FAME: Flexible Real-Time Aware Error Correction by Combining Application Knowledge and Run-Time Information

A. Heinig, F. Schmoll, B. Bönninghoff, P. Marwedel, M. Engel
Computer Science 12
TU Dortmund
D-44221 Dortmund, Germany
Email: {firstname.lastname}@tu-dortmund.de

***Abstract*—In this paper, we present FAME (Fault-Aware Microvisor Ecosystem): an ecosystem which exploits flexibility in software implemented fault tolerance to significantly reduce error handling overhead. Combining both offline and online information FAME can decide how to handle errors appropriately. Static code analysis methods provide application knowledge about possible error impacts. During runtime, data liveness and resource mapping are determined by a fine-grained resource usage model. Using our approach, we achieve flexible real-time aware error handling that enables the use of the system even under high error rates. Compared to a simple checkpoint-recovery approach the number of deadline misses are reduced significantly by up to 87.37 %.**

I. Introduction

Resilience against transient faults was traditionally only a concern for computer systems running in harsh environments or in fields where failures can lead to severe damages. Transient faults are single-shot phenomena that can lead to unintended status changes in various components of a system. Natural radioactive decay, disturbance in supply voltages, high energy cosmic particles, or overheating are examples for causes of transient faults. Due to advancements of semiconductor fabrication that lead to shrinking geometries and lowered supply voltages of semiconductor devices, transient fault rates will increase significantly for future semiconductor generations [1]. Hence, resilience against transient faults will become an issue also for everyday computing.

To cope with transient faults, error detection and correction (EDAC) will be mandatory. Fortunately, a variety of such methods already exists. However, EDAC is not free. Typically, additional resources are required for the implementation. This is a serious problem, especially in embedded systems development. On the one hand, transient faults have to be handled, and on the other hand, embedded systems possess only a limited number of resources, like processing time, memory, and energy. If every single fault is corrected, maintaining real-time properties of a system will become extremely hard.

In this paper, *FAME*, the *Fault-Aware Microvisor Ecosystem*, is presented. *FAME* implements a flexible error handling approach which is able to decide **if**, **when**, and **how** an error has to be corrected. The flexible error handling approach allows us to ignore errors if, for example, unused memory is affected. Access to unused memory can happen, if a cache fetches a new line, but, due to data alignment, memory is fetched which resides between two allocated data objects. In the opposite case, if errors can lead to a program crash, error correction will be mandatory. Examples for those kinds of errors are faults which lead to erroneous pointers, arithmetical exceptions, or control flow changes.

We focus this paper on the error correction aspect. Hence, we assume that an error detection method is present, like, e. g. Reed-Solomon codes [2]. In this paper, we will show improvement of the real-time behavior of the used benchmark application.

The contributions of this paper are as follows:

1) We show a flexible error handling approach for embedded real-time systems.
2) We combine compile time and runtime information to decide if, how, and when errors have to be handled.
3) By applying the proposed techniques, the number of deadline misses is reduced significantly.

The paper is organized as follows. Related work is presented in section II. In Section III, flexible error handling is introduced followed by a description of the realization with *FAME* in Section IV. Evaluation is presented in Section V. In Section VI, we conclude the paper.

II. Related Work

Fault tolerance methods are based on some kind of redundancy. Typically, a larger amount of redundant resources allows correcting a higher number of errors. Several approaches [3], [4], [5], and [6] trade-off reliability against resource consumption by applying error protection or detection to only a part of the application's data. The data that is protected is determined using profiling or heuristics.

The annotation of precision requirements of data in Java programs using type qualifiers is presented in [7]. Data annotated with the type qualifier `@Approx` can tolerate inaccuracies and can be processed by approximate hardware components to save energy. However, the determination of

approximate data is not automated and inaccuracies due to transient faults are not considered.

To detect multi bit errors and errors in the control flow, TMR (Triple Modular Redundancy) can be used. If errors only have to be detected, DMR (Dual Modular Redundancy) is sufficient. DMR "only" doubles the amount of required resources. Software based approaches to detect errors are, for example, EDDI [8] and SWIFT [9] which duplicate instructions important for branching and memory transactions. By comparing the results errors can be detected. In contrast to software-based approaches, all hardware-based approaches require hardware modifications or additional hardware components. Software-only methods can be executed on existing hardware.

Typically, error detection only approaches require less redundant resources than approaches including error correction. As mentioned earlier, we focus on error correction. For the rest of this paper we assume that at least one of the previously described error detection methods is available.

A framework that allows the specification of application specific error correction actions is presented by de Kruijf et al. [10]. They extent C/C++ by `relax` and `recover` blocks similar to `try-catch` blocks for exception handling. A `recover` block contains the actions to be taken if the rate of errors occurring during the execution of the preceding `relax` block exceeds a specified limit. In contrast to our approach this a code-centric approach. Also, code not included in a `relax` block is assumed to experience no faults.

Support for reliability in operating systems is often closely related to security concerns. One important development in this direction is the EROS system [11] and its follow-up project, Coyotos. EROS provides support to efficiently restructure critical applications into small communicating components. These components can then be efficiently isolated from each other and the rest of the system. Access to objects is controlled by capabilities. From our point of view, capabilities can be seen as an orthogonal approach to the subscription-based model of data object ownership used in our paper.

The idea of providing only a minimal layer of abstractions was investigated in the Exokernel project [12]. This enables software running on top of exokernels to have a greater level of control on the use of hardware abstractions. The functionality of exokernels is limited to protection and multiplexing of resources. In a sense, the operating system of our *FAME* approach can be seen as a kind of specialized exokernel which provides only those abstractions required for building fault-tolerant applications on top of it, while leaving other crucial resource allocation decisions such as scheduling and memory management to the library OS running on top of it.

III. Flexible Error Handling

The idea of flexible error handling is shown in Figure 1. In the depicted scenarios, an application is running and at some point in time an error occurs. In a naive approach every error would be handled alike, without considering the available resources. This can lead to deadline misses.

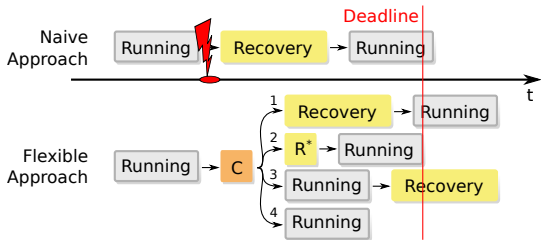

Fig. 1. Flexible Error Handling

In our approach, the error is classified first, labeled as "C". The goal of the classification is to determine **if**, **how**, and **when** the error has to be corrected.

(if) Whether errors have to be handled or not depends mainly on the error impact. If an error has a high impact, error correction will be mandatory. In contrast, if an error has only low impact at all, e.g. the color of a single pixel in the frame buffer is disturbed, further handling is optional. Handling errors in the latter case will, however, improve the quality of service (QoS). To distinguish errors by their impact, static code analysis methods are used, which are detailed later.

(how) Error handling depends on the available error correction methods, the error impact, and the available resources. In our *FAME* system the correction methods "checkpoint-and-recovery" and "ignore" are always available. "Ignore" is the empty recovery method (case 4 in Figure 1). In addition to these two methods, the application programmer can provide further error correction methods. Such a method (labeled as "R^*" in Figure 1) can be, for example, faster as the recovery of a checkpoint which increases the probability that the application adheres to the deadline.

(when) The scheduling algorithm has major influence on the question when an error correction method should be scheduled. Typically, the task with the highest priority is executed. Hence, if a high priority task is affected, error correction has to be scheduled immediately (cases 1 and 2). If a low priority task is affected, the high priority task can continue execution and the error handling will be delayed (case 3). To be able to map errors to tasks, a subscriber-based model will be used.

By considering if, how, and when an error has to be corrected, our flexible error handling approach can select an error handling method which is suited for the current situation. In the worst case, we have to immediately schedule the same error correction method as in the naive approach. Compared to the naive approach, this results in an additional overhead caused by the classification phase. In the best case, however, flexible error handling can completely ignore the error and hence save resources and adhere to the deadline.

IV. FAME

To enable flexible error handling application knowledge gathered off-line has to be incorporated with data only available at runtime. Therefore, we use a special compiler that collects information about error handling options. It encodes this information in a classification data base. The runtime components of *FAME* can efficiently extract the error handling

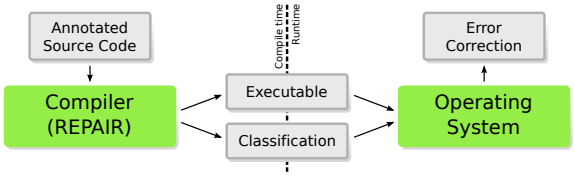

Fig. 2. FAME - Fault-Aware Microvisor Ecosystem

Listing 1. Annotated Corretion Method

```
#pragma eca qos=... wcet=...
void MoCompDefaultValue(...) {
    int offset = fault2offset(...);
    mpi->MVx[offset]  = 0;
    mpi->MVy[offset]  = 0;
}
```

Listing 2. Corretion Method Assigned to Data

```
mode_pred_info_t * mpi = (mode_pred_info_t *)
    malloc(sizeof(mode_pred_info_t));
mpi->MVx = (reliable int *)
    #pragma eca ecm = MoCompDefaultValue;
    malloc(x * y * sizeof(int));
```

information from this classification and select and schedule the error correction that is most suitable under the current runtime conditions. The resulting ecosystem is depicted in Figure 2.

A. Compile-Time Classification of Application Data

During compile-time the application's source code is analyzed to determine error correction options for the individual data objects of the application. The source code can contain two kinds of annotations with that the application programmer can express application knowledge: Reliability type qualifiers and error correction annotations.

The type qualifiers `reliable` and `unreliable` classify, whether data has to be corrected if it is affected by an error. Reliable data are expected to be error free, otherwise the system may crash. Hence, if an error affects reliable data, error correction will be mandatory. Our REPAIR (Reliable Error Propagation And Impact Restricting) compiler, a compiler for ANSI C99, can automatically classify data as reliable based on its use in the application and insert the type qualifiers into the application's source code [13]. Hence, reliability type qualifiers need not be contained in the initial application's source code.

If reliable data affected by errors are always corrected, REPAIR guarantees the correct control flow of the application without system crashes by static analysis. Data classified as unreliable need not to be corrected, but uncorrected errors can lead to huge deviations in the application's output. To limit the influence of errors on the output, the application programmer can also annotate additional data as reliable that should always be corrected. Thus, the propagation of errors to these data is inhibited as well.

With error correction annotations (eca) custom error correction methods (ecm) can be specified. In Listing 1, the function `MoCompDefaultValue` is tagged as error correction method. The purpose of this correction method is to correct corrupted data by assigning a default value. In this way, consequences like a system crash can be prevented, but a costly recovery of the fault free value can be avoided. To assign this correction to the actual data object, the annotation depicted in Listing 2 has to be used.

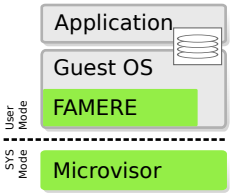

Fig. 3. Software Stack

After the classification of data and parsing of the error correction annotations, the compiler integrates the information into the binary. Using a runtime error classification library, position, size, and reliability class of static data objects as well as possible correction methods can be queried.

B. Runtime Components

Figure 3 gives an overview of the runtime components. An application with integrated classification information is running on a virtualized guest OS. The guest OS is linked against the *FAME Runtime Environment (FAMERE)*. *FAMERE* is responsible for the flexible error handling as well as the interfacing with the microvisor. The microvisor runs low-level error correction and ensures the feasibility of software-based error handling. In the next subsections, the aspects of the runtime system that are important for the implementation of flexible error handling are described in more detail.

1) Microvisor: Our flexible error handling strategy is based on a custom microvisor. The main purpose of the microvisor is to isolate critical system components from possible error propagation. Critical components in this context are resources required to keep error detection and correction running. Depending on the underlying hardware, the actual critical resources vary. If, for example, errors are signaled via interrupts, the interrupt controller will be part of the critical resources set.

The microvisor itself is also part of the critical resources set. Like in all software-based fault-tolerance mechanisms, this is a weak spot, since the microvisor cannot protect itself. If software parts critical for error handling are susceptible to errors as well, situations will occur where the error handling has to cope with errors affecting the error handling itself. This can result in a livelock. To deal with this problem, some guaranteed fault free hardware components are required to execute software-based fault-tolerance mechanisms. Those fault free hardware components and the microvisor form a minimal set, the so called Reliable Computing Base (RCB) [14]. To shield the RCB from error propagation, our microvisor uses paravirtualization. Compared to other virtualization solutions, our microvisor is tailored to the needs of embedded systems and fault tolerance. To keep the virtualization overhead low, our microvisor supports only one guest operating system during

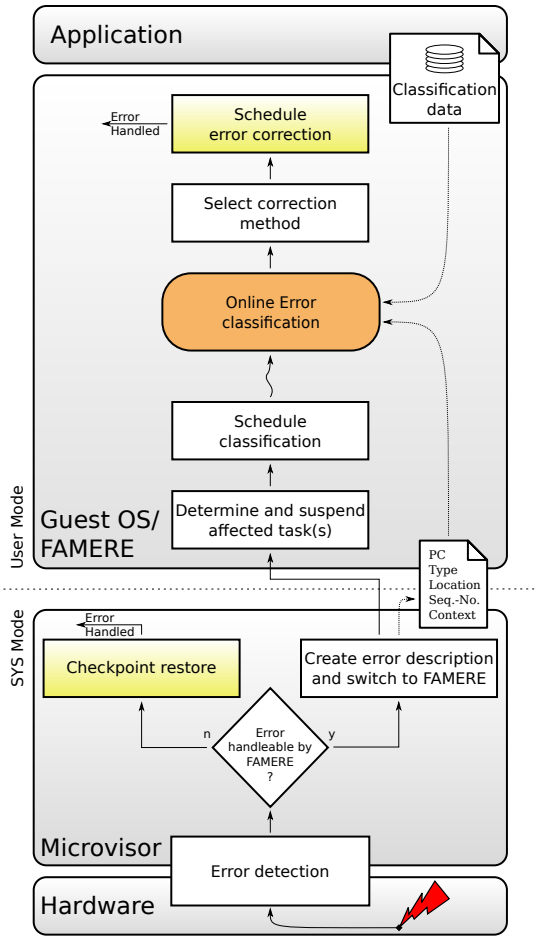

Fig. 4. *FAME* Error Handling Procedure

runtime. In contrast to other hypervisors, this releases us from the burden to provide virtual CPUs and CPU multiplexing. Also, caches and TLB entries need not to be switched between different OS instances.

A further responsibility of the microvisor is the creation of full system checkpoints. Consequently, the microvisor will be able to restore a valid state if the system is in an odd state due to a severe error. This will be the case if the error affects the error handling in *FAMERE*.

2) FAMERE: *FAMERE* is a library embedded in the guest operating system. *FAMERE* is the component where all information – compile time as well as runtime – is combined to implement our flexible error handling approach. However, the task of *FAMERE* is not only to handle errors. In addition, *FAMERE* provides support for other aspects related to virtualization and reliability. Among others, this includes a high level API enabling C code to directly call hypercalls and to replace the guest OS' heap allocator by a version supporting (un-)reliable data annotations.

However, by far the most important task of *FAMERE* is error handling. The complete error handling procedure is depicted in Figure 4. Error handling starts in the microvisor (bottom of Figure 4). After error detection, the microvisor first checks whether the fault can be handled outside the RCB or not. Errors, for example, which affect *FAMERE* are very unlikely to be handled by *FAMERE* itself – again, this is the previously described chicken-and-egg problem. In such a case the microvisor automatically restores the last system checkpoint or, if no checkpoints are available, resets the complete user space as a last resort. If *FAMERE* is not affected, error handling is delegated by sending a message to *FAMERE*. Before jumping to the *FAMERE* message handler, the microvisor creates an error description containing information about the occurred error as well as the user space context.

After switching to *FAMERE*, the tasks affected by the error have to be determined. To get a very fine-grained mapping, the subscriber model introduced by Heinig et. al [15] is used. Briefly, the subscriber model defines a new programming paradigm where tasks have to explicitly `subscribe` to data prior usage. After usage, tasks can `unsubscribe` from the data. Hence, each data object possesses a set of tasks currently using this object. If the set is empty, the subscriber model defines that the information stored in the object is not life. In the sense of our flexible error handling strategy, this means that fault affecting such objects can be ignored.

If there are tasks that are not affected by the fault and that are higher prioritized than the affected tasks, further error handling will be delayed until all higher prioritized tasks finish their execution. When the error handling is scheduled again, the online error classification will be performed. It determines possible correction scenarios based on the occurred errors and the classification data provided by the compiler.

The last steps of the flexible error handling procedure are to select one scenario and to schedule this scenario. For the selection, the current runtime conditions are considered. This includes, for example, the currently available slack. The slack time denotes the time which is not used by any real-time task. In this period of time, error correction as well as checkpointing can be performed without disturbing the real-time behavior of the application.

V. Evaluation

A. Experimental Setup

For the evaluation, we simulate an embedded system with the Synopsys CoMET cycle-accurate simulator [16]. In all experiments CoMET is configured to simulate an 1.2 GHz ARM926 system with 64 MiB RAM, 16 MiB ROM and 128 KiB reliable RAM. All components are considered reliable, except the 64 MiB of RAM.

As software load we execute an H.264 constrained baseline profile video decoder. The decoder is configured to create a checkpoint after every displayed frame. In all our experiments, we decode 600 frames in total at a rate of 10 frames per second. The frame resolution is 480x320 pixels. Although resolution and frame rate seem quite low, this setup leads to a CPU utilization of more than 65 %, since we decode H.264 in software only. However, higher resolutions and frame rates will be possible if more computing power is available. We are convinced that our results will also be valid for such platforms. To reduce jitter in the output, our H.264 decoder

Error Rate		Naive Error Handling		Flexible Error Handling		Flexible + Application Specific	
λ	Effective $[s^{-1}]$	# Avg. Deadline Miss	Avg Missed by	# Avg. Deadline Miss	Avg Missed by	# Avg. Deadline Miss	Avg Missed by
λ=1e-16	0.14	0.00	0.00 ms	0.00	0.00 ms	0.00	0.00 ms
λ=1e-15	1.44	2.86	8.15 ms	0.52	7.93 ms	0.36	4.89 ms
λ=1e-14	35.84	-	-	1,937.87	10,268.98 ms	1,887.12	9,346.16 ms

TABLE I
AVERAGE DEADLINE MISSES

buffers eight frames. Each buffer element is decoded in a separate task and the inter-frame dependencies are modeled by task dependencies. EDF* [17], [18] is used to schedule the tasks. Therefore, we extended RTEMS to support task dependencies as well as task activation time.

A simulation of one complete decoder run takes normally 20 minutes on an Intel(R) Xeon(R) E5630 CPU clocked at 2.53 GHz. Every run which takes longer than 2 hours is automatically terminated, since such a run is very likely to violate any real-time constraint due to numerous checkpoint restores. The microvisor is configured to allow a maximum of eight restores per checkpoint. Hence, if *FAMERE* requests to restore the same checkpoint the ninth time, the checkpoint will be dropped and the previous checkpoint will be restored. If a system reset is necessary, since no more checkpoints are available, the run will be terminated as well.

B. Fault Injection

To evaluate the real-time behavior under influence of transient faults, we implemented our own CoMET memory module which injects uniformly distributed transient faults. For each memory access, we simulate error detection in hardware. If the processor accesses an erroneous word, an interrupt will be raised. The number of faults to be injected is determined by a Poisson distribution with configurable parameter λ. We use three different parameters λ. Not all injected faults are visible by the application, since faults are only detected when the corresponding memory cell is accessed. In the first two columns of table I, the observed average error rates (of detected faults) are depicted. As can be seen, our injection rates range from several faults per minute to an artificially high fault rate of 36 faults per second.

C. Naive Error Handling

In this scenario, the microvisor treats every error as error which cannot be handled by *FAMERE*. Hence, a checkpoint is immediately restored. For this scenario, columns three and four in Table I show the average amount of missed deadline and the average duration of a deadline miss, respectively. For the lowest error rate, no deadline misses occur since enough slack time is available for the recovery of checkpoints.

If the error rate increases by an order of magnitude, deadline misses can be observed. On average, deadlines are missed by 8.15 ms. Considering the highest error rate, it can be noticed that no run of the experiment terminates within the two hour time limit or without resets initiated by the microvisor.

D. Flexible Error Handling

In Table I columns five and six, results for flexible error handling are shown. In this experiment, only errors affecting reliable and alive data are handled by checkpoint recovery. Errors affecting other data are ignored.

As can be seen, the flexible error handling approach reduces the number of deadline misses significantly (81.75 %). Alike, the time by which a deadline is missed is reduced as well (2.70 %). However, most importantly, flexible error handling allows for very high error rates. The number of missed deadlines is, unfortunately, very high but, according to our simulation statistics, more than 50 % of all runs terminated within the two hour time limit without reset.

E. Flexible Error Handling with Application Specific Error Correction Methods

In our last experiment we use the annotations shown in Listing 1 and 2 to enable the application specific error correction method `MoCompDefaultValue`. This method is able to transfer a corrupted motion vector into a valid state. In H.264 motion vectors are used to shift a macro block to a new location within a frame. Motion vectors are hence well suited to encode movements in the video. However, if a motion vector is corrupted, it can happen that the corresponding macro block gets shifted out of the frame. Hereby, other memory will be overwritten. Consequently, motion vectors have to be reliable. Anyway, if a motion vector moves the corresponding macro block only to a location inside the frame, no fatal consequences will happen. Therefore, `MoCompDefaultValue` provides a valid correction method by just setting the motion vector to zero. This corresponds to no movement of the macro block. By applying this application specific error correction method to erroneous motion vectors, the deadline misses are reduced by 87.37 % for the second highest error rate.

In Figure 5, we evaluate the ratio between errors which can be ignored, errors which require checkpoint restore, and errors which can be handled by `MoCompDefaultValue`. The bars are normalized for better comparison. It is clearly shown that over 56 % and up to 63 % of all transient faults can be ignored. For the flexible error handling experiment with enabled application specific error correction, it can be observed that, with higher error rates, the share of errors correctable with our application specific method is increasing.

For the highest error rate, more than 5.5 % of all errors can be corrected by `MoCompDefaultValue`. Or in other words, the amount of required checkpoint restores is reduced by over 5.5 %. This leads to the conclusion that it is worthwhile to write application specific error corrections methods.

F. Quality of Service Impact

Setting the motion vector to zero and ignoring of faults affecting only unreliable data has no impact on the control

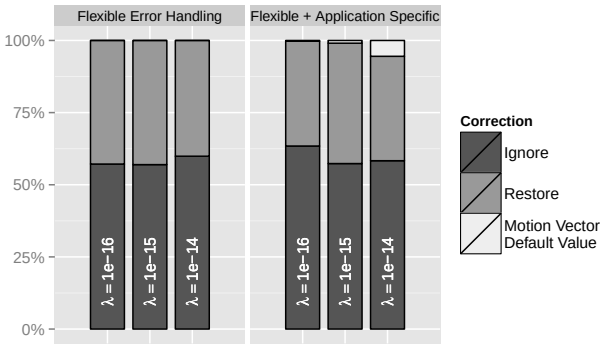

Fig. 5. Applied Error Correction Methods

	λ=1e-16	λ=1e-15	λ=1e-14
Naive Handling	36.19	36.15	-
Flexible Error Handling	36.19	36.18	29.01
Flexible + Application Specific	36.20	36.12	28.95

TABLE II
PEAK SIGNAL TO NOISE RATIO

flow of the application. However, it will definitively have an impact on the quality of service (QoS). Table II shows the measured peak signal to noise ratio (PSNR) of the different scenarios. The PSNR is a typical QoS metric used in image processing. The higher the PSNR, the better the QoS. To obtain the PSNR we compare the decoded frames with the original source images used to create the video. The average PSNR ratio achieved by a golden run is 36.20 dB. As expected, higher fault rates lead to lower PSNR.

Although no errors are ignored during the naive error handling approach, the PSNR decreases. The reason for it is that we use an optimized checkpointing which ignores data which is classified as unreliable. For the highest error rate, significant QoS deviations can be observed. Due to the fact that `MoCompDefaultValue` corrects errors not exactly, the deviation of flexible error handling with application specific correction has a higher deviation as without application specific correction.

VI. CONCLUSIONS

In this paper, we presented *FAME*, an ecosystem which combines compile time and runtime information to enable a flexible error handling strategy. To save scarce resources, especially in embedded real-time systems, our flexible error handling strategy determines **if**, **how**, and **when** errors have to be corrected. By applying our flexible approach we showed that up to 63 % off all errors can be safely ignored in our H.264 video decoding application. Our techniques improved the real-time behavior of the application under influence of transient faults. We showed the reduction of deadline misses by up to 87.37 % compared to an approach where every error is handled immediately without classification. Furthermore, flexible error handling allows coping with high error rates. Even if a deadline is missed, the difference to the original deadline is very small. Reductions by up to 40 % are shown.

The next step to improve *FAME* will be to implement more application specific error correction methods. Furthermore, other applications have to be considered as well.

ACKNOWLEDGMENTS

This work was supported by the German Research Foundation (DFG) Priority Programme SPP1500 under grant no. MA-943/10. The authors would also like to thank Synopsys for the provision of the CoMET simulation framework.

REFERENCES

[1] "Process Integration, Devices, and Structures (PIDS)," *International Technology Roadmap for Semiconductors*, 2013, http://www.itrs.net/Links/2013ITRS/Home2013.htm.

[2] I. S. Reed and G. Solomon, "Polynomial Codes Over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, Jun. 1960.

[3] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian, "Mitigating soft error failures for multimedia applications by selective data protection," in *CASES '06*, Oct. 2006, pp. 411–420.

[4] M. Mehrara and T. Austin, "Exploiting selective placement for low-cost memory protection," *ACM Trans. Archit. Code Optim.*, vol. 5, no. 3, pp. 14:1–14:24, Dec. 2008.

[5] K. Pattabiraman, Z. Kalbarczyk, and R. Iyer, "Application-based metrics for strategic placement of detectors," in *Dependable Computing, 2005. Proceedings. 11th Pacific Rim International Symposium on*, Dec. 2005.

[6] Q. Lu, K. Pattabiraman, M. Gupta, and J. Rivers, "SDCTune: A Model for Predicting the SDC Proneness of an Application for Configurable Protection," in *ESWEEK'14*, Oct. 2014.

[7] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: approximate data types for safe and general low-power computation," in *Proc. of PLDI*, Jun. 2011.

[8] N. Oh, P. Shirvani, and E. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *Trans. on Reliability*, vol. 51, 2002.

[9] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *Code generation and optimization (CGO)*, Mar. 2005.

[10] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An Architectural Framework for Software Recovery of Hardware Faults," in *ISCA '10*, June 2010, pp. 497–508.

[11] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Proceedings of the seventeenth ACM symposium on Operating systems principles (SOSP)*, Dec. 1999, pp. 170–185.

[12] D. Engler, F. Kaashoek, and J. O'Toole, "Exokernel: an operating system architecture for application-level resource management," in *Operating systems principles (SOSP)*. ACM, Dec. 1995, pp. 251–266.

[13] F. Schmoll, A. Heinig, P. Marwedel, and M. Engel, "Improving the fault resilience of an H.264 decoder using static analysis methods," *Trans. on Embedded Computing Systems (TECS)*, vol. 13, Nov. 2013.

[14] M. Engel and B. Döbel, "The Reliable Computing Base-A Paradigm for Software-based Reliability." in *SOBRES'12*, 2012.

[15] A. Heinig, F. Schmoll, P. Marwedel, and M. Engel, "Who's using that memory? A subscriber model for mapping errors to tasks," in *Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[16] Synopsys Corporation. (2014, Feb.) CoMET/METeor Models. [Online]. Available: http://www.synopsys.com/Systems/VirtualPrototyping/VPModels/Pages/CoMET-METeor.aspx

[17] J. Blazewicz, *Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines*. North-Holland Publishing Co., Oct. 1976.

[18] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *The Journal Real-Time Systems*, vol. 2, no. 3, pp. 181–194, 1990.