Who's using that memory? A subscriber model for mapping errors to tasks

Andreas Heinig, Florian Schmoll, Peter Marwedel and Michael Engel

Computer Science 12
TU Dortmund
D-44221 Dortmund, Germany
Email: {firstname.lastname}@tu-dortmund.de

***Abstract*—Software-based mitigation methods for transient faults in embedded systems have to be extremely efficient due to resource limitations. Thus, methods to reduce the overhead required for error correction are highly relevant.**

In this paper, we present a new approach to reduce the number of errors that have to be corrected by incorporating information on the use of resources. Whenever an error occurs, the error correction subsystem is informed which activity is affected by this error and if the affected location contains information relevant for future execution of this activity. We achieve this by introducing a new programming model to describe application knowledge, the subscriber model for errors. This model allows an application to advise the operating system about the currently used data working set of a process and, accordingly, about the relevance of this data.

We analyze the application of the subscriber model to the largest data structure of a running system, the system heap. Our evaluations show that applying our model requires only negligible execution time overhead. In addition, we show that using the subscriber model, the average checkpoint size for software-based checkpointing and restore approaches can be reduced significantly.

I. Introduction

The current trends of shrinking geometries as well as lowering supply voltages of semiconductor devices increase the susceptibility to transient faults [1]. Transient faults are single-shot phenomena that can lead to unintended status changes in components of a system. They are induced, for example, by natural radioactive decay, high energy cosmic particles, disturbance in supply voltages, electromagnetic inferences, or overheating. Due to the stochastic characteristics of their sources, faults affect components of a system in an unpredictable moment in time. Fortunately, transient faults can be corrected. They persist only until the next status change of the affected component.

Hardware and/or software mechanisms exist which are able to detect the occurrences of transient faults. Prominent examples for mechanisms often integrated in hardware are *Parity Bits* and *Error Check and Correction (ECC)* to safeguard main memory, registers, or caches. Further mechanisms can be *Double and Triple Modular Redundancy (D/TMR)*.

When the applied mechanism detects a transient fault which cannot be directly corrected we assume that the fault will be reported to the operating system. The operating system can then handle the fault in a flexible way. As studies suggest [2], [3], [4], not all faults have an impact on the program's execution. For example, a fault in main memory can affect an unused memory cell[1]. Error correction for this kind of faults can be skipped. In contrast, if faults may cause the application to crash, e.g., due to invalid memory accesses caused by erroneous pointer values, error correction will be mandatory [5].

Only some faults affect the overall (software) system. The majority of faults only affect a subset of all components. This is an important aspect when real-time systems are considered. If no high priority real-time activity is affected, error handling can be delayed to increase the likeliness of keeping all deadlines. In contrast, when a real-time activity is affected, the fault has to be handled as soon as and as fast as possible to continue executing the affected activity and to be able to meet real-time constraints.

However, to determine the affected activities, the operating system has to be able to map errors to tasks. Therefore, we propose a subscriber-based model in this paper. Basically, the subscriber model defines that activities have to register the resources they intend to use. In this paper we also point out a secondary usage of the subscriber model: optimizing the size of checkpoints. Since we provide a fine-grained mapping of activities to resources we are able to distinguish between used and unused resources. By only checkpointing the used resources, large storage space savings can be achieved.

In this paper we describe the subscriber model in detail and evaluate our model for a real-world H.264 application. We show significant decreases in both checkpoint sizes as well as execution times.

The contributions of this paper are as follows:

1) We introduce a new programming model which allows the operating system to perform a fine-grained mapping of resources to activities and vice versa.
2) We point out how the semantics of the subscriber model can be used to determine liveness of objects.
3) We show how the subscriber model decreases the checkpoint size without the need for special hardware.

The remainder of this paper is organized as follows. We

[1]Unused memory cells can, e.g., be accessed due to cache line fetches.

start with presenting related work in section II. In section III we detail the subscriber model. Evaluation results are shown in section IV. Thereafter, we conclude in section V.

II. Related Work

Keeping track of used resources is one of the main concerns of operating system. Typically, the OS keeps a list of resources used by a process. However, when we consider systems with only one process consisting of several activities (like in RTEMS [6]), this model is too coarse grained. Since address space and resources are common to all activities of a process, the operating system is not capable of determining the used resources of each activity.

To the best of our knowledge, our subscriber model is the first available method which not only provides a mapping between activities and resources, but also on the resource's liveness. This enables the operating system to determine whether a resource affected by an error is in use or not and which activity is using the resource.

To reduce the overhead of checkpointing, several proposed methods rely on hardware features, e.g., "dirty" and "accessed" bits of the page table to track modified and accessed memory like in libckpt [7], tracking of cache accesses in the Sequoia system [8], or page shadowing using copy-on-write semantics in Flashback [9]. However, some architectures do not have such features, e.g., ARM processors do not implement dirty and accessed bits in page table entries. The subscriber model presented in this paper is a more generic approach to reduce the checkpoint size. In contrast to hardware-supported methods, our method is architecture-independent and not restricted to, for example, page size granularity.

III. Subscriber Model

As soon as a process is started by the operating system, different resources are allocated. These consist of memory for data (`.data`, `.bss`) and, depending on the OS, a program stack for the initial activity. During execution, an activity can request additional resources from the OS via appropriate system calls. A resource is considered to be used as long as the process is not destroyed or an activity within the process explicitly releases the resource to the OS.

If an error is reported to the OS, the affected resources are determined by traversing the allocation tables of the OS. However, the affected activities cannot be determined, since resources are only mapped to processes. In other words: When a fault occurs, the OS will suspend **all** activities of **all** affected processes until the error is handled. This can be problematic when some unaffected high priority activities or real-time activities are suspended.

To solve this problem we introduce the subscriber model. With this model we are able to determine the affected activities. Furthermore, we can also decide whether an affected resource is currently in use and has live data stored.

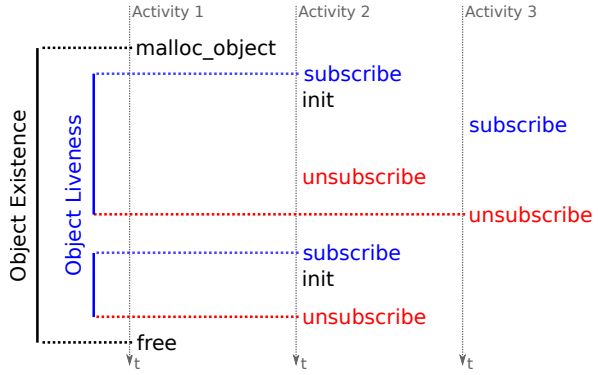


Fig. 1. Subscriber Model Example

A. Objects

For the subscriber model we define the term *object*. An object is the representation of an arbitrary resource. For example, when the process requests additional heap memory by calling `malloc()`, a new object will be created by the OS. This new object now exactly represents the allocated memory. In addition to information like the allocation's base address and size, the object subscribers are stored as well.

An object with at least one activity subscribed is called *subscribed object*. If no activity is subscribed, the object will be called *unsubscribed object*.

Another type of objects are *system objects*. System objects are always in the subscribed state. We introduce this object type to support legacy code. Therefore, we also demand that all system calls and standard library calls will produce system objects when called through the standard interfaces. To obtain an object which can be subscribed/unsubscribed, new interfaces have to be implemented. For example, the `malloc()` library call always returns system objects. In this way legacy code as well as application code will run out of the box with our new subscriber model. To support subscribable objects, a second interface in implemented: `malloc_object()`.

B. The Subscriber Model

As mentioned earlier, the resource allocation list of a typical OS contains only information on the resource state – e.g., allocated or not – and the processes/address spaces using the resource. To extend this information, we employ our *subscriber model* to express liveness of an object and activities using this object. The basic idea is that **only subscribed objects are 'live'**. This is a very important point when handling errors, since we do not have to handle errors in unused/not live resources. However, this has severe consequences for the data usage. If an activity subscribes a previously unsubscribed object again, the activity must not make any assumptions on the data stored in this object. This implies a new programming model. Before resources can be used, they have to be `subscribe`d and initialized (cf. Figure 1). If an object is used by multiple activities, each activity – even in the same process – has to subscribe to the object. When an activity does not need the object anymore, it can `unsubscribe` itself from

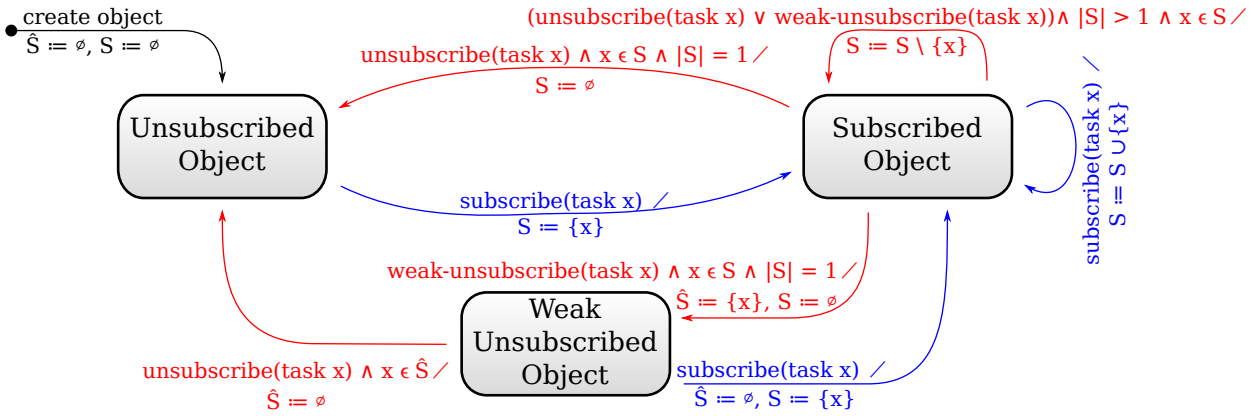
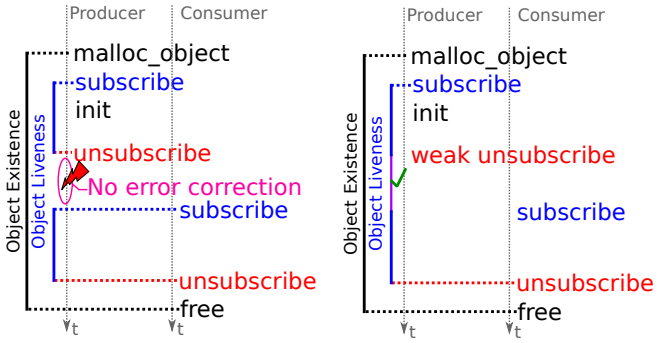


Fig. 2. Subscription States of an Object



(a) Possible data corruption due to unsubscribed object

(b) Extended live time of object due to weak unsubscribe

Fig. 3. Producer and Consumer Problem

the object. Unsubscribe generally means that the resource is of no interest for the activity anymore. As soon as all activities are unsubscribed, the liveness of the object ends.

Unfortunately, this model can provoke pitfalls. Let us assume that there is a producer P and a consumer C. To produce an object O, P allocates memory and subscribes the corresponding object. After finishing calculations, P adds the object O into a buffer and unsubscribes itself from O to produce the next object. The consumer C can now take O from the buffer to process the data. In the meantime, O is in an unsubscribed state which means that occurring errors will not be corrected (cf. Figure 3a). Hence, if data is exchanged between activities in this way, it can be corrupted by faults. One way to cope with this problem would be to insert extra synchronization, so that the producer has to delay the unsubscribe call until the consumer subscribes the object. Unfortunately, this requires either usage of semaphores or polling status variables. In our approach we solve this problem by introducing a third state: *weak unsubscribed object* (cf. Figure 3b). Weak unsubscribing an object will be only possible if the activity already holds a subscription. Literally, weak unsubscribing means that data represented by the object have no relevance for the weak unsubscribing activity anymore, but perhaps for other (unknown) activities in the future.

In Figure 2 all subscription states are depicted. Each object is associated with two sets S and $\hat{S}$ of subscribers tracking the subscribing and weak unsubscribing activities, respectively. If S contains subscribers, $\hat{S}$ will be empty and vice versa. As can be seen, a weak unsubscribed object has exactly one activity in $\hat{S}$. If more than one activity is in S ($|S| > 1$) and an activity performs an `weak unsubscribes` operation, it will be interpreted just as normal `unsubscribe` call. In the opposite case, if $|\hat{S}| = 1$ and another activity subscribes to the object, the weak unsubscribed activity gets unsubscribed automatically. This behavior perfectly matches the producer consumer problem. Before putting an object into a buffer, the producer weak unsubscribes the object. As soon as the consumer activity subscribes to the object, the producer gets unsubscribed by the OS. With this methodology there are no periods of time where vital data are unsubscribed.

To summarize, an object – and hence the resource it represents – can be in exactly one of the following states:

1) **Unsubscribed**: No activity is using the resource. The resource contains no live data.
2) **Subscribed**: Activities are using the resource. The resource contains live data.
3) **Weak unsubscribed**: No activity is using the resource. The resource contains live data.

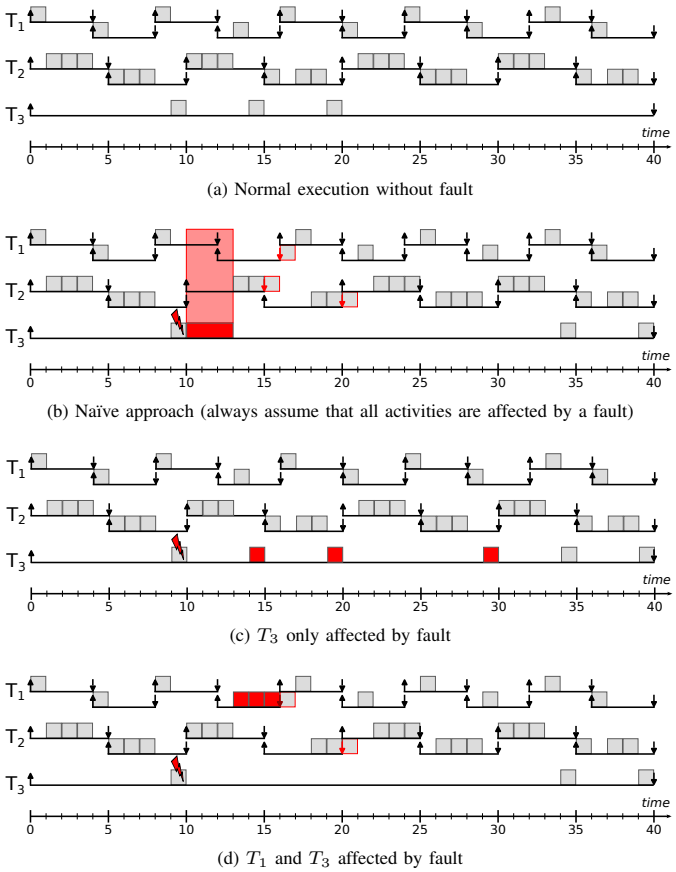
These states can directly be used in error handling. Only those activities (which are in set S) have to be suspended which are subscribed to the object that is affected by a fault. If the object is unsubscribed (S as well as Ŝ are empty), no error correction will be necessary.

C. Using the subscriber model to schedule error correction

We illustrate the advantages of the subscriber model with the example activity set defined in Table I. All activities are preemptive. As scheduling strategy we use Earliest Deadline First (EDF) [10]. If two activities have the same dynamic priority, the static priority will be used to determine the activity to execute. The static priorities of the activities are defined as: $T_1 > T_2 > T_3$. Figure 4(a) shows the corresponding EDF schedule without faults.

Activity	Period P_i	Deadline d_i	Execution Time C_i
T_1	4	4	1
T_2	5	5	3
T_3	40	40	3

TABLE I
EXAMPLE TASK SET



(a) Normal execution without fault

(b) Naïve approach (always assume that all activities are affected by a fault)

(c) T_3 only affected by fault

(d) T_1 and T_3 affected by fault

Fig. 4. Fault Correction and Real-Time

For the remaining scenarios (b), (c), and (d) a transient fault occurs in time slot nine. The corresponding error correction method is assumed to take three time slots.

If no mapping of (affected) objects to activities is available, a conservative error handling approach will be required (scenario (b)). This means error correction has to be scheduled immediately, since every activity has to be assumed to be affected by the fault. However, this schedule leads to three deadline misses.

The key to adhere to deadlines under fault influence is to schedule error correction in a flexible way. Therefore, a mapping of faulty objects to activities using that object is mandatory. If the affected activities are known, it will be possible to schedule the error correction method with the maximum priority of the affected activities. Hence, higher prioritized activities can continue execution. The subscriber model described in this paper can be used to provide the required mapping.

In scenario (c) only T_3 is subscribed to the object(s) affected by the fault. Hence, the error correction is scheduled with the priority of T_3. In this scenario all deadlines are kept.

An example case where two activities (T_1 and T_3) are affected by an error is depicted in scenario (d). Since T_2 has the highest priority and T_2 is fault-free, T_2 can be scheduled immediately. After T_2 finishes execution, T_1 has the highest priority. Due to an error affecting T_1, the error correction method has to be executed first. In this scenario two deadlines are missed. However, as can be seen at the end of time slot 20, it is possible that uninvolved activities still miss their deadlines.

To summarize, if the knowledge of activities using faulty objects is available, it will be possible to schedule error correction with respect to activity priorities. The possibility to schedule the error correction guarantees that higher prioritized activities not affected by faults can be executed before error correction. While this approach can reduce the amount of deadline misses, it is not guaranteed that all activities keep their deadlines. However, the scheduling invariant – the activity with the highest priority gets executed – is maintained all time.

D. Checkpoint and Recovery

Checkpoint and recovery [11] is a common strategy for error handling. Creating a checkpoint, however, can require a huge amount of resources. On the one hand, memory is required to store checkpoints. On the other hand, precious computation time is used to create checkpoints. To speed up the checkpoint creation time and to reduce the amount of data stored in the checkpoint, a huge variety of methods exist (e.g. [7], [8], [9]). The common idea of all such methods is to store only live data or to store recently changed data incrementally. Unfortunately, several methods rely on hardware features which are not guaranteed to be supported by all architectures.

To deal with this problem methods exists which are based on software implementations only. For example, hashes can be calculated to detect changed data. Hash calculation, however, typically requires a huge amount of computation time.

In this paper we propose to use our subscriber model to reduce the amount of data to checkpoint. As mentioned earlier, one basic mechanism to reduce the amount of data for checkpointing is to store only live data. The subscriber model is perfectly suited for this task, since the liveliness of objects is directly expressed by the subscription state. Due to the fact that only objects are assumed to be live which are either in the subscribed state or in the weak unsubscribed state, all unsubscribed objects can be excluded from checkpoints. Since the subscriber model is a pure software approach, it is applicable on any architecture.

Our subscriber model based checkpointing approach is orthogonal to other approaches. For example, the hash based method can be combined with the subscriber model to reduce the memory regions which have to be hashed.

IV. Evaluation

In this section we evaluate the overhead of the subscriber model using a real-world application example. We also measure the amount of saved data in the checkpoint.

A. Experimental Setup

For the evaluation we simulate an embedded system with the Synopsys CoMET simulator [12]. CoMET is a cycle-accurate simulator supporting a large variety of processors and periphery devices. We configured CoMET to simulate a 1.2 GHz ARM926 system with 64 MiB RAM, and 16 MiB ROM. The memory bus is clocked at 400 MHz. Furthermore, we added a 640x480 pixel frame buffer device for video output. To measure execution times on the simulator we use the built-in timing functions measuring the simulated number of cycles and the simulated time.

The software stack consists of our own microvisor, implementing the subscriber model, and the Real-Time Executive for Multiprocessor Systems (RTEMS [6]) as guest OS. RTEMS is a library real-time operating system supporting different standards such as POSIX and BSD sockets. On top of RTEMS we execute an H.264 constrained base profile video decoder [2]. We configured our application to create a checkpoint at each frame buffer output.

The subscribe/unsubscribe functionalities are exported as hypercalls from our microvisor. We have para-virtualized RTEMS and have added support for the subscriber model. We ported our H.264 video decoder application to use the subscriber model for all major dynamic data structures, such as frame buffers, slice buffers, residuals, and GUI elements. All other parts of the application as well as libraries and RTEMS are using the legacy system object support.

B. Subscriber Model Overhead

To evaluate the overhead we use our previously described experimental setup to decode 1,200 frames of an H.264 video with a frame rate of 10 frames per second.

For a single subscribe operation we measure 10,700 cycles on average, and for the unsubscribe and weak unsubscribe operation we measured 5,800 cycles on average. With a CPU clock of 1.2 GHz this translates into on overhead of 8.92 µs and 4.83 µs per call, respectively. In the context of our H.264 benchmark this results in a total overhead of 0.07 %.

Another interesting question is, why there is such a big difference between the subscribe and the (weak) unsubscribe calls? The reason for this is that after the subscribe hypercall RTEMS has to check whether the task has subscribed a faulty object or not. If the subscribed object is faulty, the task will be suspended until error correction of that object. For the check operation, RTEMS disables IRQs and dispatching to avoid race-conditions. These operations account for the additional overhead.

C. Using the Subscriber Model for Optimizing Checkpoints

To evaluate the optimization potential of the subscriber model in the context of checkpoint creation, we compare the execution of the H.264 video decoder with and without using the subscriber model. Again, we use our experimental setup with the H.264 video application decoding 1,200 frames with a frame rate of 10 fps.

	Avg. Checkpoint Size	**CPU Utilization**
Normal	15.6 MB	74.5 %
Subscriber Model	4.9 MB	56.1 %

TABLE II
Checkpoint Measurement

The measurement results are depicted in Table II. As can be seen, we were able to shrink the size of a checkpoint to one third of the original size. That is an excellent result, since this will reduce the amount of additional resources needed for storing checkpoints significantly. Another important observation is the decreased CPU utilization when using the subscriber model, since less data has to be stored in the checkpoint. In our H.264 example we decreased the CPU load by 24.7 %. Due to this large execution time savings, the overhead introduced by the subscriber model is more than mitigated.

V. Conclusions

The subscriber model has big advantages. Due to subscribed objects, activities using a faulty object can be easily determined during runtime.

To handle errors in software, it is very important to map errors to the activities which are affected by the errors. This is a basic requirement to build a flexible error handling system for real-time systems. If an object is affected by a fault, the related activities can be interrupted until the fault is handled. This enables unaffected high priority activities to continue execution. In this paper we presented the subscriber model which fulfills this requirement. We showed that the subscriber model induces only a small runtime overhead.

By interpreting the subscriber state as liveness information we are furthermore able to use the subscriber model to optimize the checkpoint size. For the used H.264 benchmark we are able to achieve savings in processor load of 24.7 % and to shrink the checkpoint size to one third.

The next step is to combine the subscriber model with the reliable and unreliable data model presented in [13].

VI. 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] International Technology Roadmap for Semiconductors (ITRS), "Executive Summary," 2009.

[2] A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, "Improving Transient Memory Fault Resilience of an H.264 Decoder," in *Proc. of ESTIMedia '10*. IEEE, 2010.

[3] 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.

[4] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, and L. C. D. Grossma, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," in *Proc. of PLDI '11*. ACM, 2011.

[5] A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, "Using Application Knowledge to Improve Embedded Systems Dependability," in *Proc. of HotDep*. Vancouver, Canada: USENIX, 2010.

[6] OAR Corporation, "RTEMS: Real-Time Executive for Multiprocessor Systems," 2013, *Online:* http://www.rtems.com, visited July 2013.

[7] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, 1995.

[8] P. Bernstein, "Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing," *IEEE Transactions on Computers*, vol. 21, no. 2, 1988.

[9] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: a lightweight extension for rollback and deterministic replay for software debugging," in *Proc of ATEC*. USENIX, 2004.

[10] W. A. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quarterly*, vol. 21, no. 1, pp. 177–185, 1974. [Online]. Available: http://dx.doi.org/10.1002/nav.3800210113

[11] K. M. Chandy and C. V. Ramamoorthy, "Rollback and recovery strategies for computer programs," *IEEE Transactions on Computers*, vol. 100, no. 6, 1972.

[12] Synopsys Corporation, "COMET/METeor Models," *Online:* http://www.synopsys.com/Systems/VirtualPrototyping/VPModels/Pages/CoMET-METeor.aspx, visited July 2013.

[13] F. Schmoll, A. Heinig, P. Marwedel, and M. Engel, "Improving the Fault Resilience of an H.264 Decoder using Static Analysis Methods," *ACM TECS*, vol. 13, no. 1s, 2013.