Using Application Knowledge to Improve Embedded Systems Dependability *

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

Semiconductor experts are convinced that the rate of soft errors occurring in electronic devices will rise to levels that regularly affect everyday operation of devices. Correcting every single error implies a significant hardware and real-time overhead, especially for embedded devices. Hence, an error classification is needed to distinguish whether an error has to be corrected or not.

In this paper, we present an approach using application knowledge. This knowledge is used to classify errors according to their *relevance* and the influence of their correction on the timing behavior of the whole system. When real-time conditions have to be met not all errors can be fixed immediately.

Using a typical soft real-time application, an H.264 video decoder, as an example, we show that error correction can be *delayed*. Furthermore, we show that the correction overhead will be significantly reduced if application knowledge is employed.

1 Introduction

Future embedded systems will be based on semiconductor devices featuring smaller structure sizes as well as reduced operating voltages. These reductions will lead to an increase in the number of transient errors in memory devices as well as logic components.

In order to create a reliable system from increasingly unreliable components, a naïve approach would be to detect and correct every single occurring error. This is a serious dilemma for embedded system developers. To correct all errors, additional amounts of the scarce resources processing power, memory and energy are required.

Often, this will exceed cost as well as performance limitations of the embedded system in question. Thus, in cases where real-time behavior is mandatory, correcting every single error makes it extremely hard to meet mandatory deadlines. Thus, a *classification* of errors according to their impact on the quality of the embedded system's output is useful. This can be achieved by correcting only certain errors. In the following sections, we describe our classification approach for transient memory errors and present our first evaluation results.

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

2 Error Classification

It is obvious that the effect of memory errors in a system varies significantly, depending on the affected memory location and the point of time the error occurs relative to a program's execution. The effect of errors can range from fatal errors like program crashes down to errors that have no effect at all, e.g., when an unused memory location is affected.

The most obvious errors are those that crash or terminate the affected application. This may be the consequence of a pointer referencing 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 whenever possible.

All other classes of errors do not cause program termination, but rather influence the result in different ways. Here, we distinguish between errors that have to be corrected and errors that may be ignored. Errors that cause a large disturbance in the output, like an incorrectly decoded frame type in a video decoder, have to be treated right away. Less significant errors are those that affect only a small part of the result or affect the result only for a short amount of time. Depending on the quality requirements, these errors may be ignored in order to adhere to the system's real-time requirements.

The least critical error class is one of the largest—errors that do not show up on the application level at all.

All errors that affect unused memory or memory that will be overwritten before reuse belong to this class.

For real-time critical systems, the effect of an error cannot be assessed according to the quality of the output alone. An additional dimension of classification, the *urgency* of error correction, is of special interest, since error correction has an influence on the real-time properties of a system. If error handling could be delayed, the scheduler of a system governing the timing behavior would be enabled to re-schedule tasks so that more timing-critical functions are performed previous to error correction. This classification is based on information provided by the application as well as the system software. Thus, real-time aware error correction systems require a *hybrid approach*. This approach has to consider information on error classification provided by the application developer as well as application-internal and system-wide timing properties.

3 Experimental Results

In this section we perform a case study to prove the concept of delayed error handling using application knowledge. Therefore, we performed an analysis of the source code of an H.264 decoder [4] that implements a subset of the H.264 base profile. Of special interest to us were locations in the code which have a huge impact on the reliability of the system.

We selected one of the manually identified critical functions of the H.264 decoder, `residual_block`. This function repeatedly reads the so-called NAL-Buffer containing the actual frame to be decoded. Frame data is encoded using variable length coding. In the worst case, a bit flip can lead to misinterpretation of the data length and any subsequent read will return also wrong values until the next frame is copied into the NAL-Buffer. As a consequence, we evaluated the decoder's behavior under varying error injection conditions affecting this buffer.

Error Injection

In order to provide a reproducible, deterministic error scenario, we added a separate error injection thread to the decoder application which injects single-bit errors into the NAL-Buffer. The error model employed uses a simple, randomized uniform distribution of flipped bits. All injected errors are registered by the injection thread and can be rolled back later on demand to simulate error correction taking place. This **delayed error correction**, however, does not correct variables holding values derived from previously corrupted inputs. Furthermore, we can suspend error injection at any arbitrary point, which corresponds to an **immediate error correction**. In any case, error injection is suspended when we copy the input into the NAL-Buffer and when the frame header gets decoded, to prevent incorrectly reading critical information like the frame type. Nevertheless, such critical errors always require immediate handling.

To investigate the effects of errors, we used an artificially high error rate of up to 240 injected errors per second. Thus, fatal faults like a crashed application can be found in a short amount of time.

Results

First evaluations have shown that delayed error handling is feasible. We disabled error injection and roll back errors, which have shown up in the NAL-Buffer, at the beginning of `residual_block`. The application is still affected by errors that have propagated to other memory locations than the NAL-Buffer. Those errors result from simulated transient faults in between subsequent calls of `residual_block`, to which delayed error handling has been applied.

The results of our experiments show that when applying error correction to the `residual_block` function, we could reduce the percentage of application crashes from 60% to 2%.

In order to distinguish between *perceptible* and *non-perceptible* errors, we used the ΔE metrics standardized in ISO 12647, which describes the distance between two colors. We define that a ΔE value less than 5.0 indicates a *non-perceptible* error. The analysis was performed by comparing each decoded frame under error injection with the expected correct frame. Based on this metric, the amount of perceptible errors in the decoded video could be reduced from an average of 11.5% to less than 0.6%.

According to a profiling of the decoder using the `gprof` tool, the `residual_block` function only accounts for 26.0% of the overall execution time of the decoder for 2880 frames (120 seconds) of our example video. Overall, the function was called 4,582,301 times.

The input copy code and frame header decode code is only executed 2880 times each, which is less than 0.2% of the overall execution time.

These values show that if error correction is performed in less then 26.2% of the overall decoder execution time, the amount of perceptible errors and crashes can be significantly reduced.

4 Related Work

Almost all error handling techniques for transient faults aim at correcting every error that is caused by a single event upset. In that way errors are prevented from propagating in the application. Popular approaches that

do so are *triple-modular redundancy* [3], *roll-forward checkpointing* [7], and *checkpointing and rollback recovery* [1]. For checkpointing, Silva and Silva [8] have shown that using application knowledge rather than applying a static scheme the overhead for checkpointing in runtime and memory footprint can be reduced significantly.

In recent years, researchers have studied the impact of soft errors on several applications. In [2] Li et al. studied the impact of transient faults on multimedia, AI, and SPECInt CPU2000 benchmarks. Polian et al. [6] considered ISCAS-89 sequential benchmark circuits. All of them make the observation that a large count of transient faults do not have any effect on the application correctness. Another fraction of faults changes the output or state of the application, but do not make the application crash and the results can still be accepted by the user. Especially for multimedia applications absolute correct results are not required, since the user will hardly recognize the difference.

These observations led to selective protections and hardening techniques against transient faults. Mehrana et al. [5] placed memory objects with a long lifetime in reliable memories, whereas all other memory objects can be effected by faults. In [9] Sundaram et al. replicated instructions, that are critical for the control path. These instructions were determined by static analysis.

Both proposed techniques are based on the assumption that the unprotected parts of the application are less susceptible to faults than the protected parts. Unfortunately, they can only estimate the fault tolerance for the different parts of an application, since they try to infer program semantics only from the application code and leave out application knowledge.

5 Conclusions and Future Work

In this paper, we presented an approach which uses application knowledge to decide which errors to correct in an embedded system. Of special interest are errors that are critical for the reliability of the system. Using a typical embedded application, we have shown that correcting only a subset of the errors that occur will still lead to a system that provides acceptable service quality for the end user while avoiding nearly all application crashes. Delayed error handling is especially useful for real-time critical embedded systems. When used as an additional dimension for error classification, the overhead of error correction in our example video decoder system could be reduced significantly.

Considering the requirements of the system in question, the next steps here are to actually create the software-based error correction implementation. An emerging question is to which extent the error handling overhead can be reduced by exploiting error classification and especially delayed error handling in general. To answer this question, an assessment of both the quality and the real-time properties of the resulting system under real-world conditions is required. Based on the classification, embedded system developers should be provided with a simple and efficient way to create application annotations, which can be used by an (semi-)automated tool flow in a later step. Here, future work will include the definition of syntax and semantics for error annotations in C as well as the provisioning of the required compiler infrastructure. In addition, the suitability of our approach for different implementations of the video player implementation (e.g., the more complex H.264 reference decoder) as well as different embedded applications is of interest. Here, we expect very similar results for real-time critical media-related applications like MP3 audio players and mobile phones.

Acknowledgments

We appreciate the support of Martin Fiedler, the author of the H.264 decoder analyzed in this paper [4], and Ingo Korb, who fixed errors in the original implementation.

References

[1] LI, C.-C. J., STEWART, E. M., AND FUCHS, W. K. Compiler-assisted full checkpointing. *Software – Practice & Experience 24*, 10 (1994), 871–886.

[2] LI, X., AND YEUNG, D. 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.

[3] LYONS, R. E., AND VANDERKULK, W. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of Research and Development 6*, 2 (1962), 200–209.

[4] MARTIN FIEDLER. Implementation of a basic H.264/AVC Decoder (Seminar Paper), TU Chemnitz, 2004.

[5] MEHRARA, M., AND AUSTIN, T. Exploiting selective placement for low-cost memory protection. *ACM Transactions on Architecture and Code Optimization 5*, 3 (2008), 1–24.

[6] POLIAN, I., REDDY, S. M., POMERANZ, I., TANG, X., AND BECKER, B. No Free Lunch in Soft Error Protection? In *Proc. of the 2nd Workshop on Dependable and Secure Nanocomputing* (Anchorage, Alaska, USA, 2008).

[7] PRADHAN, D. K., AND VAIDYA, N. H. Roll-Forward Checkpointing Scheme: A Novel Fault-Tolerant Architecture. *IEEE Transations on Computers 43*, 10 (1994), 1163–1174.

[8] SILVA, L. M., AND SILVA, J. G. System-Level Versus User-Defined Checkpointing. In *SRDS '98: Proc. of the The 17th IEEE Symp. on Reliable Distributed Systems* (Washington, DC, USA, 1998), IEEE Comp. Society, p. 68.

[9] SUNDARAM, A., AAKEL, A., LOCKHART, D., THAKER, D., AND FRANKLIN, D. 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, 2008), ACM, pp. 339–346.