

Failure by Design: Influence of the RTOS Interface on Memory Fault Resilience

Martin Hoffmann, Christian Dietrich, Daniel Lohmann
{hoffmann, dietrich, lohmann}@cs.fau.de
FAU Erlangen-Nürnberg

Abstract: Soft errors are emerging with the ongoing reduction of structure sizes in current and future hardware designs. This problematic is generally tackled by employing fault detection or tolerance measures from an applications' point of view. At the same time, research commences to harden the operating system, often considered as remaining single point of failure. Certainly, these measures can effectively treat the symptoms of hardware faults. However, we argue that the operating system design per se can offer an intrinsic resilience against errors. Dynamic operating system designs, often resembling Unix-like interfaces, are obliged to cope with pointers and list-based data structures to provide the demanded flexibility. In contrast, especially in the domain of embedded systems this flexibility is often not needed. Here, static system designs can be deployed, which allow to avoid error-prone pointer-based memory operations. We believe, that a fully static system design can enhance the resilience against memory errors solely by reduced memory consumption and inherently more robust data structures. This paper studies the influences of memory faults on both, a dynamic and a fully static embedded operating system. Extensive injection campaigns, covering the entire fault space within the kernel data structures, will show that even when applying hardware-based fault detection mechanisms to a dynamic kernel, a static kernel design is still more than 75 percent less susceptible to silent data corruptions.

1 Introduction

New operating systems today tend to be just ways of reimplementing Unix. If they have a novel architecture – and some do – the first thing to build is the Unix emulation layer. Rob Pike: “Systems Software Research is Irrelevant” [Pik]

With the ongoing reduction of structure sizes, future hardware designs for embedded systems will exhibit more performance and parallelism at the price of being less and less reliable. In embedded control systems, the handling of soft errors (i.e., bit flips in memory) is becoming mandatory for all SIL3 or SIL4 categorized safety functions [SKK⁺02, MFV06, IEC98]. Established solutions stem mostly from the avionics domain and employ extensive hardware redundancy or specifically hardened hardware components [MM88, Yeh96] – both of which are too costly to be deployed in commodity products, like cars.

Software-based redundancy techniques, especially the redundant execution in terms of *triple modular redundancy* (TMR), is a well-established countermeasure against soft errors

[GRRV06] on the application level. By combining it with further techniques (such as, arithmetic codes) even the voter as the *single point of failure* (SPOF) can be eliminated [UHK⁺12]. 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). Many partial solutions have been suggested by the OS community to increase the resilience also of the operating-system kernel against transient and permanent hardware faults. Examples include the employment of watchdog timers [DC07], graceful degradation with respect to RAM or CPU errors [SNK⁺11, RK12], the fine-grained ex-post hardening of OS data structures [BSS13], the idea of self-stabilizing (but fairly limited) systems [DY08], or the transfer of essential operating system code to dedicated reliable computing base [DH12].

1.1 About this Paper

We argue, however, that it is, first and foremost, the *design* of the RTOS kernel that determines its sensitivity to soft errors: Design decisions influence the organization and amount of the mutable kernel state and, hence, its “attack surface” with respect to transient faults. In essence, a small and completely statically allocated kernel state (as suggested by the automotive OSEK/AUTOSAR OS standards [OSE05, AUT06]) provides a significantly higher inherent robustness against soft errors than the common list-based dynamic organisation of kernel objects employed by the vast majority of RTOS kernels, which tend to resemble Unix in their concepts and interfaces.

In this paper, we compare both approaches with respect to resilience to memory errors. In particular, we claim the following contributions:

- Complete fault injection campaign, covering the whole effective fault space of our realistic evaluation scenario.
- Detailed examination of the occurred failures in three systems with different object allocation strategies.
- Comparison of fault rates with respect to different hardware-based fault detection and software-based fault avoidance strategies.

2 Dynamic vs. Static Kernel Object Allocation

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*. This is presumably caused by the fact that these systems resemble Unix on the API-level.¹

However, while the dynamic allocation of kernel objects provides a flexibility that clearly makes sense for an interactive multi-user computer, it is hardly ever needed for embedded

¹Examples include FreeRTOS, eCos, RTEMS, QNX, vxWorks, LynxOS, and many more, which are inspired by or even explicitly claim POSIX [IEE95] conformance/compatibility.

	eCos (Unix-like)	OSEK (completely static)
app.c	<pre>void TaskA(cyg_addrword_t data) { cyg_mutex_lock(&SPIBus); ... cyg_mutex_unlock(&SPIBus); }</pre>	<pre>TASK(TaskA) { GetResource(SPIBus); ... ReleaseResource(SPIBus); }</pre>
main.c	<pre>#define STACKSIZE 128 static cyg_uint8 TaskA_stack[STACKSIZE]; static cyg_thread TaskA_thread; static cyg_handle_t TaskA_handle; static 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); }</pre>	
app.oil		<pre>RESOURCE SPIBus { RESOURCEPROPERTY = STANDARD; }; TASK TaskA { AUTOSTART = TRUE; PRIORITY = 2; StackSize = 128; RESOURCE = SPIBus; };</pre>

Figure 1: Static vs. dynamic kernel object allocation on the examples of OSEK and eCos. eCos initializes the system objects with static data at runtime, while OSEK can deduce the system objects from a configuration file (app.oil) ahead-of-time.

real-time control systems! An alternative design is suggested by the automotive OS-EK/AUTOSAR operating system standards [OSE05, AUT06], 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.

In Figure 1 the differences regarding the kernel interface are sketched on behalf of a small application with two kernel objects (a task TaskA and a mutex SPIBus) implemented for (left column) eCos [Mas02], a widely used POSIX-like RTOS for embedded appliances [Tur06] and (right column) OSEK:

While the actual application code (app.c) is pretty similar, the eCos version requires some extra startup code to allocate and initialize all kernel objects at run time (main.cc).

Kernel objects are identified by their run-time address and passed as pointers to the kernel. Additional kernel objects might come and go as the application proceeds.

In the OSEK version, all kernel objects have to be specified in a dedicated *OSEK Implementation Language* (OIL) [OSE04] configuration file (`app.oil`), which is evaluated at compile-time by some generator to create a tailored kernel. Kernel objects are identified and passed to the kernel by compile-time constants.

These contrary design of static and dynamic kernel-object allocation eventually manifests in different implementation variants: 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 and provide appropriate data structures.

Static systems, on the other hand, can 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. Compared to dynamic systems such a static design concept may lead to less error-prone indirections, probably with beneficial impacts on the robustness of the system.

This paper aims to evaluate the influences of memory faults on the behavior of embedded operating systems implementing those contrary designs. We will show that exploitation of static knowledge increases the error resilience of a system.

3 Operating Systems Under Test

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 thread. As already mentioned, 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*, as a typical representative of a dynamic embedded operating system, and *CiAO* [LHSP⁺09], a fully static embedded operating system implementing the OSEK specification.

3.1 *eCos* – Embedded Configurable Operating System

eCos, 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* does manage dynamic data structures in form of data pointers and linked lists. For this study, an interesting aspect is the possibility to configure kernel internals, which have different resilience behaviours regarding to soft-errors. Here, the essential system component is the scheduler. *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 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.

3.2 *CiAO* – CiAO is Aspect Oriented

The *CiAO* operating system is, similar to *eCos*, designed for extensive configurability, and additionally aims for fully static configuration and data allocation. 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. Further resources are handled in a similar manner.

4 Terminology – Fault, Error, Failure

To clarify the terms fault, error and failure, that are consistently used in the next sections, we follow the terminology of the ISO 26262 specification [ISO11], which is based on the definitions by Avizienis [ALRL04]. According to the standard, a *fault* is an “abnormal condition that *can* cause an element or item to fail”. In our study, a fault can be seen as the flipped bit in an arbitrary memory cell of the system. Such a bit flip may not always affect the system behavior, for example, if the affected bit is never read.

The fault only evolves to an *error*, if a “deviation between a computed, observed or measured value or condition, from a theoretically correct value or condition” can be recognized. Therefore, in our study, an error is a bit flip that is actually read by the system,

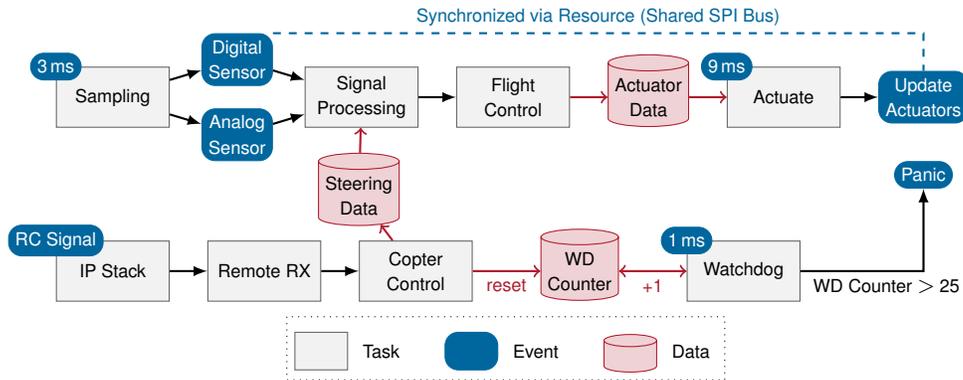


Figure 2: Simplified representation of the *I4Copter* task and resource constellation. The system resembles a real-world safety-critical embedded system.

potentially leading to a system behavior different from the unaffected golden run.

Ultimately, an error leads to a *failure*, when it causes the “termination of the ability of an element, to perform a function as required”. Generally, the required functional behavior of a system is specified on a high abstraction layer, integrating the total behavior of the operating system and the application tasks. Depending on the application, in some cases, a variance in the task execution may still lead to a correct functional behavior.

In our study, inspecting the operating system itself, we specify the correct functional behavior more strictly, namely in the strict task execution order and resource usage, which is exactly determined by our evaluation scenario.

5 Evaluation

In order to achieve realistic system workloads, considering all essential OS features, we chose the task setup of the *I4Copter* [UKH⁺11], a real-world safety-critical embedded system, 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. We evaluate 3 hyperperiods of the *I4Copter* application which consists of 178 task switches, 72 resource accesses and 225 event operations.

5.1 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

Memory Consumption in Bytes	<i>eCos</i>		<i>CiAO</i>
	MLQ	bitmap	bitmap
Text Segment	42,591 (100%)	40,671 (95.5%)	18,636 (43.8%)
Kernel Data	2,976 (100%)	2,668 (89.6%)	744 (25%)

Table 1: Memory consumption of the respective system configurations. The text segment of the *eCos* variants is more than twice the size of the *CiAO* system. Regarding the kernel data, *CiAO* consumes only 25% of RAM compared to *eCos*-MLQ.

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.

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. We will show that, if this flexibility is not needed, the exchange of a system component can improve the dependability without replacing the whole operating system.

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. Table 1 shows the resulting text segment sizes and the memory consumption of the kernel data structures. In our evaluation scenario the static *CiAO* system consumes only 25% of memory for kernel data, compared to the *eCos*-MLQ variant, to realize the same application scenario.

On the basis of overall memory consumption, an increased robustness of the *CiAO* system, having only one-quarter of the OS state memory compared to the *eCos* systems, is to be expected. Especially the strongly pointer-based implementation of the *eCos* kernel raises the expectation of a high amount of invalid memory accesses and traps. On the other hand, errors in the condensed *CiAO* kernel state probably have a more severe effect on the resulting workload execution.

5.2 Fault Model

We inject single bit flips into all relevant bits of the kernel state, rather than choosing memory addresses randomly. More precisely, we concentrate on faults that can actually escalate to errors. For identifying these faults and their injection times, we utilized the fault-space pruning methods offered by the Fail* framework [SHK⁺12]. Fail* single-steps the

program and schedules a fault injection for all memory words that are read in the following instruction. This allows us to minimize the number of ineffective faults, concentrate on actual errors, and cover the *entire* fault space of potentially effective errors.

5.3 Classification of Results

The expected injection results can be grouped into three main categories:

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

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 behavior, like fill-bits used for memory alignment.

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 corruptions influencing the correct functional behavior of the operating system, thus leading to failures. Regarding the concrete workload this functional behavior is evaluated by observing the resulting task execution in terms of completeness, order, and activation times. All systems under test do not make use of hardware memory protection or execution time monitoring in our evaluation. Therefore, timeouts and invalid memory accesses also imply silent data corruptions.

6 Analysis of the Obtained Results

For all three system configurations 952,872 single bit flips were injected with Fail*. All systems were compiled with gcc (4.7.2; -O2) for an x86 hardware target.

6.1 Vulnerability through Memory Accesses

As described in Section 5.2 the pruning method of Fail* traces all memory reads to kernel data structures and maps them directly to injection experiments. In all systems the chosen operating system provides the same service on the same hardware. Therefore a lower number of memory accesses is an indicator for higher robustness, since a smaller fraction

Fault Injection Results	<i>eCos</i>		<i>CiAO</i>
	MLQ	bitmap	bitmap
No Influence	73,722 (17.7%)	77,663 (20.1%)	69,662 (46.3%)
Detected Errors	50,718 (12.2%)	51,777 (13.4%)	8,203 (5.5%)
Hardware Trap	17,163 (4.1%)	16,114 (4.2%)	278 (0.2%)
Jump Outside Text Segment	33,555 (8.1%)	35,663 (9.2%)	7,925 (5.3%)
Silent Data Corruption	291,888 (70.1%)	256,624 (66.5%)	72,615 (48.3%)
Invalid Memory Access	131,211 (31.5%)	95,525 (24.7%)	4,691 (3.1%)
Execution Timeout	46,581 (11.2%)	48,057 (12.4%)	18,570 (12.3%)
Differing Task Activation	114,096 (27.4%)	113,042 (29.3%)	49,354 (32.8%)
Σ Total Fault Injections	416,328 (100%)	386,064 (100%)	150,480 (100%)

Table 2: For the system configurations, described in Section 5.1, the evaluation scenario from Section 5 is examined. The fault injection experiments, that cover the whole fault space, fall into three main categories: benign faults having no influence; detected errors and silent data corruptions.

of the whole fault space can get an effective error. The results in Table 2 show that *CiAO* requires 63.9 percent respectively 61 percent less memory accesses (i.e. fault injections) than the *eCos* variants to realize the same workload, therefore indicating a smaller surface susceptible for errors.



Figure 3: The results in Table 2 fall in different classes of errors, which show different relative ratios in the three systems: In *CiAO* 46.3 percent are, even after the Fail* pruning, without influence; Due to massive pointer usage the *eCos* variants exhibit a big number (> 24%) of invalid memory accesses.

6.2 Vulnerability through Memory Utilization

Besides the absolute results, Table 2 shows the distribution of error classes relative to the total sum of fault injections for each system. These relations are further depicted in Figure 3.

Regarding the *eCos* MLQ system only 17.7 % of the 416,328 injected faults remain benign. On the other hand, a considerable ratio of 31.5 % invalid memory accesses occurred, resulting from the pointer based implementation. Regarding hardware traps and the execution of wrong instructions, both *eCos* systems show a similar distribution. This is an effect of the strong use of pointers, leading to invalid instruction traps, for example when dereferencing wrong function pointers directly, or indirectly via corrupted TCB pointers.

Comparing the MLQ and bitmap implementations, a tendency seems discernible towards an increase of differing task activations, as expected due to the condensed system state, and decrease of invalid memory accesses owing to less pointer handling.

Conversely, almost half of the injected faults show no effect in the *CiAO* system. A further investigation reveals that injecting TCB memory often leads to ineffective faults. This is caused by the fact that in our scenario, tasks are rarely preempted, therefore the values in the TCB are often of no relevance. This explains the higher relative number of ineffective faults in *CiAO* compared to *eCos*: The TCBs occupy the same amount of memory in both systems. Nevertheless, *eCos* needs more additional memory for kernel structures than *CiAO* does.

The statically allocated *CiAO* shows a significantly higher robustness, compared to the dynamic *eCos* variants. *CiAO* needs around 60 % less memory accesses to realize the same workload scenario. The mainly pointer-based implementation of the *eCos* systems, on the other hand, bear a high susceptibility against undetectable invalid memory accesses.

7 Discussion

One threat of validity to our results is clearly the question: Do we compare apples to oranges, when comparing the complex and flexible design of *eCos* to the static design of *CiAO*? In our chosen realistic scenario (Section 5), both systems provide the same task interaction. The flexibility of *eCos* is not needed, but enlarges the effective fault space. The higher number of memory accesses is caused by the system design, rather than the application on top.

Also a bitmap scheduler, which is less flexible than a MLQ scheduler, does not decrease the attack surface of *eCos* significantly (7.3 % less memory accesses). The real problem is located in the dynamic Unix-like system interface, that imposes a need for dynamic system objects. In embedded control systems the involved objects are well known at compile time, but *eCos* enforces you to neglect this knowledge in favor of a Unix-like interface. *CiAO*, on the other hand, can exploit this knowledge more profound, and therefore reveals a significantly smaller “attack surface” for hardware faults.

This static benefit is most obvious in the placement of constant configuration data, like

		Better Design →		
Better Hardware ↓	% of Silent Data Corruptions	<i>eCos</i> MLQ	<i>eCos</i> bitmap	<i>CiAO</i> bitmap
	w/o Hardware Protection	100	87.9	24.9
	with Watchdog	84	71.5	18.5
	with MPU	55	55.2	23.3
	with MPU, Watchdog	39.1	38.7	16.9

Table 3: Hardware features as well as the system design have a major influence on the number of undetectable errors. Switching to a static design reduces the rate to 24.9 percent. Adding an MPU and a watchdog implies a decrease to 39.1 percent. When applied both improvements the rate goes down to 16.9 percent.

the entry function of a task. As *eCos* initializes system objects at runtime, they have to be placed in RAM, where *CiAO* places them in constant read-only memory and the compiler can even optimize out code paths that cannot be reached with the chosen configuration.

Diving more into the different errors that arose by the injected bit flips, there are two basic cases: errors detected by the hardware and undetected silent data corruptions. For the first case, the system can react in some way, for example restart the whole system. The errors that cannot be detected by the system itself are much more dangerous. In our scenario we used a simple CPU model without hardware-based isolation support for space and time. Therefore invalid memory accesses, execution timeouts and differing task activations cannot be detected.

Table 3 shows how the error resilience changes, when we enhance our hardware to detect more error conditions. Adding a hardware watchdog timer detects situations where the execution takes too long or the system hangs in an endless loop. Adding a *memory protection unit* (MPU) would detect all memory accesses outside the data and text sections. We observe that the pointer-based *eCos* benefits much more (-45 %) from an MPU than the array-based *CiAO* (-1.6 %) does. It is also notable that, when both hardware improvements are brought into the *eCos* variants (39.1 %; 38.7 %) the error resilience of *CiAO* is still better (24.9 %) without additional hardware support.

So why do we use dynamic Unix-like system interfaces for static embedded systems, even if they use more memory and are more fragile in the presence of hardware faults? In our opinion there are two reasons: Unix-like general-purpose operating systems is what we teach our students; and everybody wants to use the well-known. Also for a static, OSEK-like design, more external code generation tools are needed, where an *eCos*-like design can mostly be implemented within the system language itself. We conclude that we need to make better tools for exploiting static knowledge, but also have to pass on the knowledge of static system design.

8 Related Work

The effects of faults affecting operating systems have been investigated from various points of view in the last years.

Koopmann et al. [KD99] examined fifteen *Commercial off-the-shelf* (COTS) POSIX conform operating systems together with various standard C library implementations to evaluate the potential use of N-version software on OS level. To evaluate the robustness of the different POSIX implementations, combinations of invalid and exceptional function parameters have been generated and injected into the systems under test. This evaluation concentrates more on software inherent robustness with regard to parameter plausibility and API conformance, rather than physically induced errors into the hardware.

Madeira et al. [MSM⁺02] 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. The test scenarios comprised synthetic and real world task sets. Injections into the OS kernel execution showed a fairly high amount (60%) of ineffective faults, mainly caused due to the random fault distribution. The remaining effective errors manifested as overall system crashes (29.5%) or abnormal task terminations. Only about 1.3 percent of the injected faults into the OS kernel led to silent data corruptions, mainly corrupting the execution of file access operations. Considering the faults injected while executing the application tasks, the experiments showed, besides a similar high amount of ineffective faults, mainly abnormal task terminations caused by corrupted system call parameters.

Aidemark et al. [AFK04] 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. The evaluation consisted of around 26,000 single bit flips into the address and data registers of a Motorola 68340 microcontroller. The injection points were randomly distributed during the execution of two tasks and several specific operating system calls. The results show around 80 percent ineffective faults mainly owing to the random fault distribution. The remaining effective errors divide into hardware detected faults or CPU crashes, and wrong calculation results. In contrast to this paper, it is worth to be noted, that the space of effective errors is not covered widely in the applied experiments. Furthermore, the experiments compare hardened static data structures with unhardened list implementations.

Borchert et al. [BSS13] applied an ex post hardening of the dynamic data structures of an existing embedded operating system. With the help of aspect oriented programming, different dependability measures could be applied and evaluated without touching the original code base. Extensive fault injection campaigns were conducted, covering the entire fault space of effective memory errors within the scheduler and thread data structures of the *eCos* operating system. With the cost of a significant code bloat, the applied measures significantly reduced the amount of silent data corruptions.

9 Conclusion

In essence, a small and completely statically tailored operating system provides a significantly better resilience to soft errors than the common list-based dynamic organization of kernel objects employed by the vast majority of real-time operating systems, which tend to resemble Unix in their concepts and interfaces. Especially in embedded systems the component organization in the application is static, therefore there is no need for a dynamic interface. Conversely, as complete fault injection campaigns show, a pointer-based dynamic operating system has a major drawback with regard to dependability to the static system design. Choosing the static approach even outperforms hardware-based fault detection mechanisms applied to a dynamic system, by avoiding error-prone memory operations. Based on the static knowledge more elaborate fault detection/tolerance methods can be developed to completely protect the remaining small kernel state.

References

- [AFK04] Joakim Aidemark, Peter Folkesson, and Johann Karlsson. Experimental Dependability Evaluation of the Artk68-FT Real-time Kernel. In *Proceedings of the International Conference on Real-Time and Embedded Computer Systems and Applications*, August 2004.
- [ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, August 2004.
- [AUT06] AUTOSAR. Specification of Operating System (Version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.
- [BSS13] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-based Memory Error Detection and Correction for Operating System Data Structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Computer Society Press, June 2013.
- [DC07] Francis M. David and Roy H. Campbell. Building a Self-Healing Operating System. In *DASC '07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 3–10, Washington, DC, USA, 2007. IEEE Computer Society.
- [DH12] Björn Döbel and Hermann Härtig. “Who watches the watchmen? – Protecting Operating System Reliability Mechanisms”. In *International Workshop on Hot Topics in System Dependability (HotDep)*, 2012.
- [DY08] Shlomi Dolev and Reuven Yagel. Towards Self-Stabilizing Operating Systems. *IEEE Transactions on Software Engineering*, 34(4):564–576, 2008.
- [GRRV06] Olga Goloubeva, Maurizia Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer-Verlag, Secaucus, NJ, USA, 2006.
- [IEC98] IEC. *IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, December 1998.

- [IEE95] IEEE. *IEEE 1003.1c-1995: Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language)*. 1995.
- [ISO11] ISO - International Organization for Standardization, Geneva, Switzerland. Road vehicles – Functional safety, 2011.
- [KD99] Philip Koopman and John DeVale. Comparing the Robustness of POSIX Operating Systems. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing, FTCS '99*, pages 30–, Washington, DC, USA, 1999. IEEE Computer Society.
- [LHSP⁺09] Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat, Jochen Streicher, and Olaf Spinczyk. CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association.
- [Mas02] Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002.
- [MFV06] R. Mariani, P. Fuhrmann, and B. Vittorelli. Fault-robust microcontrollers for automotive applications. In *Proceedings of the 12th International Symposium on On-Line Testing (IOLTS '06)*, page 6 pp., Washington, DC, USA, 2006. IEEE Computer Society Press.
- [MM88] A. Mahmood and E. J. McCluskey. Concurrent Error Detection Using Watchdog Processors-A Survey. *IEEE Transactions on Computers*, 37:160–174, February 1988.
- [MSM⁺02] Henrique Madeira, Raphael R. Some, F. Moreira, D. Costa, and David Rennels. Experimental Evaluation of a COTS System for Space Applications. In *in Proc. Int. Conference on Dependable Systems and Networks*, pages 325–330. IEEE CS Press, 2002.
- [OSE04] OSEK/VDX Group. OSEK Implementation Language Specification 2.5. Technical report, OSEK/VDX Group, 2004. <http://portal.osek-idx.org/files/pdf/specs/oil25.pdf>, visited 2009-09-09.
- [OSE05] OSEK/VDX Group. Operating System Specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. <http://portal.osek-idx.org/files/pdf/specs/os223.pdf>, visited 2011-08-17.
- [Pik] Rob Pike. Systems Software Research is Irrelevant. Talk.
- [RK12] Karthik Raghavan and V. Kamakoti. ROSY: recovering processor and memory systems from hard errors. *ACM SIGOPS Operating Systems Review*, 45(3):82–84, January 2012.
- [SHK⁺12] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. Gesellschaft für Informatik, March 2012.
- [SKK⁺02] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proceedings of the 32nd International Conference on Dependable Systems and Networks (DSN '02)*, pages 389–398, Washington, DC, USA, June 2002. IEEE Computer Society Press.
- [SNK⁺11] Horst Schirmeier, Jens Neuhalfen, Ingo Korb, Olaf Spinczyk, and Michael Engel. RAMpage: Graceful Degradation Management for Memory Errors in Commodity Linux Servers. In *Proceedings of the 17th International Symposium on Dependable Computing (PRDC '11)*, Washington, DC, USA, December 2011. IEEE Computer Society Press.

- [Tur06] Jim Turley. Operating Systems on the Rise. *embedded.com*, June 2006. <http://www.embedded.com/columns/showArticle.jhtml?articleID=187203732>.
- [UHK⁺12] Peter Ulbrich, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, Wolfgang Schröder-Preikschat, and Reiner Schmid. Eliminating Single Points of Failure in Software-Based Redundancy. In *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*, pages 49–60, Washington, DC, USA, May 2012. IEEE Computer Society Press.
- [UKH⁺11] Peter Ulbrich, Rüdiger Kapitza, Christian Harkort, Reiner Schmid, and Wolfgang Schröder-Preikschat. I4Copter: An Adaptable and Modular Quadrotor Platform. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*, pages 380–396, New York, NY, USA, 2011. ACM Press.
- [Yeh96] Y.C. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the 1996 IEEE Aerospace Applications Conference*, pages 293–307, Washington, DC, USA, February 1996. IEEE Computer Society Press.