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation (PLDI '99), pp. 192–203. ACM, New York (1999)
8. Heinig, A., Engel, M., Schmoll, F., Marwedel, P.: Improving transient memory fault resilience of an H.264 decoder. In: Proceedings of the Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia 2010). IEEE Computer Society Press, Scottsdale (2010)
9. Heinig, A., Engel, M., Schmoll, F., Marwedel, P.: Using application knowledge to improve embedded systems dependability. In: Proceedings of the Workshop on Hot Topics in System Dependability (HotDep 2010). USENIX Association, Vancouver (2010)
10. Heinig, A., Mooney, V.J., Schmoll, F., Marwedel, P., Palem, K., Engel, M.: Classification-based improvement of application robustness and quality of service in probabilistic computer systems. In: Proceedings of the 25th International Conference on Architecture of Computing Systems (ARCS'12), pp. 1–12. Springer, Berlin (2012)
11. Heinig, A., Schmoll, F., Bönninghoff, B., Marwedel, P., Engel, M.: Fame: flexible real-time aware error correction by combining application knowledge and run-time information. In: Proceedings of the 11th Workshop on Silicon Errors in Logic-System Effects (SELSE) (2015)
12. Heinig, A., Schmoll, F., Marwedel, P., Engel, M.: Who's using that memory? A subscriber model for mapping errors to tasks. In: Proceedings of the 10th Workshop on Silicon Errors in Logic-System Effects (SELSE), Stanford, CA, USA (2014)

13. de Kruijf, M., Nomura, S., Sankaralingam, K.: Relax: an architectural framework for software recovery of hardware faults. In: Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10), pp. 497–508. ACM, New York (2010)
14. Ramanathan, P.: Overload management in real-time control applications using (m, k)-firm guarantee. IEEE Trans. Parallel Distrib. Syst. **10**(6), 549–559 (1999)
15. Schmoll, F., Heinig, A., Marwedel, P., Engel, M.: Improving the fault resilience of an H.264 decoder using static analysis methods. ACM Trans. Embed. Comput. Syst. **13**(1s), 31:1–31:27 (2013)
16. Whitaker, A., Shaw, M., Gribble, S.: Denali: lightweight virtual machines for distributed and networked applications. In: Proceedings of the 2002 USENIX Annual Technical Conference (2002)
17. Yayla, M., Chen, K., Chen, J.: Fault tolerance on control applications: empirical investigations of impacts from incorrect calculations. In: 2018 4th International Workshop on Emerging Ideas and Trends in the Engineering of Cyber-Physical Systems (EITEC), pp. 17–24 (2018)