Dependability Aspects in Configurable Embedded Operating Systems

Horst Schirmeier, Christoph Borchert, Martin Hoffmann, Christian Dietrich, Arthur Martens, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk

1 Introduction

Future hardware designs for embedded systems will exhibit more parallelism and energy efficiency at the price of being less reliable, due to shrinking structure sizes, increased clock frequencies, and lowered operating voltages [9]. In embedded control systems, the handling of soft errors—e.g., transient bit flips in the memory hierarchy—is becoming mandatory for all safety integrity level (SIL) 3 or SIL 4 categorized safety functions [30, 35]. Established solutions stem mostly from the avionics domain and employ extensive hardware redundancy or specifically hardened hardware components [55]—both of which are too costly to be deployed in commodity products.

H. Schirmeier (✉)
Embedded System Software Group, TU Dortmund, Dortmund, Germany
e-mail: horst.schirmeier@tu-dortmund.de

C. Borchert · O. Spinczyk
Embedded Software Systems Group, Osnabrück University, Osnabrück, Germany
e-mail: christoph.borchert@uos.de; olaf.spinczyk@uos.de

M. Hoffmann
System Software Group, FAU Erlangen, Erlangen, Germany
e-mail: hoffmann@cs.fau.de

C. Dietrich · D. Lohmann
Systems Research and Architecture Group, Leibniz University Hannover, Hannover, Germany
e-mail: dietrich@sra.uni-hannover.de; lohmann@sra.uni-hannover.de

A. Martens · R. Kapitza
Institute of Operating Systems and Computer Networks, TU Braunschweig, Braunschweig, Germany
e-mail: martens@ibr.cs.tu-bs.de; kapitza@ibr.cs.tu-bs.de

© The Author(s) 2021

J. Henkel, N. Dutt (eds.), *Dependable Embedded Systems*, Embedded Systems,
https://doi.org/10.1007/978-3-030-52017-5_4

Software-based redundancy techniques, especially redundant execution with majority voting in terms of triple modular redundancy (TMR), are well-established countermeasures against soft errors on the application level [24]. By combining them with further techniques—such as arithmetic codes—even the voter as the single point of failure (SPOF) can be eliminated [53]. However, all these techniques "work" only under the assumption that the application is running on top of a soft-error-resilient system-software stack.

In this chapter, we address the problem of software-stack hardening for three different points in the system-software and fault-tolerance technique design space:

- In Sect. 3 we investigate soft-error hardening techniques for a statically configured OS, which implements the automotive OSEK/AUTOSAR real-time operating system (RTOS) standard [5, 40]. We answer the research question what the *general reliability limits* in this scenario are when aiming at *reliability as a first-class design goal*. We show that harnessing the static application knowledge available in an AUTOSAR environment, and protecting the OS kernel with AN-encoding, yields an extremely reliable software system.
- In Sect. 4 we analyze how programming-language and compiler extensions can help to modularize fault-tolerance mechanisms. By applying the resulting fault-tolerance modules to a *dynamic* commercial off-the-shelf (COTS) embedded OS, we explore how far reliability can be pushed when a legacy software stack needs to be maintained. We show that aspect-oriented programming (AOP) is suitable for encapsulating generic software-implemented hardware fault tolerance (SIHFT) mechanisms, and can improve reliability of the targeted software stack by up to 79%.
- Looking beyond bit flips in the memory hierarchy, in Sect. 5 we investigate how a system-software stack can survive even more adverse fault models such as whole-system outages. Using persistent memory (PM) technology for state conservation, our findings include that software transactional memory (STM) facilitates maintaining state consistency and allows fast recovery.

These works have been previously published in conference proceedings and journals [8, 29, 36], and are presented here in a summarized manner. Section 6 concludes the chapter and summarizes the results of the *DanceOS* project, which was funded by the German Research Foundation (DFG) over a period of 6 years as part of the priority program SPP 1500 "Dependable Embedded Systems" [26] (Fig. 1).

2 Related Work

Dependable Embedded Operating Systems While most work from the dependable-systems community still assumes the OS itself to be too hard to protect, the topic of RTOS reliability in case of transient faults has recently gained attention. The C^3 μ-kernel tracks system-state transitions at the inter-process communication (IPC) level

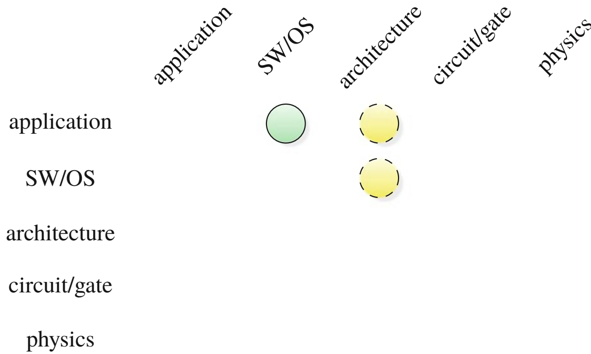

Fig. 1 Main abstraction layers of embedded systems and this chapter's major (green, solid) and minor (yellow, dashed) cross-layer contributions

to be able to recover system components in case of a fault [50]. Their approach, however, assumes that faults are detected immediately and never turn into silent data corruptions (SDCs), and that the recovery functionality itself is part of the reliable computing base (RCB). L4/Romain [19] employs system-call interception to provide transparent thread-level TMR—and, hence, error detection,—but still requires a reliable μ-kernel. The hypervisor approach of Quest-V [34] reduces the *software*-part of the RCB even further—at the price of increasing the *hardware*-part for the required virtualization support. In the end, however, all these approaches assume early and reliable detection of faults and their strict containment inside the RCB, which our three approaches provide.

Software-Based Soft-Error Detection and Correction The concept of AN-encoding has been known for quite a while and has been taken up in recent years in compiler- and interpreter-based solutions [45]. Yet, these generic realizations are not practicable for realizing a RCB—not only due their immense runtime overhead of a factor of 10^3 up to 10^5, but also due to the specific nature of low-level system software. Thus, following our proven CoRed concept [28], we concentrate the encoded execution to the *minimal necessary points*. Besides AN-encoding, several more generic *error detection and recovery mechanisms* (EDMs/ERMs) exist and have been successfully deployed. Shirvani et al. [48] evaluate several software-implemented error-correcting codes for application in a space satellite to obviate the use of a low-performance radiation-hardened CPU and memory. Read-only data segments are periodically scrubbed to correct memory errors, whereas protected *variables* must be accessed manually via a special API to perform error correction. Similarly, *Samurai* [41] implements a C/C++ dynamic memory allocator with a dedicated API for access to replicated heap memory. Programmers have to manually invoke functions to check and update the replicated memory chunks. The latter approach exposes the heap allocator as single point of failure, which is not resilient against memory errors. To automate the hardening process, some works extend *compilers* for transforming code to add fault tolerance [44]. These

approaches are based on duplicating or even triplicating important *variables* of single-threaded user-level programs. Our work differs in that we use the general-purpose AspectC++ compiler that allows us to focus on the implementation of software-based EDM/ERMs in the OS/application layer, instead of implementing special-purpose compilers. AOP also allows to separate the "business logic" from fault-tolerance implementations, which has, e.g., been pioneered by Alexandersson et al. [2]—however at the cost of 300% runtime overhead.

State Consistency in Non-volatile Memories Maintaining state consistency in persistent memory has been achieved on the level of process-wide persistence [10, 39] and specialized file systems [13, 20]. Our *DNV Memory* approach shares the most similarities with libraries that provide safe access to a persistent heap [6, 12, 54]. Mnemosyne [54] shows the overall steps that are needed to build a persistent heap, while NV-Heaps [12] focuses mainly on usability aspects. Both libraries rely on a transactional-memory model that stores logs in persistent memory and executes expensive flush operations to ensure data consistency in presence of power failures. In order to improve performance, the memory allocator of Makalu [6] guarantees the consistency of its own meta data without the need of transactions. However, it does not extend this ability to the data stored within. Thus, library support, similar to Mnemosyne [54], is still needed to enforce durability. *DNV Memory* shares with these approaches the transactional model and the goal to provide a persistent heap, but aims at improving performance and lifetime of persistent applications by reducing the amount of writes to persistent memory. Additionally, *DNV Memory* provides transparent dependability guarantees that none of the previous work has covered.

3 *d*OSEK: A Dependable RTOS for Automotive Applications

In the following, we present the design and implementation of *d*OSEK, an OSEK/AUTOSAR-conforming [5, 40] RTOS that serves as reliable computing base (RCB) for safety-critical systems. *d*OSEK has been developed from scratch with dependability as the first-class design goal based on a two-pillar design approach: First we aim for strict *fault avoidance*[1] by an in-depth static tailoring of the kernel towards the concrete application and hardware platform—without restricting the required RTOS services. Thereby, we constructively minimize the (often redundant) vulnerable runtime state. The second pillar is to then constructively reintegrate redundancy in form of dependability measures to eliminate the remaining SDCs in the essential state. Here, we concentrate—in contrast to others [4, 50]—on reliable *fault detection* and fault containment within the kernel execution path (Sect. 3.2) by

[1] Strictly speaking, we aim to avoid *errors* resulting from transient hardware faults.

employing arithmetic encoding [23] to realize self-contained data and control-flow error detection across the complete RTOS execution path.

We evaluate our hardened *d*OSEK against ERIKA [21], an industry-grade open-source OSEK implementation, which received an official OSEK/VDX certification (Sect. 3.3). We present the runtime and memory overhead as well as the results of extensive fault-injection campaigns covering the *complete* fault space of single-bit faults in registers and volatile memory. Here, *d*OSEK shows an improvement of four orders of magnitude regarding the SDC count, compared to ERIKA.

3.1 Development of a Fault-Avoiding Operating System

Essentially, a transient fault can lead to an error inside the kernel only if it affects either the kernel's control or data flow. For this, it has to hit a memory cell or register that carries *currently alive* kernel state, such as a global variable (always alive), a return address on the stack (alive during the execution of a system call), or a bit in the status register of the CPU (alive only immediately before a conditional instruction). Intuitively, the more long-living state a kernel maintains, the more prone it is to transient faults. Thus, our first rule of fault-avoiding OS development is: ❶ **Minimize the time spent in system calls and the amount of volatile state, especially of global state that is alive *across* system calls.**

However, no kernel can provide useful services without any runtime state. So, the second point to consider is the containment and, thus, detectability of data and control-flow errors by local sanity checks. Intuitively, bit flips in pointer variables have a much higher error range than those used in arithmetic operations; hence, they are more likely to lead to SDCs. In a nutshell, any kind of indirection at runtime (through data or function pointers, index registers, return addresses, and so on) impairs the inherent robustness of the resulting system. Thus, our second rule of fault-avoiding operating-system development is: ❷ **Avoid indirections in the code and data flow.**

In *d*OSEK, we implement these rules by an extensive static analysis of the application code followed by a subsequent dependability-oriented "pointer-less" generation of the RTOS functionality. Our approach follows the OSEK/AUTOSAR system model of static tailoring [5, 40], which in itself already leads to a significant reduction of state and SDC vulnerability [27]. We amplify these already good results by a flow-sensitive analysis of all application–RTOS interactions [17, 18] in order to perform a partial specialization of system calls: Our system generator specializes each system call *per invocation* to embed it into the particular application code. This facilitates an aggressive folding of parameter values into the code. Therefore, less state needs to be passed in volatile registers or on the stack (rule ❶). We further achieve a pointer-less design by allocating all system objects statically as global data structures, with the help of the generator. In occasions where pointers would be used to select one object out of multiple possible candidates, an array at a constant address with small indices is preferred (rule ❷).

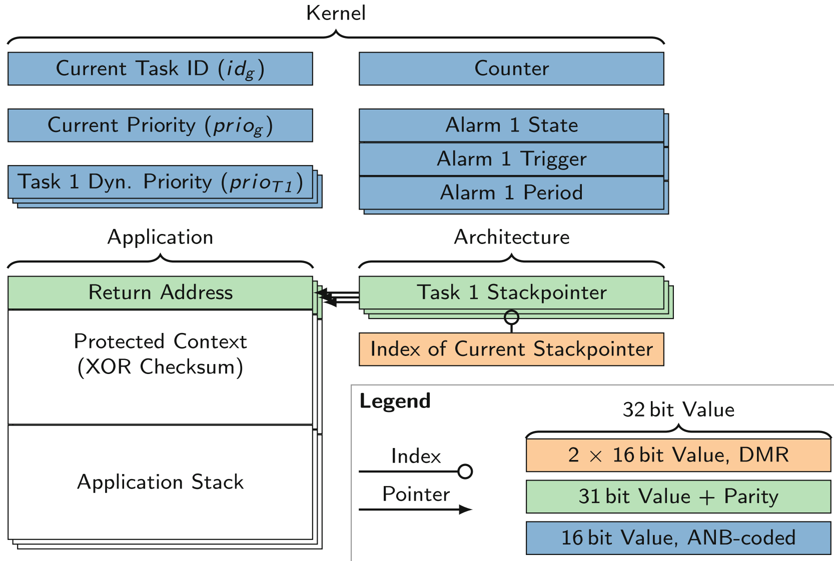

Fig. 2 Overview of the OS data kept in RAM of an example system composed of three tasks and two alarms. Each box represents a 32-bit memory location. All kernel data are hardened using an ANB-Code. The remaining application- and architecture-specific values are safeguarded by dual modular redundancy (DMR) or parity bits

Figure 2 depicts the resulting state of this analysis by the example of a system consisting of three tasks and two alarms: The remaining volatile state variables are subsumed under the blocks *Application*, *Architecture*, and *Kernel*. The architecture-independent minimal *Kernel* state is condensed to two machine words for the current task's priority, its id, and one machine word per task for the task's dynamic priority according to the priority ceiling protocol. Depending on the requirements of the application, the kernel maintains the current state of additional resources: in this case two alarms (three machine words each) and one counter (one machine word). The *Architecture* blocks are related to the dispatching mechanism of the underlying processor. In case of the IA-32, this is reduced to the administration of one stack pointer per task.

The most frequently used (but far less visible) pointers are the stack pointer and the base pointer. Albeit less obvious, they are significant: A corrupted stack pointer influences all local variables, function arguments, and the return address. Here, we eliminated the indirection for local variables by storing them as static variables at fixed, absolute addresses, while keeping isolation in terms of visibility and memory protection (rule ❷). Furthermore, by aggressively inlining the specialized system calls into the application code, we reduce the spilling of parameter values and return addresses onto the vulnerable stack, while keeping the hardware-based spatial isolation (MPU/MMU-based AUTOSAR memory protection) between applications and kernel using inline traps [15] (rule ❶).

3.2 Implementing a Fault-Detecting Operating System

*d*OSEK's *fault-detection* strategies can be split up into two complementary concepts: First, coarse-grained hardware-based fault-detection mechanisms, mainly by means of MPU-based memory and privilege isolation. Second, fine-grained software-based concepts that protect the kernel-internal data/control flows.

Hardware-based isolation by watchdogs and memory protection units (MPUs) are a widely used and a proven dependability measure. Consequently, *d*OSEK integrates the underlying architecture's mechanisms into its system design, leveraging a coarse-grained fault detection between tasks and the kernel. We furthermore employ hardware-based isolation to minimize the set of kernel-writable regions during a system call, which leverages additional error-detection capabilities for faulty memory writes from the kernel space. With our completely generative approach, all necessary MPU configurations can be derived already at compile time and placed in robust read-only memory (ROM).

The execution of the *d*OSEK kernel itself is hardened with a fine-grained arithmetic encoding. All kernel data structures are safeguarded using a variant of an AN-code [23] capable of detecting both data- and control-flow errors. The code provides a constant common key A, allowing to uncover errors when calculating the remainder, and a variable-specific, compile-time constant signature B_n detecting the mix-up of two encoded values as well as the detection of faulty control flows—the ANB-Code:

$$\underbrace{n_{enc}}_{\text{Encoded Value}} = \underbrace{A}_{\text{Key}} \cdot \underbrace{n}_{\text{Value}} + \underbrace{B_n}_{\text{Signature}}$$

A particular feature of arithmetic codes is a set of code-preserving arithmetic operations, which allow for computation with the encoded values. Hence, a continuous sphere of redundancy is spanned, as the corresponding operands remain encoded throughout the entire kernel execution.

In addition to the existing elementary arithmetic operations, *d*OSEK also requires an encoded variant of the mandatory OSEK/AUTOSAR fixed-priority scheduling algorithm [40]: The encoded scheduler is based on a simple prioritized task list. Each task's current dynamic priority is stored at a fixed location (see also Fig. 2), with the lowest possible value, an encoded zero, representing the suspended state. To determine the highest-priority task, the maximum task priority is searched by comparing all task priorities sequentially. Thus, the algorithm's complexity in space and time is linear to the constant number of tasks. Figure 3 shows the basic concept for three tasks: The sequence processes a global tuple of ANB-encoded values storing the current highest-priority task id found so far, and the corresponding priority ($\langle \text{id}_g, \text{prio}_g \rangle$, see Fig. 2). Sequential compare-and-update operations, based on an encoded greater-equal decision on a tuple of values (*ge_tuple*), compare

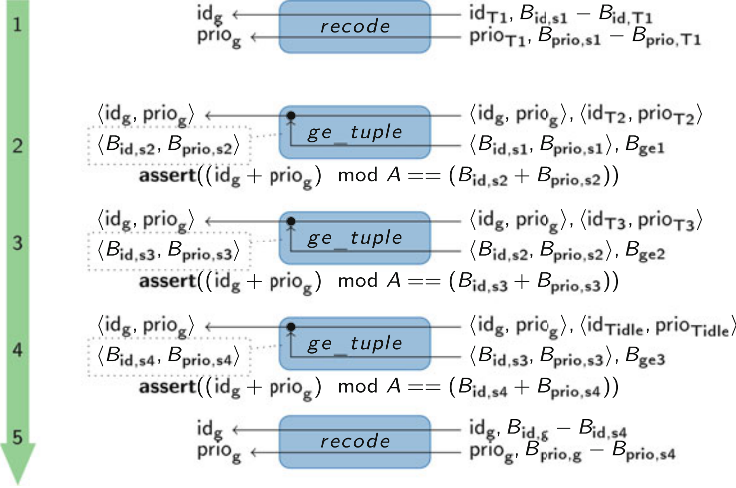

Fig. 3 General sequence of the encoded scheduling operation on the example of three tasks (T1, T2, T3). All operations on signatures B are calculated already at compile time

the tuples' priority value and update the global values, if necessary. The sequence consists of five steps, as shown in Fig. 3:

(1) Initialize $prio_g$ and id_g to the first task.
(2–3) For all further tasks, compare the task's priority to $prio_g$: If greater or equal, update $\langle id_g, prio_g \rangle$.
(4) Repeat the last step for the idle task.
(5) Recode the results to their original signatures.

The idle task priority is constantly bound to an encoded zero that is representing a suspended state. Thus, if all previous tasks are suspended, the last comparison (in step 4) will choose the idle task halting the system until the next interrupt.

Aside from the actual compare-and-update operation on fully encoded values, the *ge_tuple* function additionally integrates control-flow error detection. For each step, all signatures of the input operands ($B_{id,s1..s4}$, $B_{prio,s1..s4}$) and the signature of the operation itself ($B_{ge1..4}$) are merged into the resulting encoded values of the global tuple. Each corresponding signature of a step is then applied in the next operation accordingly. Thus, the dynamic values of the result tuple accumulate the signatures of all preceding operations. As the combination of these compile-time constant signatures is known before runtime, interspersed assertions can validate the correctness of each step. Even after the final signature recode operation (step 5), any control-flow error is still detectable by the dynamic signature. Thus, the correctness of the encoded global tuple can be validated at any point in time. In effect, fault detection is ensured, as all operations are performed on encoded values.

The remaining dynamic state highly depends on the underlying architecture. Regarding the currently implemented IA-32 variant, we were able to reduce this

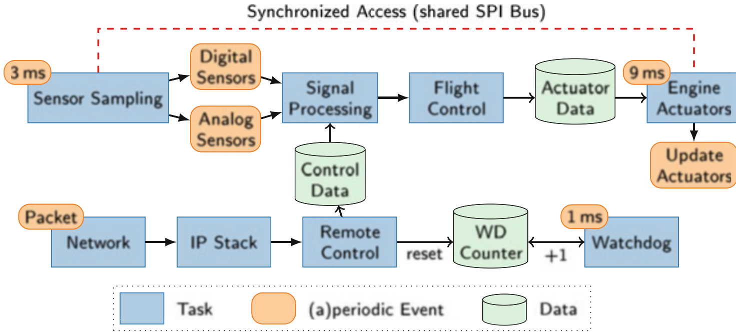

Fig. 4 Simplified representation of the *I4Copter* task and resource constellation used as evaluation scenario

runtime state to an array storing the stack pointers of preempted tasks, and an corresponding index variable, as shown in Fig. 2. The variables are used within each interrupt entry as well as during the actual dispatch operation. As they are not involved in any arithmetic calculations, but only read and written, we can avoid the overhead of the ANB-encoding in these cases and protect them by DMR or parity checks, respectively.

3.3 Evaluation

For comparison, we chose ERIKA Enterprise [21], an industry-grade (i.e., formally certified) open-source implementation of the automotive OSEK standard [40].

The evaluation is based on a realistic system workload scenario considering all essential RTOS services, resembling a real-world safety-critical embedded system in terms of a quadrotor helicopter control application (cf. Fig. 4). The scenario consists of 11 tasks, which are activated either periodically or sporadically by one of four interrupts. Inter-task synchronization is done with OSEK resources and a watchdog task, observing the remote control communication. We evaluated several variants of ERIKA and *d*OSEK, all running the same task set. As ERIKA does not provide support for hardware-based memory protection, we also disabled the MPU in *d*OSEK:

ERIKA Standard version of ERIKA with enabled sanity checks (SVN r3274).

*d*OSEK (unprotected) For the *d*OSEK base version only the indirection avoidance and the generative approach are used against SDCs.

*d*OSEK (FT) The safeguarded kernel execution with encoded operations.

*d*OSEK (FT+ASS) Like FT, but with additional assertions obtained by a flow-sensitive global control-flow analysis [18].

The application flow is augmented with 172 checkpoints. Every RTOS under test executes the application for three hyper periods, while, at the same time a trace of visited checkpoints is recorded. It is the mission of the systems under test to reproduce this sequence, without corrupting the application state. If the sequence *silently* diverges in the presence of faults, we record a silent data corruption.[2] The application state (task stacks) is checked for integrity at each checkpoint. To evaluate the fault containment within the kernel execution, we further recorded an SDC in case of violated integrity. Both SDC detection mechanisms were realized externally by the FAIL* fault-injection framework [47] without influencing the runtime behavior of the systems under test. Since FAIL* has the most mature support for IA-32, we choose this architecture as our evaluation platform. FAIL* provides elaborate fault-space pruning techniques that allow to cover the *entire* space of effective faults, while keeping the total number of experiments manageable. The evaluated fault space includes all *single-bit faults* in the *main memory*, in the *general-purpose registers*, the *stack pointer, and flags registers*, as well as the *instruction pointer*.

3.3.1 Fault-Injection Results

All OS variants differ in code size, runtime, and memory consumption—parameters that directly influence the number of effective injected faults. To directly compare the robustness independent of any other non-functional properties, we concentrate on the resulting absolute SDC count, which represents the number of cases in which the RTOS did not provide the expected behavior. Figure 5 shows, on a logarithmic scale, the resulting SDC counts.

The results show that, compared to ERIKA, the *unprotected* *d*OSEK variant already faces significantly fewer control-flow and register errors. This is caused by the means of constructive fault avoidance, particularly the avoidance of indirections in the generated code. The activation of fault tolerance measures (*d*OSEK *FT*) significantly reduces the number of memory errors, which in total reduces the SDC count compared to ERIKA by **four orders of magnitude**. The remaining SDCs can further be halved by adding static assertions (*d*OSEK *FT+ASS*).

3.3.2 Memory- and Runtime Costs

On the downside, aggressive inlining to avoid indirections, but especially the encoded scheduler and kernel execution path leads to additional runtime and memory costs, which are summarized in Table 1. Compared again to ERIKA, the SDC reduction by four orders of magnitude is paid for with a 4$\times$ increase in runtime and a 20$\times$ increase in code size. As most of the code bloat is caused by the inlining

[2] Faults that lead to a hardware trap are *not* counted as silent, as they are handled by the kernel.

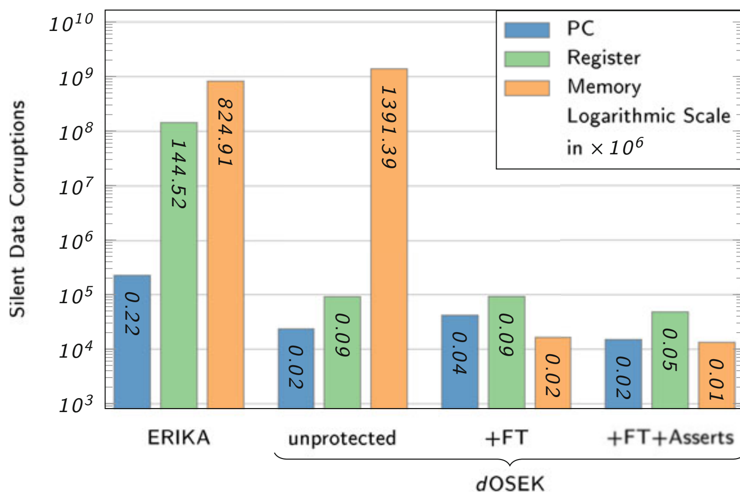

Fig. 5 SDC distribution for the evaluated variants of the *I4Copter* scenario (Fig. 4 on a logarithmic scale; pruned experiments are factored in). The encoded *d*OSEK system achieves an improvement in the SDC count by four orders of magnitude compared to ERIKA (base)

Table 1 Memory- and runtime cost

System	Code size (bytes)	Runtime (instructions)
ERIKA	3782	38,912
*d*OSEK (unprotected)	14,985	29,223
*d*OSEK FT	53,956	110,524
*d*OSEK FT+ASS	71,049	121,583
*d*OSEK FT+ASS+OPT	24,955	90,106

of the encoded scheduler at each call site, we have added a fifth variant (*d*OSEK *FT+ASS+OPT*) that employs further whole-program static optimizations to exclude unnecessary scheduler invocations (see [17] for further details). This version is still $10^4\times$ less vulnerable to SDCs, but reduces the runtime overhead to $2.5\times$ and the code overhead to $8\times$.

4 Modularizing Software-Based Memory Error Detection and Correction

The *d*OSEK approach in the previous section showed the general reliability limits when designing a static OS from scratch, focusing on reliability as a first-class design goal. However, a different and quite common use case is that the requirements entail using a preexisting COTS embedded OS, which is often *dynamic* in the sense that it provides an interface for creating and destroying threads or memory

allocations at runtime. To protect this class of system-software stacks against transient hardware faults—e.g., bit flips—in memory, we propose a software-based memory-error recovery approach that exploits application knowledge about memory accesses, which are analyzed at compile time and hardened by compiler-generated runtime checks.

A central challenge is the *placement* of these runtime checks in the control flow of the software, necessitating an analysis that determines which program instructions access which parts of the memory. In general, this is an undecidable problem for pointer-based programming languages; however, if we assume an *object-oriented programming model*, we can reason that non-public data-structure members are accessed only within member functions of the same class. Consequently, data structures—or, *objects*—can be examined for errors by inserting a runtime check *before* each member-function call.

In this section, we describe our experiences with devising such an object-level error recovery in AspectC++ [51]—an AOP extension to C++,—and applying it to the embedded Configurable operating system (eCos) [37]. Our software-based approach, called Generic Object Protection (GOP), offers the flexibility to choose from an extensible toolbox of error-detecting and error-correcting codes, for example, CRC and Hamming codes.

4.1 Generic Object Protection with AspectC++

Our experience with the embedded operating system eCos shows that OS kernel data structures are highly susceptible to soft errors in main memory [8]. Several kernel data structures, such as the process scheduler, persist during the whole OS uptime, which increases the chance of being hit by a random soft error.

As a countermeasure, OS kernel data structures can contain redundancy, for example, a separated Hamming code [48]. Before an instance of such a data structure—an *object* in object-oriented jargon—is used, the object can be examined for errors. Then, after object usage, the Hamming code can be updated to reflect modifications of the object.

Manually implementing such a protection scheme in an object-oriented programming language is a tedious and error-prone task, because *every* program statement that operates on such an object needs careful manipulation. Therefore, we propose to integrate object checking into existing source code by AOP [32]. Over the last 19 years, we have developed the general-purpose *AspectC++* programming language and compiler [51] that extends C++ by AOP features. A result of the SPP-1500's *DanceOS* project is AspectC++ 2.0, which provides new language features that allow for a completely modular implementation of the sketched object protection scheme—the GOP. In the following, we describe these programming-language features taking the example of GOP.

```
1  aspect GenericObjectProtection {
2    pointcut critical() = "Cyg_Scheduler" || "Cyg_Thread"; // list of types
3    advice critical() : slice class { // generic class extension
4      HammingCode<JoinPoint> code; // template meta-program
5    };
6
7    advice construction(critical()) : after() {
8      tjp->target()->code.update(); // generic advice
9    }
10
11   pointcut trigger_check() = call(member(critical())) ||
12     get(member(critical())) || set(member(critical()));
13   pointcut trigger_update() = call(member(critical())) ||
14     set(member(critical()));
15
16   advice trigger_check() : before() {
17     if (tjp->that() != static_cast<void*>(tjp->target())) {
18       tjp->target()->code.check(); // check callee
19   }}
20   advice trigger_update() : after() {
21     if (tjp->that() != static_cast<void*>(tjp->target())) {
22       tjp->target()->code.update(); // update callee
23   }}
24
25   pointcut any_call() = call("% ...::%(...)");
26   advice any_call() && within(member(critical())) : before() {
27     if (tjp->that() != static_cast<void*>(tjp->target())) {
28       tjp->that()->code.update(); // update caller
29   }}
30   advice any_call() && within(member(critical())) : after() {
31     if (tjp->that() != static_cast<void*>(tjp->target())) {
32       tjp->that()->code.check(); // check caller
33   }}
34 };
```

Fig. 6 A simplified implementation of the GOP mechanism written in AspectC++

4.1.1 Generic Introductions by Compile-Time Introspection

Figure 6 shows the source code for a highly simplified implementation of the GOP. The keyword `aspect` in the first line declares an entity similar to a C++ class that additionally encompasses `pointcut` expressions and pieces of `advice`. A pointcut expression is a reusable alias for names defined in the program. For example, the pointcut `critical()` in line 2 lists two classes, namely `"Cyg_Scheduler"` and `"Cyg_Thread"`, from the eCos kernel. This pointcut is used by the following line that defines `advice` that those two classes get extended by a `slice` introduction, which inserts an additional member into these classes. The inserted member `"code"` is an instance of the template class `HammingCode<typename>`, whose template argument is bound to the built-in type `JoinPoint`. This type is only available in the body of advice code and offers an interface to a compile-time introspection API.

AspectC++'s introspection API [7] provides the programmer with information on the class type that is being extended by the `slice` introduction. We use this information within the template class `HammingCode` to instantiate a generative

C++ template metaprogram [14] that compiles to a tailored Hamming code for each class. In particular, we use the number of existing data members (`MEMBERS`) *prior* to the slice introduction, their types (`Member<I>::Type`) to obtain the size of each member, and a typed pointer (`Member<I>::pointer(T *obj)`) to each data member to compute the actual Hamming code. Furthermore, for classes with inheritance relationships, we recursively iterate over all base classes that are exposed by the introspection API. To simplify the iteration over this API, we implemented a Join-Point Template Library (JPTL) that offers compile-time iterators for each API entry.

4.1.2 Advice for Control Flow and Data Access

Once the Hamming code is introduced into the classes, we need to make sure that the code is checked and updated when such an object is used. At first, the Hamming code needs to be computed *whenever an object of a protected class is instantiated*. The `advice` for `construction` in line 7 implements this requirement: `after` a constructor execution, the `update()` function is invoked on the "`code`" data member. The built-in pointer `tjp->target()` yields the particular object being constructed (`tjp` is an abbreviation for **t**his **j**oin **p**oint).

The lines 11–14 define further pointcuts that describe situations where the objects are used. The pointcut function `member(...)` translates the existing pointcut `critical()` into a set of all data members and member functions belonging to classes matched by `critical()`. Thus, `call(member(critical()))` describes all procedure calls to member functions of the particular classes. Likewise, the pointcut function `get(...)` refers to all program statements that read a member variable, and the other way around, `set(...)` matches all events in the program that write to a particular member variable. The `get/set` pointcut functions are new features of the AspectC++ language that notably allow observing access to data members declared as `public`.

The `advice` in line 16 invokes the `check()` routine on the Hamming-code sub-object based on the `trigger_check()` pointcut, that is, *whenever a member function is called, or a member variable is read or written*. Similarly, the `advice` in line 20 invokes the `update()` function `after` member-function calls or writing to a member variable. Both pieces of advice invoke these routines only if the caller object (`tjp->that()`) and the callee object (`tjp->target()`) are not identical. This is an optimization that avoids unnecessary checking when an already verified object invokes a function on itself.

A call to *any* function is matched by the wild-card expression in line 25. Therewith, the advice definition in line 26 updates the Hamming code *whenever a function call leaves a critical object*, as specified by `within(member(critical()))`, and when the caller object is not identical to the callee object. When the function returns, the Hamming code gets checked by the advice in line 30.

By defining such generic pieces of advice, AspectC++ enables a modular implementation of the GOP mechanism, completely separated from the remaining

source code. More advice definitions exist in the complete GOP implementation, for instance, covering `static` data members, non-blocking synchronization, or virtual-function pointers [8].

4.2 *Implementation and Evaluation*

In the following, we describe the implementation of five concrete EDMs/ERMs based on the GOP mechanism. Subsequently, we demonstrate their configurability on a set of benchmark programs bundled with eCos. We show that the mechanisms can easily be adapted to protect a specific subset of the eCos-kernel data structures, e.g., only the most critical ones. After applying a heuristic that benchmark-specifically chooses this data-structure subset, and protecting the corresponding classes, we present fault injection (FI) experiment results that compare the five EDMs/ERMs. Additionally, we measure their static and dynamic overhead, and draw conclusions on the overall methodology.

4.2.1 EDM/ERM Variants

We implemented the five EDMs and ERMs listed in Table 2 to exemplarily evaluate the GOP mechanism. For instance, a template metaprogram generates an optimal Hamming code tailored for each data structure and we applied a bit-slicing technique [48] to process 32 bits in parallel. Thereby, the Hamming-code implementation can correct multi-bit errors, in particular, all burst errors up to the length of a machine word (32 bits in our case). Besides burst errors, the CRC variants (see Table 2) cover all possible 2-bit and 3-bit errors in objects smaller than 256 MiB by the CRC-32/4 code [11]. Each EDM/ERM variant is implemented as a generic module and can be configured to protect any subset of the existing C++ classes of the target system.

In the following subsections, we refer to the acronyms introduced in Table 2, and term the unprotected version of each benchmark the “Baseline.”

Table 2 EDM/ERM variants, and their effective line counts (determined by `cloc`)

Variant	Description (mechanisms applied *on data member granularity*)	LOC
CRC	CRC-32, using SSE 4.2 instructions (EDM)	163
TMR	Triple modular redundancy: two copies + majority voting (EDM/ERM)	124
CRC+DMR	CRC (EDM) + one copy for error correction (ERM)	210
SUM+DMR	32-Bit two’s complement addition checksum (EDM) + one copy (ERM)	198
Hamming	SW-implemented Hamming code (EDM/ERM), processing 32 bits in parallel	355
Framework	GOP infrastructure, basis for all concrete EDM/ERM implementations	2371

4.2.2 Evaluation Setup

We evaluate the five EDM/ERM variants on eCos 3.0 with a subset of the benchmark and test programs that are bundled with eCos itself, namely those 19 implemented in C++ and using threads (omitting CLOCK1 and CLOCKTRUTH due to their extremely long runtime). More details on the benchmarks can be found in previous work [8]. Because eCos currently does not support x64, all benchmarks are compiled for i386 with the GNU C++ compiler (GCC Debian 4.7.2–5), and eCos is set up with its default configuration.

Using the FAIL* FI framework [47], we simulate a fault model of uniformly distributed transient single-bit flips in data memory, i.e., we consider *all* program runs in which one bit in the data/BSS segments flips at some point in time. Bochs, the IA-32 (x86) emulator back end that FAIL* currently provides, is configured to simulate a modern 2.666 GHz x86 CPU. It simulates the CPU on a behavior level with a simplistic timing model of one instruction per cycle, also lacking a CPU cache hierarchy. Therefore the results obtained from injecting memory errors in this simulator are pessimistic, as we expect a contemporary cache hierarchy would mask some main-memory bit flips.

4.2.3 Optimizing the Generic Object Protection

As described in Sect. 4.1.1, the generic object-protection mechanisms from Table 2 can be configured by specifying the classes to be protected in a pointcut expression. Either a wild-card expression selects all classes automatically, or the pointcut expression lists a subset of classes by name. In the following, we explore the trade-off between the subset of selected classes and the runtime overhead caused by the EDM/ERMs.

We cannot evaluate all possible configurations, since there are exponentially many subsets of eCos-kernel classes—the power set. Instead, we compile each benchmark in *all* configurations that select only a single eCos-kernel class for hardening. For these sets that contain exactly one class each, we measure their simulated runtime, and subsequently order the classes from the least to most runtime overhead individually for each benchmark. This order allows us to *cumulatively* select these classes in the next step: We compile each benchmark again with increasingly more classes being protected (from one to all classes, ordered by runtime). Observing the cumulative runtimes of the respective class selections [8], the benchmarks can be divided into two categories, based on their absolute runtime:

1. **Long runtime (more than ten million cycles):** For any subset of selected classes, the runtime overhead stays negligible. The reason is that the long-running benchmarks spend a significant amount of time in calculations on the application level or contain idle phases.

2. **Short runtime (less than ten million cycles):** The EDM/ERM runtime overhead notably increases with each additional class included in the selections. These benchmarks mainly execute kernel code.

After conducting extensive FI experiments on each of the cumulatively protected programs, it turns out that for our set of benchmarks, the following heuristic yields a good trade-off between runtime and fault tolerance: *We only select a particular class if its protection incurs less than 1 percent runtime overhead*. Using this rule of thumb can massively reduce the efforts spent on choosing a good configuration, as the runtime overhead is easily measurable without running any costly FI experiments. However, in 6 of the initial 19 benchmarks, there are *no* classes that can be protected with less than 1% overhead. Those programs are most resilient without GOP (see Sect. 4.3 for further discussion).

4.2.4 Protection Effectiveness and Overhead

Using this optimization heuristic, we evaluate the EDM/ERM mechanisms described in Table 2. Omitting the aforementioned six benchmarks that our heuristic deems not protectable, Fig. 7 shows FI results from an FI campaign entailing 46 million single experiment runs, using the extrapolated absolute failure count (EAFC) as a comparison metric that is proportional to the unconditional failure probability [46]. The results indicate that the five EDM/ERMs mechanisms are similarly effective in reducing the EAFC, and reduce the failure probability by up to 79% (MBOX1 and THREAD1, protected with CRC) compared to the baseline. The total number of system failures—compared to the baseline without GOP—is reduced by 69.14% (CRC error detection), and, for example, by 68.75% (CRC+DMR error correction). Note that some benchmarks (e.g., EXCEPT1 or MQUEUE1) show very little improvement; we will discuss this phenomenon in Sect. 4.3.

Of course, the increase in system resiliency comes at different static and dynamic costs. With the GOP in place, the static binary sizes (Fig. 8) can grow quite significantly by on average 57% (CRC) to 120% (TMR) (up to 229% in the case of TMR and the KILL benchmark)—showing increases in the same order of magnitude as those observed in the *d*OSEK evaluation (Sect. 3.3.2). Looking closer, the DATA sections of all baseline binaries are negligibly tiny (around 450 bytes) and increase by 5% up to 79%. The BSS sections are significantly larger (in the tens of kilobytes), and vary more between the different benchmarks. They grow more moderately by below 1% up to 15%. In contrast, the code size (TEXT) is even larger in the baseline (23–145 kiB), and the increases vary extremely between the different variants: While CRC increases the code by an average of 114%, CRC+DMR on average adds 204%, SUM+DMR 197%, Hamming 200%, and TMR is the most expensive at an average 241% code-size increase.

But although the static code increase may seem drastic in places, low amounts of code are actually executed at runtime, as we only protected classes that introduce

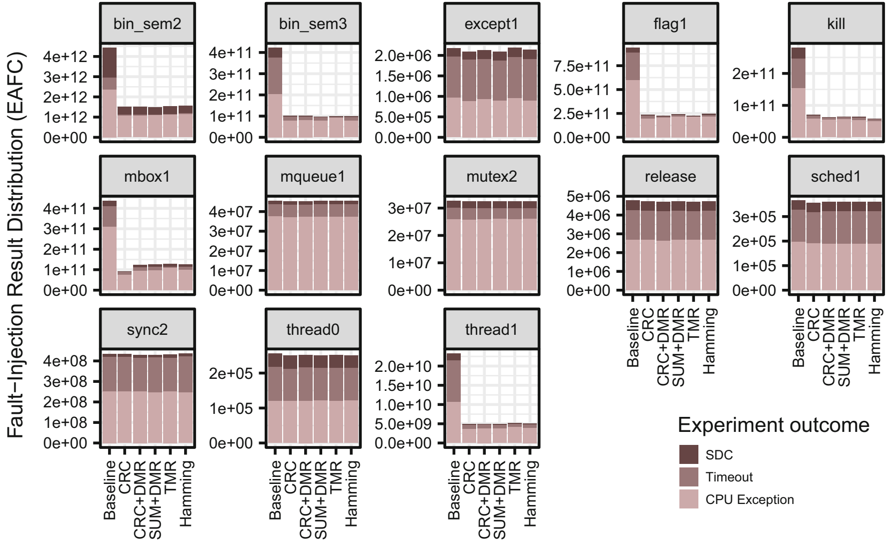

Fig. 7 Protection effectiveness for different EDM/ERM variants

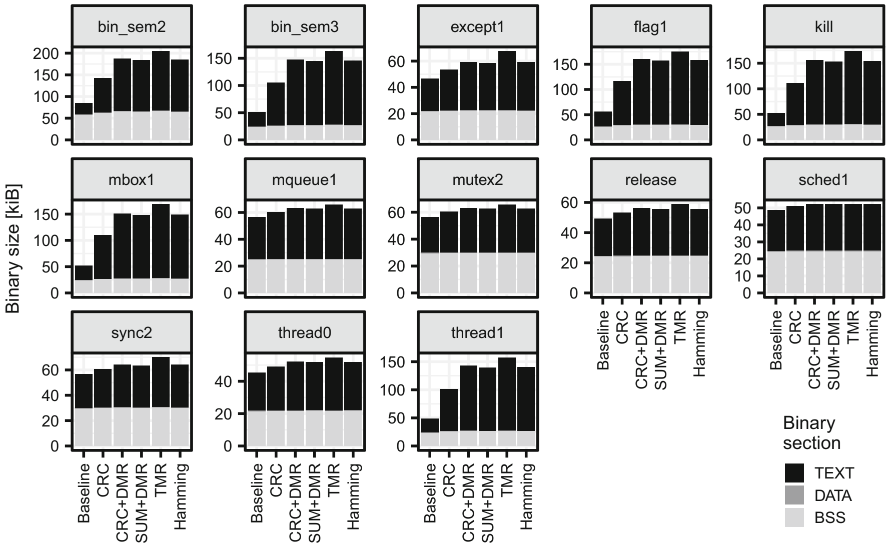

Fig. 8 Static code and data/BSS segment size of the EDM/ERM variants: the code (TEXT) segment grows due to additional CPU instructions, with CRC (detection only) being the most lightweight

less than 1% runtime overhead (see Sect. 4.2.3). Verifying the runtime on real hardware (an Intel Core i7-M620 CPU running at 2.66 GHz), we confirm that the real-world runtime overhead totals at only 0.36% for all variants except for TMR (0.37%). The results indicate that the GOP—when configured appropriately—involves negligible runtime overhead on real hardware.

4.3 Discussion

As software-implemented error detection and correction always introduces a runtime overhead, protected variants naturally run longer than their unprotected counterparts, increasing the chance of being hit by memory bit flips (assuming them to be uniformly distributed). Consequently, there exists a break-even point between, metaphorically, quickly crossing the battlefield without protection (and a high probability that a hit is fatal), and running slower but with heavy armor (and a good probability to survive a hit). The benchmarks in our initial analysis [8] we identified to be *not* effectively protectable with the GOP are on the unfavorable side of this break-even point: The additional attack surface from the runtime and memory overhead outweighs the gains from being protected for all configurations. Also, some benchmarks are just *barely* profiting from the GOP, such as, e.g., EXCEPT1 or MQUEUE1 (see Fig. 7).

A more detailed analysis of what distinguishes these benchmarks from the others reveals that they actually represent the *pathologic worst case* for GOP: Unlike "normal" applications that spend a significant amount of time in calculations on the application level, or waiting for input or events from the outside, this subset of benchmarks *only* executes eCos system calls. This reduces the time frame between an `update()` after the usage of a system object, and the `check()` at the begin of the next usage (cf. Sect. 4.1.2), to a few CPU cycles. The fault resilience gains are minimal, and the increased attack surface all in all increases the fault susceptibility significantly. Nevertheless, we do not believe the kernel-usage behavior of these benchmarks is representative for most real-world applications, and do not expect this issue to invalidate our claim that GOP is a viable solution for error detection and correction in long-living data structures.

For the remaining benchmarks, the analysis in Sect. 4.2.4 shows that the EDM/ERMs mainly differ in their static overhead. CRC is clearly the best choice when detection-only suffices. For error correction, the Hamming code turns out best. The high redundancy of the DMR variants and TMR are overkill—at least unless much more adverse fault models are considered.

5 Conserving Consistent State in Persistent Memory with Software Transactional Memory

Recent advances in persistent memory (PM) enable fast, byte-addressable main memory that maintains its state across power-cycling events. To survive power outages and prevent inconsistent application state, current approaches introduce persistent logs and require expensive cache flushes. In fact, these solutions can cause a performance penalty of up to 10× for write operations on PM. With respect to wear-out effects, and a significantly lower write performance compared to read operations, we identify this as a major flaw that impacts performance and lifetime of PM. Being already persistent, data corruptions in PM cannot be resolved by simply restarting a system. Without countermeasures this limits the usability of PM and poses a high risk of a permanently inconsistent system state.

In this section, we present ***DNV Memory***, a library for PM management. For securing allocated data against power outages, multi-bit faults that bypass hardware protection and even usage violations, *DNV Memory* introduces *reliable transactions*. Additionally, it reduces writes to PM by offloading logging operations to volatile memory, while maintaining *durability on demand* by an early detection of upcoming power failures. Our evaluation shows a median overhead of 6.5%, which is very low considering the ability to repair up to 7 random bit-errors per word. With durability on demand, the performance can be even improved by a factor of up to 3.5 compared to a state-of-the-art approach that enforces durability on each transaction commit.

5.1 System Model

We assume that hybrid system architectures equipped with both, volatile and persistent main memory, will become a commodity. This implicates that the execution state of processes will be composed of volatile and persistent parts.

While Phase Change Memory (PCM) is the most promising PM technology today, PM modules can also be built using resistive random-access memory (RRAM), spin-transfer-torque magnetoresistive random-access memory (STT-MRAM), or even battery-backed DRAM. Thereby, all processes in a system should be able to access PM directly through load and store operations in order to achieve optimal performance.

CPU caches can be used to further speed up access to persistent data. However, in order to survive power failures, cache lines containing data from PM must be flushed and the data must reach the *Durability Domain* of the PM module before the machine shuts down due to a power loss. This requires platform support in form of an asynchronous DRAM refresh (ADR) [49] or a *Flush Hint Address* [1]. Under these premises, we assume that word-level power failure atomicity is reached.

Depending on the used main-memory technology, various effects exist that may cause transient faults as previously outlined. Additionally, PCM and RRAM have

a limited write endurance that lies in the range of 10^6 up to 10^{10} operations [31]. Once worn out, the cell's value can only be read but not modified anymore.

We assume that all static random-access memory (SRAM) cells inside the CPU are guarded by hardware fault tolerance and are sufficiently reliable to ensure correct operation. Of course reliable DRAM supporting hardware error correction code (ECC) exists and PM can be protected by hardware solutions too. However, the common hardware ECC mechanisms only provide single-bit-error correction, double-bit-error detection (SECDED) capabilities, which is not always sufficient [52]. We assume that due to economic reasons not every PM module will support the highest possible dependability standard, leaving a fraction of errors undetected. Some PM modules may even lack any hardware protection. This paves the way for software-based dependability solutions.

5.2 Concepts of DNV Memory

The main goal of our design is to provide the familiar *malloc* interface to application developers for direct access to PM. At the same time, we want data stored in PM to be robust against power failures, transient faults, and usage errors.

Our core API functions (see Table 3(a) and (b)) resemble the interface of `malloc` and `free`. The only additional requirement for making legacy volatile structures persistent with *DNV Memory* is using our API functions and wrapping all persistent memory accesses in atomic blocks (see Table 3(e)).

These atomic blocks provide ACID[3] guarantees for thread safety, and additionally preserve consistency in case of power failures. Furthermore, *DNV Memory* combines software transactional memory (STM) with the allocator to manage

Table 3 Overview of the *DNV Memory* application programming interface (API)

Category	Function	Description	Ref.
Core API	void* dnv_malloc(size_t sz)	Allocates persistent memory like **malloc(3)**	(a)
	void dnv_free(void* ptr)	Releases persistent memory like **free(3)**	(b)
Static Variables	DNV_POD *variable*	Statically places *plain old data* in PM at definition	(c)
	DNV_OBJ *variable*	Statically places the object in PM at definition	(d)
Transactions	__transaction_atomic{...}	Atomic block with ACID guarantees and reliability	(e)

[3] Atomicity, consistency, isolation, durability.

software-based ECC. Every data word that is accessed during a transaction is validated and can be repaired if necessary.

In order to store entry points to persistent data structures that survive process restarts, *DNV Memory* provides the possibility to create static persistent variables (Table 3(c) and (d)). On top of this core functionality, *DNV Memory* introduces the concepts *durability on demand* and *reliable transactions* that are explained in the following.

If a power failure occurs during the update of persistent data structures, the *DNV Memory* might be in an inconsistent state after restart. To prevent this, *DNV Memory* follows the best practices from databases and other PM allocators [12, 54] and wraps operations on PM in atomic blocks. This can be achieved with STM provided by modern compilers or libraries like TinySTM [22]. The transactions must also be applied to the allocator itself, as its internal state must be stored in PM as well.

Different to previous works, *DNV Memory* aims at minimizing write accesses to PM. We store all transaction logs in volatile memory and utilize a power-failure detection to enforce *durability on demand*. When a power outage is imminent, the operating system copies the write-back logs back to PM in order to prevent state inconsistency. Therefore, every thread has to register its volatile memory range for the write-back log at our kernel module, which in turn reserves a PM range for a potential backup copy. After restart, the write-back logs are restored from PM, and every unfinished commit is repeated.

Since durability is actually required only in case of a power failure or process termination, memory fences and cache flushing can be performed on demand. This preserves persistent data inside the CPU cache and consequently reduces writes to PM. Additionally, since memory within a CPU is well protected by hardware, persistent data inside the cache is less susceptible to transient faults and can be accessed faster.

Enforcing durability on demand requires the ability to detect power failures in advance. For embedded devices, the power-outage detection is a part of the *brownout* detection and state of the art [43]. On servers and personal computers, power outages can be detected via the PWR_OK signal according to the ATX power supply unit (PSU) design guide [3]. Although the PWR_OK signal is required to announce a power outage at least 1 ms in advance, much better forecasts can be achieved in practice. For instance, some Intel machines provide a power-failure forecast of up to 33 ms [39]. An even better power-failure detection can be achieved by inspecting the input voltage of the PSU with a simple custom hardware [25]. With this approach, power failures can be detected more than 70 ms in advance, which leaves more than enough time to enforce durability and prevent further modification of persistent data.

Crashes that are not caused by power failures can be handled just like power failures if durability can be secured. For instance, our kernel module is aware of any process using PM that terminates and enforces durability in that case. Crashes in the operating-system kernel can be handled either as part of a kernel-panic procedure, or by utilizing a system like Otherworld [16].

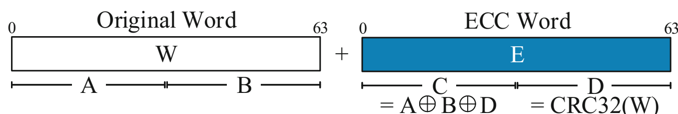

Fig. 9 *DNV Memory* ECC

In order to protect persistent data from corruption, *DNV Memory* reserves additional memory in each allocation that is meant to store ECC data. Afterwards fault tolerance is provided through *reliable transactions*.

As described in the previous section, all accesses to PM should be wrapped by atomic blocks in order to protect persistent data from power failures. These atomic blocks simply wrap all read and write operations in `TM_LOAD` and `TM_STORE` functions provided by the STM library, which in consequence control every word access. In combination with support from the memory allocator, this can be exploited to provide transparent fault tolerance.

Essentially, any ECC can be used to provide fault tolerance in software. For instance, we considered the SECDED Hamming code that is common in hardware protected memory. It protects 64-bit words with additional 8 bits, resulting in a 12.5% memory overhead. However, if implemented in software, the Hamming code would highly impact the performance of the application. Additionally, as already mentioned, we do not think that SECDED is enough to protect persistent data. Consequently, we decided to implement an ECC that provides a high multi-bit error correction with a memory overhead no more than dual modular redundancy. In addition, we want a fast error detection in software by exploiting commonly available hardware support. In general, whenever a *data word W* is written inside an atomic block, an *ECC word E* is created and stored in the additional space that the allocator has reserved. In theory, any fault-tolerant encoding is possible as long as error detection can be conducted in a few CPU cycles.

For *DNV Memory* we combine cyclic redundancy check (CRC) for fast error detection with an error location hint. Thus, we subdivide E into two halves C and D as shown in Fig. 9. The *error detection* half word D is generated with CRC32c ($D = CRC32c(W)$). We chose CRC as hardware support is available on many architectures, including most commodity CPUs. Additionally, with CRC32c—which is supported by *SSE 4.2*,—a Hamming distance of 8 is achieved on a word length of 64 bits [33]. Without further assistance, error correction of up to 3 bits can be achieved by guessing the error location. However, by augmenting the CRC-based error detection with an *error location hint* C, less trials are needed and more bit-errors can be corrected. Inspired by RAID level 5 [42], we subdivide the data word W into two halves A and B and compute C according to Eq. (1).

$$C = A \oplus B \oplus D \tag{1}$$

The data validation takes place during a transaction whenever a word W is read for the first time. At that point, we recompute E' from W and compare its value with E. Normal execution can continue if both values match. Otherwise error correction is initiated.

Since errors can be randomly distributed across W and E, we start the error correction by narrowing the possible locations of errors. Therefore, we compute the *error vector* F via Eq. (2), which indicates the bit position of errors.

$$F = A \oplus B \oplus C \oplus D \tag{2}$$

This information is, however, imprecise, as it is unknown whether the corrupted bit is located in A, B, C, or D. Thus, for f *errors detected by* F, 4^f repair candidates R_i are possible, and are computed via Eq. (3). The *masking vectors* M_a, M_b, M_c, M_d are used to partition F between all four half words.

$$\begin{aligned} R_i &= W_i \| E_i \\ W_i &= A \oplus (F \wedge M_a) \| B \oplus (F \wedge M_b) \\ E_i &= C \oplus (F \wedge M_c) \| D \oplus (F \wedge M_d) \end{aligned} \tag{3}$$

To find the repair candidate R_s that contains the right solution, each R_i needs to be validated by recomputing E_i' from W_i and compare it to E_i. In order to repair all errors, exactly one R_s must be found with matching E_i' and E_i. For instance, if all errors are located in A, the repair candidate using $M_a = F$ and other masking vectors set to zero will be the correct result. Additionally, all combinations need to be considered that have an error at the same bit position in two or all half words, as these errors extinguish each other in C.

Please note that the set of repair candidates may yield more than one solution that can be successfully validated if more than three errors are present. To prevent a false recovery, all repair candidates must be validated for up to n errors. As an optimization step, we estimate n by counting the population in $E \oplus E'$ and limit the result to a maximum of $n = 7$.

To optimize the performance in a cache-aware way, we store the ECC words interleaved with the original words W as presented in Fig. 10. However, this interleaved data layout cannot be accessed correctly outside atomic blocks because the original layout is always expected here. Unfortunately, omitting atomic blocks around PM access is a very common mistake. We encountered such usage errors in every single STAMP benchmark [38], and whenever we ported or wrote persistent applications ourselves. Since the access to PM outside atomic blocks should be prevented to keep data consistent during power failures, we introduce the concept of a *transaction staging (TxStaging) section* as shown in Fig. 10. All memory that is allocated by *DNV Memory* has addresses belonging to the TxStaging section. The same applies to the location of persistent static variables. The TxStaging section is only a reserved virtual address space without any access rights. Consequently, any access to this segment will cause a segmentation fault that is easy to debug.

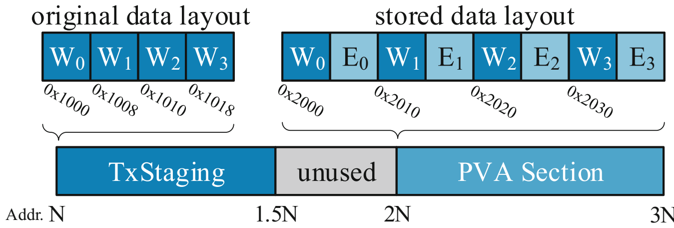

Fig. 10 *DNV Memory* persistent data layout and memory sections

However, inside an atomic block every access to the TxStaging section is intercepted by the STM library and redirected to the persistent virtual address (PVA) section where the actual persistent data is stored. To simplify the address transformation, the PVA section should be located at the address of the TxStaging section multiplied by 2. For instance, assuming the TxStaging section begins at address `0x1000` the PVA section should be placed at `0x2000`. In that case a 32-byte object that is located in the address range from `0x1000` to `0x101f` will be transformed into the address space `0x2000` to `0x203f` as shown in Fig. 10.

5.3 Evaluation

We implemented *DNV Memory* on Linux in the form of a user-space library with a small companion kernel module and a hardware power-failure detector. Our design does not require any changes to the operating-system kernel or the machine itself. All components are pluggable and can be replaced by more extended solutions if needed. All user-space code is written in C++ and compiled with an unmodified GCC 5.4.0. A small linker-script extension provides additional sections like the TxStaging or the PVA section as shown in Fig. 10.

To show the feasibility of *durability on demand*, we artificially introduced power failures and measured the time between the detection of a power failure and the eventual machine shutdown. This period is referred as the shutdown forecast, and the results of 100 experiments are shown in Fig. 11. Additionally, the time of critical tasks in the event of a power failure is shown here. As can be seen, power failures can be detected sufficiently early to conduct all necessary durability measures. Counterintuitively, an idling CPU has a negative impact on the feasibility of the approach because the CPU enters the a power-saving mode with reduced performance. Additionally, less energy is stored within the power supply in the event of a power failure, thus leading to a quicker shutdown.

The performance impact of durability on demand was evaluated with applications from the STAMP benchmark suite [38] and the Memcached key-value store that was retrofitted with transactions. Figure 12 shows for each application the average relative runtime out of 100 measurements together with the 90% quantile that is

Fig. 11 Duration of critical tasks. A Heavy workload is achieved through kernel compilation

Measurement	Workload	Time in ms min	Time in ms max
Stop CPU and Flush Cache	Heavy	2.3	3.3
	Idle	4.4	5.6
Store Write-Back Log	Heavy	3.8	4.8
	Idle	7.4	8.6
Shutdown Forecast	Heavy	34.6	39.4
	Idle	25.2	36.8

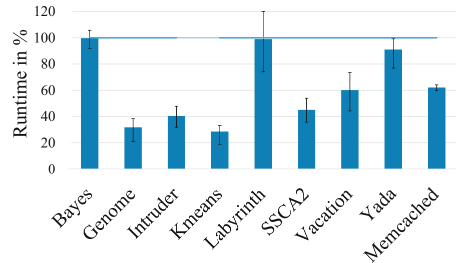

Fig. 12 Application runtime under *durability on demand* in comparison to *durability on commit* (100% baseline)

indicated by the error bars. As the 100% baseline we used the state of the art, which enforces durability on each transaction commit. The results highly correlate with the cache efficiency of the application. For instance, little to no performance impact was achieved for Bayes, Labyrinth, and Yada, which operate on large work sets and show large transactions. If the transactions become large, they do not fit well into the cache and therefore do not benefit from locality, which severely impacts performance. Enforcing durability in this case has a low impact because the overhead from memory barriers and cache flushing becomes negligible. The other benchmarks, however, have moderate to small work sets, therefore a significant performance increase of up to 3.5× can be observed.

To investigate the error detecting and correcting capabilities of *DNV Memory*, we conducted one billion fault-injection experiments, for one to seven-bit errors each. Every fault-injection experiment used a random word and bit-error positions that were randomly distributed over the original data and its corresponding ECC word. Only in the case of 7-bit errors, a small fraction of 0.000012163% fault injections produced ambiguous repair solutions that prevented a correction. In all other cases, including all errors up to 6-bit, a detection and correction was always successful. As can be seen in Fig. 13 the repair time increases exponentially with the number of flipped bits. However, even for correcting seven-bit errors, the mean error-repair time is less than 1.4 ms, which is acceptable considering the low probability of errors. Without any error, the validation only takes 34 ns.

For the performance evaluation of reliable transactions we again used STAMP benchmark applications [38] and Memcached. The bars depicted in Fig. 14 show

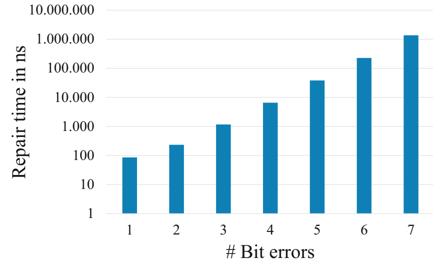

Fig. 13 Time to repair bit errors with reliable transactions

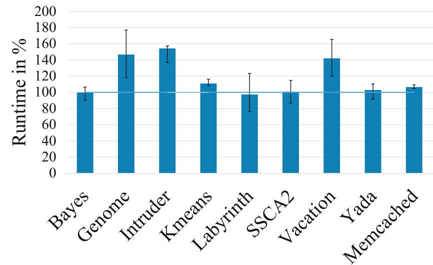

Fig. 14 Performance impact of reliable vs. traditional transactions (100% baseline)

the mean runtime of each benchmark. All values are relative to plain transactional execution (the 100% baseline), and the error bars represent the 95% and the 5% quantile. Over all applications, a median runtime of 106.5% is achieved with reliable transactions. Applications above this median have a workload that is dominated by reads or short transactions, hence the overhead of data verification has a higher impact here. Applications with a balanced or write-driven workload, however, have a higher runtime impact from transactions in general, thus the overhead that comes from reliable transactions is less prevalent. In summary, these results indicate a very acceptable performance impact—especially when considering the error-correcting capabilities of the approach.

5.4 Discussion

DNV Memory provides system support for dependable PM. Unlike previous approaches, *DNV Memory* enforces *durability on demand*, which in turn reduces write operations on PM and therefore improves reliability, lifetime, and performance. For tolerating power failures, *DNV Memory* uses software transactions that also include and secure the allocator itself. Our system even goes one step further and provides fault tolerance via software transactional memory. As our evaluation showed, *DNV Memory* protects data at word granularity, with an ECC word that is capable of detecting and correcting *a random distributed seven-bit error*, which is by far more than common hardware protection offered by server-

class volatile main memory. We also demonstrated that power failures can be detected early, allowing to conduct all necessary cleanup operations.

6 Summary

The work presented in this chapter has gained high visibility in the international research community. It was on the programme of all major conferences in the field and the authors received a number of best paper, best poster, and best dissertation awards, culminating in the renowned Carter Award for Christoph Borchert.

A reason for this success might be the focus on design principles and methods for hardening the operating system—and only the operating system. Most of previous research did not consider the specific properties of this special execution environment, such as different kinds of concurrent control flows, or assumed the reliable availability of underlying system services.

In our work we made a huge effort to design and implement an embedded operating system from scratch with the goal to explore the limits of software-implemented hardware fault tolerance in a reliability-oriented static system design. As a result we were able to reduce the SDC probability by orders of magnitude and found the remaining spots where software is unable to deal with hardware faults.

For existing embedded operating systems we have developed and evaluated Generic Object Protection by means of "dependability aspects," which can harden operating systems at low cost without having to change the source code, and also addressed faults that crash the whole system by means of reliable transactions on persistent memory.

Finally, the authors have developed a fault-injection framework for their evaluation purposes that implements novel methods, which also advanced the state of the art in this domain.

Acknowledgments This work was supported by the German Research Foundation (DFG) under priority program SPP-1500 grants no. KA 3171/2-3, LO 1719/1-3, and SP 968/5-3.

References

1. Advanced Configuration and Power Interface Specification (Version 6.1) (2016). http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf
2. Alexandersson, R., Karlsson, J.: Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In: Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11), pp. 303–314. IEEE Press, Piscataway (2011). https://doi.org/10.1109/DSN.2011.5958244
3. ATX12V Power Supply Design Guide (2005). http://formfactors.org/developer%5Cspecs%5CATX12V_PSDG_2_2_public_br2.pdf

4. Aussagues, C., Chabrol, D., David, V., Roux, D., Willey, N., Tornadre, A., Graniou, M.: PharOS, a multicore OS ready for safety-related automotive systems: results and future prospects. In: Proceedings of the 4th International Conference on Embedded Real Time Software and Systems (ERTS2 '10) (2010)
5. AUTOSAR: Specification of operating system (version 5.1.0). Tech. rep., Automotive Open System Architecture GbR (2013)
6. Bhandari, K., Chakrabarti, D.R., Boehm, H.J.: Makalu: fast recoverable allocation of non-volatile memory. In: Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2016)
7. Borchert, C., Spinczyk, O.: Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In: Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15), pp. 1–7. ACM Press, New York (2015). https://doi.org/10.1145/2818302.2818304
8. Borchert, C., Schirmeier, H., Spinczyk, O.: Generic soft-error detection and correction for concurrent data structures. IEEE Trans. Dependable Secure Comput. **14**(1), 22–36 (2017). https://doi.org/10.1109/TDSC.2015.2427832
9. Borkar, S.Y.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6), 10–16 (2005). https://doi.org/10.1109/MM.2005.110
10. Cassens, B., Martens, A., Kapitza, R.: The neverending runtime: using new technologies for ultra-low power applications with an unlimited runtime. In: International Conference on Embedded Wireless Systems and Networks, NextMote Workshop (EWSN 2016) (2016)
11. Castagnoli, G., Brauer, S., Herrmann, M.: Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. IEEE Trans. Commun. **41**(6), 883–892 (1993). https://doi.org/10.1109/26.231911
12. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In: SIGARCH Computer Architecture News (2011). https://doi.org/10.1145/1961295.1950380
13. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better I/O through byte-addressable, persistent memory. In: Proceedings of the Symposium on Operating Systems Principles (2009)
14. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Boston (2000)
15. Danner, D., Müller, R., Schröder-Preikschat, W., Hofer, W., Lohmann, D.: Safer Sloth: efficient, hardware-tailored memory protection. In: Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '14), pp. 37–47. IEEE Press, Piscataway (2014)
16. Depoutovitch, A., Stumm, M.: "Otherworld" - giving applications a chance to survive OS kernel crashes. In: Proceedings of the European Conference on Computer Systems (EuroSys) (2010)
17. Dietrich, C., Hoffmann, M., Lohmann, D.: Cross-kernel control-flow-graph analysis for event-driven real-time systems. In: Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '15). ACM Press, New York (2015). https://doi.org/10.1145/2670529.2754963
18. Dietrich, C., Hoffmann, M., Lohmann, D.: Global optimization of fixed-priority real-time systems by RTOS-aware control-flow analysis. ACM Trans. Embed. Comput. Syst. **16**, 35:1–35:25 (2017). https://doi.org/10.1145/2950053
19. Döbel, B., Härtig, H.: Who watches the watchmen?—protecting operating system reliability mechanisms. In: International Workshop on Hot Topics in System Dependability (HotDep) (2012)
20. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of the 9th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '14) (2014)
21. ERIKA Enterprise. https://erika.tuxfamily.org. Accessed 29 Sept 2014

22. Felber, P., Fetzer, C., Riegel, T., Marlier, P.: Time-based software transactional memory. IEEE Trans. Parallel Distrib. Syst. **21** (2010). https://doi.org/10.1109/TPDS.2010.49
23. Forin, P.: Vital coded microprocessor principles and application for various transit systems. In: Proceedings of the IFAC IFIP/IFORS Symposium on Control, Computers, Communications in Transportation (CCCT ’89), pp. 79–84 (1989)
24. Goloubeva, O., Rebaudengo, M., Reorda, M.S., Violante, M.: Software-Implemented Hardware Fault Tolerance. Springer, New York (2006). https://doi.org/10.1007/0-387-32937-4
25. Heiser, G., Le Sueur, E., Danis, A., Budzynowski, A., Salomie, T.l., Alonso, G.: RapiLog: reducing system complexity through verification. In: Proceedings of the 8th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys ’13) (2013). https://doi.org/10.1145/2465351.2465383
26. Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., Herkersdorf, A., Kapitza, R., Lohmann, D., Marwedel, P., Platzner, M., Rosenstiel, W., Schlichtmann, U., Spinczyk, O., Tahoori, M., Teich, J., Wehn, N., Wunderlich, H.J.: Design and architectures for dependable embedded systems. In: Dick, R.P., Madsen, J. (eds.) Proceedings of the 9th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS ’11), pp. 69–78. ACM Press (2011). https://doi.org/10.1145/2039370.2039384
27. Hoffmann, M., Borchert, C., Dietrich, C., Schirmeier, H., Kapitza, R., Spinczyk, O., Lohmann, D.: Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In: Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC ’14), pp. 230–237. IEEE Press, Piscataway (2014). https://doi.org/10.1109/ISORC.2014.26
28. Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D., Schröder-Preikschat, W.: A practitioner’s guide to software-based soft-error mitigation using AN-codes. In: Proceedings of the 15th IEEE International Symposium on High Assurance Systems Engineering (HASE ’14), pp. 33–40. IEEE Press, Miami (2014). https://doi.org/10.1109/HASE.2014.14
29. Hoffmann, M., Lukas, F., Dietrich, C., Lohmann, D.: dOSEK: the design and implementation of a dependability-oriented static embedded kernel. In: Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS ’15), pp. 259–270. IEEE Press, Piscataway (2015). https://doi.org/10.1109/RTAS.2015.7108449
30. IEC: IEC 61508 – functional safety of electrical/electronic/programmable electronic safety-related systems. International Electrotechnical Commission, Geneva (1998)
31. Kannan, S., Gavrilovska, A., Schwan, K.: pVM: persistent virtual memory for efficient capacity scaling and object storage. In: Proceedings of the 11th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys ’16), pp. 13:1–13:16. ACM, New York (2016). https://doi.org/10.1145/2901318.2901325
32. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Akşit, M., Matsuoka, S. (eds.) Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ’97). Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer, Berlin (1997). https://doi.org/10.1007/BFb0053381
33. Koopman, P.: 32-Bit cyclic redundancy codes for Internet applications. In: Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’02) (2002). https://doi.org/10.1109/DSN.2002.1028931
34. Li, Y., West, R., Missimer, E.: A virtualized separation kernel for mixed criticality systems. In: Proceedings of the 10th USENIX International Conference on Virtual Execution Environments (VEE ’14), pp. 201–212. ACM Press, New York (2014). https://doi.org/10.1145/2576195.2576206
35. Mariani, R., Fuhrmann, P., Vittorelli, B.: Fault-robust microcontrollers for automotive applications. In: Proceedings of the 12th International On-Line Testing Symposium (IOLTS ’06), 6 pp. IEEE Press, Piscataway (2006). https://doi.org/10.1109/IOLTS.2006.38
36. Martens, A., Scholz, R., Lindow, P., Lehnfeld, N., Kastner, M.A., Kapitza, R.: Dependable non-volatile memory. In: Proceedings of the 11th ACM International Systems and Storage

Conference, SYSTOR ’18, pp. 1–12. ACM Press, New York (2018). https://doi.org/10.1145/3211890.3211898
37. Massa, A.: Embedded Software Development with eCos. Prentice Hall, Upper Saddle River (2002)
38. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: Stamp: Stanford transactional applications for multi-processing. In: Proceedings of the International Symposium on Workload Characterization (IISWC) (2008)
39. Narayanan, D., Hodson, O.: Whole-System Persistence, pp. 401–410. ACM Press, New York (2012). https://doi.org/10.1145/2189750.2151018
40. OSEK/VDX Group: Operating system specification 2.2.3. Tech. rep., OSEK/VDX Group (2005). http://portal.osek-vdx.org/files/pdf/specs/os223.pdf. Accessed 29 Sept 2014
41. Pattabiraman, K., Grover, V., Zorn, B.G.: Samurai: protecting critical data in unsafe languages. In: Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys ’08), pp. 219–232. ACM Press, New York (2008). https://doi.org/10.1145/1352592.1352616
42. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). SIGMOD Rec. **17**(3), 109–116 (1988). https://doi.org/10.1145/971701.50214
43. Philips Semiconductors: AN468: Protecting Microcontrollers against Power Supply Imperfections (2001)
44. Rebaudengo, M., Sonza Reorda, M., Violante, M., Torchiano, M.: A source-to-source compiler for generating dependable software. In: Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation, pp. 33–42. IEEE Press (2001). https://doi.org/10.1109/SCAM.2001.972664
45. Schiffel, U., Schmitt, A., Süßkraut, M., Fetzer, C.: ANB- and ANBDmem-encoding: detecting hardware errors in software. In: Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP ’10), pp. 169–182. Springer, Berlin (2010)
46. Schirmeier, H., Borchert, C., Spinczyk, O.: Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In: Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN ’15), pp. 319–330. IEEE Press, Piscataway (2015). https://doi.org/10.1109/DSN.2015.44
47. Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., Spinczyk, O.: FAIL*: an open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In: Proceedings of the 11th European Dependable Computing Conference (EDCC ’15), pp. 245–255. IEEE Press, Piscataway (2015). https://doi.org/10.1109/EDCC.2015.28
48. Shirvani, P.P., Saxena, N.R., McCluskey, E.J.: Software-implemented EDAC protection against SEUs. IEEE Trans. Reliab. **49**(3), 273–284 (2000). https://doi.org/10.1109/24.914544
49. SNIA NVDIMM Messaging and FAQ (2014)
50. Song, J., Wittrock, J., Parmer, G.: Predictable, efficient system-level fault tolerance in C^3. In: Proceedings of the 34th IEEE International Symposium on Real-Time Systems (RTSS ’13), pp. 21–32. IEEE Press (2013). https://doi.org/10.1109/RTSS.2013.11
51. Spinczyk, O., Lohmann, D.: The design and implementation of AspectC++. Knowl.-Based Syst. **20**(7), 636–651 (2007). Special Issue on Techniques to Produce Intelligent Secure Software. https://doi.org/10.1016/j.knosys.2007.05.004
52. Sridharan, V., DeBardeleben, N., Blanchard, S., Ferreira, K.B., Stearley, J., Shalf, J., Gurumurthi, S.: Memory errors in modern systems: the good, the bad, and the ugly. In: Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’15), pp. 297–310. ACM Press, New York (2015). https://doi.org/10.1145/2694344.2694348
53. Ulbrich, P., Hoffmann, M., Kapitza, R., Lohmann, D., Schröder-Preikschat, W., Schmid, R.: Eliminating single points of failure in software-based redundancy. In: Proceedings of the 9th European Dependable Computing Conference (EDCC ’12), pp. 49–60. IEEE Press, Piscataway (2012). https://doi.org/10.1109/EDCC.2012.21

54. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: lightweight persistent memory. In: SIGARCH Computer Architecture News, vol. 39, pp. 91–104. ACM Press, New York (2011). https://doi.org/10.1145/1961295.1950379
55. Yeh, Y.C.: Triple-triple redundant 777 primary flight computer. In: Proceedings of the IEEE Aerospace Applications Conference, vol. 1, pp. 293–307 (1996). https://doi.org/10.1109/AERO.1996.495891

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.