Improving Transient Memory Fault Resilience of an H.264 Decoder

Andreas Heinig, Michael Engel, Florian Schmoll, and Peter Marwedel
Computer Science 12
TU Dortmund
D-44221 Dortmund, Germany
{andreas.heinig,michael.engel,florian.schmoll,peter.marwedel}@tu-dortmund.de

Abstract

Traditionally, fault-tolerance has been the domain of expensive, hard real-time critical systems. However, the rates of transient faults occurring in semiconductor devices will increase significantly due to shrinking structure sizes and reduced operating voltages. Thus, even consumer-grade embedded applications with soft real-time requirements, like audio and video players, will require error detection and correction methods to ensure reliable everyday operation.

Cost, timing and energy considerations, however, prevent the embedded system developer from correcting every single error. In many situations, however, it will not be required to create a totally error-free system. In such a system, only perceptible errors will have to be corrected. To distinguish between perceptible and non-perceptible errors, a classification of errors according to their *relevance* to the application is required. When real-time conditions have to be observed, the current timing properties of the system will provide additional contextual information.

In this paper, we present a structure for an error-correcting embedded system based on a real-time aware classification. Using a cross-layer approach utilizing application annotations of error classifications as well as information available inside the operating system, the error correction overhead can be significantly reduced. This is shown in a first evaluation by analyzing the achievable improvements in an H.264 video decoder under error injection and simulated error correction.

I. Introduction

According to forecasts of semiconductor experts [1], future computer systems will be exposed to non-negligible rates of transient faults. A transient fault is a fault that occurs at some unpredictable point in time due to external events like a high-energy particle hitting a part of a semiconductor. Other causes, like overheating conditions, can also be considered culprits for transient faults. As a consequence, a temporary malfunction of that part of the semiconductor—e.g., a transistor in a memory cell—occurs and one or several bits are flipped in the system, resulting in an error showing up. Due to ever decreasing semiconductor structure sizes and reduced operating voltages, the probability that such a particle impact affects stored information in the semiconductor circuit increases significantly. Thus, even the operation of non safety-critical systems may be affected in everyday use and some form of error detection and correction (EDAC) will be required.

This work was partially supported by the Artist Design Network of Excellence EU FP7 ICT Grant No. 39316

A significantly large fraction of these systems consists of video and audio processing equipment, which require soft real-time capabilities. Most of these devices, like DVD and MP3 players or mobile phones, can be considered typical embedded systems. The inherent limitations of these systems—cost, real-time, memory and energy constraints—restrict the amount of effort that can be put into error correction. As a consequence, correcting every occurring error is often not feasible; we expect that especially cost-sensitive devices like audio and video players will be affected. Instead of providing perfect error correction, these systems can employ a *best-effort* error correction to improve resilience against transient faults.

Overall, the possible types of errors vary greatly. In order to choose a meaningful foundation for our error correction system, we first restrict ourselves to the most common error case: transient errors showing up in RAM. The classification of the severity and urgency of transient RAM errors cannot be determined by the restricted information obtained from an error detection system alone. Rather, additional information is required to determine the context of an error; this information is only available on the application level. Thus, our system employs a cross-layer approach that consults application annotations as well as information provided by the error detection system to determine which errors are to be corrected at what points in time.

In this paper, we devise a classification approach and, based upon this classification, describe a system structure that will enable an embedded system developer to build a soft real-time system that implements this best-effort error correction approach. The system structure is based on our earlier results on error classification published in [2], which are included in this paper in an extended manner, adding new metrics and a revised evaluation of our data.

The rest of this paper is organized as follows. In section II, our error classification approach is detailed, followed by a description of our conceived system structure to support a classification-based EDAC approach in section III. To ensure the viability of our approach, an analysis of an H.264 video decoder application is presented in section IV, followed by an evaluation of the experimental results in section V. Section VI discusses related work, and section VII concludes the paper and gives an outlook to future work.

II. Error Classification

Memory faults can manifest themselves in many different ways. Both the memory location affected and the point in time at which the error occurs relative to a program's execution are crucial parameters that decide if an error will crash an application or will not show any noticeable effect (e.g., when an unused memory location is affected). As a consequence, not all errors that are detected—by well-known hard- or software methods not considered in this paper—have to be treated alike. Thus, a *classification* is required that provides a basis to determine a precedence for error correction. While some early approaches to classify errors according to their impact exist, to the best of our knowledge, no system provides an important feature of our approach: the flexibility to handle the same type of error in a different way depending on the *context* it appears in and the current timing conditions of the system.

To support different types of countermeasures, an application has to provide annotations for different parts of the application to classify possible errors. These annotations include the urgency of error correction, the worst-case impact on the quality of service (QoS) for the whole system, and the possible error correction methods which may be applied to correct the errors.

By annotating the urgency of error handling, the user specifies, whether an immediate handling is required, or the handling can be delayed until a certain event takes place. In the latter case, the system can continue with subsequent computations, if the overhead introduced by an immediate error handling would jeopardize the adherence to the real-time constraints. Nevertheless, if the correction of the error is postponed, it may propagate to further memory locations. An estimate of the impact of this effect has to be acounted for in the annotations.

The remaining component of an annotation is a list of possible error correction methods. Each method can reduce the negative impact on the QoS to a certain amount, but in turn requires some overhead. Depending on these parameters, the system will be enabled to choose an appropriate error handling method for the current context.

In practice, annotations are highly application dependent. To demonstrate the error classification approach in the context of a typical application, a video decoder is an excellent example, since a certain amount of errors in its output is tolarable while soft real-time properties are required. We will use a H.264 decoder as an example throughout this paper to illustrate possible effects of errors, the resulting classification and the appropriate corrective action.

An overview of the error classification for our H.264 decoder is given in table I. The most obvious class of errors are those that crash or terminate the affected application. This may be the consequence of a pointer pointing to an incorrect location, an arithmetical exception like a division by zero, or an incorrectly used system service like a memory allocation using an incorrect size specification. Obviously, these errors have to be corrected immediately by recovering the last known error-free state of the system.

	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 motion vector	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

TABLE I
Error Classification for our H.264 decoder

Errors that do not belong to this first class do not cause the application to crash. However, they may cause serious deviations its the inner state and the generated output, which is directly visible in the QoS. If, for example, the frame type field in the frame header is decoded incorrectly, the whole frame will be misinterpreted. Correction of this type of error has to be completed before the frame is finally displayed on screen.

In comparison, the impact of an incorrect motion vector on the QoS in much smaller, however, the urgency for this kind of error is higher. If the motion vector is not corrected immediately, the macroblock will be copied to a random area in the frame. Although the affected range only has the size of a macroblock, it is hard to restore. This example shows that we cannot assume a given correlation between the impact of an error on the QoS and the urgency error correction.

Incorrectly decoded intra macroblocks in a P-Frame have a very limited effect on the QoS. They affect only a small part of the result or affect the result only for a short amount of time. The outcome will still be acceptable to the user, in many cases the derivation will be hardly noticeable at all. With the next decoded frame, the defective image range is replaced in the output and future outputs are not affected. Such errors may be ignored in order to adhere to the system's real-time constraints, depending on the quality requirements. Nevertheless, depending on the urgency of the error, a delayed error handling is possible, e.g. after the whole frame has been decoded. Thus, if there is any slack time available until the

frame will be displayed, a simple correction method could be applied to reduce the negative impact on the QoS, e.g. by copying a neighbor block.

In the following sections, we analyze critical parts of our example H.264 video decoder application according to the impact of errors. Following this, an evaluation of the application's behavior under error conditions is performed to show the applicability of our classification approach.

III. System Structure

A. System Components

Today, an embedded media player system, like a DVD player, consists of various software components which have to interact in order to provide the required functionality. The most important components are the operating system, handling real-time constraints, resource allocation, memory and I/O management and the media player application, which decodes the media data for live replay. Further components include less timing critical parts like network and mass storage interfaces and software that handles the user interface. The timing properties of the overall system are determined by the interaction of the various components, controlled by the operating system scheduler. In our case, an additional component is the EDAC subsystem.

As a prerequisite for this paper, we assume that the OS and EDAC subsystem are not affected by errors. Hence, only the application is exposed to errors. This might be true for a system that is equipped with different memories. On the one hand, there might be a small robust memory space manufactured in a way to show only a negligible fault rate or protected by additional hardware features. On the other hand, there might be larger, low-cost memories with a significant fault-rate that are used for application and data. The OS and EDAC subsystem would then be placed in the robust memory. An example for such a system design can be found in [3]. To extend EDAC to protect the OS and EDAC subsystem is a challenging topic for future research.

B. Error Detection and Correction

A differentiating approach to error handling under real-time constraints requires on the one hand information on the *semantics* of an error, provided by a classification approach detailed above, and on the other hand information on the current *timing properties* of the system, including the task of the affected application and all related tasks. Error semantics are a property of the application involved, whereas the timing of a system is controlled by the scheduler of the underlying operating system. Thus, a real-time system that is able to employ context-dependent error handling behavior has to take classification handling requires information from the application as well as additional context from the system layer into account.

Existing work describes many different methods for error detection and correction (EDAC). Our common criteria for suitable methods is that the methods are low-cost and should have low overhead. Among the detection methods are simple, hardware-based methods like memory parity bits as well as software-based methods like redundant execution or checksumming of data. Approaches to correct encountered errors may include checkpointing and recovery and redundant execution of critical functions. In order to assess the overhead introduced by the error detection and correction method, methods that provide worst-case timing information will help to calculate the overall timing dependencies of the system. To create an approach that is as flexible as possible, functions implementing EDAC will be contained in separate libraries to be included with the OS.

The system should be highly configurable as to the error detection and correction methods supported. For error detection, possible approaches are low-cost hardware-based approaches like parity bits and software-based approaches like redundant instruction execution. Error correction can use any of several proven approaches, e.g. checkpointing and recovery.

A system that improves reliability to transient errors thus has to implement the components error detection, error correction, application annotation and scheduler control. In the following paragraphs, we describe details of the intended application annotation and the interaction of the system components.

C. Application Annotations

As explained above, the semantics of an error can not be deduced from information usually available inside the operating system alone. This information includes data like the address of the memory location that is affected by an error or the current process context. In order to reach a decision on how a specific error has to be handled, the OS thus has to correlate this data with additional information on the semantics of the specific error.

This information is specific to and has to be provided by the affected application. In our system, *application annotations* are used to provide information on the semantics of an error that can not be inferred from information available inside the OS alone.

As a first approach, annotations are created manually by inserting calls to an annotation library into the application code. These library functions augment an application-specific database of error annotations which can in turn be retrieved on demand by the operating system using callbacks into the library. An example showing a potential implementation for an application written in C is shown in fig. 1. Here, function calls mark the beginning and end of a section of code with specific error semantics. Whenever these functions are called, the application-specific annotation data is updated by the library. In case of a detected error, the OS calls the library to check if an annotation is available for the current code section and can then invoke the appropriate handling function from a provided EDAC library.

To create a system providing maximum dependability, we follow the pessimistic assumption that all code sections not

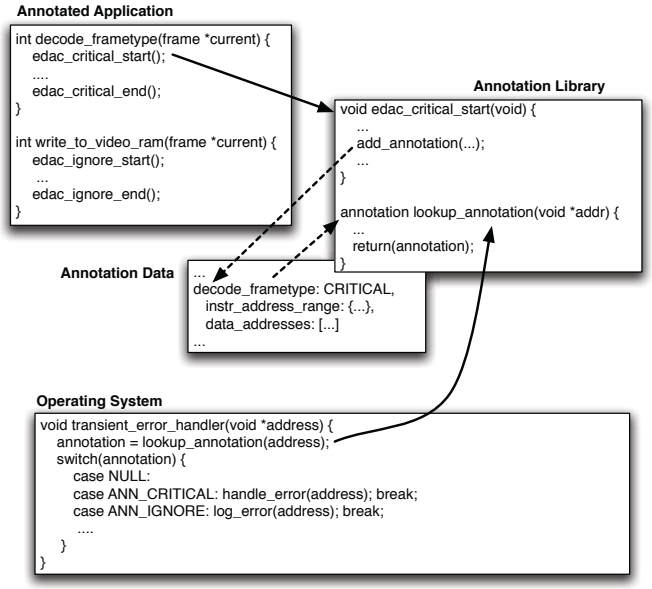


Fig. 1. Annotation interaction

annotated at all will require immediate error correction.[1]

However, the manual creation of annotations is a tedious and potentially error-prone task. Thus, in future versions of our system, the application developer shall be supported by a compiler-based static code analysis tool that provides automated annotations for sections of the application source code. For example, if the application developer annotates an intermediate result as less important for the quality of the final output, with the help of data-flow and pointer analysis this tool can identify all operations and inputs that contribute to this result. However, more global information on the influence of a piece of code to the quality of the output is not likely to be deducible from a static analysis. This will still require manual annotations. In that way, the application developer will only have to annotate some parts of the source code instead of having to review the whole source code.

D. OS Integration

The structure described in the previous paragraphs already provides a simple integration of the annotations into the underlying operating system. However, this integration alone can not influence the temporal properties of our system. Hence, a tighter integration of error handling and scheduling is required in order to gain the appropriate timing flexibility.

When the system detects one or several errors, the scheduler first has to interact with the error correction system to gather the annotated error classification. Thereafter, the scheduler has a complete list of all arisen error containing the urgency, the impact on the QoS, and possible correction methods for each error.

[1]However, there may be cases when a different default behavior is desirable. This default behavior should thus be easily adaptable.

The second step is to use these information to create an order in which the errors have to be corrected. Faults which have to be corrected with a high urgency shall be handled first. In cases where those errors have also a huge impact on the overall system, like a program termination, the scheduler must take intermediate action. When several errors of such a severity occur in parallel, the adherence to the real-time constraints will be of minor importance.

In general, the scheduler has to solve a multi-objective problem. The first object is keeping the real-time constraints, while the second object is to maximize the QoS. When several errors occur in a short time interval, the scheduler has to choose correction methods which keep the real-time constrains but eventually lower the quality of the output. For example, if three macroblocks are decoded incorrectly due to an error-affected motion vector and there is no time to recalculate those macroblocks, it would be sufficient to display grey blocks or a neighbor block to keep the dead-lines. In a worst-case scenario, all errors on these macroblocks would have to be ignored.

The third step for the scheduler is to integrate the error correction methods into the system schedule. Depending on the specific type of scheduler used, error handling might be added to the system schedule with a high priority in a priority-based scheduler, whereas a TDMA-based scheduler might reserve time slots for error correction. The scheduler has to be sufficiently flexible to change the error correction order on demand if new errors show up; e.g. when a recently occured error has a higher urgency than the error which is already scheduled for recovery.

In the following sections, we investigate functions of our H.264 decoder which are important for the QoS and give first hints where to annotate the source code.

IV. Application Analysis

In this section, we analyze an H.264 decoder, which we consider typical embedded application with soft real-time requirements. H.264 or *MPEG-4/Part 10, ISO/IEC 14496-10* [4] is a standard for video compression used for BluRay discs, digital video broadcasting, videoconferencing, and many other applications. The standard describes different profiles ranging from the Constrained Baseline Profile (CBP), which defines a subset of the Main Profile (MP), to the High 4:4:4 Predictive Profile (Hi444PP), which supports advanced features like up to 4:4:4 chroma sampling, up to 14 bits per sample, and additionally supports efficient lossless region coding and the coding of each picture as three separate color planes.

Here, we analyze the source code of a simple H.264 decoder [5] consisting of about 3000 lines of C code and 1000 lines of header files. The decoder implements a subset of CBP:

- I and P Slices (restricted to one back reference)
- CAVLC Entropy Coding
- 4:2:0 Chroma Format
- 8 Bit Sample Depth

Multiple reference frames, arbitrary slice ordering, and redundant slices are not supported in this implementation.

For complexity reasons, we currently restrict our transient error impact analysis to the functions that are used directly after one frame is copied to the NAL unit input buffer of our H.264 decoder. Here, we assume that the transfer of the input stream is error-free. Faults occurring, e.g. over packet-lossy networks, are already investigated in various publications. An overview of various approaches is given in [6]. For this first evaluation, we only consider memory errors occurring in the data segment of the application. Since the amount of memory required for the video itself is usually orders of magnitude larger than the memory required for the decoder application, we expect to catch a significant number of errors using this approach. In addition, it can be assumed that program code in a typical embedded system is usually contained in flash memory, which is far less susceptible to transient errors than dynamic RAM.

In the following paragraphs, we show selected excerpts from the analyzed H.264 video decoder and explain which specific error impacts are to be expected.

Frame decoding

The frame decoding routine is shown in listing 1. According to our assumption that errors occur only after the NAL unit has been copied to the input buffer, line 3 is not affected by memory faults. Thus, in the following paragraphs, we will concentrate on lines 8 and 9.

```
1  frame_t * h264_decode_frame()
2  {
3    while(get_next_nal_unit(&nalu))
4    {
5      if(nalu.nal_unit_type==1 || nalu.nal_unit_type
           ==5)
6      {
7        ++frame_no;
8        decode_slice_header(&sh,&sps,&pps,&nalu);
9        decode_slice_data(&sh,&sps,&pps,&nalu,this,ref
             ,mpi);
10       frame_t *temp;
11       temp = this; this = ref; ref = temp;
12       return temp;
13     }
14   }
15   return NULL;
16 }
```

Listing 1. Decode Frame Routine

Decode Header Information

The most critical part of decoding is the processing of header information. Here, a flipped bit can have fatal consequences for the decoding of the whole frame and even of subsequent frames. If, for example, the frame type is decoded incorrectly, the header bits will be misinterpreted, since different frame types use different layouts for header data.

The majority of header items are integers encoded with *Exponential-Golomb codes*, which can encode arbitrary positive integer numbers. The construction scheme favors small numbers by assigning them shorter codes. A value of x has a representation of:

$$x = 2^n - 1 + v$$

where n is the number of "0" bits followed by one "1" bit, and v represents an offset of the next n bits. For example, the decimal value "5" is encoded as "00**1**10". In our decoder, the decoding is implemented as follows:

```
1  int get_unsigned_exp_golomb()
2  {
3    int exp;
4    for(exp=0; !input_get_one_bit(); ++exp);
5    if(exp)
6      return (1<<exp)-1+input_get_bits(exp);
7    else
8      return 0;
9  }
```

Listing 2. Exponential-Golomb Calculation Routine

Here, the occurrence of a transient memory error can have different impacts, depending on the program code affected:

1) An error which hits the n bits representing v (`input_get_bits(exp)` – line 6) only has an impact on the returned value. However, the value may be used later on to calculate offsets in address operations. This, of course, will lead to a crash with a high probability.
2) If, however, the part which calculates n (`for(exp=0;!input_get_one_bit();++exp);` – line 4) is affected by an error, the complete application flow is disturbed, i.e., not just the output is different than expected. Reading the wrong value for n implies to read the input stream either not far enough or too far. As a consequnce, subsequent calls of any `input_get_...` function (cf. listing 3) will read at the wrong input offset.

```
1  int input_get_one_bit()
2  {
3    int res=(nal_buf[nal_pos]>>(7-nal_bit))&1;
4    if(++nal_bit>7)
5    {
6      ++nal_pos;
7      nal_bit=0;
8    }
9    return res;
10 }
11
12 int input_get_byte()
13 {
14   return nal_buf[nal_pos++];
15 }
```

Listing 3. Input Reading Functions

Especially the second impact can completely change the *timing behavior* of a system. In a worst-case scenario, `input_get_one_bit()` always returns "0", which results in an endless loop. Such kind of application code is an indication of a section in which protecting memory data reads from errors is of major importance.

Decode Frame Data

Decoding frame data is not as critical as decoding the header. In most cases, only incorrect pixel values will be read from the input. However, there are also critical functions. One of these is `get_code`, which is shown in listing 4.

```
int get_code(code_table *table)
{
  unsigned int code=input_peek_bits(24)<<8;
  int min=0, max=table->count;
  while(max-min>1)
  {
    int mid=(min+max)>>1;
    if(code>=table->items[mid].code)
      min=mid;
    else
      max=mid;
  }
  input_step_bits(table->items[min].bits);
  return table->items[min].data;
}
```

Listing 4. Get Code Table Entry

This function implements a binary search in a code table. Due to different coding sizes, the number of bits is also stored in the table. Here, a transient memory error can cause the return of a corrupted data item and the invalid bit count, which results in an incorrect reading of the input by any subsequent `input_get_...` function.

Another critical function we found is `residual_block`, which is used for parsing transform coefficient levels. This function uses code tables and hence `get_code()` to decode the coefficients.

Hence, a error affecting the `get_code()` function can have a different impact on the QoS, depending on the *current context* of the function call. If the function is used to get data of a motion vector, the impact on the QoS and the urgency of error correction will be higher than if data for an intra macroblock is extracted from the input stream. Thus, annotations should not be placed inside of `get_code()`, since then always the highest impact and urgency would have to be assumed, and the choice of alternative error correction methods would be limited. Instead, it is more reasonable to place an annotation in the calling function, or even another function further up in the call hierarchy. Here, the intention of the `get_code()` is known and the annotations can be more specific.

Our evaluation has shown that the majority of application crashes were caused by errors affecting `residual_block`, `get_(un)signed_exp_golomb`, and `decode_slice_header`.

V. Evaluation

A. Application

To evaluate the error classification approach, we have developed an application that decodes a video using the decoder analyzed in the previous section. This application compares a frame decoded under the influence of error injection with the corresponding correctly decoded frame. The results are visualized using an in-house developed tool shown in fig. 2. The tool displays the correctly decoded frame, the decoded frame under error injection, and the difference between the two frames. In addition, the metrics described below are calculated and displayed for each frame.

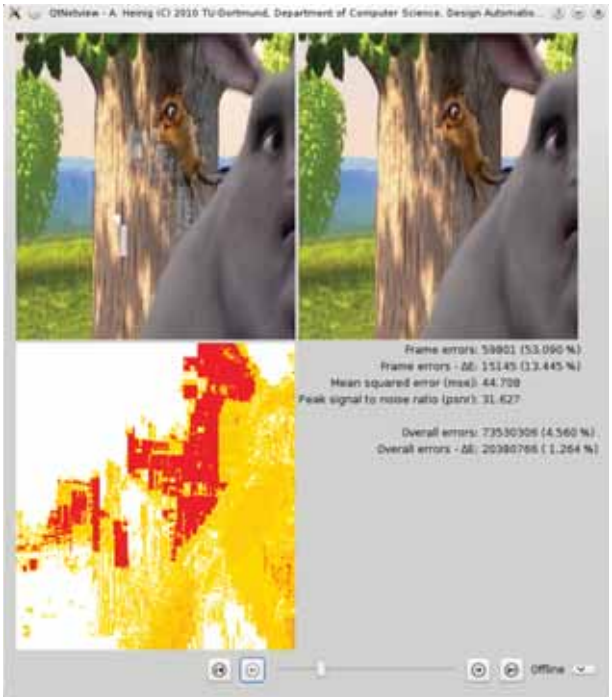


Fig. 2. Error Analysis Tool

B. Error Metrics

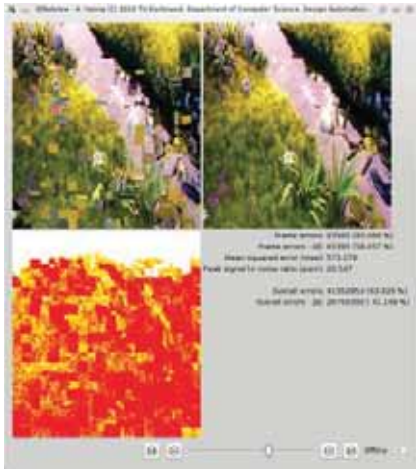
Since we implement a best-effort approach to error correction, it is to be expected that some of the occuring errors will propagate to the output. Nevertheless, we have to evaluate whether the output is acceptable for the user. In order to distinguish between *perceptible* and *non-perceptible* errors, we used different metrics:

1) ΔE:
   The ΔE metric is standardized in ISO 12647, which describes the distance between two colors. We define that a ΔE value smaller than 5.0 indicates a *non-perceptible* error. Our visualization tool shows the frame under error injection, the correct decoded frame and a difference frame showing perceptible and non-perceptible errors (indicated by the intensity of the colors) by using this metric; in addition, statistics giving the percentage of errors accumulated over the whole video and statistics of the current frame are displayed.
2) PSNR:
   The peak signal-to-noise-ratio is most commonly used to measure the quality of reconstruction of lossy compression codecs. PSNR is defined by

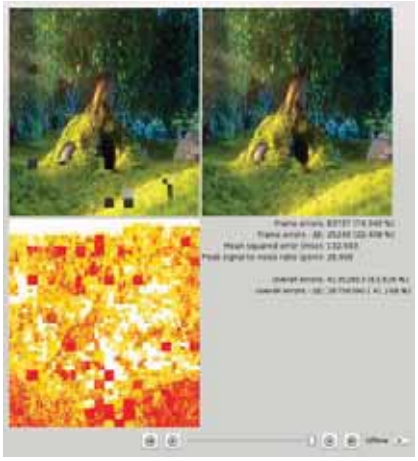
$$\mathrm{PSNR} = 10\log_{10}\frac{2^B-1}{MSE}\mathrm{dB}$$

   where MSE denotes the mean squared error between reconstructed and original frame, and B is the number of bits per sample. A higher PSNR indicates higher quality. In fig. 3, we give examples for different PSNR values. In (a), the original frame is not recognizable anymore,

(a) PSNR = 13.5 dB



(b) PSNR = 20.5 dB



(c) PSNR = 26.9 dB

Fig. 3. Comparision Of Different PSNR Values

whereas in (b) and (c), one can still recognize the content of the frame.

C. Error Injection

In order to provide a reproducible, deterministic error scenario, we added a separate error injection thread to the video decoder application. The memory ranges where the errors are injected are freely configurable. All injected single-bit errors are registered by the injection thread and can be reverted on demand to simulate error correction taking place, which corresponds to a *delayed error correction*. However, this will not correct errors which already have been propagated to other locations. The error model employed is quite simple, a randomized uniform distribution of flipped bits was implemented. Additionally, our application allows to suspend error injection at any arbitrary point, which corresponds to an *immediate error correction*. Finally, a simple flush of the rollback buffer simulates *ignoring* of errors.

To investigate the effects of errors, we used an artificially high error rate of up to 240 injected errors per second. Thus, fatal errors like a crashed application can be found in a short amount of time. In addition, a video decode run can be analyzed to generate the results. This analysis was performed by comparing each decoded frame under error injection with the expected correct frame.

D. Injection Scenarios

In our first evaluation, we decided to activate error injection only in `decode_slice_data` (cf. listing 1 line 9). Thus, it is ensured that the header was decoded correctly (line 8) and the current NAL unit was copied without errors. This simulates an immediate error correction with rollback and recalculation.

We enabled the injection of errors in the NAL buffer and in the frame buffers. However errors injected in the frame buffers have shown very little impact on the application, since they modify only single pixels in the displayed frame or in the reference frame. Hence, we will only concentrate on NAL buffer errors for the rest of this paper.

In the previous chapter, we manually identified `residual_block` as a critical function of the H.264 decoder. This function is called repeatedly from inside a `while` loop inside `decode_slice_data`. We analyzed various interesting combinations, where we selectively switched off error injection and/or corrected errors at that location to obtain hints for future application annotation possibilities:

A) *No error correction in* `decode_slice_data()` (cf. listing 1 line 9)

B) *Error correction in* `residual_block`
In this scenario all errors in the NAL-Buffer that occurred so far are corrected and no new errors are injected while reading the NAL-Buffer.

Scenario A and the perfectly decoded video are the extrema for our evaluation. They are used to evaluate the results of scenario B. With scenario B we like to show that the protection of the manually identified function `residual_block` will lead to significant improvements of the QoS.

E. Results

According to a profiling of the decoder using the `gprof` tool, the investigated function `residual_block` accounts for 26.0% of the overall execution time of the decoder for 2,880 frames (120 seconds at 24 fps) of our example video. Overall, the function was called 4,582,301 times. The NAL-Buffer copy code and the header decoding function, where we do not inject errors, were called 2,880 times each and only account for less than 0.02% of the overall run time. Taken together, we investigated error correction in about 26.2% of the application runtime.

We concentrate on the influence of our error correction approach on the QoS. Therefore, we use two error injection rates: 240 errors and 80 errors per second, which results in 10 respectively 3.33 injected errors per frame on average. The results of the previously described scenarios A and B are shown in table II.

Injected faults per second	240		80	
Scenario	A	B	A	B
Program Termination (crashes) [%]	60.07	16.79	48.69	1.88
Frames with perceptible errors ($\Delta E > 5$) [%]	57.49	28.11	41.99	3.77
Average PSNR of frames with errors [dB]	32.87	34.35	35.86	38.83

TABLE II
SIMULATION RESULTS

As expected, the impact on the QoS grows with a growing error rate. The number of crashes of scenario A will increase by a factor of 1.2 and the amount of frames with perceptible errors increase by a factor of 1.3 if we triple the number of injected errors. Those factors are significantly higher for scenario B. Here, the crashes will increase by a factor of 8.9 and the visible errors increase by a factor of 7.4 if we triple the injection rate. These values clearly show that the underlying error model—in our case the random error distribution in the NAL buffer—has a huge impact on the error handling method to be used. Further research is required to extend the error classification approach with respect to the error model of the underlying hardware components.

The comparison of scenario A and B shows a significant improvement of the QoS. Especially in the case where we inject 80 errors per second the amount of crashes is reduced by a factor of 25.8 times when enabling error handling in the `residual_block` function, and the numer of perceptible errors is reduced by a factor of 11.1.

Taken all scenarios into account, we can also determine an increasing PSNR. This means, that even if the frame has perceptible errors, those errors will be harder to notice.

These results back up our manual analysis. We clearly identified functions that are extremely fault-critical. However, in at least 1.88% of our test cases, the application is still crashing. Those crashes are mainly caused by other, non-annotated functions which also read from the NAL-Buffer, since here errors are currently ignored.

VI. RELATED WORK

Currently, almost all existing error handling strategies against transient faults treat every error alike. They correct every error using techniques like *triple- modular redundancy* [7], *roll-forward checkpointing* [8], or *checkpointing and roll-back recovery* [9].

Various frameworks have already been proposed that can implement software-based techniques automatically and relieve the programmer from doing this manually. In [9], Li et al. present a compiler that inserts library calls into the program code that check whether a certain amount of time has elapsed since the last checkpoint and trigger a new checkpoint if required. Benso et al. built a software development kit [10] that comprises a compiler, middleware, a device emulator tool, and a fault injection environment. The compiler performs source-to-source transformations like code-reordering or variable duplication that increase the reliability of code. A software wrapper is employed as middleware to protects memory accesses against faults by adding redundancy.

Alfonso et al. developed a framework [11] with a focus on multithreading. It provides a scheduler and a set of fault tolerance strategies. The user still has to provide an implementation for subfunctions, like saving and restoring of checkpoints or an acceptance test, that are needed by the chosen fault tolerance strategy. The scheduler will apply EDF (Earliest Deadline First) as a scheduling strategy.

An adaptive scheduling approach is presented in [12]. If no task of a task set will miss its deadline, deadline-monotonic scheduling is used. However, if the overhead for error handling increases so that tasks will miss their deadline, the scheduling behavior is changed to a value-based strategy. In that way, critical tasks will obtain a higher priority and have a higher chance to finish within their deadlines, while less important tasks will be delayed.

Several other scheduling approaches for real-time systems ([13], [14], [15]) address the scheduling problem from a different point of view. Here, a fixed scheduling strategy is chosen and a schedulability check is performed offline. All cited approaches assume that checkpointing is applied for error handling and that for each task the distance between checkpoints is always the same. Whereas Zhang et al. [13] check whether a certain amount of errors will not cause a task to violate its deadline given a constant checkpoint interval, Kwak et al. [14] and Punnekkat et al. [15] determine the optimal checkpointing intervals for the tasks. None of the mentioned scheduling approaches propose a tight integration of error handling into the scheduler or facilitate delayed error handling.

The idea of our classification approach to ignore some errors completely (i.e., not to handle these errors at all) is supported by the observations made my Li et al. [16]. They analyzed which impact transient faults can have on the output of multimedia and AI applications as well as SPECInt CPU2000 benchmarks. Also, Polian et al. investigate errors in ISCAS-89 sequential benchmark circuits [17] and the motion estimation part of an MPEG2 encoder circuit [18] from the same point of view. Both conclude that, besides the fraction of faults that make the system crash or lead to invalid results, a large number of transient faults do not have any effect on the output of the application. Some faults lead to deviations in the output or the inner state of the application, but they are within a tolerable range for the user. Especially for multimedia applications, absolutely correct results are not required, since the user will hardly recognize the difference. Unfortunately, in both papers the resiliency of different parts of the applications has not been investigated.

Some approaches already exploit these observations for selective error protection techniques. Sundaram et al. [19] assume that the memory is protected using Error Correcting Codes and only instructions can be effected by faults. They de-

cided to protect only arithmetic instructions with an influence on the control path and address calculations by replication. To determine these instructions, they use static analysis. In this way, the replication overhead can be reduced by up to 33% compared to a system were every instruction is protected, the fidelity degrades only by 1%. Since our approach focuses on faults in memory, this is a complementary approach. Mehrana et al. [3] assume that only a part of the memory system can be affected by transient faults. Thus, there are reliable and unreliable memories. Their approach is to place memory objects with a long lifetime in reliable memories, whereas all other memory objects are placed in the remaining memory space. As a consequence, these can be affected by faults. About 99% of the transient faults are covered by this approach, but only 36% of the memory objects have to be protected against faults on average. This approach is based on the assumption that objects that reside in memory for a longer time span are exposed to faults with a higher likelihood than objects with a short lifespan. Nevertheless, the approach does not consider the relevance of objects. In contrast to our approach, the objects are protected against faults and no error handling is applied.

Methods for cross-layer resiliency are the topic of current research projects at Intel [20] and IBM [21]. The authors claim that cross-layer approaches to build resilient systems can significantly reduce the cost of such systems and may also offer possibilities for increased performance and reduced power consumption. While these publications cover a wider range—from the silicon level up to the application—than envisioned in our approach, selective error handling for a best-effort error recovery approach, which may enable the system designer to further lower overheads, is not considered here.

VII. Conclusions and Future Work

In this paper, we proposed an approach to error classification that enables embedded systems developers to handle errors in a flexible way. Using a cross-layer error handling approach, errors can be handled according to their context as well as the current overall timing conditions of the system, employing a best-effort approach in order to adhere to timing and quality requirements whenever possible. The soundness of our proposed system structure is backed up by an analysis of a typical, soft real-time critical H.264 decoder application.

Evaluating our error classification approach using an H.264 video decoder shows that using application knowledge can significantly reduce the amount of errors that have to be corrected while still maintaining acceptable quality of the resulting output video stream. This provides the justification for our system design, which knits a close connection between application semantics and scheduling decisions. Embedded system designers can so exploit the error classification and reduce the related correction overhead significantly while still increasing the reliability of the system.

The corner cases of our evaluation show that a typical multimedia application in many cases depends on valid input data to function correctly. For example, in many cases range checks of input parameters in a data stream are omitted for the sake of performance. There are even corner cases to be found where such an unchecked value is used directly to calculate some pointer offsets, leading to application crashes when a parameter exceeds an expected value range. This creates an additional challenge for dependable systems, since application crashes require more overhead to fix and thus are highly likely to violate real-time constraints. These programming practices, however, do not only affect the *resiliency* of a system, but also the *security*, thus safer programming practices may also improve the resiliency of an application.

The next steps required in building our system include the definition of a more precise semantics of error classifications and an assessment of the timing overhead different EDAC approaches require. Currently, the most tedious task is the manual creation of application annotations. We do not expect this procedure to be fully automatable. However, static analysis tools may be able to support an application developer in creating annotations to indicate error classification.

To evaluate the automated annotation mechanism to be developed and show its general applicability, it will in addition be important to analyze applications showing similar real-time characteristics such as audio players, media recording software or variants of the H.264 decoder analyzed in this paper, e.g., the decoder reference implementation.

We have not considered code segment or OS resiliency in the context of this paper. For typical embedded systems, code often resides in flash memory, which is far less susceptible to transient errors than RAM. However, transient errors in RAM will also affect the reliability of the OS itself. Handling cases of OS RAM corruption in a real-time context will be a challenging topic for future research.

Acknowledgments

We appreciate the support of Martin Fiedler, the original author of the H.264 decoder analyzed in this paper [5], and Ingo Korb, who found and fixed errors in the original implementation.

References

[1] International Technology Roadmap for Semiconductors (ITRS), "International Technology Roadmap for Semiconductors, 2009 Edition, Executive Summary, http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf," 2009.

[2] A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, "Using Application Knowledge to Improve Embedded Systems Dependability," in *Proceedings of HotDep 2010*. Usenix, 2010 (to appear).

[3] M. Mehrara and T. Austin, "Exploiting selective placement for low-cost memory protection," *ACM Transactions on Architecture and Code Optimization*, vol. 5, no. 3, pp. 1–24, 2008.

[4] T. Wiegand, G. Sullivan, J. Reichel, H. Schwarz, and M. Wien, "Joint Draft ITU-T Rec. H.264 — ISO/IEC 14496-10 / AMD.3 Scalable video coding," Joint Video Team (JVT) of ISO/IEC MPEG & ITU-T VCEG, Geneva, Switzerland, Doc. X201, Jun./Jul. 2007.

[5] M. Fiedler, "Implementation of a basic H.264/AVC Decoder," *TU Chemnitz (Seminar Paper)*, 2004.

[6] T. Stockhammer and T. Wiegand, "Video Coding And Transport Layer Techniques For H.264/AVC-Based Transmission Over Packet-Lossy Networks," in *IEEE International Conference on Image Processing (ICIP)*, 2003.

[7] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM J. Res. Dev.*, vol. 6, no. 2, pp. 200–209, 1962.

[8] D. K. Pradhan and N. H. Vaidya, "Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture," *IEEE Trans. Comput.*, vol. 43, no. 10, pp. 1163–1174, 1994.

[9] C.-C. J. Li, E. M. Stewart, and W. K. Fuchs, "Compiler-assisted full checkpointing," *Software – Practice & Experience*, vol. 24, no. 10, pp. 871–886, 1994.

[10] A. Benso, S. Chiusano, and P. Prinetto, "A software development kit for dependable applications in embedded systems." IEEE Comp. Society, 2000, pp. 170 –178.

[11] F. Afonso, C. A. Silva, A. Tavares, and S. Montenegro, "Application-level fault tolerance in real-time embedded systems," in *SIES*, 2008, pp. 126–133.

[12] P. Richardson, L. Sieh, and A. Elkateeb, "Fault-tolerant adaptive scheduling for embedded real-time systems ," *Micro, IEEE*, vol. 21, no. 5, pp. 41 –51, sep/oct 2001.

[13] Y. Zhang and K. Chakrabarty, "Fault Recovery Based on Checkpointing for Hard Real-Time Embedded Systems," in *In Proc. of the 18 th IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003, pp. 320–327.

[14] S. W. Kwak, B. J. Choi, and B. K. Kim, "Checkpointing strategy for multiple real-time tasks," in *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, 2000, pp. 517 –521.

[15] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Syst.*, vol. 20, no. 1, pp. 83–102, 2001.

[16] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *Proc. of the 13th Int'l Symp. on High Performance Comp. Architecture*, 2007, pp. 181–192.

[17] I. Polian, S. M. Reddy, I. Pomeranz, X. Tang, and B. Becker, "No Free Lunch in Soft Error Protection?" in *Proc. of the 2nd Workshop on Dependable and Secure Nanocomputing*, Anchorage, Alaska, USA, 2008.

[18] I. Polian, B. Becker, M. Nakasato, S. Ohtake, and H. Fujiwara, "Low-Cost Hardening of Image Processing Applications Against Soft Errors," in *DFT '06: Proc. of the 21st IEEE Int'l Symp. on Defect and Fault-Tolerance in VLSI Systems*. Washington, DC, USA: IEEE Comp. Society, 2006, pp. 274–279.

[19] A. Sundaram, A. Aakel, D. Lockhart, D. Thaker, and D. Franklin, "Efficient fault tolerance in multi-media applications through selective instruction replication," in *WREFT '08: Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies*. New York, NY, USA: ACM, 2008, pp. 339–346.

[20] N. P. Carter, H. Naeimi, and D. S. Gardner, "Design Techniques for Cross-Layer Resilience," in *Proceedings of IEEE/ACM Design Automation and Test in Europe, Dresden, Germany*. IEEE Press, March 2010, pp. 1023–1028.

[21] S. Mitra, K. Brelsford, and P. N. Sanda, "Cross-Layer Resilience Challenges: Metrics and Optimization," in *Proceedings of IEEE/ACM Design Automation and Test in Europe, Dresden, Germany*. IEEE Press, March 2010, pp. 1029–1034.