Multi-layer software reliability for unreliable hardware

1 Introduction

Technology scaling in the nano-era has led to various reliability threats (like increased susceptibility to soft errors, aging, manufacturing-induced variations, power density and thermal issues, etc.) that hamper the cost-effective applicability of scaled technologies. As a result, reliability/dependability has emerged as a first-class design constraint. Earlier approaches have mitigated reliability threats mainly at the device, circuit, and architecture layers. Within the past decade, several cross-layer techniques have evolved that engage two or more adjacent layers (e.g., circuit and architecture or architecture and operating system) to improve the full system reliability. A comprehensive survey of reliability techniques at different design abstractions can be found in [1].

These reliability threats mainly arise at the device level and propagate to the upper design layers, ultimately jeopardizing the correct application software execution [1, 2]. In particular, such errors are of serious concern for the safety-and mission-critical applications. Moreover, a real-world system executes tasks with mixed criticality that requires special reliability considerations.

The resilience of a system highly depends upon how errors propagate from the lower layers to the higher system layers because different system layers manifest distinct masking effects like, logical masking at the circuit level, architecture-level masking, and program-level masking (e.g., due to the control and data flow). Therefore, in order to effectively mitigate multiple reliability threats, besides multiple hardware layers, it is crucial to develop reliability methods at multiple software layers (like application, compiler, and operating system) and to engage them in a synergistic design and operational flow. Exploration of synergistic interactions between all software layers and available run-time tradeoffs have not yet been explored in the literature and it requires proactive optimization and adaptation of multiple software layers to each other.

Recently the work in [3] introduced the notion of multi-layer reliability that refers to engaging, optimizing, and/or adapting typically more than two hardware and/or software layers (adjacent or non-adjacent in the system hierarchy) at design or run-time to optimize the full system reliability in a cost-effective way. Following this notion, in this paper, we present a holistic modeling and optimization...
Figure 1: Multi-Layer Software Reliability: An Overview of the Design and Optimization Flow.

flow for multi-layer software reliability targeting unreliable or partially hardened hardware.

This paper is part of the DFG funded priority program SPP 1500 “Dependable Embedded Systems”\(^1\) [4, 5] and covers the software-related projects namely GetSURE [6], DanceOS [7, 8], ASTEROID [9], and FEHLER [10]. This paper is part of the IT special issue on “Embedded Systems”. Other papers from SPP 1500 in this issue are: “Adaptive Multi-Layer Techniques for Increased System Dependability” [11] and “Application-aware Cross-Layer Reliability Analysis and Optimization” [12].

2 Multi-layer software reliability

Figure 1 presents an overview of our multi-layer software reliability framework for homogeneous/heterogeneous processors consisting of unprotected and partially/fully hardened cores. Depending upon the varying resilience properties of different applications, an appropriate core can be allocated under constrained scenarios. In the software layers, Figure 1 illustrates different system abstractions like (1) Aspect-Oriented Programming; (2) Reliability-Driven Compilation; (3) Resilient Application Design; (4) Reliable Operating System (OS) Services and Architecture; and (5) Dynamic Reliability Analysis.

The application and operating system can be programmed using aspect-oriented programming (Section 4) that allows programmers to implement fault-tolerant mechanisms within the application software. Similarly both application and operating system can be compiled using a reliability-driven compiler (Section 5) to improve their fault tolerance. At run-time, the resilient or non-resilient application code is executed using an OS (Section 6) which is not only itself reliable but also provides other services for reliable application execution that is evaluated using dynamic reliability analysis (e.g., using fault injection). Enabling efficient reliability optimization at these software layers also requires corresponding reliability models at appropriate granularity (Section 3).

The reliability modeling and estimation flow is bottom-up because the errors occur at the hardware-layer which propagate and manifest (e.g., as bit flips) at the software-layer. For reliability optimization, we follow a top-down flow...
as functional and timing correctness depend upon application characteristics and user requirements. For efficient reliability modeling and optimization, our framework requires key information from the hardware layers along with the chip process variation and thermal maps.

In the following, we explain the key components of our multi-layer software reliability framework.

### 3 Multi-layer software reliability modeling

Figure 2 illustrates our concept of multi-layer reliability modeling with the information exchange across different layers for accurate reliability estimation. Software-level reliability models need to be adapted to the appropriate granularity of each particular software layer while also accounting for the hardware-level knowledge in order to bridge the gap between the hardware and software. At the hardware-layer, we consider the parameters like (1) probability of fault of different processor components obtained through a detailed gate-level soft error analysis [13], (2) aging estimation of different logic elements at the circuit-level [14], (3) spatial and temporal vulnerabilities of different instructions executing through different processor components [2, 15]. At the software-level, we account for control and data flow graphs (CDFG), basic block execution probabilities, register live-in and live-out information, etc. Note, the hardware-level parameters may be highly affected by the software characteristics, like switching activity depends upon the input data and CDFG. Similarly soft error rates and aging can be affected by temperature and voltage-frequency levels that are affected by the decisions taken by the operating system. Therefore, joint consideration of hardware- and software-level parameters and their interdependencies is of key importance.

Towards this end, we developed reliability estimation models at instruction and basic block levels for reliability-driven compilation, and at function/task level for the reliability-driven run-time system. Our Instruction Vulnerability Index (IVI), Instruction Masking Index, and Error Propagation Index models estimate the error vulnerabilities, error masking probabilities, and error propagation effects, respectively, at the instruction granularity (see model details in [2, 14, 16]). These models are used to characterize the reliability importance of instructions in a given CDFG and are also used as input to derive the reliability at function/task granularity.

### 4 Language support for reliability

A software developer can implement fault-tolerance mechanisms within the application software. For example, critical data structures can be protected by checksums to detect memory errors. Exploiting the developer’s knowledge, less critical data is left unprotected to optimize for performance. However, the resulting error detection code tends to be scattered across many modules where a checksum is needed. In general, intermixing fault-tolerance with the core logic of the application not only increases software complexity, but it is also a tedious and error-prone task.

To address this problem, we developed the AspectC++ [17] programming language and compiler, which supports the modular implementation of reliability mechanisms. The key concept is to separate a reliability mechanism’s implementation (i.e., the what) from the source code locations it is applied to (i.e., the where). Figure 3 illustrates this separation with an example: A memory-error detection aspect constitutes a reusable module that augments certain data structures by a checksum.

![Figure 2: Multi-Layer Software Reliability Modeling and Estimation Considering Hardware-Level Information.](image)

![Figure 3: Modular and generic memory-error detection implemented in the AspectC++ programming language.](image)
The pointcut expression textually describes the software components to be hardened by error detection. This example considers the Scheduler data structure from the operating-system layer (OS) and the Task1 data structure from the application layer to be critical. The following advice extends those data structures by a checksum, whereas the remaining pieces of advice implement the actual error detection, which is carried out before and after function calls to such data structures.

We successfully applied similar dependability aspects to the embedded Configurable operating system (eCos) [18] and the memcached [8] application software. Extensive fault-injection experiments indicate that operating-system failures caused by errors in the kernel data structures are thereby reduced to below 0.06%.

Furthermore, we have developed transparent majority voting [8] and two fault-tolerance aspects that avoid control-flow errors caused by hardware faults in main memory, cache, bus, and CPU. In particular, the dynamic dispatch of virtual functions in C++ [19] and procedure calls in C/C++ code [20] are protected against hardware faults.

The strengths of aspect-oriented fault tolerance are the applicability to every layer in the software stack, and the exploitation of programmer’s high-level knowledge. On the other hand, source code alone provides no insights into low-level details such as the machine code and compiler optimizations as discussed below.

5 Reliability-driven compilation

A reliability-driven compiler offers significant opportunities to generate reliable application code for unreliable or partially hardened hardware platforms. For effective reliability improvements, both hardware knowledge (e.g., number of available registers, area and logical masking properties of different processor components) and software characteristics (e.g., variable lifetime, instruction types and dependencies, basic block execution probabilities) need to be considered. In the following, we will discuss how different front-/middle-/back-end algorithms in a compiler can be re-designed under reliability considerations. In particular, we will present an overview of our reliability-driven transformations, instruction scheduling, and selective instruction protection techniques. Our reliability-driven compiler (see Figure 4) applies these techniques to generate multiple reliable code versions with diverse reliability and execution time properties. These multiple versions facilitate the run-time system (see Section 6.1) to explore the reliability-performance optimization space at run-time, considering varying fault rates and resilience/performance properties of the concurrently executing applications [6].

5.1 Reliability-driven transformations

The key goal of our reliability-driven transformations is to minimize the error probabilities (both SDCs – Silent Data Corruptions and Application Failures) by reducing the spatial and temporal vulnerabilities (e.g., by reducing the register content and control flow vulnerabilities) and number of critical instruction executions (such as load, store, branches, etc.) under user-provided performance overhead constraints. In [2, 15], we proposed the following four transformations and investigated their impact on the software code reliability (see Figure 4).

1. Reliability-Driven Data Type Optimization targets reducing the number of critical instruction executions by transforming the smaller bit-width data types into larger bit-width data types for given data structures, while minimizing the function vulnerabilities.
2. **Reliability-Driven Loop Unrolling** determines an unrolling factor that jointly minimizes the spatial and temporal vulnerabilities of different instructions under performance and code size constraints. It explores the tradeoff between reduced loop evaluation instructions and vulnerabilities of live register variables.

3. **Reliability-Driven Common Expression Elimination and Operation Merging** reduces the vulnerabilities by removing identical expressions and/or merging partially common sub-expressions. It investigates the reliability effects (e.g., reduced SDCs) of recomputation and register variables with increased lifetime while also accounting for register spilling.

4. **Reliability-Driven Online Table Value Computation** evaluates whether precomputed table values with increased memory vulnerability would be beneficial from the overall function reliability perspective or the online computation with increased instruction vulnerabilities in the pipeline.

When comparing to performance-optimized transformations, our reliability-driven transformations result in 60% reduced application failures and on average 57% reduced application vulnerabilities leading to reduced SDCs [15].

### 5.2 Reliability-driven instruction scheduling

The instruction scheduler determines the instruction execution sequence that directly influences their vulnerabilities in different processor components. A performance-driven scheduler may degrade the software reliability, for instance, by scheduling critical instructions after a pipeline stalling instructions or increased spatial vulnerability due to increased register usages. In contrast, our soft error driven instruction scheduler [21] improves the software reliability by prioritizing the instructions with the highest reliability-weight, which is a joint function of statically estimated spatial and temporal vulnerabilities [22], instruction’s criticality, probabilities of different program errors, and number of dependent instructions (see Figure 4). It employs a lookahead-based heuristic to evaluate the reliability weights of different scheduling candidate instructions in conjunction with their dependent instructions, thus minimizing the risk of scheduling a critical instruction after a multi-cycle or a pipeline stalling instruction (if possible). On average, our scheduler provides 22% reduced application program failures compared to state-of-the-art schedulers [21].

### 5.3 Reliability-driven instruction protection

Full-scale instruction redundancy techniques like EDDI [23] incur significant energy, code size/memory, and performance overhead and thus cannot be applied in resource-constrained (embedded) systems. Moreover, different instructions in an application program exhibit varying reliability importance due to their diverse error vulnerability and masking properties as a result of changing data and control flow behavior. For such scenarios, we employ a constrained instruction protection technique [16] that selectively applies redundancy to a subset of instructions with the highest reliability profit value under a given performance overhead constraint. The reliability profit is computed as a joint function of instruction vulnerabilities, error masking probabilities, error propagation effects, and protection overhead. In case of sequential dependencies, it may be beneficial to jointly protect a group of instructions as the voting and checking is only required at the end of the sequential group, thus incurring a reduced protection overhead. Compared to state-of-the-art software protection schemes, our constrained instruction protection technique improves the reliability efficiency by 30%–60% [16] on average.

### 5.4 Reliability-driven type annotation

To reduce the data protection overhead, our compiler automatically classifies error-affecting data in the front-end through source-to-source compilation techniques and static code analysis. The data is classified according to the impact of soft errors on the application execution, i.e., resulting in application failures (e.g., due to erroneous pointers) or incorrect output (e.g., bit flips in the output). To enable this, the reliability-driven compiler employs two reliability type qualifiers: **reliable** and **unreliable**. Reliable data is expected to require error-protection mechanisms and will otherwise result in application failures, while the unreliable data can be left unprotected while allowing for output errors, if tolerable by the application user. Moreover, the unreliable data should not lead to application failures or unrecoverable system states. To increase the potential of unreliable annotations, the compiler only annotates data as reliable which are used as pointers, in offset calculations, can have influences on the control flow, or can lead to arithmetical exceptions. With this method, up to 64% of all data objects of a 720p H.264 video decoder were classified as unreliable [10]. A complementary approach for selective data protection at the hardware-level is proposed in [24]. Furthermore, a classification database
is created for the operating system to enable flexible error handling based on the error-affected data and the current system conditions, like available slack time and resources for error handling.

6 Reliability-driven operating systems

The operating system has a key role in reliable execution of the software. On one hand, it has the potential to dynamically implement reliability-aware scheduling and resource management policies (Section 6.1) and can implement generic services that enhance the reliability of applications that are oblivious to potential hardware faults (Section 6.2). On the other hand, reliability extensions imply additional constraints on real-time scheduling analysis (Section 6.3). Furthermore, the OS forms the Reliable Computing Base [25] of all other software-implemented fault tolerance techniques and therefore requires special care to be protected against hardware failures itself (Section 6.4).

6.1 Run-time reliability management

In real world scenarios, a system is subjected to multiple reliability threats, i.e., aging, process variation, and soft errors. Towards this issue, we employ a dynamic reliability management system dTune [14] that optimizes the reliability at run-time while jointly accounting for soft errors and cores’ performance variations due to design-time process variation and runtime aging-induced performance degradation. It performs three key operations. (1) First, it activates or deactivates the redundant multi-threading (RMT) mode (i.e., thread execution with or without thread replication on different cores) for a subset of application tasks out of many concurrently executing applications while accounting for their resilience properties, the history of encountered errors at run-time, and the available number of cores. (2) Afterwards, it performs a variation-aware task-to-core mapping considering the tasks’ deadlines, task execution properties, and cores’ performance variations while reducing the synchronization overhead when performing comparison/voting for redundant thread executions on different cores with frequency variations. (3) At the end, it selects an appropriate reliable compiled version that improves the soft error resilience while considering the task deadline and core’s frequency. Compared to four different state-of-the-art techniques, our dTune system achieves on average 44% (maximum 63%) improved system reliability for different chip configurations with numerous variability maps at different aging years [14].

6.2 Replication for binary-only programs

The previously discussed mechanisms require application source code to be available.

To accommodate for the large range of binary-only applications, we developed L4/Romain, an operating system service that transparently replicates binary programs using a software implementation of redundant multithreading [9]. L4/Romain splits the application into a master process and N replica processes to leverage MMU-based fault isolation. To minimize execution time overhead, replicas execute independently on different CPU cores.

The L4/Romain master process (1) compares replicas’ states whenever they generate an event that becomes externally visible, such as a page fault, a system call or any other CPU exception. (2) The master acts as a resource manager for all replicas. It maintains dedicated copies of each resource (e.g., memory) for each replica to avoid relying on hardware-level protection, such as ECC. (3) If the replicas agree to perform a system call, the master proxies this call and returns the result to all replicas before they resume execution.

We showed that the accompanying execution time overhead for the SPEC CPU 2006 suite of benchmarks is less than 14% for TMR execution. Our experiments indicate that CPU assignment needs to be done with care because replication overhead is influenced by application- and hardware-specific properties, such as the event rate as well as cache usage [26].

When replicating multithreaded applications, scheduling-induced non-determinism may lead to diverging application behavior even in the non-faulty case. L4/Romain enforces deterministic execution across replicas by instrumenting pthread lock operations and enforcing identical ordering of lock acquisitions [27]. This instrumentation increases replication overhead to up to 65% for...
four worker threads in TMR mode, and heavily depends on the replicated program's lock acquisition frequency.

Software-level replication relies on an efficient voting mechanism to compare replica states. We integrated a hardware-level fingerprinting technique [28] into the CPU pipeline as shown in Figure 5. Our fingerprints improve previous solutions by allowing the calculation of per-context data and instruction fingerprints. Thereby every replica generates a dedicated fingerprint and state comparison becomes fast.

6.3 Real-time analysis for replication

Voting between replicas imposes several problems with respect to real-time analysis. The biggest problem arises as a consequence of the complex scheduling interactions in large potentially heterogeneous multicore. Depending on the task mapping to cores, other processes interfere with replicas. This interference can lead to additional stall times when replicas need to wait for others to catch up. The resulting scheduling problem is similar to other applications, such as high-performance computing (map-reduce, OpenMP). These applications exhibit a fork-join structure and have been studied by the real-time community for a long time (e.g., [29]) but related work falls short to obtain response-time bounds.

We provided an approach to determine a conservative bound for the response-times for such fork-join tasks under fixed priority scheduling [30]. We found that mapping replicas to distinct cores does not always outperform redundancy in space. This heavily depends on higher priority interference, actual mapping parameters, and the rate of state comparisons.

6.4 Generating a dependable RTOS

While L4/Romain transparently provides reliability to even existing applications, it still requires the underlying OS kernel itself to be executed on reliable hardware to constitute a reliable computing base (RCB) [25]. Our approach primarily targets at the prevention and reliable detection of faults across the complete OS, based on (a) constructive measures to reduce the vulnerability of OS code and (b) strict tailoring of the OS functionality towards the application and hardware. We address the remaining SDCs of the kernel execution path by pointer-less control-flow structures, fine-grained assertions, and AN-encoding [31] (an arithmetic data encoding also including control flow and temporal information).

The concept of AN-encoding has already been taken up in compiler- and interpreter-based solutions [32, 33]. Yet, these generic realizations are not practicable for realizing a RCB – not only due to their immense overhead of a factor of $10^3 - 10^5$, but also due to the specific nature of low-level system software. Thus, following our CoRed concept [34], we concentrate the encoded execution to the *minimal necessary points*, to keep the overhead on a bearable level [35].

The efficient application of such measures requires a generative approach with system-wide optimizations in the generation process, based on a detailed control-flow model acquired by static analysis. It includes, for instance, the task-specific specialization of system calls – similar to Synthesis [36], but already at compile time (see Figure 6). However, existing static analysis techniques are restricted till the *application* → *kernel* boundary (with the notable exception of [37] that partly carries out the analysis into the kernel, but not out of it, i.e., it stops at the *kernel* → *application* boundary). In contrast, we facilitate (based on techniques adopted from [38] and the stringent OSEK/AUTOSAR system model [39, 40]) static per-CPU control-flow analysis across the kernel and its scheduler for the aggressive tailoring of our kernel and the applied dependability aspects.

The tailoring process is exemplified in Figure 6: By means of a system-wide analysis (Figure 6b), calls into the RTOS (such as the ActivateTask($X$) in Figure 6a) in the application code are specialized per call-instance and transformed so that they pass encoded parameters (such as $X_{enc}$ in Figure 6c) into the kernel; tasks that employ AN encoding also at the application level can directly interact with the kernel. The point here is that in either case the complete kernel execution path is protected. All faults that corrupt the kernel state or kernel control flow are reliably

![Figure 6](https://example.com/figure6.png)

**Figure 6:** On the basis of the original application (app.c) and the system description (app.oil) a system analysis reveals the task dependencies and the expected runtime behavior according to the OSEK specification. Subsequently a tailored kernel (dosek.c) is generated, where the expected behavior information is merged into encoded system call parameters.
detected – the RTOS can be trusted as a RCB for the implementation of application-level dependability measures.

We could already show [7] that employing a static RTOS design (all tasks and resources are known at compile time, as mandated, e.g., by the automotive OSEK or AUTOSAR software stacks) alone leads to a five times higher inherent kernel resilience than a more UNIX-inspired dynamic design (that is, tasks and resources are allocated at startup-time or run-time, as implied by POSIX semantics and implemented in, e.g., eCos or FreeRTOS). These results were obtained using a realistic embedded control-system (the flight controller of an autonomous flight vehicle), consisting of 14 OSEK tasks and several resources, events, and alarms.

However, first results with our additional measures (strict tailoring, pointer-less kernel design, AN-encoded kernel path) show that thereby we can significantly reduce the number of SDCs.

With these techniques, we push software-based OS dependability to its limits, as all remaining silent data corruptions (SDCs) stem from faults that occur in a (very small) window before the kernel is left and register values have to be decoded to continue execution of the user-mode program. At this point, hardware-measures have to take over.

7 Conclusion

In this paper, we discussed how multiple software layers (i.e., application, compiler, and operating system) can be leveraged synergistically to improve the overall system reliability. Our optimization flow spans over multiple software layers and requires important information from the underlying hardware layers to accurately model the reliability at software abstractions in order to bridge the gap between hardware and software. Our software-level techniques can be employed in conjunction with other hardware-level techniques as – in a complex and highly dependable on-chip system – all abstraction layers need to be engaged to provide cost-effective reliability.

Funding: This work is supported in parts by the German Research Foundation (DFG) as part of the priority program “Dependable Embedded Systems” (SPP 1500 – http://spp1500.itec.kit.edu).

References

8. A. Martens et al., Crosscheck: Hardening replicated multi-threaded services, Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV), 2014.
18. C. Borchert, H. Schirmeier, and O. Spinczyk, Generative software-based memory error detection and correction for operating system data structures, IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2013.


Bionotes

Dr. Muhammad Shafique
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems (CES), Germany
muhammad.shafique@kit.edu

Muhammad Shafique received the Ph.D. degree in computer science from the Karlsruhe Institute of Technology (KIT), Germany, in Jan. 2011. He is currently a Research Group Leader and Lecturer at KIT. He holds one U.S. patent, received the ACM SIGDA Outstanding New Faculty Award 2015, six gold medals and several Best Paper Awards.

Philip Axer
TU Braunschweig, Institut für Datentechnik und Kommunikationssysteme, Germany
axer@ida.ing.tu-bs.de

Philip Axer is currently pursuing his Ph.D. at the Embedded System Design Automation Group (IDA), TU Braunschweig, Germany.

Christoph Borchert
TU Dortmund, Germany
christoph.borchert@tu-dortmund.de

Christoph Borchert is pursuing his Ph.D. at the Embedded System Software group at TU Dortmund, Germany.
Prof. Dr. Jian-jia Chen
Department of Informatics, TU Dortmund, Germany
jian-jia.chen@cs.uni-dortmund.de

Jian-Jia Chen is Professor at Department of Informatics in TU Dortmund, Germany. He was Juniorprofessor at Department of Informatics in KIT, Germany from May 2010 to March 2014. He received his Ph.D. degree from Department of Computer Science and Information Engineering, National Taiwan University, Taiwan in 2006. He received several Best Paper Awards.

Kuan-Hsun Chen
Department of Informatics, TU Dortmund, Germany
kuan-hsun.chen@tu-dortmund.de

Kuan-Hsun Chen is pursing his Ph.D. at the Chair for Design Automation of Embedded Systems, TU Dortmund, Germany.

Björn Döbel
TU Dresden, Dept. of Computer Science, Germany
doebel@os.inf.tu-dresden.de

Björn Döbel is pursuing his Ph.D. at the TU Dresden, Germany and his work focuses on implementing operating system services to protect applications against the effects of hardware faults.

Prof. Dr. Rüdiger Kapitza
TU Braunschweig, Germany
kapitza@ibr.cs.tu-bs.de

Rüdiger Kapitza is professor at the Technische Universität Caro- Wilhelmina zu Braunschweig. There he leads the Distributed Systems Group of the Institute of Operating Systems and Computer Networking since January, 2012. He received his M.Sc. and Ph.D. degree from the Department of Computer Sciences, University of Erlangen-Nuremberg in 2001 and 2007, respectively.

Florian Kriebel
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems (CES), Germany
florian.kriebel@kit.edu

Florian Kriebel is currently pursuing the Ph.D. degree from the Chair for Embedded Systems, KIT, Germany. He was the recipient of the CODES + ISSS 2011 Best Paper Award.

Dr. Daniel Lohmann
FAU, Germany
dl@cs.fau.de

Daniel Lohmann is an Assistant Professor at the Chair of Distributed Systems and Operating Systems at FAU, Germany. In 2003 he joined the group of Wolfgang Schröder-Preikschat at FAU, where he received his PhD (Dr.-Ing.) in 2009 and his venia legendi (Dr.-Ing. habil.) in 2014.

Prof. Dr. Peter Marwedel
TU Dortmund, Dept. of Computer Science, Germany
peter.marwedel@tu-dortmund.de

Peter Marwedel is professor at the department of computer science at TU Dortmund, Germany. He received his PhD in physics from the University of Kiel in 1974 and habilitated in informatics in 1987. Prof. Marwedel is an IEEE Fellow and ACM Senior Member.

Semeen Rehman
Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems (CES), Germany
rehman@ira.uka.de

Semeen Rehman is pursuing the Ph.D. degree from the Chair for Embedded Systems, KIT, Germany, since 2008. She was the recipient of the CODES + ISSS 2011 Best Paper Award and several HiPEAC Paper Awards.

Florian Schmoll
TU Dortmund, Dept. of Computer Science, Germany
florian.schmoll@cs.tu-dortmund.de

Florian Schmoll is pursuing his Ph.D. at TU Dortmund, Germany with a focus on embedded systems, compilers, and WCET optimization.
Prof. Dr. Olaf Spinczyk
TU Dortmund, Germany
olaf.spinczyk@tu-dortmund.de

Olaf Spinczyk is professor of Computer Science at Technische Universität Dortmund, Germany, where he leads the Embedded System Software Group. Before 2007, Olaf worked at the University of Erlangen-Nuremberg and the University of Magdeburg, where he received his PhD for his research on “Operating System Construction by Aspect-Orientation” in 2002.