Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs

Martin Hoffmann[1], Christoph Borchert[2], Christian Dietrich[1], Horst Schirmeier[2], Rüdiger Kapitza[3], Olaf Spinczyk[2], Daniel Lohmann[1]
[1]Friedrich-Alexander-Universität Erlangen-Nürnberg, [2]TU Dortmund, [3]TU Braunschweig,
[1]{hoffmann,dietrich,lohmann}@cs.fau.de, [3]kapitza@ibr.cs.tu-bs.de
[2]{christoph.borchert,horst.schirmeier,olaf.spinczyk}@tu-dortmund.de

***Abstract*—Developers of embedded (real-time) systems can choose from a variety of operating systems. While some embedded operating systems provide very flexible APIs, e.g., a POSIX-compliant interface for run-time management, others have a completely static structure, which is generated at compile time by utilizing detailed application knowledge. A prominent example for the latter class from the domain of automotive operating systems is OSEK/OS and its successor AUTOSAR/OS. As we have shown in previous work, the design of the operating system has a strong impact on its vulnerability for system failure caused by hardware faults. This observation is gaining importance, because there is an ongoing trend towards low-power and low-cost, yet less reliable, hardware. This work quantifies the difference in vulnerability for soft errors in main memory of a flexible (dynamic) operating systems (eCos) and a static system (CiAO), which has an OSEK-compliant structure. We also analyze the additional degree of robustness that is achieved by hardening an operating system with software-based and hardware-based fault-tolerance measures and the corresponding costs. Covering this design space gives developers a better chance for good design decisions with respect to the trade-off between fault tolerance, resource consumption, and interface convenience. Our results indicate that with a combination of hardware- and software-based fault-tolerance measures, silent data corruptions in both operating systems can be reduced to below one percent (compared to eCos). However, the analyzed fault-tolerance mechanisms are expensive for the dynamic system, whereas the statically designed operating system can be hardened at much lower price.**

I. Introduction

Many innovations in recent industrial and automotive applications rest upon elaborate and complex software systems. Their realization demands modern and fast processor architectures at the price of being less and less reliable, due to shrinking structure sizes, increased clock frequencies, and reduced operating voltages [1]. In embedded control systems, the handling of soft errors (i.e., transient bit flips in memory) is becoming mandatory for all SIL3 or SIL4 categorized safety functions [2], [3], [4]. Established solutions stem mostly from the avionics domain and employ extensive hardware redundancy or specifically hardened hardware components [5], [6] – both of which are too costly to be deployed in commodity products.

Software-based redundancy techniques, especially the redundant execution with majority voting in terms of *triple modular redundancy* (TMR), are well-established countermeasures against soft errors on the application level [7]. By combining them with further techniques (such as, arithmetic codes) even the voter as the *single point of failure* (SPOF) can be eliminated [8]. However, all these techniques "work" only under the assumption that the application is running on top of an soft-error–resilient *real-time operating system* (RTOS).

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

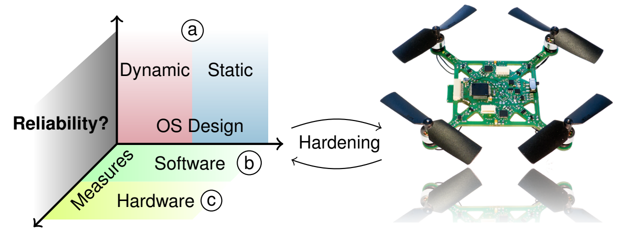


Fig. 1. Investigated dimensions of system-software robustness in a real-world setting: (a) OS design without explicit hardening, (b) software-based dependability measures, (c) hardware-based fault detection.

Many partial solutions have been suggested by the systems community to increase the resilience also of the operating-system kernel against transient and permanent hardware faults. Examples include using watchdog timers [9], graceful degradation with respect to RAM or CPU errors [10], [11], the fine-grained ex-post hardening of *operating system* (OS) data structures [12], or the transfer of essential operating system code to a dedicated reliable computing base [13].

A. About this Paper

In a recent workshop paper [14], we figured out that also the RTOS *design* and *kernel interface* has a strong influence on the sensitivity to soft errors. The design directly influences the resulting mutable kernel state, which has to face possible transient memory faults: In essence, a completely statically allocated kernel state, tailored to the sole needs of a specific application, as suggested by the automotive OSEK/AUTOSAR OS standards [15], [16], provides a significantly higher inherent robustness against soft errors than the dynamic organization of kernel objects implied by a POSIX interface and employed by the vast majority of RTOS kernels. Our experimental fault-injection evaluation with CiAO [17], a completely statically configured implementation of AUTOSAR OS, and eCos [18], which offers a POSIX-compliant dynamic interface, revealed that the dynamic eCos system is affected by four times as many *silent data corruptions* (SDCs) as the static CiAO system. However, our preliminary results in [14] have two shortcomings:

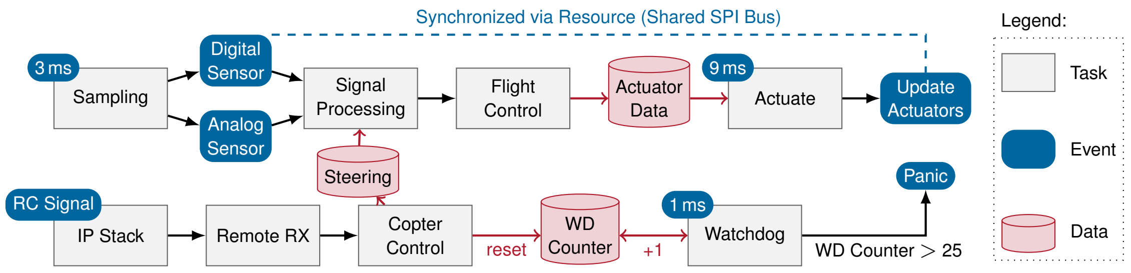


Fig. 2. Simplified representation of the *I4Copter* task and resource constellation, resembling a real-world safety-critical embedded system. The system acts a common workload scenario for all operating system variants under test.

(1) They do not reflect bit flips in the kernel's stack memory, and (2) we did not evaluate any explicit software-based reliability measures that may mitigate these results.

B. Our Contribution

The main contribution of this paper is to extend these considerations by a further dimension in terms of explicit and well-known hardware- and software-based *dependability measures* applied to the different operating system variants (Figure 1). On the example of a real-world safety-critical system (the I4Copter, an autonomous flight vehicle), we quantify the effects of these measures. Here, it is worth to be noted that we explicitly focus on the bare OS execution, rather than considering any (still possible) dependability measures on the application layer, which have been presented elsewhere [8].

In order to reach comprehensive results we conduct exhaustive *fault injection* (FI) campaigns, covering the *entire effective fault space* of single-bit memory bit flips. In a first step, we compare the inherent robustness of two contrasting OS designs, similar to the experiments from [14] (this time including bit flips in the stack space) as a baseline. Then, we extend both kernels with proven software-based fault-detection mechanisms to quantify their effectiveness on the robustness of real-world, safety-critical application scenarios. In a last step, we extend the evaluation on to hardware-based measures. The underlying research questions are: Can the application of proven dependability measures on a dynamic system help to reach the level of inherent robustness of a static system design? And, on the other hand: Can the extension of a condensed static system with additional measures further increase its robustness, considering the increased usage of susceptible memory?

II. Application Scenario and Experiment Setup

The evaluation scenario is based on a common workload that all OS variants have to realize properly. The systems under test are executed with the help of an emulation-based fault-injection framework that allows for deterministic and reproducible experiments.

A. Real-World Application Scenario

In order to achieve realistic system workloads, considering all essential OS features, we chose the task setup of the *I4Copter* [19], a representative for a real-world safety-critical application, as evaluation scenario. The system, as illustrated in Figure 2, consists of three periodically activated tasks and a group of tasks synchronized via resources and events, interacting in a reproducible execution order. In total 14 tasks, 13 events, one resource and 4 alarms are defined. In our evaluation, we evaluate 3 hyperperiods of the *I4Copter* application.

B. Fault Model

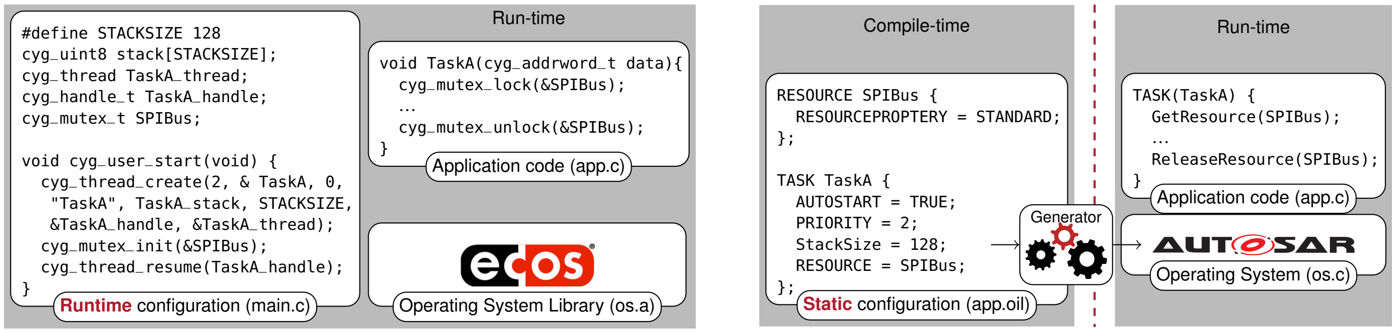
We systematically inject *single bit flips* into all relevant bits of the *kernel state*, rather than choosing memory addresses randomly. In contrast to the fault model of our previous evaluation in [14], here, we additionally include the kernels' stack memory. During an application's runtime, we inject exactly one independent bit flip at a particular time/memory-address coordinate and observe its impact. Afterwards, the application is restarted and the next bit flip is injected at the succeeding point in the fault space. The evaluation scenarios in this paper would emit $3.84 \cdot 10^{13}$ possible faults in total. To reach feasible campaign run times, we concentrate on faults that can actually escalate to errors. For identifying these faults and their injection times, we utilize the fault-space pruning methods offered by the Fail* framework [20]. Based on a x86 CPU simulation, Fail* traces a golden run and schedules fault injections only for memory words that are actually used. In essence, the fault space pruning allows us to minimize the number of ineffective faults, concentrate on actual errors, and still cover the *entire* fault space of potentially effective errors, that is failures. As a result, the overall experiment count is reduced to a feasible amount of 106 million single experiments.

C. Result Categories

The corresponding fault-injection results are grouped into three main categories:

1) Errors not resulting in any observable failure. The task activation order is preserved.
2) Errors detected by hardware traps.
3) Undetected failures, silently influencing the expected behaviour.

As already mentioned, we minimize the number of results belonging to the first category by filtering out faults that can be proved to not leading to any error. Nevertheless there may still occur benign faults, for example if the flipped *bit* never affects the actual system behaviour, as for example a low-order bit that is lost in an upcoming integer division. Regarding the resulting effective FI experiments, we further differentiate between detected and undetected errors. Error detection occurs in terms of hardware traps, that is, for example a division by zero or the execution of illegal instructions. The most severe effects arising from memory faults are the undetected *silent data*



```
#define STACKSIZE 128
cyg_uint8 stack[STACKSIZE];
cyg_thread TaskA_thread;
cyg_handle_t TaskA_handle;
cyg_mutex_t SPIBus;

void cyg_user_start(void) {
  cyg_thread_create(2, & TaskA, 0,
   "TaskA", TaskA_stack, STACKSIZE,
   &TaskA_handle, &TaskA_thread);
  cyg_mutex_init(&SPIBus);
  cyg_thread_resume(TaskA_handle);
}
```

```
void TaskA(cyg_addrword_t data){
  cyg_mutex_lock(&SPIBus);
  ...
  cyg_mutex_unlock(&SPIBus);
}
```

```
RESOURCE SPIBus {
  RESOURCEPROPTERY = STANDARD;
};

TASK TaskA {
  AUTOSTART = TRUE;
  PRIORITY = 2;
  StackSize = 128;
  RESOURCE = SPIBus;
};
```

```
TASK(TaskA) {
  GetResource(SPIBus);
  ...
  ReleaseResource(SPIBus);
}
```

(a) Dynamic OS Design

(b) Static OS Design

Fig. 3. The dynamic *eCos* system initializes kernel objects with static data at runtime. A static *OSEK*/AUTOSAR system deduces its kernel data from a configuration file (`app.oil`) *ahead-of-time*.

corruptions (SDCs) influencing the correct functional behaviour of the operating system, thus leading to failures. Regarding the concrete workload, this functional behaviour is defined by the resulting task execution in terms of completeness, order, and activation times. We also count a terminating and correct task execution order that writes outside of the kernel memory as a SDC.

III. The OS Design Paradigm Dimension

Real-time operating systems provide a common ground of basic features, such as priority-based thread scheduling, management of time, and synchronization primitives to model interdependencies between threads. These functions can be realized in different ways, depending on the design concepts of the particular operating system. For our study, we chose the off-the-shelf operating system *eCos* [18] as a typical representative of a *dynamic* embedded operating system, and a fully *static* implementation of the OSEK specification in terms of the *CiAO* [17] RTOS. Figure 3 contrasts both OS concepts on behalf of a small application with two kernel objects (a task `TaskA` and a mutex `SPIBus`).

A. Dynamic Operating System: eCos (POSIX)

Most RTOS are conceived as **dynamic**: All kernel objects, such as threads, semaphores, or timers, are set up at *run time*; their number is considered as *unbounded*. Dynamic systems typically implement data structures in the form of linked lists and make high use of pointer-based data structures. Especially regarding embedded systems, it is often up to the developer to allocate all necessary resources and hand them over to the operating system, to avoid dynamic memory allocation within the kernel. Moreover, this allocation is typically carried out only once within an initialization phase, while the rest of the system's lifetime, the number of resources remains unchanged. Nevertheless, the system still has to cope with the dynamic allocation strategy, which in fact is a heritage of POSIX semantics, and provide appropriate data structures. Figure 3a illustrates such a dynamic OS design: The *eCos* operating system requires some startup code to initialize all kernel state at run time (`main.c`). These kernel objects are then identified by their run-time address and passed as pointers to the kernel.

eCos (embedded configurable operating system), as the name implies, is designed for configurability, allowing to select and configure various system components like file systems, networking, and many more at compile time. Anyhow at run time *eCos* still manages dynamic data structures in form of data pointers and linked lists. Yet, an interesting aspect is the possibility to configure kernel internals, which have different resilience behaviors regarding to soft errors. Here, the essential system component is the scheduler [12]. *eCos* offers different real-time schedulers to choose from: a *multi-level queue* (MLQ) scheduler and a *bitmap* scheduler.

The multi-level queue scheduler implements an array of pointers to list headers. Each array element represents a priority level, whereby each level comprises a dynamic number of threads within a linked list. This allows for an arbitrary number of threads within each priority level.

The bitmap scheduler variant, on the other hand, realizes the ready "list" in terms of a map, each bit position representing an index, which also corresponds to the priority, in an array of thread pointers. Consequently, the number of threads is fixed and each priority can only be applied to a single thread. Nevertheless the systems interface remains the same and suggests an unbounded number of threads per priority.

B. Static Operating System: CiAO (OSEK)

While the dynamic allocation of kernel objects provides a flexibility that clearly makes sense for uncritical interactive applications, it is hardly ever needed for embedded real-time control systems. An alternative design is suggested by the automotive OSEK/AUTOSAR operating system standards [15], [16], which describe a completely **statically** configured RTOS: All kernel objects have to be declared and configured at *compile time*; their number is implicitly *bounded*. Hence, all objects can be allocated (and to a large degree also initialized) at system generation time. This allows to make high use of static data representations, which can be condensed in tailored arrays offering constant indexing of data. As all needed resources are known at compile time, the data structures can be initialized before run time. Static configuration data can be put into read-only memory and the remaining kernel structures are more robust. For the automotive industry, the driving motivation for such a static design concept has been to keep memory requirements (i.e., hardware costs) as low as possible. However,

TABLE I. EXECUTION ANALYSIS OF THE GOLDEN RUNS.

Golden Run **Memory Usage Statistics**	Unhardened			CRC32 Hardened			CRC32 + Stack protection	
	CiAO-BM	*eCos*-BM	*eCos*-MLQ	*CiAO*-BM	*eCos*-BM	*eCos*-MLQ	*eCos*-BM	*eCos*-MLQ
Text Segment Size (Bytes)	17,056	24,697	26,169	20,207	58,205	65,181	60,125	67,037
Number of Kernel Instructions	1,122	1,072	1,285	1,494	4,183	4,951	4,454	5,218
Executed Kernel Instructions	58,329	80,742	88,456	325,327	1,815,457	2,078,016	1,897,093	2,163,250
Time in Kernel	0.1 %	0.1 %	0.1 %	0.5 %	2.8 %	3.2 %	2.9 %	3.3 %
Executed Instructions per Syscall	100.7	122.9	134.6	561.9	2,763.3	3,162.9	2,887.5	3,292.6
Allocated Kernel Memory (Bytes)	838	1,992	2,308	884	3,880	4,616	3,880	4,616
Accessed Kernel Memory (Bytes)	491	857	949	829	2,267	2,987	2,271	2,991
Kernel Memory Reads per Syscall (Avg. Bytes)	49.5	77.3	83.9	1,706.2	3,182.3	3,509.4	3,226.4	3,563.6
Kernel Memory Writes per Syscall (Avg. Bytes)	39.3	39.1	41.9	76	1,413.9	1,646.1	1,415.3	1,647.6

compared to dynamic systems, this can also lead to less mutable kernel state and fewer error-prone indirections with beneficial impacts on the robustness of the system [14].

Figure 3b exemplifies this design concept with the *CiAO* operating system implementing the *OSEK* specification. Here, all kernel objects have to be specified in a dedicated *OSEK Implementation Language* (OIL) [21] configuration file (`app.oil`), which is evaluated at compile-time by a generator that creates a tailored kernel. Kernel objects are identified and passed to the kernel by compile-time constants. The *CiAO* operating system is, similar to *eCos*, designed for extensive configurability, and additionally aims for fully static configuration and data allocation. With the help of *Aspect Oriented Programming* (AOP) [22], the system can be tailored to the sole requirements of the application, without the need for dynamic data structures within the kernel, as all necessary system information can be described before run time using the OIL description. The automatic tailoring process allocates statically a *task control block* (TCB) for each thread and sorts them into an array ordered by the (static) thread priority. The ready state of each thread is represented by a single bit position in a bitmap scheduler. Since all tasks are defined statically, most configuration data, like the entry point of the task or its preemption behavior, is stored in a read-only data section.

C. System Configurations

The task and resource constellation of the evaluation scenario can be directly mapped to both *eCos* and *CiAO*. Three different system configurations with different gradations of allocation strategies are evaluated:

1) The *eCos* default configuration comprising only dynamic data structures, implementing a MLQ scheduler, alarm and resource lists (*eCos*-MLQ).
2) The same *eCos* default variant, but with a static bitmap scheduler (*eCos*-BM), and
3) a completely static *CiAO* system, also providing a bitmap scheduler (*CiAO*-BM).

We chose these system configurations to examine how the design of the operating system design influences the resilience to soft errors. The default MLQ variant of *eCos* is the most flexible one in terms of tasks per priority. All configurations have been tailored as close a possible to the application scenario. Both *eCos* variants only provide the bare minimum requirements, leaving out all additional drivers and components. Where possible, configurable maximum values of resources have been reduced to a feasible minimum value.

IV. RESULTS OF THE UNHARDENED OS DESIGNS

Before evaluating the FI results of the unhardened OS variants, we first consider the memory usage statistics gathered from the golden runs.

A. Memory and Time Requirements

Table I shows the impact of the differing OS designs on the concrete kernel memory usage. Regarding the unhardened variants, the bare number of kernel instructions in the text section lies at a similar level. However, the total number of executed kernel instructions within the golden run of the *eCos* variants shows an 38.4 % (resp. 51.6 %) increase compared to *CiAO*. The increased amount of instructions per system call further reflects the inherent, more complex nature of *eCos*. Nevertheless, concerning the application run time, the unhardened kernel execution is still neglectable for all variants.

The *eCos* variants allocate at least 2.4 times as much kernel memory as the *CiAO* system. Nevertheless, the actually accessed memory at runtime shows at most an increase of 93.3 percent, both for reading and writing accesses. *CiAO* utilizes more than a half of its previously allocated kernel memory (491 of 838 Bytes), while the *eCos* variants touch less than 41 percent. Here, the impact of the strong tailoring facility of the static *CiAO* system comes to light, condensing the necessary kernel state to the bare requirements of the application.

B. Fault-injection Results

The FI results shown in Table II confirm our findings from [14] to a large extent. Without any explicit dependability measures applied, the detected errors of the unhardened variants can be directly related to CPU hardware traps. Error hook events, signaling a detected error by a software measure, cannot occur.

Again, according to our previous results, the OS design has a considerable influence on the number and explicit manifestation of silent data corruptions. Errors within the condensed kernel state of the *CiAO* system directly lead to differing task activations, while the pointer-based kernel data of the *eCos* variants also suffer from many timeouts after invalid memory reads. At the same time, the partly static *eCos*-BM variant shows an increased inherent robustness, compared to

TABLE II. Overall Fault-injection results showing total numbers also factoring in pruned experiments.

Fault Injection Results (in 10^9 Errors)	Unhardened			CRC32 Hardened			CRC32 + Stack protection	
	CiAO-BM	*eCos*-BM	*eCos*-MLQ	*CiAO*-BM	*eCos*-BM	*eCos*-MLQ	*eCos*-BM	*eCos*-MLQ
No influence	36,476.26	37,888.9	38,000.11	36,139.54	38,276.54	38,412.09	37,732.19	37,864.55
Detected Error	24.24	140.29	171.07	409.91	897.44	1,147.69	1,457.57	1,708.57
Hardware Traps	24.24	140.29	171.07	2.71	167.69	219.23	113.63	159.68
Error Hook	–	–	–	407.20	729.75	928.46	1,343.94	1,548.89
Silent Data Corruption	36.98	181.62	204.06	14.65	36.24	34.76	20.83	21.83
Invalid Memory Write	0.59	1.72	1.74	0.73	1.53	1.44	0.01	0.02
Execution Timeout	16.23	57.57	53.85	9.81	29.02	29.01	20.56	21.55
Differing Task Activation	20.16	122.33	148.46	4.11	5.69	4.32	0.26	0.26

the fully dynamic *eCos*-MLQ implementation. Nevertheless, on an overall basis, the total number of SDCs of the *eCos* systems exceeds the SDCs of *CiAO* by a factor of five. Notably, the additional stack memory injection, that was omitted in [14], has a higher negative influence on the SDCs of the *eCos* variants, than on the *CiAO* system.

V. Software-based Dependability Measures

Dealing with memory faults, we chose the established 32-bit *cyclic redundancy check* (CRC) as error-detection mechanism for both the static *CiAO*, as well as the dynamic *eCos* variants. Both software implementations utilize the `crc32` instruction, implemented by the `SSE4.2` instruction set extension provided by recent `x86` processors.

A. Hardening eCos

The robustness of the *eCos* kernel against memory faults is improved by *Generic Object Protection* [12]. In-memory kernel objects are enriched by a CRC32 error-detecting code, which is allocated together with each instance of a kernel data structure, such as the scheduler and thread objects. The object-oriented software structure of *eCos* allows that the CRC code is checked only *before* function calls to so protected objects, because data access is restricted to member functions only. *After* a member function of a CRC-protected kernel object has executed (and certainly modified its data), the CRC code is updated. By these rules, all memory faults that accumulate while a kernel object is not in-use are detectable.

We implemented the *Generic Object Protection* by means of *Aspect Oriented Programming* (AOP), which allows for a completely modular implementation separated from the *eCos* source code. An aspect compiler, in our case *AspectC++* [23], automatically inserts the protection rules at compile time. Thus, every function call to a protected kernel object can be instrumented with minimal effort from the programmer. Figure 4 shows a simplified[1] `aspect` comprising these protection rules in the language of AspectC++. The `pointcut` expression quantifies all kernel data structures, that is, all C++ classes that match the name pattern `"Cyg_%"`[2], such as `Cyg_Scheduler` and `Cyg_Alarm`. These classes are extended via `slice` introductions by the `crc32` field as well as functions to calculate and check the CRC code for that particular data structure. The aforementioned rules to check/update the CRC code on function calls to kernel objects are specified by the remaining pieces of advice. Thereby, the built-in function `tjp->target()` of AspectC++ yields the instances to be checked/updated.

```
aspect GenericObjectProtection {
  pointcut protectedClasses() = "Cyg_%"; // eCos' kernel classes
  advice protectedClasses() : slice class { // class extension
    unsigned int crc32;
    void check() { MemberIterator<JoinPoint, Check>::exec(this); }
    void update() { MemberIterator<JoinPoint, Update>::exec(this); }
  };
  advice construction(protectedClasses()) || call(protectedClasses())
    : after() { tjp->target()->update(); } // calculate crc32
  advice call(protectedClasses())
    : before() { tjp->target()->check(); } }; // detect errors
```

Fig. 4. Simplified implementation of the *Generic Object Protection*, as applied to all the eCos' kernel data structures (prefixed with `"Cyg_"`), shown in the programming language of AspectC++.

Additionally, we provide a second aspect-oriented mechanism that protects the runtime stacks of preempted threads. When a thread is preempted, a CRC32 code for its used stack memory is stored until the thread is scheduled again, which triggers a check of the stored CRC code. We evaluate hardened variants of the *eCos* kernel both with and without *Preempted Thread Protection*.

These aspect-oriented dependability measures had been carefully chosen because they allow for the *automated* hardening of the roughly 1 million lines of legacy C/C++ code found in eCos – a manually implemented protection would be infeasible.

B. Hardening CiAO

The *Generic Object Protection*, which is used for *eCos*, is not directly applicable to *CiAO* since AspectC++ is currently not capable of recursive code insertion. *CiAO* itself is constructed mostly with aspects [17], and in the current implementation of AspectC++, aspect code cannot be augmented by further aspects. Since this problem disallows automated hardening, a much more coarse-grained approach was applied.

For the hardened *CiAO* all kernel objects are also enriched by CRC error-detecting codes. Every time the kernel is entered, all CRC codes (i.e., the *complete* kernel state) are checked for forged data. After the control flow returns to the application (or into the idle loop), the CRC codes are updated, again for the complete kernel state. This ensures that the data remains protected during the application execution as well as the system idle time.

[1]The complete implementation consists of about 1,800 lines of code.

[2]In AspectC++, `"%"` serves as a wildcard for any number of characters.

In contrast to the *Generic Object Protection*, the data structures are not only vulnerable during the time a specific kernel object is used, but during the whole kernel execution. This approach is also only applicable to small kernel states to keep the constant overhead for entering the kernel low. As a benefit from checking the integrity of all objects every time, faults in seldom used kernel objects can be detected earlier, decreasing the overall fault latency.

C. Memory and Time Requirements of the Hardened Systems

The fine-grained *Generic Object Protection* applied to the *eCos* variants is directly reflected by a considerably increased number of kernel instructions, as shown in Table I. While the *CiAO* variant only emits a CRC check around the entire system call, the dependability aspects intersperse the whole *eCos* kernel execution with loop-unrolled CRC-calculation code. In consequence, the kernel execution time of the hardened *eCos* system has a more considerable impact, compared to the application run time.

The coarse-grained CRC applied to *CiAO* only slightly increases the allocated kernel memory, as only few CRC codes are generated for major data chunks. Accordingly, the number of memory writes per system call, to additionally store these CRC codes, remain near to the level of the unhardened variant.

The fine-grained *Generic Object Protection* applied to *eCos*, on the other hand, roughly doubles the kernel-memory usage. Further, checking more but smaller data chunks, both reads and writes increase considerably. The additional stack protection aspect, covering the used stack space of the respective threads is rather coarse-grained, compared to the kernel object protection. Therefore, the respective overhead remains low in this case.

D. Fault-injection Results

The results of the fault-injection campaign are shown in Table II. The coarse-grained hardening of the *CiAO* system reduces the amount of SDCs by 60.4 percent mainly detecting errors affecting the concrete execution behavior, that is differing task activations and execution timeouts. Clearly, the fine-grained hardening of the *eCos* systems directly manifests in a respectively high amount of detected errors. But, more important, the *Generic Object Protection* also effectively reduces the amount of SDCs by 83 percent. While differing task activations are detected to a large extent, the dynamic *eCos* systems still suffer from a certain amount of undetected execution timeouts. Surprisingly, the more complex *eCos*-MLQ system gains a greater benefit from the *Generic Object Protection* than the *eCos*-BM variant. This is due to the fact, that the pointer-based list implementation of the multi-level queue algorithm handles self-contained thread objects, performing CRC code calculations on each particular access. The bitmap scheduler, on the other hand, maintains a thread pointer array, which is checked as a whole on entry and exit of any scheduler method. Thus, during the method execution, more kernel data is left unprotected for a longer period of time. Here, the object-oriented MLQ design perfectly utilizes the strengths of the *Generic Object Protection*.

The additional stack protection further reduces differing task activation effectively, but the fully-fledged *eCos* variants, providing CRC protection for all kernel objects and stacks, can still not reach the low level of SDCs of the hardened *CiAO* system. Nevertheless, most of the remaining SDCs manifest as execution timeouts, which might be detectable with additional hardware-based measures.

TABLE III. THE SYSTEM DESIGN, SOFTWARE PROTECTION AND USED HARDWARE FEATURES HAVE A MAJOR INFLUENCE ON THE NUMBER OF UNDETECTABLE ERRORS.

Operating System Design →

	% of SDCs	*eCos*-MLQ	*eCos*-BM	*CiAO*-BM	
SW-Protection ↓	w/o Protection	100.00	89.01	18.12	
	with CRC(coarse)	–	–	7.18	
	with CRC (fine)	17.04	17.76	–	
	with CRC (fine), Stack	10.70	10.21	–	
	with Watchdog	73.61	60.79	10.17	Hardware Features ↓
	with MPU	71.31	68.86	16.80	
	with MPU, Watchdog	52.43	48.81	8.88	
SW-Protection ↓		w/ CRC+Stack		w/ CRC	
	with Watchdog	0.14	0.13	2.38	
	with MPU	0.84	0.44	5.70	
	with MPU, Watchdog	0.06	0.06	1.00	

VI. Enabling HW-based Measures

In Table III the influence of the different measures is shown relative to the number of SDCs for the unhardened *eCos*-MLQ variant. The static system design of *CiAO* has only 18.12 percent SDCs, which can be further decreased to 7.18 percent by using a coarse-grained error-correcting CRC code. For the *eCos* systems, hardened with CRC for all kernel objects and stacks, the SDC reduction is equally effective, but does not fall below 10 percent. A more detailed examination of the SDCs for the hardened *eCos* reveals, that, not the differing task activations account for the this lower limit, but execution timeouts (see Table II) caused by faults injected in virtual function pointers. Such pointers are generated by the C++ compiler, as needed for *Generic Object Protection*, and are invisible to the programmer, but could be protected in software as well by *Virtual Function Pointer Protection* [24].

These virtual function pointer failures are also easily caught by dedicated hardware, such as a *memory protection unit* (MPU), which detects roughly 31 percent of the SDCs affecting the unhardened *eCos*. On the other hand, the unhardened *CiAO* only improves by 1.32 percent using the MPU. This is certainly the result of the static system design avoiding indirect memory accesses via pointers as far as possible. However, both unhardened operating systems share a significant amount of timeouts causing SDCs. A hardware watchdog timer detects such timeouts effectively in both systems (see middle part of Table III). Still, the unhardened dynamic *eCos*, using hardware MPU and watchdog timer, fails almost three times more often than the unhardened *CiAO* without any hardware protection.

The combination of these hardware measures together with the CRC software measures is depicted in the lower third of Table III. The CRC-hardened *CiAO* benefits mostly from the watchdog timer, while the MPU reduces the SDCs only by less than two percent. The other way around, the fully protected *eCos* – all kernel objects and stacks covered by CRC – benefits again largely from both the MPU and the watchdog timer. Either hardware measure independently detects the control-flow errors caused by faults in virtual function pointers, which manifest as execution timeouts after an invalid memory read. Finally, considering full hardware and software protection, the SDCs for *CiAO* are reduced by 99.00 percent and by 99.94 percent for the *eCos* variants. Thus, the stack protection applied to *eCos* provides a true advantage over the software hardening applied to *CiAO*.

VII. Discussion

Common *dynamic* system designs clearly profit from their high flexibility and often provide the programmer with the well-known POSIX interface and semantics. But this comfort is won at the expense of robustness against memory faults, while the flexibility remains generally unused for real-time systems. On the other hand, static system designs (like OSEK) make it easy to condense kernel state and to avoid complex operations. This was already discussed earlier [14], where we found the dynamic system to be affected by four times as many SDCs. When also considering the execution stack, as we do in this work, this factor increases even to five.

However, choosing a static system design is not the only tool in the toolbox of software-based measures. Applying active detection through *Generic Object Protection* and protected stack frames results in an resilience improvement against SDCs of 90 percent for *eCos*-MLQ, while not touching the scheduled task set. This brings the dynamic system even above the inherent robustness of an unhardened static design. But this catch-up is bought at the price of 23 times more executed kernel instructions (compared to the unhardened versions) and a doubled kernel-memory consumption. However, most applications spend only very little of their computation time in the kernel, so regarding CPU utilization these numbers are less dramatic in practice: In the I4Copter scenario, the hardened *eCos*-MLQ variant spends less than four percent of the total CPU time inside the kernel. Moreover, the *Generic Object Protection* approach could be applied to the most critical kernel objects only, to trade off reliability for performance. Clearly, the overhead scales with the number of protected kernel objects, but we have not quantified this opportunity, yet. But also the static system approach can be further improved by checking the kernel objects for integrity – at a generally much lower overhead.

Applying additionally hardware-based methods, like an MPU and a watchdog timer, nearly eliminates SDCs in the *eCos* system. Together with the fine-grained CRC aspects, even the coarse-grained checking of the *CiAO* system is outperformed. This is caused by the fact, that the fine-grained protection leaves behind much smaller vulnerability intervals, while with the coarse-grained approach a big interval remains intact for every system call. Errors in the condensed kernel state of *CiAO* during this interval directly affect the system behavior, where errors in the pointer-based dynamic system lead more often to execution timeouts after invalid memory reads. Hence, dynamic systems benefit supremely from hardware-based protection.

With respect to the RTOS robustness impact of the three analysed dimensions (1) OS design, (2) software measures, and (3) hardware measures we conclude:

(1) The **OS design has a major influence** on the robustness of a system. Considering the unhardened systems, the statically designed *CiAO* revealed **4.9 times fewer** SDCs than the dynamic *eCos*-BM.

(2) It is, however, possible to compensate for this initial "design penalty" by **automated software measures**: A *hardened* dynamic system *can* reach (and even outperform) the inherent robustness of a static design. To reach **8.7 times fewer** SDCs in *eCos*-BM, fine-grained – and therefore costly – software measures are necessary. Compared to the unhardened *CiAO*, the hardened *eCos*-BM requires 3.5 times more code bytes, 4.6 times more kernel data (RAM), and 32.5 times more executed kernel instructions.

(3) Much cheaper in terms of memory and run-time (but not necessarily hardware costs) is the application of **hardware-based measures**. In the case of the *eCos* system, applying MPU-based memory protection and in particular a hardware watchdog can significantly decrease the SDCs.

The major thread to validity of these findings is that they are based on single case study only. The flight control of our I4Copter quadrocopter is a real-world safety-critical system, so we consider it as a good representative for these kind of control applications. Nevertheless, the exact quantitative findings will be different for other applications or RTOSs. However, given the order of the numerical differences between the static and the dynamic system, we are convinced that our qualitative findings regarding the robustness and cost impact of (1) OS design, (2) software measures, and (3) hardware measures can be generalized.

VIII. Related Work

There is a limited number of works that investigated how faults effect operating systems.

Koopmann et al. [25] focused on N-version software at the OS level. They examined fifteen *Commercial off-the-shelf* (COTS) POSIX conform operating systems together with various standard C library implementations. In contrast to the presented work, their evaluation concentrates on software inherent robustness with regard to parameter plausibility and API conformance.

Madeira et al. [26] focused on physically induced errors to evaluate the use of COTS systems for space applications. Randomly distributed faults have been injected into both registers and memory of a system running LynxOS, a POSIX conform RTOS. In the course of this paper, we focus on the operating system and evaluate specific hardened variants.

Ferreira and colleagues [27] discussed operating-system vulnerability to memory errors in the context of HPC. Comparable to our results, they conclude that a lightweight operating system might be less vulnerable and easier to harden. They base, however, their observations solely on source-code statistics.

Aidemark et al. [28] compared the effects of transient faults affecting two variants of a custom small-sized embedded real-time kernel. The evaluated variants are a kernel comprising conventional dynamic data structures, namely linked lists, and a kernel using static arrays, additionally extended with spare information enabling fault detection, and triple modular redundant execution of the application tasks. While Aidemark et al. did a substantial evaluation by injecting 26,000 single bit flips into the address and data registers, they did this randomly. Furthermore, the conducted experiments compare hardened static data structures with unhardened list implementations. We, in contrast, compare vanilla and hardened variants of a static and a dynamic RTOS on the base of systematic fault injection during kernel execution that covers the *complete* fault space of $3.84 \cdot 10^{13}$ possible bit flips.

IX. Conclusions

There is an ongoing development towards more and more sophisticated applications in the area of embedded systems with the automotive sector as a trendsetter. This development forces the use of modern and fast processor architectures that are less reliable due to shrinking structure size, increased clock frequencies, and reduced operating voltages than their predecessors. To ensure safe operation of the RTOS, resilient software designs and software-based measures are mandatory.

We have analyzed three dimensions regarding their impact on the robustness of the RTOS: (1) OS design, (2) software measures, and (3) hardware measures. Our results show that the OS design has a major influence on the robustness of a system – the static OSEK-like *CiAO* revealed five times fewer SDCs than the dynamic POSIX-like *eCos*. However, we could also show that it is possible to compensate for this initial "design penality" by further and automated software measures: A hardened dynamic system can reach (and even outperform) the inherent robustness of a static design – at the price of much higher run times and memory consumption. If these software measures are furthermore combined with additional hardware-based protection facilities (MPU, watchdog) the number of SDCs can be reduced to 0.06 percent. However, in general a statically designed RTOS can be hardened at much lower price.

In retrospect, the automotive industry's choice for static systems, originally driven by resource and cost constraints, pays also off tackling the present and future challenge of more and more unreliable hardware.

References

[1] S. Y. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," vol. 25, no. 6, 2005.

[2] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic," in *DSN '02*, Jun. 2002.

[3] R. Mariani, P. Fuhrmann, and B. Vittorelli, "Fault-robust microcontrollers for automotive applications," in *12th On-Line Testing (IOLTS '06)*, 2006.

[4] IEC, *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. Int. ElectroTech. Commission, Dec. 1998.

[5] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," vol. 37, February 1988.

[6] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *1996 IEEE Aerospace Appl. Conf.*, Feb. 1996.

[7] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*, Secaucus, NJ, USA, 2006.

[8] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schröder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *EDCC '12*, May 2012.

[9] F. M. David and R. H. Campbell, "Building a self-healing operating system," in *DASC '07: 3rd IEEE Int. Symp. on Dependable, Autonomic and Secure Computing*. Washington, DC, USA: IEEE Comp. Society, 2007.

[10] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro, "Assessment of the effect of memory page retirement on system RAS against hardware faults," in *DSN '06*, Jun. 2006.

[11] K. Raghavan and V. Kamakoti, "ROSY: recovering processor and memory systems from hard errors," vol. 45, no. 3, Jan. 2012.

[12] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *DSN '13*. IEEE Comp. Society Press, Jun. 2013.

[13] B. Döbel and H. Härtig, ""who watches the watchmen? – protecting operating system reliability mechanisms"," in *Int. W'shop on Hot Topics in Syst. Dependability (HotDep)*, 2012.

[14] M. Hoffmann, C. Dietrich, and D. Lohmann, "Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience," in *2nd GI W'shop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, ser. Lecture Notes in Informatics. German Society of Informatics, 2013.

[15] OSEK/VDX Group, "Operating system specification 2.2.3," OSEK/VDX Group, Tech. Rep., Feb. 2005, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, visited 2011-08-17.

[16] AUTOSAR, "Specification of operating system (version 2.0.1)," Automotive Open Syst. Architecture GbR, Tech. Rep., Jun. 2006.

[17] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk, "CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems," in *2009*, Jun. 2009. [Online]. Available: http://www.usenix.org/event/usenix09/tech/full_papers/lohmann/lohmann.pdf

[18] A. Massa, *Embedded Software Development with eCos*. New Riders, 2002.

[19] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat, "I4Copter: An adaptable and modular quadrotor platform," in *SAC '11*, 2011.

[20] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *, W'shop Proceedings*, vol. 200, Mar. 2012.

[21] OSEK/VDX Group, "OSEK implementation language specification 2.5," OSEK/VDX Group, Tech. Rep., 2004, http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf, visited 2009-09-09.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP '97*, vol. 1241, Jun. 1997.

[23] O. Spinczyk and D. Lohmann, "The design and implementation of AspectC++," *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, vol. 20, no. 7, 2007.

[24] C. Borchert, H. Schirmeier, and O. Spinczyk, "Protecting the dynamic dispatch in C++ by dependability aspects," in *1st GI W'shop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, ser. Lecture Notes in Informatics. German Society of Informatics, Sep. 2012.

[25] P. Koopman and J. DeVale, "Comparing the robustness of posix operating systems," in *FTCS-29*, 1999.

[26] H. Madeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, "Experimental evaluation of a cots system for space applications," in *DSN '02*, Jun. 2002.

[27] K. B. Ferreira, K. Pedretti, R. Brightwell, P. G. Bridges, D. Fiala, and F. Mueller, "Evaluating operating system vulnerability to memory errors," in *2nd Runtime and Operating Systems for Supercomputers (ROSS '12)*. New York, NY, USA: ACM, 2012.

[28] J. Aidemark, P. Folkesson, and J. Karlsson, "Experimental dependability evaluation of the artk68-ft real-time kernel," in *RTCSA '04*, Gothenburg, Sweden, Aug. 2004.