Temporal Properties of Error Handling for Multimedia Applications

Michael Engel, Andreas Heinig, Florian Schmoll, and Peter Marwedel
TU Dortmund
Faculty of Computer Science
Design Automation of Embedded Systems
Email: {firstname.lastname}@tu-dortmund.de

***Abstract*—In embedded consumer electronics devices, cost pressure is one of the driving design objectives. Devices that handle multimedia information, like DVD players or digital video cameras require high computing performance and real-time capabilities while adhering to the cost restrictions. The cost pressure often results in system designs that barely exceed the minimum requirements for such a system.**

Thus, hardware-based fault tolerance methods frequently are ignored due to their cost overhead. However, the amount of transient faults showing up in semiconductor-based systems is expected to increase sharply in the near future. Thus, low-overhead methods to correct related errors in such systems are required. Considering restrictions in processing speed, the real-time properties of a system with added error handling are of special interest. In this paper, we present our approach to flexible error handling and discuss the challenges as well as the inherent timing dependencies to deploy it in a typical soft real-time multimedia system, a H.264 video decoder.

I. Introduction

Hardware design for consumer electronics devices is a challenging task. Since most of the devices include embedded systems, the typical restrictions inherent to embedded development apply—the digital components of a consumer electronics device are limited in factors like processing power, memory, and available energy. In addition, rapid development cycles do not leave much room for extensive optimizations.

An additional constraint will aggravate hardware development in the near future. According to the ITRS roadmap [1], the amount of faults showing up during the regular operation of semiconductor components is expected to rise sharply. This can be attributed to two major causes: shrinking semiconductor structure sizes (to drive down manufacturing costs) and reduced operating voltages (to prolong battery life). Another point might also be valid for cost-sensitive devices. Here, some hardware devices will not pass all tests perfectly. They might show faults e.g. in extended temperature ranges. If these devices could also be sold to customers instead of being scrapped, this could reduce the cost pressure significantly.

However, hardware-based fault tolerance methods like ECC protection of memories or providing redundant components are in most cases too cost-intensive. Software-based fault-tolerance methods, on the other hand, require a certain overhead that increases a systems' requirements of e.g. processing speed and memory. In a video decoder, for example, a simple checkpoint-and-recovery approach might involve saving the video frame buffer before a new frame is decoded and restoring the previous checkpoint (and, possibly restarting the decoding of the current frame) in case any faults shows up.

In a previous publication [2], we identified sections in an H.264 decoder that are very vulnerable to faults. Using a simulation with an extraordinary high fault injection rate it was shown that if error correction is only applied to these very critical sections, the rate of unwanted program terminations can be reduced dramatically and the video quality will increase significantly.

Based on this, the approach presented in this paper intends to reduce the overhead of software-based fault-tolerance methods while providing acceptable quality-of-service in case of errors. This is achieved by applying a *classification* indicating the impact of an error—will the system crash, will only a pixel in video memory be altered or might there be no effect at all? Since the point in time at which an error shows up is unpredictable, correcting an error disturbs the scheduling of tasks in such a soft real-time system. Thus, in addition to observing the semantics of an error, the *current timing properties* of the system at the time the error shows up as well as the *timing properties of error correction* have to be considered in a flexible, software-based error handling approach. We describe the semantics of errors showing up in an embedded H.264 video decoder application and discuss the influence of video decoding timing properties on error handling.

The paper is organized as follows. We describe the underlying problem of transient faults in section II. The basics of our classification approach are described in section III. The timing properties of H.264 video decoding and their impact on error correction are described in section IV. Section V discusses related work and section VI concludes the paper.

II. The Transient Fault Problem

Faults in semiconductor devices can be divided into two classes. *Permanent* errors show up at a certain point in time

This work is supported by the German Research Foundation SPP1500 priority program under grant no. MA943/10-1

and persist until repair. Those might be caused e.g. by bad solder joints or electromigration. *Transient* errors are spontaneous events that result in a single disturbance in the system (e.g., a flipped bit in memory) at a certain point in time. Traditionally, cosmic radiation and atomic decay were major causes for transient faults; recently, thermal and electromagnetic effects have also shown to be a major cause, since semiconductors with shrinking structure sizes and reduced operating voltages are less robust to these effects. A number of simulation approaches exist to predict when permanent errors will show up and research in the area of *graceful degradation* intends to design systems that are able to handle such effects.

Transient faults, however, are single-shot phenomena. They affect a component of a system at a given, unpredictable moment in time and only persist until a status change. A transient fault in RAM, for example, will cease to exist when the affected memory cell is overwritten. While several transient faults might show up in a short time interval, it is generally unpredictable when such an error will hit a system and the probability that the same component is affected by two unrelated transient faults is expected to be rather low.

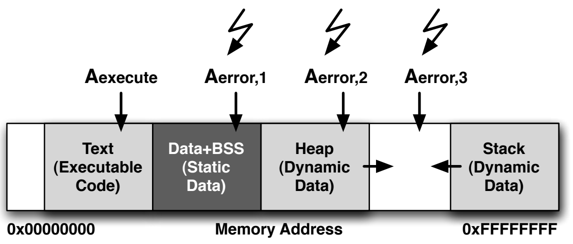


Fig. 1. Possible impact of a transient fault

An example for the impact of a transient fault is shown in fig. 1. A program consists of several memory sections: the *text* section contains executable program code, the *data* section contains statically allocated data (having a constant memory location), while the *heap* and *stack* sections contain dynamically allocated data allocated under program control. The contents and sizes of stack and heap change over time.

Here, three cases for bit flip in memory (e.g., a bit changed from '1' to '0' or vice-versa) at time t_{error} are depicted. The three depicted error cases $A_{error,1}, \ldots, A_{error,3}$ show errors affecting different memory sections. In the first case, a transient fault affects memory location $A_{error,1}$, which is in the data segment. The second case hits $A_{error,2}$ in the heap segment. The third case, $A_{error,3}$ is the easy one. Here, memory that is currently unused (in the "empty" area between stack and heap) is affected. The first two memory locations contain an object with differing lifetimes and semantics, while the third one is not allocated to any object in the system.

Simple error handling methods (like hardware-based ECC) incur the same overhead for every error, since they have no way to determine the consequences of an error. A software-based *flexible* error handling approach should at least be able to distinguish between the cases described above and apply the required minimum amount of corrective actions to fix the error. Advanced methods, described below, will be able to take the current execution state of an application, the current timing conditions of a system and the severity of an error into account.

The system we analyze in the following sections is an example of a typical embedded system with soft real-time requirements and related embedded constraints. We analyzed an H.264 video decoder [3] that is able to decode the H.264 constrained basic profile. This H.264 subset is typical for small embedded devices like pocket video players or simple mobile phones which have to restrict the displayable format due to memory, computing power and energy constraints. The decoder is analyzed running on the Synopsys CoMET [4] simulation platform simulating a single-core ARM926 CPU at 1,2 GHz.

III. Flexible Error Handling

A. Error Semantics

Error detection mechanism (e.g., parity bits or software-based error detection) indicate the *location* of an error in the system at a memory address A_{error}[1]. The three error cases $A_{error,1}, \ldots, A_{error,3}$ in fig. 1 each belong to objects with differing characteristics. The location $A_{error,1}$, lies in the data segment. Since elements in the data segment do not change their location, we can apply a simple mapping from address ranges to objects to obtain the affected object. The second case, $A_{error,2}$, is located the heap segment[2]. Since heap contents change over time, we need runtime information (the current memory map of the heap) to obtain the related object, which requires some overhead. The third location, $A_{error,3}$ affects memory that is currently unused (in the "empty" area between stack and heap). If we know that a memory location is currently unused, we can safely ignore the related error.

From information about the affected object alone we are not able to deduce the *impact* of an error. This impact depends on a set of parameters describing the *error context*. Most important are the point in time t_{error} at which an error that is the result of a transient fault shows up related to a program's execution, the program code that was being executed when the error hit ($A_{execute}$) and the data or code affected by the error.

The examples above show that handling transient errors cannot be statically decided except for the brute-force approach to handle every error regardless of its impact. Thus, a *classification* of transient errors is required. The classification according to the *location* A_{error} of an error can be performed statically by (automatically or manually) annotating code and data sections of an application. The temporal properties, i.e. the impact of an error showing up at time t_{error}, have to be analyzed ad-hoc while the affected program is being executed.

One question that has not been discussed so far is how to actually mitigate the effects of an error. Literature on dependable systems provides a plethora of possible approaches, including

[1]For the sake of simplicity, we only consider transient *memory* errors in the context of this paper. Our approach can be extended to errors in other components like registers, busses, etc.

[2]The same would be valid if $A_{error,2}$ is located in the stack segment.

hardware redundancy, voting decisions, and error-correction codes. One particularly simple approach is checkpointing and recovery. This method takes snapshots of the mutable state of an application in regular intervals. In case of an error, the most recent snapshot is restored and the actions since the snapshot repeated. In the case of our video decoder, a checkpoint could be taken after each frame is successfully decoded. If an error shows up during the decoding of the subsequent frame, the decoding could be terminated, the snapshot containing the last frame restored and the decoding of the current frame would start from the beginning. Obviously, this has impact on the system's timing. This impact is analyzed in section IV.

B. Error Modeling

We now present a model of our classification approach that takes the location A_{error} of a transient error into account. Further on, we describe how the timing properties of H.264 video encoding affect our model.

In most cases, the only information provided by error detection methods is the affected memory address A_{error} and the location in the application's address space $A_{execution}$ from where the related memory access was initiated. However, the hardware does not provide information about the semantics of those locations. In our error correction approach, the error correction mechanism has to consult a mapping function f_{map} to deduce the affected object. This information alone, however, is not sufficient to obtain information about the impact of the error. Here, a *semantics function* f_{sem} that provides a *classification* of the error impact is required. In general, f_{map} has to consider the error location, the code location executed at the time of the error and the time of the error itself:

$$Object := f_{map}(A_{error}, A_{execution}, t_{error})$$
$$Classification := f_{sem}(Object)$$

The classification returned by the f_{sem} function consists of a triplet of data describing the impact and error handling:

$$Classification := (Impact, Urgency, \{M | M \in Methods\})$$

The *Impact* is an indicator for the loss in quality-of-service that results: $Impact = \Delta QoS$, the *Urgency* is an indicator telling how soon this error has to be handled. The set of *Methods* consists of the following triplets M:

$$M := (f_{handle}, \Delta t_{handle}, \Delta QoS)$$

Thus, the error handling system can select from a set of error handling methods f_{handle} with differing QoS gains ΔQoS and timing properties Δt_{handle}.

The basis for selecting a certain error handling method depends on the current timing properties of the system. For our video decoder, only a certain amount of time t_{corr} is available to correct the error. The value of t_{corr} is the amount of time from the detection of the error until the planned start of the subsequent frame. In case error correction requires more time, the visible effect of performing the correction is *jitter*, i.e., the point in time at which the following frame (and, possibly, subsequent frames) start deviates from the planned time deduced from the frame rate.

$$t_{corr} := f(t_{current}, t_{start,frame}, t_{frame})$$

Depending on the value of t_{corr}, the *selection function* f_{sel} can select the appropriate and affordable error handling method:

$$Method := f_{sel}(Methods, t_{corr})$$

The selected method can then be executed in order to correct the error. To ensure that executing the correction method will not exceed the real-time constraints, information on the *run-time of the correction method* and the *available time to correct an error* t_{corr} is required.

The estimation of t_{corr} is an important factor that increases the complexity of this method. Our approaches to calculate values for t_{corr} for our H.264 video decoder example and the related complications are described in the following section. The run-time of an error correction method, in comparison, is rather easy to obtain using standard methods. An approach for the checkpointing and recovery method described above is also shown below.

IV. Timing Properties of H.264 Videos

In order to perform error correction while keeping the given real-time constraints, we would need to know how long the decoding of the current frame will take. The basic real-time constraint for the system is the *frame rate* of the video. E.g,. with 25 frames per second, the decoding time t_{decode} for each frame must be $\leq$ 40 ms, the frame duration t_{frame}. A simple model for the distribution of t_{decode} among the various tasks involved is shown in fig. 2.

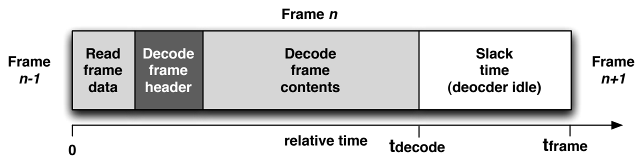


Fig. 2. Decoding of a single frame

The processing of each frame consists of several steps. After decoding the current frame, the decoder can stay idle until t_{frame} has passed and the decoding of the subsequent frame begins. In reality, the decoder will not decode every frame on demand, but rather *buffer* a number of decoded frames. However, the amount of bufferable frames is restricted by the amount of memory available in the system. Thus, reducing the amount of frames to buffer reduces the memory requirements and the cost of a system. Thus, slack time in a system might accumulate over a small number of decoded frames; following the simple model of fig. 2, we ignore the accumulation of slack.

In order to assess the amount of time available for error correction, it would now be beneficial if the slack time $t_{frame} - t_{decode}$, which would be available for our correction method, could be calculated.

A. WCET estimation

A first approach to obtain timing information is trying to calculate an *upper bound* for the decoding time of a frame, the so-called *worst case execution time* (WCET) [5]. In general, the WCET of a given program is not precisely computable, thus WCET analysis tools generate a *safe upper bound* WCET_{EST} for the WCET. This safe upper bound is guaranteed to be larger than any execution time for arbitrary input data sets, however, it is not guaranteed to be close to the real WCET. Fig. 3 (adapted from [6]) shows the relation of the various execution times of a process.

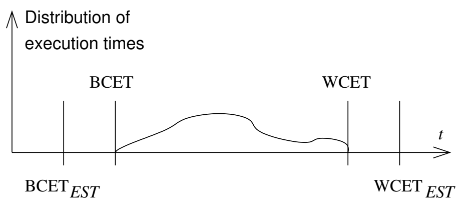


Fig. 3. Timing properties of a process

One of the major problems in WCET estimation is the calculation of upper bounds for the number of iterations of a given loop. Here, H.264 poses a rather large set of problems including the large variability of parameters for videos (even using the restricted base profile) and the run-length encoding of numerical values that is used in H.264.

Using the code of our H.264 decoder, we performed an analysis of the decoder function using the aiT [7] WCET analysis tool for an ARM7-based platform and compared the results with the measured execution time. This WCET estimation yielded a WCET_{EST} that is ten times higher than the highest observed frame decoding time. Obviously, this estimation is useless for our low-overhead error correction approach.

B. Variations in frame sequences

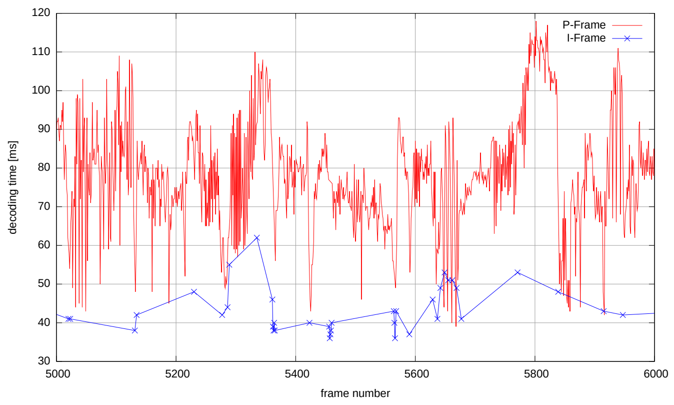


Fig. 4. Frame decoding time for successive frames

The subsequent approach employed was to try finding relations between subsequent frames by analyzing typical H.264 encoded videos, as shown in fig. 6[3]. Fig. 4 depicts the required decoding time in ms per frame for a section of video C. The figure shows that we cannot expect to find a dependency between subsequent frames. The diagram shows that it is useful to distinguish between P- and I-frames. In contrast to P-frames, I-frames are decoded faster and the number of I-frames is smaller by an order of magnitude or more.

	Video	Resolution	frame count
A	TV series	320×352	15499
B	Computer animated movie	320×352	14314
C	Music video	640×360	9083
D	TV movie	620×476	62952
E	Computer animation	848×480	21244

Fig. 6. Analyzed videos

C. Encoded frame size vs. decoding time

Neither static WCET analysis nor knowledge about frame sequences are a suitable basis to estimate the decoding time of a frame. Thus, our next approach was to use the size of the encoded NAL unit[4] to estimate the time needed for decoding. The results for our sample videos are depicted in fig. 5. In the case of I-frames, we are able to define a probabilistic upper bound. For example, if we have a frame with an average macroblock size of 40 bytes, then we will know that in most cases, we do not need more than 70μs to decode this macro block. For a complete NAL unit, if the decoded I-frame is 4000 bytes large representing 100 macroblocks, then it is likely that we need at most 7 ms for decoding.

We cannot define such an upper bound for P-frames. Hence, before decoding a P-frame, we do not know how long the decoding will take. Thus, when an error hits, we can obtain the amount of time left until the decoding of the next frame should start, but we do not know if the restoration of a previous checkpoint and restarting of the decoding of the affected frame will violate the timing constraints. If one could deduce the decode time of the current frame from available data, this question could be answered. Unfortunately, no information about the NAL unit content like the number of encoded motion vectors or intra macroblocks is known until the complete NAL unit has been decoded.

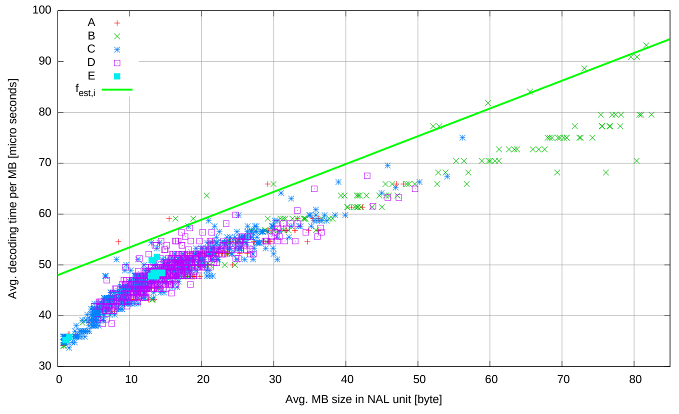
D. Encoded frame size vs. decoding time, classified

Among the information that is available *while decoding a frame* is the overall size of the frame represented by the current NAL unit and the amount of data that has already been processed for that frame. To obtain more information on the relation between the NAL unit size and the decoding time of a frame, we analyzed the videos shown in table 6.

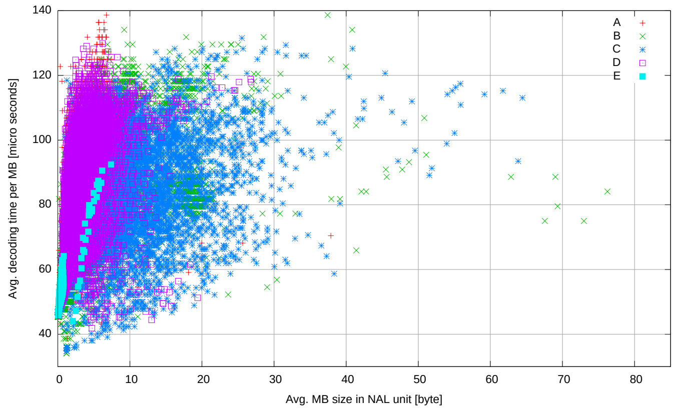
The decoding progress of P-frames is depicted in fig. 7, which shows measurements taken during decoding of a frame.

[3]The encoding of all videos was performed by the x264 encoder running on Linux with parameters generating the required restricted base profile.

[4]Network Abstraction Layer



(a) I-frames



(b) P-frames

Fig. 5. Decoding time in relation to NAL unit size

On the x axis, we show the ratio of the NAL unit part that has already been decoded to the complete NAL unit size. On the y axis, the ratio of the elapsed time to the decoding time for the complete frame is displayed.

We are interested in a lower boundary which represents a prediction of the complete decoding time based on the knowledge of the elapsed time when a certain amount of the NAL unit has been decoded. For the diagram in fig. 7 (a), it is very hard to find such a boundary. Since a significant disturbance is observed, a precise estimation is not possible.

However, if we filter NAL units which are smaller then 2 KiB (b) resp. 4 KiB (c), then we can define a lower boundary for the relative execution time by the curve $f_{est,p}$. Now, we can estimate the decoding time of the frame depending on the decoding progress of the NAL unit so far.

If, for example, a transient fault affects the decoder and error correction has to be applied, the fraction x of the NAL unit size that has been decoded is calculated. Using the function $f_{est,p}$ the fraction of time that has been elapsed so far is determined with a high probability. In this way the time that is required to decode the complete NAL unit can be estimated:

$$f_{decode} = \frac{t_{elapsed}}{f_{est,p}(x)}$$

It is obvious that the precision of this estimation increases with the amount of data already decoded. Nevertheless, this estimation is very vague. With our definition of $f_{est,p}$, we chose to be conservative. For NAL units with a size less than 2 KiB, no meaningful estimation seems possible, however, these frames are expected to have a short decoding time.

E. Timing analysis of checkpointing and recovery

For checkpoint and recovery we implemented a simple method that simply copies relevant data structures to a second storage location. Relvant data structures include the decoded frame header and the buffers that contain the reference frame, the NAL unit to be decoded and the input data of the next NAL units that has been received so far. The time for checkpointing and recovery is quite constant, since we record checkpoints only directly before a frame is decoded. At this point, the amount of data that has to be saved or restored is known rather exactly and is restricted to some global data structures.

For the video with the highest resolution, video E, the measured execution time for the creation of a checkpoint is 21.5 ms. The same amount of time is needed for recovery, since the checkpointed data is just copied in the opposite direction. The average decoding time for a single frame is 85 ms and the largest observed decoding time is 147 ms. Thus, the time required for recovery is a significant fraction of the decoding time for a complete frame. While recovery will only be applied in case of a fault, checkpointing will cause a repeating significant overhead. Thus, lightweight error correction techniques are required here.

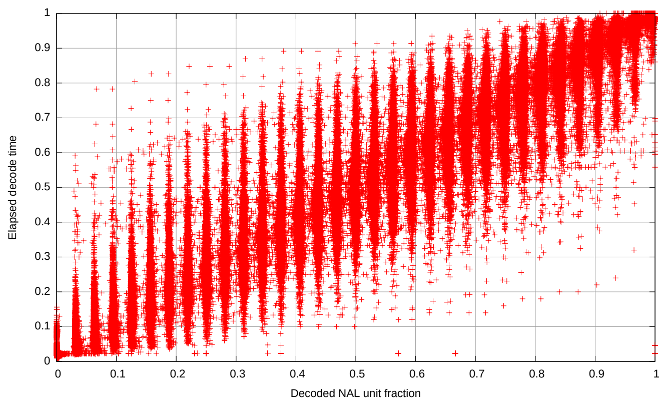
V. Related Work

So far, only few publications address error correction for multimedia applications. Polian et al. [8] analyze the effects of transient faults on the motion estimation curcuits of an MPEG-2 encoder. They observe that several faults are not critical, since their effects disappear after a few cycles. However, faults that affect these circuits will only result in a lower compression rate, while the correctness of the output is retained.

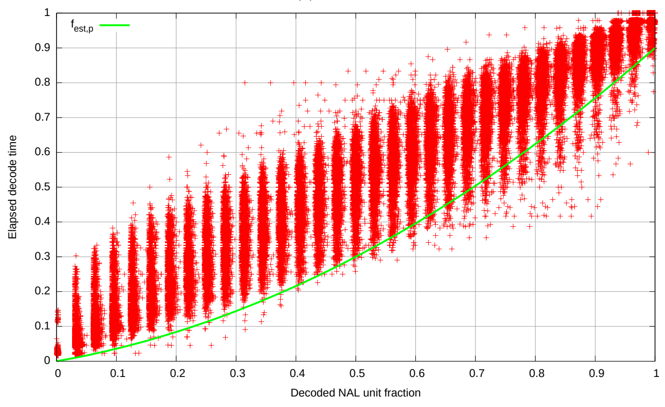
There are many approaches that suggest common error corrections techniques ([9], [10], [11], [12]). Since they are not designed to consider specific features and requirements of embedded multimedia applications, they can scarcely be applied and would result in excessive resource requirements.

Moreover, some publications analyze the runtime performance of multimedia applications. Hughes et al. [13] analyze a couple of video and audio encoders as well as decoders. They measure the execution time and count the executed instructions per frame. Large variations in these metrics can be observed for the H.263 decoder. They come to the conclusion that the variations depend on the applications and the inputs. Features of the execution platform have only little influence.

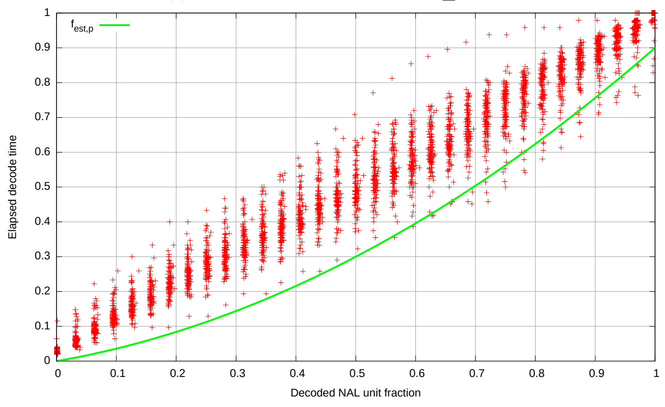
An H.264 decoder is studied in [14] and [15]. In [14], an optimized main and high profile decoder is compared with an H.264 reference implementation, MPEG-4, and MPEG2



(a) P-frame



(b) P-frame filtered: NAL unit size $\geq$ 2 KiB



(c) P-frame filtered: NAL unit size $\geq$ 4 KiB

Fig. 7. Relative decoding times within one frame

decoders. Here, the characteristics and average performance parameters of the decoders are studied intensively from points of view. In contrast to our observations, they state that the variations per frame type within a video are very small.

Horovitz [15] suggests an analytical approach to estimate the time and space complexity of a baseline profile decoder that uses UVLC encoding. Here, the H.264 decoder is also analyzed to create inputs for the estimation model and to evaluate it. Whereas the storage requirements of the decoder can be estimated easily due to the static nature of the decoder, the estimation of the execution time lacks accuracy.

To our knowledge, our publication is the first one that analyzes the runtime performance of a video decoder regarding the slack time that is available for error correction.

VI. Conclusions

In this paper, we discussed the problems of error correction in a real-time sensitive H.264 decoder. Our model extends our previous approaches [2], [16] that describe the idea of error classification and its implications for soft real-time systems.

The structure of the H.264 video format is obviously optimized for size. Some of the optimizations, like run-length encoding of numerical values, prove to be very hard cases for timing analyses. The WCET analysis of the H.264 decoder shows that a useful static estimation of frame decoding times cannot be performed, since the resulting overhead just to perform error correction with given real-time constraints would be unacceptable. However, the WCET estimation for the *error correction* approach works well, so that only the time available for correction has to be estimated differently.

We propose statistical methods to estimate the available amount of time. However, finding correlations to deduce this value from known parameters has proven to be difficult. The models described above need improvements to be used as basis for real-time calculations in a flexible, software-based error correction system. This is an important future research topic.

References

[1] ITRS, “Intl. Technology Roadmap for Semiconductors, http://www.itrs.net/Links/2009ITRS/2009Chapters_2009Tables/2009_ExecSum.pdf.”

[2] A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, “Improving transient memory fault resilience of an H.264 decoder,” in *Proc. of the Workshop on Embedded Systems for Real-time Multimedia (ESTIMedia 2010)*. Scottsdale, AZ, USA: IEEE Computer Society Press, Oct. 2010.

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

[4] Synopsys, Inc., “Virtual prototyping tools – http://www.synopsys.com/Systems/VirtualPrototyping/Pages/VP-Tools.aspx.”

[5] R. Wilhelm et al., “The worst-case execution-time problem - overview of methods and survey of tools,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 36:1–36:53, May 2008.

[6] P. Marwedel, *Embedded System Design*. Springer, 2011.

[7] AbsInt Ang. Informatik GmbH, “ait – http://www.absint.com/ait/.”

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

[9] 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.

[10] C.-C. Jim Li et al, “Compiler-assisted full checkpointing,” *Software–Practice&Experience*, vol. 24, no. 10, pp. 871–886, 1994.

[11] 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.

[12] 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.

[13] C. J. Hughes et al., “Variability in the execution of multimedia applications and implications for architecture,” in *Proc. of the 28th Intl. Symp. on Computer Architecture*, 2001, pp. 254–265.

[14] M. Alvarez, E. Salam, A. Ramirez, and M. Valero, “A performance characterization of high definition digital video decoding using h.264/avc,” in *Proc. IEEE Workload Characterization Symp.*, 2005, pp. 24–33.

[15] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, “H.264/avc baseline profile decoder complexity analysis,” *IEEE Trans. Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 704 – 716, July 2003.

[16] A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, “Using application knowledge to improve embedded systems dependability,” in *Proc. of the Workshop on Hot Topics in System Dependability (HotDep 2010)*. Vancouver, Canada: USENIX Association, Oct. 2010.