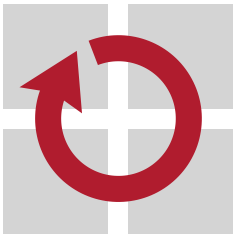
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 4

Verteilte Systeme und Betriebssysteme



Florian Lukas

Design and Implementation of a Soft-error Resilient OSEK Real-time Operating System

Masterarbeit im Fach Informatik

19. Mai 2014

Please cite as:
Florian Lukas, “Design and Implementation of a Soft-error Resilient OSEK Real-time Operating System,” Master’s Thesis, University of Erlangen, Dept. of Computer Science, May 2014.

Friedrich-Alexander-Universität Erlangen-Nürnberg

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Verteilte Systeme und Betriebssysteme

Martensstr. 1 · 91058 Erlangen · Germany

www4.cs.fau.de

Design and Implementation of a Soft-error Resilient OSEK Real-time Operating System

Masterarbeit im Fach Informatik

vorgelegt von

Florian Lukas

geb. am 22. Juni 1988
in Weiden i. d. Opf.

angefertigt am

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dipl. Ing. Martin Hoffmann, Dr. Daniel Lohmann**
Betreuender Hochschullehrer:	**Prof. Dr.-Ing. Wolfgang Schröder-Preikschat**
Beginn der Arbeit:	**1. Dezember 2013**
Abgabe der Arbeit:	**19. Mai 2014**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Florian Lukas)
Erlangen, 19. Mai 2014

Abstract

Advances in manufacturing processes steadily reduce the structure size of computer chips, which improves their processing power. However, this smaller size also increases the likelihood of *soft-errors*, such as the flip of a single bit in memory. If such an error remains undetected, it can result in invalid system behavior. This is unacceptable for *dependable* systems which have strict reliability and safety requirements.

To detect data and control flow errors, *arithmetic coding* is an effective method but incurs significant computational overhead. Another widely-used solution is *triple modular redundancy*, which replicates data and computation followed by a voter to determine a correct result. However, these methods are usually implemented in the application and depend on the correct execution of the underlying operating system.

The operating system itself should not only support the detection of errors, but also introduce as few possible failures as possible. Using a-priori knowledge about the system and application, a static operating system can be generated with fewer indirections, which increase the susceptibility to soft-errors.

In this work, the *d*OSEK real-time operating system is presented, which leverages a static design and arithmetic coding to avoid and detect soft-errors while implementing the widely-used OSEK specification.

*d*OSEK is evaluated using extensive fault injection campaigns, comparing the robustness and overhead of multiple system variants for several benchmarks. Additionally, *d*OSEK is evaluated against *ERIKA Enterprise*, an open-source OSEK real-time operating system.

Kurzfassung

Die Rechenleistung von Computersystemen steigt stetig durch die fortschreitende Miniaturisierung der Chip-Strukturen. Jedoch führen die schrumpfenden Strukturgrößen auch zu einer stärkeren Anfälligkeit für *transiente Fehler*, wie das spontane Invertieren eines Speicherbits. Bleibt ein solcher Fehler unerkannt, kann er zu unzulässigem Verhalten des Gesamtsystems führen. Dies ist inakzeptabel für Systeme mit Zuverlässigkeits- und Sicherheitsanforderungen.

Für die Erkennung von Fehlern im Daten- und Kontrollfluss ist *Arithmetische Kodierung* ein mächtiges Verfahren, jedoch mit signifikantem Overhead. Eine weit verbreitete Methode ist *dreifach redundante Ausführung*, bei der Daten und Operationen mehrfach gespeichert und ausgeführt werden. Ein anschließendes Votum ermittelt das korrekte Ergebnis. Diese Techniken sind jedoch üblicherweise auf Anwendungsebene implementiert und setzen die Korrektheit des darunterliegenden Betriebssystems voraus.

Das Betriebssystem selbst sollte nicht nur die Fehlererkennung unterstützen, sondern auch selbst so wenig potentielle Fehler wie möglich verursachen. Mit apriori Wissen über das System und die Anwendung kann ein statisches System mit weniger Indirektionen generiert werden, welche die Anfälligkeit für Fehler erhöhen.

In dieser Arbeit wird das *d*OSEK Echtzeitbetriebssystem vorgestellt, das ein statisches Design und arithmetische Kodierung einsetzt um transiente Fehler zu vermeiden und zu erkennen. Dabei implementiert *d*OSEK den verbreiteten OSEK Standard.

*d*OSEK wird anhand von extensiven Fehlerinjektionskampagnen evaluiert, wobei die Zuverlässigkeit und der Overhead verschiedener Systemvarianten anhand mehrerer Benchmarks verglichen werden. Außerdem wird *d*OSEK gegen *ERIKA Enterprise* evaluiert, einem Open-Source OSEK Echtzeitbetriebssystem.

Contents

Chapter 1

Introduction

Shrinking structure sizes and the resulting performance increases have enabled the introduction of computer systems in many safety-critical applications. This chapter first lists the requirements of such dependable software systems and discusses the problem of soft-errors caused by shrinking chip structures. Before presenting the structure of this thesis, the reliable computing base as well as the design goals for the soft-error resilient real-time operating system presented in this work are explained.

1.1 Dependable Software Systems

The usage of software systems embedded in automotive or other safety-critical scenarios requires these systems to be *dependable*. A dependable software system can be trusted to perform its tasks correctly through all of the following aspects: [1,2]

The system must be *available* by being ready to perform its service at all times. This service must also be *reliable*, that is, delivered continuously without interruptions. Under all circumstances, the *safety* of the users and the environment must be guaranteed by preventing catastrophic malfunctions. Safety relies on the *integrity* of the system, the correctness of all internal state of the system. Finally, the system must also be *maintainable* to allow repairs and adjustments after the initial deployment.

Especially in the area of safety, standards like ISO 26262 [3] and IEC 61508 [4] have become mandatory for manufacturers.

Real-time operating systems are usually used to build such dependable embedded systems by ensuring timing and other non-functional constraints which allow the system to react in a predicable manner.

1.2 Soft-Errors

The widespread usage of embedded software systems would not have been possible without the continued miniaturization of chip structures, which increases their computing power. At the same time, operating voltages are reduced as well, for thermal and power consumption reasons.

However, smaller structure sizes and lower voltages increase the susceptibility for *transient hardware errors* as the absolute electrical charge representing one stored bit decreases as well. High-energy particles (natural background radiation or cosmic rays) striking chip structures are more likely to change information as the relative difference in charges decreases. This can lead to non-permanent errors called *soft errors* or *single event upsets* [5].

From a software point of view, such soft errors manifest as bit-flips at unpredictable times and memory locations. Through the addition of redundancy, like the usage of error-detecting codes, these errors can be detected in hardware as well as in software. However, if such an error remains undetected when checks are skipped or impossible, the integrity of the entire system is violated. This *Silent Data Corruption (SDC)* is a serious dependability threat as these errors endanger both the safety and reliability of the system.

In case enough redundancy is available, a system may also be able to *mitigate* the error and continue correct operation, resulting in *fault tolerance* [6].

1.3 Reliable Computing Base

Dependable software systems must take measures against transient hardware faults to prevent failures of the system.

The usage of hardened or redundant hardware is a simple yet expensive solution to prevent or mask transient faults. Examples are hardware techniques like Error-Correcting Code (ECC) memory and lockstep processors.

Alternatively, errors can be detected by software-based measures after a transient fault has occurred. Redundant execution in time and space or error-correcting codes can be used to check for invalid system states. Error detection in software can be used to selectively protect only critical data on a system which also processes non-critical data (*mixed-criticality systems*).

Such methods depend on a *Reliable Computing Base (RCB)* [7], which must not be affected by transient fault to ensure the correctness the software error detection itself. Most software-based methods developed so far assume that the operating system is part of the RCB.

1.4 Design Goals

The goal of this work is to design, implement and evaluate *d*OSEK, a real-time operating system which is robust against soft-errors.

- To serve as a reliable computing base, transient errors in the operating system itself must be prevented from corrupting the application. This means *the application needs to be protected from the operating system* as the correct execution of the operating system is worthless once the integrity of the application is violated.

- Application tasks must be *isolated* from each other temporally and spatially to stop the propagation of errors between different parts of the application.

- To increase robustness by *avoiding* potential errors, the system state must be minimized. For this reason, a static system design, based on the OSEK[1] specification [8], is chosen to prevent errors caused by dynamic system structures.

- Another source of potential errors are indirections such as pointers or loops, which must be reduced as much as possible. Indirections introduced by the specific hardware architecture must be considered as well.

- Remaining potential errors must be *detected* through the addition of redundancy. Arithmetic coding is used for operating system structures while specialized measures must be developed for architecture-dependent structures.

- Once a error has occurred, the system aborts to a error handling routine before the integrity of the application is endangered (*fail-stop*). Potential recovery procedures (besides a full system reset) are out of scope of this work.

- Although architecture-specific optimizations are necessary, the general system design must be extensible and not limited to a single architecture.

- For the detection of functional errors (software bugs), automatic tests should cover all system components. These tests are also used for continuous integration during the development process.

- To evaluate the achieved robustness, fault injection campaigns with complete fault space coverage are performed. In these campaigns, single-bit flips in RAM, CPU registers and the instruction pointer are simulated.

[1] OSEK: Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen

1.5 Thesis Structure

The next chapter introduces the fundamentals of real-time operating systems, the OSEK specification and arithmetic coding, followed by a discussion of related work. Chapter 3 presents the general architecture and design choices of *d*OSEK, including an encoded scheduler and measures to reduce indirections. The implementation of these error-avoidance and error-detection techniques for the Intel i386 architecture is described in Chapter 4. In Chapter 5, the robustness and overhead of *d*OSEK is evaluated using fault injection campaigns for several benchmarks and compared against the open-source *ERIKA Enterprise* operating system. Finally, a conclusion and possible future work is given in Chapter 6.

Chapter 2

Fundamentals

In this chapter, the general components of a real-time operating system are presented, followed by a description of how the OSEK specification maps to these concepts.

Then the fundamentals of arithmetic coding are introduced, followed by a discussion of related work.

2.1 Operating System Components

This section gives a short overview of the core components of a Real-Time Operating System (RTOS).

2.1.1 Scheduler

A primary task of the operating system is to manage the control flow between several tasks. The decision which task has to run next is made by the *scheduler*, according to a specific mechanism, like task priorities or deadlines. If running tasks can be interrupted temporarily for later resumption they are called *preemptable*.

Time-triggered systems use periodic or deadline-specific timers to schedule tasks after a predefined amount of time has passed. In event-triggered systems, priorities are used to switch to the highest-priority task at every point of rescheduling.

2.1.2 Dispatcher

The *dispatcher* is the architecture-specific part of the kernel which actually transfers control flow between tasks after the scheduler has reached a decision.

In general, the context of the previous task has to be saved (unless it cannot resume) and the context of the next task has to be restored or initialized. A task context consists of the general-purpose CPU registers as well as the stack and instruction pointers, amongst other architecture-specific parts.

2.1.3 Interrupts

Interrupts can be used to switch CPU execution to a special handler when certain events occur. This prevents the overhead of polling, but requires the operating system to ensure the interrupted task is not influenced by this asynchronous control flow change.

In case multiple interrupts arrive at the same time, they are usually handled on the basis of an assigned priority. Depending on the implementation and hardware, interrupt handlers may be preemptable as well.

2.1.4 Isolation Mechanisms

Additionally, to prevent applications from interfering with each other in unintended ways, isolation mechanisms are provided by the operating system. These mechanisms are used to detect errors and other invalid operations in tasks, and possibly in the kernel as well.

Many operating systems support spatial isolation using a Memory Protection Unit (MPU), which prevents tasks from accessing the data of other tasks. This requires data to be assigned to the respective tasks, which can usually be derived from the application.

Especially in real-time operating systems, temporal isolation is used as well, which ensures every task gets a certain, guaranteed amount of execution time, even when other tasks are malfunctioning. This is usually achieved by deadline monitoring and/or watchdog timers.

Privilege isolation is used to prevent applications from performing critical operations which can endanger the execution of the entire system, like reconfiguring hardware devices or the MPU.

2.2 OSEK Specification

The OSEK specification describes a model and Application Programming Interface (API) for static, event-triggered real-time operating systems. It originates from the requirements of the automotive sector and specifies several conformance classes, distinguished by the respective implemented features.

The specification describes a simple system model with a small, precise API while lower-level (architecture-specific) design choices are left to the implementation.

OSEK systems are static, as system components cannot be added or removed at run-time. Only few parameters, like dynamic priorities or alarm limits are changeable at run-time.

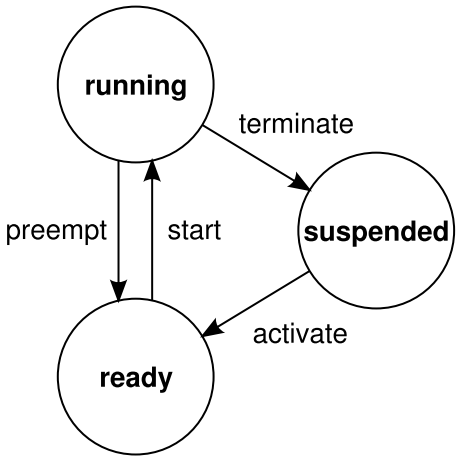


Figure 2.1 – OSEK task states and possible transitions. [8]

2.2.1 Tasks

Application functionality is mapped to OSEK tasks, which are assigned a static priority. While the set of tasks is fixed at run-time, the individual tasks can change between several states as shown in Figure 2.1:

- At most one task can be *running*, i. e. currently executing on the CPU, until it is preempted by a higher-priority task or terminates.
- *Ready* tasks can be started or resumed once no higher-priority tasks are active.
- *Suspended* tasks are not ready to run until they are activated.

While the static priority of a task cannot be changed, the effective priority of a task can change when resources are used, as described in Section 2.2.3.

Tasks can be activated (set to be *ready*) by API calls from other tasks, interrupts or alarms. The implementation is free to specify whether tasks may be activated multiple times and how those requests are queued.

Tasks can be set to be *non-preemptive*, in which case they continue running even when higher priority tasks are ready until they explicitly invoke the scheduler.

2.2.2 Interrupts

The OSEK specification defines two classes of interrupts:

ISR1 interrupts are not allowed to use operating system services, thus not requiring synchronization with the operating system. Consequently, the OSEK specification imposes no further restrictions on them.

ISR2 interrupts are similar to tasks and can use almost all operating system services. This requires the synchronization of system structures to prevent corruption from concurrent accesses.

Like tasks, interrupts are assigned a static priority, which decides the order of processing. While not enforced, it is recommended that ISR2 priorities are higher than any task priority and ISR1 priorities higher than any ISR2 priority.

Since the activation of interrupts is hardware-dependent the OSEK specification does not require interrupt handlers to be preemptable.

2.2.3 Resources

For synchronization between tasks, a static set of *resources* can be defined for the system. Tasks can acquire and release a resource before and after a critical section, respectively. This prevents other tasks from entering a critical section guarded by this resource. Multiple resources can be held at the same time, but must be released in opposite order of acquisition (LIFO).

The scheduler handles resources using the OSEK *priority ceiling protocol*, which assigns each resource the maximum priority of all tasks which can possibly acquire it. While a task holds a resource, it is scheduled as if its priority is equal to the highest acquired resource priority. This is the only form of dynamic task priority allowed in an OSEK system.

2.2.4 Alarms

To react on repeating events like timer ticks, the OSEK specification defines counters and alarms.

Counters are unsigned integer values, increased (asynchronously) by external events. Typically, at least one counter is linked to the system timer to enable periodic activities.

In order to react on counter changes, *alarms* can be defined statically. Each alarm specifies a trigger value (for one specific counter) and starts a task or callback routine once this value is reached, similar to an interrupt. Optionally, alarms can then automatically be reset using a given period. While the set of alarms is defined statically, tasks can change their parameters at run-time.

2.2.5 Events

The OSEK specification also defines conformance classes for an *extended system*, which allows tasks to synchronize by waiting for a set of *events*, which can be set by other tasks. This adds a new *waiting* task state and API functions.

When using events, the usual run-to-completion semantics of OSEK tasks can be replaced by tasks waiting for events in a (possibly infinite) loop.

2.3 Arithmetic Coding

Arithmetic coding is a method of encoding integer values to detect errors while still allowing some arithmetic operations to be performed on these transformed values. [9]

2.3.1 AN coding

The simplest form of arithmetic coding is AN encoding, named after the transformation equation:

$$v_c = A \cdot n \tag{2.1}$$

where n is the original value, v_c the encoded result and A a multiplicative factor.

The upper half of Figure 2.2 illustrates this transformation. Spreading the original values by the factor A introduces gaps with invalid values (gray dots). If an encoded value is corrupted the chance to result in another valid encoded value is small, allowing these errors to be detected. The amount of detectable errors depends heavily on the choice of A, which needs to be chosen with respect to a good hamming distance between encoded values. [10] [11]

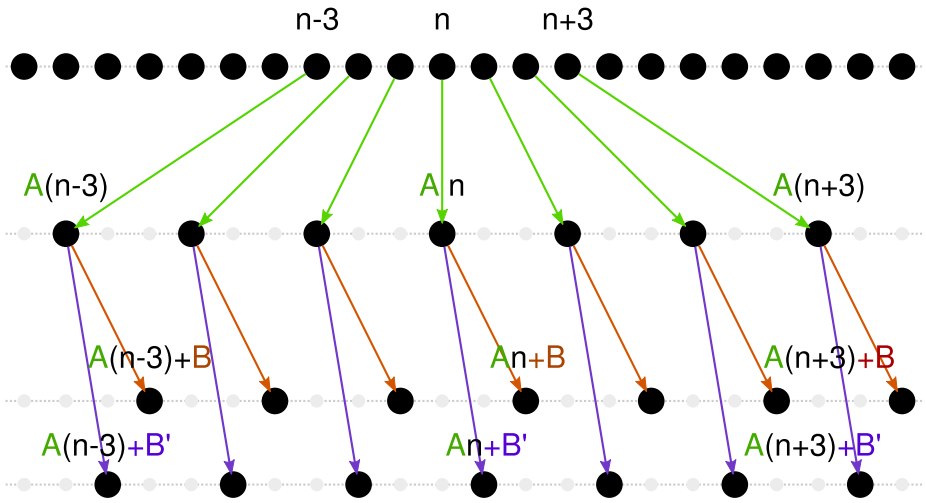


Figure 2.2 – Transformation steps from unencoded values (top row) to AN-encoded values (second row) and two different ANB-encoded values (last two rows). Black dots represent valid, gray dots invalid codewords. Figure is not to scale, as values for A usually are much larger.

The encoded values are bigger than the original values because of the multiplication, but unlike other error detection methods, actual data and error-correction information is mixed and cannot be separated.

Basic arithmetic operations can be performed on the values without decoding them, thus preventing them from being unprotected during the calculation. Equation (2.2) shows how addition and subtraction directly results in a new encoded value while the encoded multiplication in Equation (2.3) requires the result to be corrected by division of A.

$$(A \cdot n_1) \pm (A \cdot n_2) = A \cdot (n_1 \pm n_2) \tag{2.2}$$

$$(A \cdot n_1) \cdot (A \cdot n_2) = A \cdot (A \cdot (n_1 \cdot n_2)) \tag{2.3}$$

2.3.2 ANB coding

Errors might not only corrupt data values, but also change the control flow or pointers, resulting in the use of a *different yet valid* encoded value. To detect such "confusions", AN**B** coding adds a unique, static signature B to each encoded variable:

$$v_c = A \cdot n + B \tag{2.4}$$

This transformation is illustrated in the lower half of Figure 2.2. Valid codewords with signature B are invalid if expected to have signature B' and vice versa.

The encoded arithmetic operations now result in a value with a new, compound signature:

$$(A \cdot n_1 + B_1) \pm (A \cdot n_2 + B_2) = A \cdot (n_1 \pm n_2) + (B_1 \pm B_2) \tag{2.5}$$

This enables implicit data flow checks, as the final result of a combination of operations differs from the expected compound signature if any of the operands is mistaken.

2.3.3 ANBD coding

Errors may also cause *lost updates*, in which the memory write of an encoded value is silently lost. If an old value of the same encoded variable was stored at this location, this old value cannot be distinguished from a correct new value. To solve this problem, ANB**D** coding adds a dynamic timestamp D to the signature:

$$v_c = A \cdot n + B + D \tag{2.6}$$

This value is usually incremented at each write and the current, correct value must be known for all accesses. The encoded operations have to respect the timestamp value as well, which is handled like a modified B, but must be calculated at run-time. In real systems, it can be challenging to keep track of the various timestamps in a robust and efficient manner.

2.3.4 Control Flow Checking

Arithmetic coding can also be used to detect control flow errors implicitly by checking the signatures of computed results.

If a linear block of code (without branches) performs a sequence of encoded operations on ANB or ANBD values, the data and control flow is implicitly checked through the static signatures (*B*s). This is caused by the fact that every partial result receives a static signature *B* derived from the source operands' signatures. If the control flow deviates from the original (which is used to determine all signatures) the final values at the end of this block are calculated from a different sequence of operations. Thus, the results in case of an error are encoded using a different combination of signatures. Control flow errors can then be detected by checking if

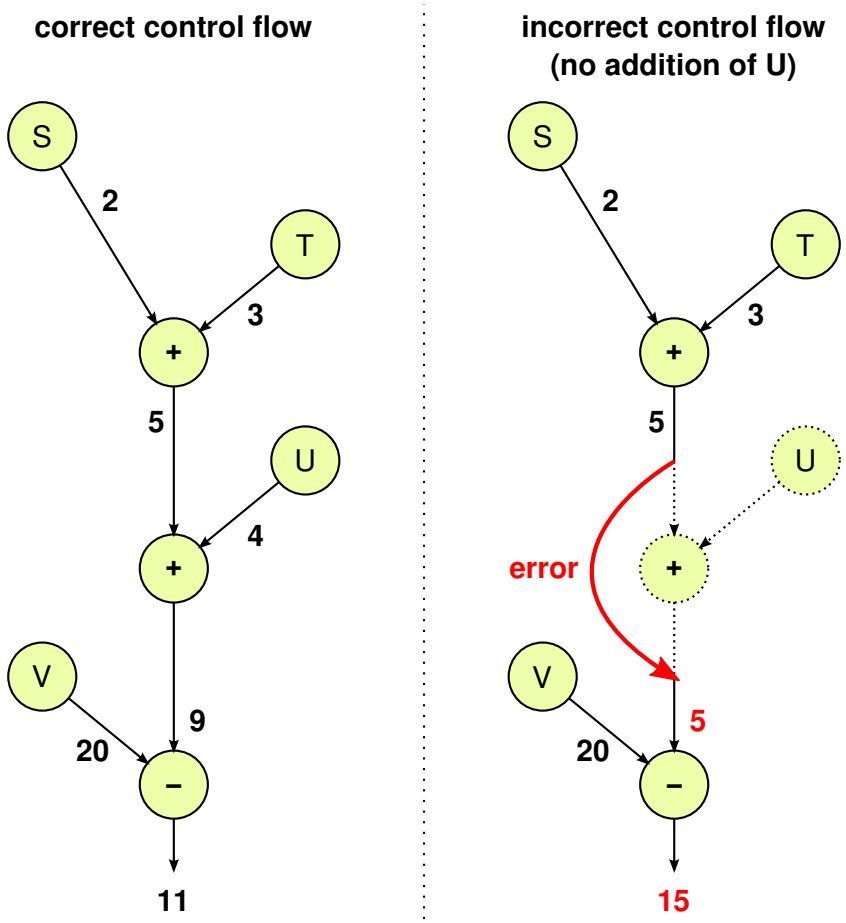


Figure 2.3 – Implicit control flow check through static signatures. Bold numbers represent static signatures (*B*s) of source values and results of arithmetic operations. Skipping an operation (red arrow on right side) leads to a different signature for the final result.

the results are correct encoded values with respect to the signatures resulting from the correct operations. Figure 2.3 illustrates how a control flow error leads to a different final signature.

This method can be extended to code using conditional branches, as illustrated in Figure 2.4. Before the actual, unencodable branch, a signature sig_{Cond} must be derived from the branch condition (variable). This signature must be different for

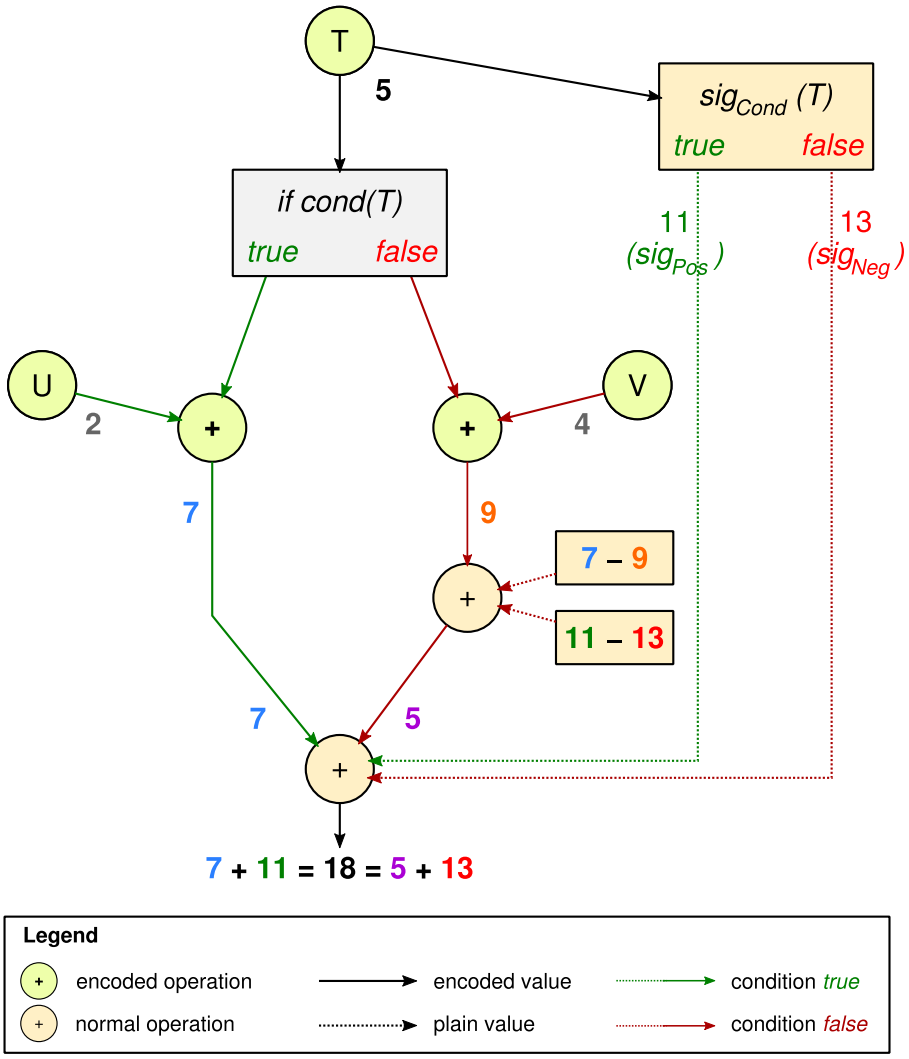


Figure 2.4 – Implicit control flow checks for conditional branches. Bold numbers represent static signatures (*Bs*) of encoded values. sig_{Cond} calculates a conditional signature value from the ANB-encoded condition variable. Control flow errors within and between branches lead to a different signature for the final result.

the true condition (sig_{Pos}) and the false condition (sig_{Neg}). It is added to the encoded variable to include the condition calculation in the protected data flow.

Inside the branches, the linear control flow is protected by the method described above, resulting in different signatures at each branch end. When the control flow merges again, it must be ensured that the resulting signature is the same no matter which branch was taken. For this, at the end of the negative branch, the difference in expected signatures must be (atomically) subtracted from the encoded variable. This way, any control flow error between branches is detected through an unexpected resulting signature.

Since these (implicit) control flow checks are tightly coupled to the data flow, they are only sufficient as long as the code does not perform any side effects. Code with side effects is not part of the encoded data flow and has no effect on signatures. To add (explicit) control flow checks to such code, an encoded variable can be transformed (e. g. by simple additions) after each side effect operation to create a linked and protected data and control flow.

2.4 Related Work

Related work for fault-tolerant operating systems falls in two mostly non-overlapping categories: methods to harden operating systems against non-transient errors and methods to detect or tolerate transient errors at a higher level above the operating system.

Several works have analyzed the effect of soft-errors on operating systems: Soft-errors occurring in the kernel of the *MicroC* RTOS are shown to have a major impact on the system's behavior [12]. Correct control flow, a major responsibility of the operating system, is shown to be much more critical than correct data, especially in signal processing applications [13]. Similarly, control flow assertions can detect most transient errors [14].

The dependability of commodity micro-kernel systems is evaluated in [15] and wrappers are added around system calls to detect errors by formal assertions. Also, a static system design has been identified to be very advantageous while hardening an operating system kernel against soft errors [16].

Several operating systems provide sophisticated spatial and temporal isolation or partitioning mechanisms, such as memory protection, deadline monitoring and virtualization. Examples for these systems include Integrity, PikeOS and PharOS. However, the kernel responsible for switching between isolation domains is part of the reliable computing base for these systems.

Other approaches use hardware support to provide error-detection. Hardware fingerprinting can be used to check the control flow against hashed values with little

overhead [17]. Hypervisor features of modern CPUs can be used to implement flexible, virtual lockstep with additional fingerprinting [18]. System partitioning using a hypervisor can reduce the size of the software RCB by depending on additional hardware features [19].

Some operating systems can recover from a detected error through micro-restarts or transparent thread-level Triple Modular Redundancy (TMR) [20, 21]. These systems depend not only on the correct execution of the micro-kernel, but also on a reliable and fast detection of errors.

2.5 Summary

This chapter has introduced the OSEK model of static real-time operating systems and arithmetic coding as an effective method of detecting data and control flow errors. This static system design and encoded operations will be combined in the next chapter to provide a reliable operating system, on which most existing dependability methods depend.

Chapter 3

Architecture

This chapter presents the architecture of the *d*OSEK operating system, beginning with an overview of the system components.

To avoid faults, many parts of the system are static and stored in Read-only Memory (ROM). This process is described in Section 3.2. Section 3.3 lists isolation techniques used to contain the effects of errors after their occurrence, while Section 3.4 presents methods for error detection. Section 3.5 shows how the internal data structures of *d*OSEK are using these error detection methods. Finally, Section 3.6 describes how remaining indirections are reduced to prevent vulnerabilities.

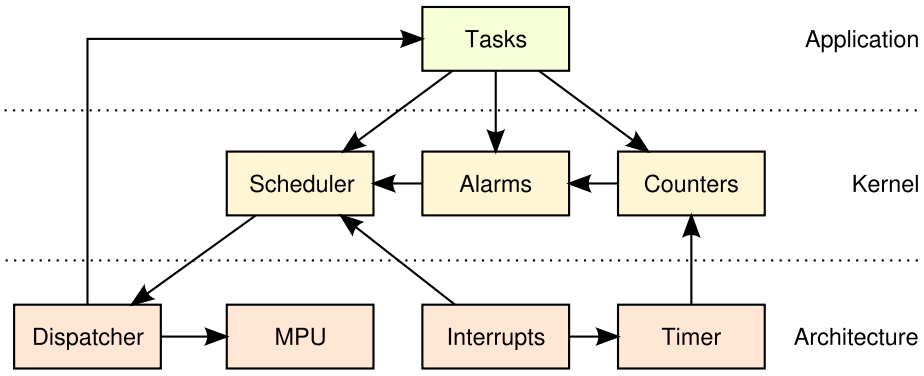


Figure 3.1 – Overview of the *d*OSEK architecture, split into application, kernel (hardware-independent) and architecture (hardware-dependent) parts.

3.1 Overview

The architecture of *d*OSEK is divided into a hardware-specific and a hardware-independent (kernel) part. Figure 3.1 shows an overview of all system components and their interactions.

Some cross-cutting system aspects are not shown as they need to be addressed both in the kernel and in the architecture-dependent parts, such as system calls or isolation mechanisms.

3.2 Static System Design

Since the CPU and RAM of the system are vulnerable to soft-errors at run-time, it is preferable to generate as much system data statically at build-time as possible.

Some system structures are derived from the hardware architecture and can be generated statically only for a specific target system, while others are dependent on the application running on the operation system. Prior knowledge about the application can be used to generate tailored code and data blocks at build-time.

All generated, static structures can be stored in read-only memory, which is assumed to be reliable, i. e. part of the Reliable Computing Base (RCB) of *d*OSEK. As long as the ROM is reliable, this procedure prevents soft-errors and increases robustness. The tradeoff for this is reduced flexibility. Generating static structures in the build process is usually more complicated, thus increasing the effort necessary for adding features or porting the system to new architectures. Furthermore, the operating system needs to be rebuilt for every change of target system or application.

3.2.1 Application/System Analysis

*d*OSEK analyzes the application to extract information about the usage of operating system services and shared resources.

For this, the analysis back-end of the Real-Time Systems Compiler (RTSC) [22] is used. The RTSC compiles the application code using the Low-Level Virtual Machine (LLVM)[2] framework and extracts Atomic Base Blocks (ABBs). One ABB can represent a block of "pure" computation as well as a call to another application function or system service.

This analysis allows every *invocation* of a system call to be specialized. This is used to generate a tailored system call implementation for each call site, which can use a-priori knowledge to reduce the amount of system state accessed during the execution. Listing 3.1 and Listing 3.2 show the application and system code before and after this transformation, respectively.

[2] `http://llvm.org/`

```
1  /* Application */
2  TASK1() {
3    ActivateTask(TASK2);
4    ActivateTask(TASK3);
5
6    TerminateTask();
7  }
8
9  TASK2() {
10   ActivateTask(TASK3);
11
12   TerminateTask();
13 }
14
15
16 /* System */
17 ActivateTask(TaskType task) {
18
19
20
21
22   // generic implementation
23
24
25
26
27 }
28
29 TerminateTask() {
30
31
32   // generic implementation
33
34
35 }
```

Listing 3.1 – Original application and sytem call code

```
1  /* Application */
2  TASK1() {
3    ActivateTask__ABB1();
4    ActivateTask__ABB2();
5
6    TerminateTask__ABB3();
7  }
8
9  TASK2() {
10   ActivateTask__ABB4();
11
12   TerminateTask__ABB5();
13 }
14
15
16 /* System */
17 ActivateTask__ABB1() {
18   // activate TASK2 from TASK1
19 }
20
21 ActivateTask__ABB2() {
22   // activate TASK3 from TASK1
23 }
24
25 ActivateTask__ABB4() {
26   // activate TASK3 from TASK2
27 }
28
29 TerminateTask__ABB3() {
30   // terminate TASK1
31 }
32
33 TerminateTask__ABB5() {
34   // terminate TASK2
35 }
```

Listing 3.2 – Transformed application and sytem call code after specialization

As most OSEK system calls use parameters known at build-time, these are compiled as constants into the individual implementations. This prevents parameter corruption, which would be possible if the parameters were passed in registers or on the stack. Other system calls (only two[3] in our OSEK implementation) require dynamic parameters and these are passed in registers and not stored in memory if possible (the compiler may choose to save them temporarily on the stack if registers are exhausted during execution).

Additionally, the RTSC analysis allows to extract information about the usage of shared resources between tasks. This is used to automatically derive resource priorities according to the OSEK priority ceiling protocol. In the future, this information

[3] *SetAbsAlarm* and *SetRelAlarm*

could be leveraged to skip (scheduling) operations for impossible system control flows by reducing the set of possible system states during application execution.

3.2.2 System Generation

Using the information from system analysis, program code as well as system structures are *generated* during the build process.

For each specialized system call the tailored kernel implementation as well as a unique invocation function is generated and the original non-tailored call is replaced by a call to the new invocation function, as seen in Listing 3.1 and Listing 3.2.

The operating system also requires several data structures for system components like tasks, interrupts or alarms. If such a structure will never change during run-time, it is generated in the build process and stored in read-only memory.

3.3 Isolation

In order to mutually protect application tasks as well as the kernel from invalid behavior, *d*OSEK uses several isolation mechanisms.

3.3.1 Data Isolation

For the spatial isolation of tasks, the data of application tasks is kept in separate RAM regions. This enables memory protection to restrict read and write accesses only to the region of the currently running task. Likewise, each task has its own stack, which is secured by memory protection as well. When a task is suspended, the task context is saved on its stack, thus ensuring that all task state is confined in the stack and data regions allocated to the task.

This method not only prevents tasks from interfering with each other, but also protects the application from the kernel.

3.3.2 Temporal Isolation

Error can lead to significantly increased execution time of a task, or even to infinite loops. In an event-triggered system, this prevents tasks with lower priority from running in time, leading to deadline violations. To isolate other parts of the system from the effects of such failures, temporal isolation is used.

Ideally, the maximum execution time for every task is determined statically and a watchdog timer is used to terminate the task if it erroneously exceeds this time. If error-detection is sufficient (*fail-stop*), a global, possibly external, watchdog timer can be reset periodically when the entire system is working correctly. When

deadlines are violated as the result of an error, the timer will expire and trigger a recovery or reset.

If a watchdog timer is built in software, it must not be maskable by disabling interrupts and should ideally only be resettable by a command sequence which is unlikely to be called even after an error occurred.

Currently, a software watchdog is not implemented in *d*OSEK, but an external watchdog is simulated for evaluation purposes. In the future, this can also be implemented in software.

3.3.3 Privilege Isolation

Privilege isolation is used to prevent tasks from performing operations which can bring the entire system into an invalid or undesired state. Examples for such operations are globally disabling memory protection or halting the system with interrupts disabled. All modern hardware architectures support a supervisor/non-supervisor privilege separation, where potentially dangerous operations can only be performed in supervisor mode.

*d*OSEK uses this isolation to run tasks and system calls in a non-privileged mode. Only the dispatcher, the system call entry function and interrupt handlers are run in supervisor mode. The dispatcher and system call entry functions require supervisor privileges to reconfigure the memory protection for the next task.

3.4 Error Detection

When an error occurs, the operating system needs to detect this as fast and reliably as possible to reduce harmful consequences. After detection, the current implementation of *d*OSEK brings the system into a safe end state (*fail-stop*).

3.4.1 Data Constraints

Errors leading to invalid data can be detected by run-time assertions checking known constraints of the specific data type. When an assertion fails, the application can cause a software trap to signal the error to the operating system.

This is especially effective for arithmetic coding, as several bit-flips can be detected reliably. To check if an ANB-encoded value is correct, the signature B is subtracted and the remainder of the division with A is calculated, which must be zero for all valid codewords. Additionally, the control flow of operations affecting ANB-encoded variables is checked implicitly. On a 32 bit machine, arithmetic coding can be used efficiently for values up to $2^{16} - 1$ (65 535), as the resulting encoded value fits inside hardware registers.

In cases where bigger values are necessary or the run-time overhead is unsuitable, parity calculations can be used as a simpler error-detection method. For example, the highest bit of a 32 bit variable can be used to ensure the value has odd parity. In this case, the parity of the other 31 bits is calculated when storing a value and the parity bit is set if the result is even. When checking the value, the parity over all 32 bits is calculated and an error is detected if the result is not odd. Modern architectures often support parity calculation in hardware, preventing the usage of corruptible temporary values during calculation. If used for pointers on a 32 bit architecture, dedicating one bit to parity reduces the addressable memory from 4 GByte to 2 GByte, but allows to detect any odd amount of bit-flips. However, an even count of bit flips in the same variable cannot be detected.

If errors in an array of values must be detected, a checksum can be derived and stored. A very simple and efficient checksum method is to combine all values using XOR. This can be calculated incrementally without any temporary values. The XOR operation is comparable to calculating the parity over all values for each bit position. As a result, an even count of bit flips occurring in the same bit of the values is not detected. All other error patterns, including an even count of bit flips in different bits, are detected. To check for data errors, the checksum is re-calculated and compared to the stored value. A more robust checksum algorithm is the CRC32 cyclic redundancy check, which is accelerated by some hardware architectures.

Table 3.1 compares the properties of these error-detection methods.

3.4.2 Memory Protection Violations

Corruption in the instruction pointer or data structures, especially inside the operating system, can lead to invalid RAM accesses or jumps to invalid code in ROM. One example for this are damaged return addresses used by the dispatcher.

Many of these errors can be detected by the MPU as the access is not within allowed regions and are signaled through software traps to the operating system.

	Arithmetic coding	Parity Bit	XOR Checksum
Detectable bit-flips	up to 5 bits†	any odd number	any odd number
Encoded operations	possible	not possible	not possible
Original data size	16 bit	31 bit	unlimited
Storage overhead	16 bit	1 bit	32 bit
Run-time overhead	large	small	small

† depends on chosen A

Table 3.1 – Comparison of error-detection techniques used in *d*OSEK. Numbers are given for a 32 bit architecture.

Executing system calls with memory and privilege isolation allows errors to be detected in operating system code as well.

Since the valid data and code regions are kept as small as possible, only errors in the lowest bits of affected data are not detected using this method. All errors in higher bits must result in addresses which are outside of the allowed regions.

3.4.3 Padding with Invalid Code

In order to detect control flow errors, like invalid jumps or skipped instructions, the individual functions within the code ROM can be padded with invalid instructions, which will always cause a trap when executed.

3.4.4 Watchdog Timer

Transient errors can also lead to violations of timing constraints. Corruption in the instruction pointer might cause an infinite loop or the system could erroneously halt the CPU. As described in Section 3.3.2, *d*OSEK currently depends on an external watchdog to detect hangs and other timing violations in the operating system.

3.5 Encoded Data Structures

Even after reducing the vulnerable system state space by static design and implementing various isolation and error detection measures, data and control flow errors can still result in silent data corruption.

Many of these errors can be detected using encoded data structures at the cost of increased code size, execution time and memory usage. Notably, the use of arithmetic coding allows data *and* control flow errors to be detected.

Due to analysis and static system generation, the remaining amount of data which needs to be encoded is already smaller compared to a generic system which must encode all data structures.

The simple and static design of *d*OSEK reduces the dynamic data structures needed at run-time to the (prioritized) ready-list of tasks, alarm and counter state, and saved task contexts.

3.5.1 Encoded Alarms

OSEK alarms and counters have dynamic values and state which must be managed at run-time.

Counters require only their current value to be stored in memory. For counters based on hardware timers, this value is incremented periodically in an interrupt handler.

The dynamic state of alarms consists of their status (armed/not armed), trigger counter value and cycle reload value. These values are only changed by application system calls or when an alarm is triggered.

By limiting their respective maximum values (as permitted in the OSEK specification), all these values can be ANB-encoded while still fitting into practical limits. For 32 bit systems, 16 bit counter and alarm values can be stored and manipulated efficiently.

When a counter ticks, an encoded addition is used to increment its value. Then, alarms are triggered as necessary by encoded equality comparisons between counter and alarm trigger values.

3.5.2 Encoded System Calls

System calls are the points where data and control flow passes between the application and the operating system. Most system call arguments in an OSEK system are static and need not be passed at run-time (see Section 3.2.2).

To protect the remaining, dynamic arguments during the transition from application to kernel, *d*OSEK uses ANB-encoded values. As before, arguments up to 16 Bit wide can easily be encoded and passed in a 32 Bit register.

3.5.3 Encoded Scheduler

The most important aspect of an operating system is the management of (running) tasks. For this, it is necessary to track the set of running processes and their respective priorities.

Since *d*OSEK implements an event-driven OSEK system, the information which tasks are running at a given point is time is not known at system design, in contrast to time-triggered systems. This requires the system to store task state and priority information in memory at run-time.

3.5.3.1 Task Queue

Conceptually, running tasks are enqueued in a priority queue, sorted by their dynamic priority. The scheduler uses this task queue to determine the next (highest-priority) task to run and may remove or modify entries due to termination or resource usage.

*d*OSEK encodes this critical system structure to detect data and control flow errors in the scheduler. Since encoded operations incur a significant overhead, encoding was added to a very basic implementation of a prioritized task queue. The implementation stores the current dynamic priority of each task at a fixed location, with the lowest possible value (zero) representing a suspended task. To determine

the highest-priority task, the maximum task priority is found by comparing all task priorities sequentially.

By assigning the lowest possible priority to suspended tasks, the ready state of tasks is stored implicitly in the priority value. This makes another structure to keep task states unnecessary, which would only introduce more vulnerabilities to transient errors.

Additionally, the simple algorithm easily allows to work only on a subset of all tasks by skipping unneeded comparisons. In the future, static system analysis can be used to determine which tasks cannot be ready for each system call invocation. For example, the scheduling after a *TerminateTask* system call does not need to consider the task just terminated[4]. Skipping such unneeded comparisons reduces overhead and increases robustness.

3.5.3.2 Encoded Task Queue

This algorithm has a space and time complexity linear to the number of tasks in the system (which is *constant* at run-time). While more sophisticated implementations are known, this priority-array approach is very suitable to encoding. On the one hand, the control flow is simple and linear, which reduces overhead as every encoded conditional branch or arithmetic operation increases complexity. On the other hand, no additional data structures are introduced and all data values are small unsigned integers, for which encoded operations are available. Also, in contrast to desktop operating systems, the number of tasks in a RTOS is not only known statically, but usually rather small (less than 100).

*d*OSEK implements this task queue using ANB-encoded priority values, which allows both data and control flow checks to detect corruption.

3.5.3.3 Encoded Retrieval of Highest-Priority Task

The most important and most frequently performed operation on the task queue is the retrieval of the task with the highest dynamic priority. If all tasks are suspended, the idle task should be returned.

The implementation shown in Algorithm 3.1 uses two ANB-encoded variables to store the highest-priority task ID found so far (id) and the corresponding priority (prio).

[4] unless multiple activation were allowed

```
1:  sig_id ← B_id + sig_1
2:  sig_prio ← B_prio + sig_1
3:  id ← task1.id + sig_id − B_task1.id
4:  prio ← task1.prio + sig_prio − B_task1.prio
5:  updateMax(sig_2, sig_id, id, task2.id, sig_prio, prio, task2.prio)
6:  assert ((id + prio) mod A) = sig_id + sig_prio
7:  updateMax(sig_3, sig_id, id, task3.id, sig_prio, prio, task3.prio)
8:  assert ((id + prio) mod A) = sig_id + sig_prio
    ...
9:  updateMax(sig_4, sig_id, id, idle.id, sig_prio, prio, idle.prio)
10: assert ((id + prio) mod A) = sig_id + sig_prio
11: id ← id − (sig_id − B_id)
12: prio ← prio − (sig_prio − B_prio)
```

Algorithm 3.1 – Retrieve the highest-priority task from the encoded priority queue. The encoded result ID is stored in sig_{id} with the corresponding encoded priority in sig_{prio}. The arguments to *updateMax* are passed **by reference**. Grayed lines are operations on constants, performed at compile-time, not run-time.

The algorithm consists of three steps:

1. Initialize prio and id to the first task (potentially suspended).
2. For all other tasks, compare the task's priority to prio:
   (a) If bigger or equal than prio, update prio and id to the new task.
   (b) If smaller, do not change the values of prio and id.
3. After all tasks, perform step 2 for the idle task.

As the values of id and prio change during one execution, their static signatures are changed accordingly after each operation. Although the algorithm explicitly shows the changes of these signatures (sig_{id}, sig_{prio}) they are not calculated at run-time. The compiler can determine the constant values of these signatures at each step and only uses the resulting constants in the actual code. To detect data and control flow errors (as described in Section 2.3.4), each step of the algorithm not only combines the signatures of the used variables, but also adds a unique, generated signature (sig_i).

Step 1 The algorithm initializes id and prio to the first task in lines 3 and 4. This is valid even if this task is suspended and must not be the final result as the comparison in step 3 succeeds in this case. In lines 1 and 2 the corresponding signatures are calculated from the original static signatures (B_{id}, B_{prio}).

Step 2 In lines 5 and 7, the priority of each task is compared to prio and updated if bigger or equal. In this case, id is changed accordingly. The arguments are passed by reference, as the values of id, sig_{id}, prio and sig_{prio} are changed in place. The function ensures that the same, unique signature is added to sig_{id} and sig_{prio} for any result of the comparison.

Any corruption in the arguments during the execution of *updateMax* results in values of id and prio which do not correspond to sig_{id} and sig_{prio}, allowing the error to be detected. However, fault injection experiments of the entire algorithm have shown that a small number of bit-flips remain undetected. In these cases, a *following* call (still with corrupted data) can cancel out[5] the effects of the first corrupted *updateMax* call. As each single *updateMax* call can reliably detect these errors, an assertion is added after each call in lines 6 and 8 to check if both signatures are correct.

As indicated by the ellipsis in Algorithm 3.1, the algorithm can be extended to any number of tasks by adding one *updateMax* call and assertion per task.

Step 3 Since the *updateMax* function can only check for a greater *or equal* priority, the maximum must be compared to the idle priority *after* all other tasks. This is required to return the idle task if all tasks are suspended. As shown in lines 9 and 10, the call to *updateMax* is identical to a normal task. Unlike other priorities, the idle priority is a constant (encoded zero), which allows additional compiler optimizations.

Finally, the signatures of id and prio must be restored by subtracting the difference between the resulting and original value. If an error occurred during the algorithm's execution, a different (incorrect) signature results, and id and prio remain invalid, as only the constant signature for a correct run is subtracted.

This last step could also be performed by another part of the system, but must be done before the algorithm can run again. In the future, dynamic signatures (ANB**D** encoding) could be used to prevent further errors by adding uniqueness to each invocation of the algorithm.

An example run of the algorithm for three possible tasks is shown in Figure 3.2. The figure illustrates the data flow of the id and prio variables, beginning at the top and ending in the result at the bottom. For easier understanding, the colors are identical to those used in Algorithm 3.1 and a value of 1000 is used for A. Italic numbers are static signatures (B) and bold numbers the resulting encoded value (v_c). Normal numbers show the unencoded values of encoded variables. In this example, task 1 has a dynamic priority of 1, task 2 is suspended (priority 0) and task 3 has priority 5 (possible when a resource is occupied).

[5] The resulting signatures are bigger and smaller by the *same* amount compared to the correct signature. Adding the signatures thus masks the error.

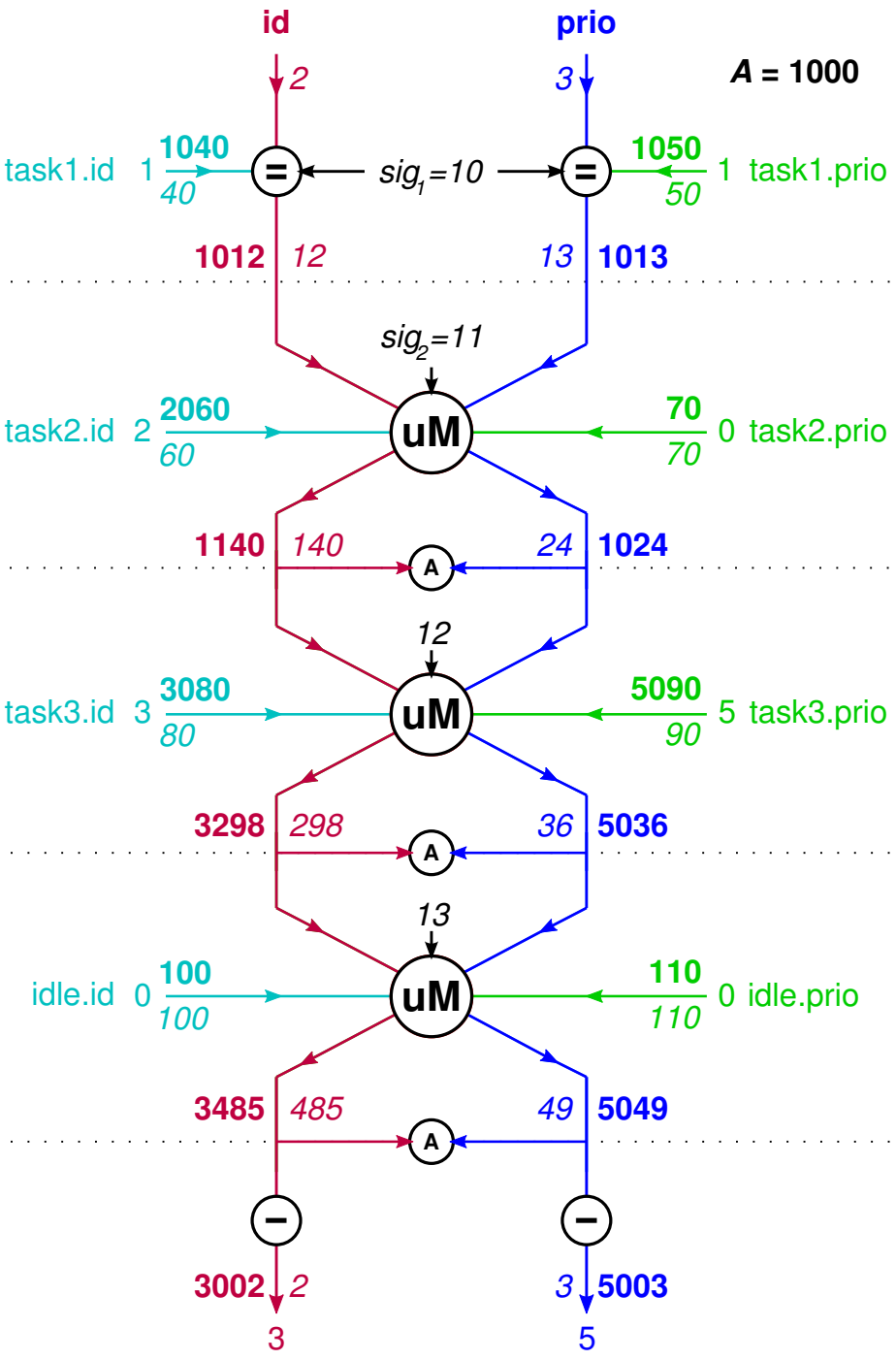


Figure 3.2 – Data flow of an example run of Algorithm 3.1. Italic values are signatures, bold values the resulting encoded values. Normal numbers show the unencoded values of encoded variables.

In step 1, id and prio are initialized to an encoded 1 (from task1.id and task1.prio) with the signatures 12 and 13 combined from the original values $(2, 3)$ and sig_1 (10).

The first *updateMax* call of step 2 does not change the value of id and prio since task 2 is suspended and has the lowest priority (task2.prio $= 0$). However, the signatures of both values are always adjusted and checked by the subsequent assertion: $(1140 + 1024)$ **mod** $1000 \overset{!}{=} 140 + 24$

The following comparison with task 3 results in an update of id (to task3.id $=$ 3) and prio as it has a higher priority (task3.prio $= 5$). The final comparison with the idle task in step 3 would only succeed if all tasks were suspended, which is not the case in this example.

After all comparisons, the static signature of id (sig_{id}) is 485 and sig_{prio} is 49. To restore these to the original values of 2 and 3, the constants 483 and 46 are subtracted, respectively. At the end, the correct encoded highest priority task ID (3) and priority (5) are stored in id and prio.

3.5.3.4 Encoded Update of Maximum

The *updateMax* method implements the core operation used when determining the highest-priority task: an encoded greater-or-equal comparison which updates the maximum priority and task ID. By hardening this base operation against data and control flow errors, the resulting task queue implementation is also hardened against these errors.

The implementation is based on the ANBD less-or-equal operation as presented by Schiffel [11, p. 68]. Instead of returning an encoded boolean comparison result, the *updateMax* method is extended to change both the compared variable (prio) and a corresponding variable (id).

The base for the algorithm is an encoded *if* condition, as described in Section 2.3.4. The algorithm consists of three steps:

1. Calculate a signature sig_{cond} from the current maximum priority prio and the task's priority task.prio.

2. Perform the actual unencoded comparison:

   (a) If the task's priority task.prio is greater or equal than prio, update prio to task.prio and id to task.id.

   (b) If task.prio is smaller, do not change the values of prio and id, but adjust them to match the signatures of the previous case.

3. Add the conditional signature sig_{cond} to prio and id in order to detect data and control flow errors.

```
1:  diff ← task.prio − prio
2:  sig_cond ← diff mod A
3:  sig_pos ← B_task.prio − sig_prio
4:  sig_neg ← ((2^32 mod A) + sig_pos) mod A
5:  comparison ← (prio − sig_prio ≤ task.prio − B_task.prio)
6:  if comparison then
7:      prio ← sig_prio + (task.prio − B_task.prio) + (sig − sig_pos)
8:      id ← sig_id + task.id + sig
9:  else
10:     prio ← prio + (sig_pos − sig_neg) + (sig − sig_pos)
11:     id ← id + (sig_pos − sig_neg) + B_task.id + sig
12: end if
13: prio ← prio + sig_cond
14: id ← id + sig_cond
15: sig_prio ← sig_prio + sig
16: sig_id ← sig_id + sig + sig_pos + B_task.id
```

Algorithm 3.2 – Update encoded maximum task priority and ID. Grayed lines are operations on constants, performed at compile-time, not run-time.

The implementation in Algorithm 3.2 shows how these steps are performed in detail:

Line 1–4 For the encoded *if* condition, it is necessary to derive a signature sig_{cond} from the input values, which has a known, fixed value for both the positive (sig_{pos}) and negative (sig_{neg}) comparison result.

This signature is based on the difference of prio and task.prio (line 1). As such a subtraction of unsigned 32 bit integers can underflow, the following values can result:

$$\text{diff} = \begin{cases} \text{task.prio} - \text{prio}, & \text{prio} \leq \text{task.prio} \\ 2^{32} - (\text{prio} - \text{task.prio}), & \text{prio} > \text{task.prio} \end{cases}$$

The signature sig_{cond} is the remainder of this difference divided by the encoding constant A (line 2). This way, the resulting values only depend on the static signatures, not actual run-time values:

$$\text{sig}_{\text{cond}} = \begin{cases} \text{sig}_{\text{pos}} = B_{\text{task.prio}} - \text{sig}_{\text{prio}}, & \text{prio} \leq \text{task.prio} \\ \text{sig}_{\text{neg}} = ((2^{32} \bmod A) + \text{sig}_{\text{pos}}) \bmod A, & \text{prio} > \text{task.prio} \end{cases}$$

As these values for sig_{pos} and sig_{neg} are constants, the calculations in line 3 and 4 are performed by the compiler, not at run-time. To prevent underflows in these values, the *updateMax* method requires that $B_{\text{task.prio}} > \text{sig}_{\text{prio}}$.

Line 5–6 The actual branch is chosen by an unencoded comparison of prio and task.prio after subtracting their static signatures. The resulting AN-encoded values can be compared directly, as they differ only by the factor A to the unencoded value.

Line 7–8 If the task's priority task.prio is greater or equal than the current maximum prio, the new value is stored in prio (line 7). The previous value of prio must be overwritten in this case, but any previous errors in this value would be propagated and detected through sig_{cond}.

To calculate the new value of prio, task.prio is added to sig_{prio}, which represents an encoded zero. Additionally, the signature of sig_{prio} must be adjusted. To ensure the precondition $B_{\text{task.prio}} > \text{sig}_{\text{prio}}$ still holds for subsequent invocations of *updateMax*, $\text{B}_{\text{task.prio}}$ and sig_{pos} are subtracted from prio. Finally, the unique signature of this *updateMax* invocation (sig) is added.

If the maximum in prio is replaced, id must also be updated to task.id (line 8). As before, the new value is the sum of an encoded zero (sig_{id}), the value of task.id and the invocation's signature (sig). However, it is not necessary to subtract any signatures as the algorithm has no constraints on sig_{id}. Thus, the signatures $\text{B}_{\text{task.id}}$ and sig_{pos} are included in the new value for id. This way, the algorithm's result signatures depend on all input signatures, as required to detect errors.

Line 10–11 If the task's priority task.prio is less than the current maximum prio, the encoded values of prio and id remain unchanged and only their signatures are adjusted.

Both values must subtract the expected value sig_{neg} of the condition signature sig_{cond} and add the value of the other possible branch (sig_{pos}). The unique signature of this *updateMax* invocation (sig) is added as well.

For prio, no further adjustments are necessary as the signature of task.prio is subtracted in the other branch. For id however, the signature $\text{B}_{\text{task.id}}$ has to be added to result in the same static signature as the addition of task.id in the other branch.

Line 13–14 The calculations in the branches use the corresponding constant (sig_{pos} or sig_{neg}) in place of sig_{cond} to derive a common result signature. After the control flow of both branches merges again, the condition signature sig_{cond} is added to both result values (prio and id). If the wrong branch was taken or a value was corrupted, this addition does not result in the expected value, which allows the error to be detected.

Line 15–16 After the algorithm is finished, the signatures of prio and id have changed to include the compared values. These changes are performed at run-time in lines 7–8 or 10–11. As it is necessary for the compiler to know the new values of sig_{prio} and sig_{id}, the new constant values are specified in the last two lines of the algorithm. These calculations are performed during compilation, not at run-time.

Example 3.3 shows the calculations of a concrete run of the *updateMax* algorithm for the comparison with task 3 in Figure 3.2.

1: $\text{diff} \leftarrow \text{task.prio} - \text{prio} \quad = \quad 5090 - 1024 = 4066$
2: $\text{sig}_{cond} \leftarrow \text{diff} \bmod A \quad = \quad 4066 \bmod 1000 = 66$
3: $\text{sig}_{pos} \leftarrow B_{\text{task.prio}} - \text{sig}_{prio} \quad = \quad 90 - 24 = 66$
4: $\text{sig}_{neg} \leftarrow ((2^{32} \bmod A) + \text{sig}_{pos}) \bmod A \quad = \quad (296 + 66) \bmod 1000 = 362$
5: $\text{comparison} \leftarrow (\text{prio} - \text{sig}_{prio} \leq \text{task.prio} - B_{\text{task.prio}}) \quad = \quad (1000 \leq 5000) = \textit{true}$
6: **if** *true* **then**
7: $\quad \text{prio} \leftarrow \text{sig}_{prio} + (\text{task.prio} - B_{\text{task.prio}}) + (\text{sig} - \text{sig}_{pos})$
$\quad\quad = 24 + (5090 - 90) + (12 - 66) = 4970$
8: $\quad \text{id} \leftarrow \text{sig}_{id} + \text{task.id} + \text{sig}$
$\quad\quad = 140 + 3080 + 12 = 3232$
9: **else**
10: $\quad \text{prio} \leftarrow \text{prio} + (\text{sig}_{pos} - \text{sig}_{neg}) + (\text{sig} - \text{sig}_{pos})$
$\quad\quad = 1024 + (66 - 362) + (12 - 66) = 674$
11: $\quad \text{id} \leftarrow \text{id} + (\text{sig}_{pos} - \text{sig}_{neg}) + B_{\text{task.id}} + \text{sig}$
$\quad\quad = 1140 + (66 - 362) + 80 + 12 = 936$
12: **end if**
13: $\text{prio} \leftarrow \text{prio} + \text{sig}_{cond} \quad = \quad 4970 + 66 = 5036$
14: $\text{id} \leftarrow \text{id} + \text{sig}_{cond} \quad = \quad 3232 + 66 = 3298$
15: $\text{sig}_{prio} \leftarrow \text{sig}_{prio} + \text{sig} \quad = \quad 24 + 12 = 36$
16: $\text{sig}_{id} \leftarrow \text{sig}_{id} + \text{sig} + \text{sig}_{pos} + B_{\text{task.id}} \quad = \quad 140 + 12 + 66 + 80 = 298$

Example 3.3 – Concrete run of *updateMax* algorithm for the comparison with task 3 in Figure 3.2. Grayed lines are operations on constants, performed at compile-time, not run-time. Lines colored in dark red are not executed since the other branch is taken.

3.6 Reduction of Indirection

Experiments clearly show that any added indirection in the data or control flow will lead to an amplification of errors. For this reason, it is a primary goal of *d*OSEK to identify and remove such indirections.

3.6.1 Pointers

One of the most obvious sources of indirection are pointers [23, 24]. As pointer values are usually rather large compared to application or OS data values, it is less practicable to encode them, both in terms of space usage as well as execution time. Additionally, encoded operations cannot be performed with or on pointers. For these reasons, *d*OSEK is built without the usage of pointers for data. However, the hardware architecture usually requires the use of some pointers, such as stack pointers when switching contexts.

3.6.2 Loops

Thanks to the static design of *d*OSEK, the usage of loops as control flow structures is not necessary. The resulting code is equivalent to fully unrolling loops operating on system structures. This leads to (significantly) increased code size, but prevents errors introduced by looping, such as endless loops caused by corruption in index variables or the result of the abort condition check. Likewise, *d*OSEK does not use recursion, which suffers from similar problems as looping and is usually not allowed in safety-critical applications.

3.6.3 Stack

The stack pointer, which is used heavily in normal (C) system code, is another major source of errors. It shares the same problems with standard pointers described above, but stack accesses are often implicitly added by the compiler or required by the hardware architecture. *d*OSEK tries to reduce stack usage in the OS to an absolute minimum. This way, not only errors in the stack pointer itself are prevented, but also corruption of values temporarily saved on the stack.

3.6.4 Function Calls

One typical usage of the stack is a function call, which pushes the return address and jumps to a different code block. Both the stack pointer as well as the return address are pointers, leading to a significant amount of errors introduced for every

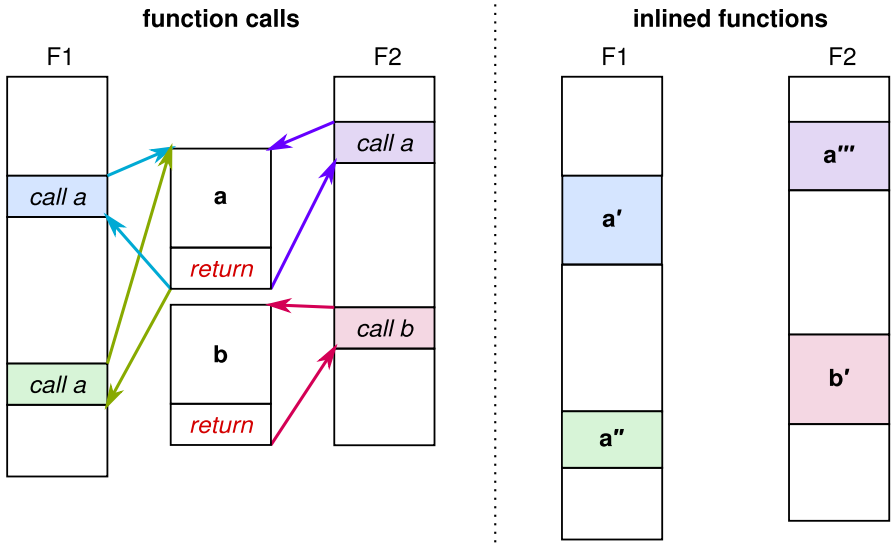


Figure 3.3 – Illustration of inlined and non-inlined function calls. Control flow errors result if the stored return address is corrupted before returning from a non-inlined function. Inlined versions of the same function ($a', a'' a'''$) usually differ because of possible compiler optimizations.

single executed call. Additionally, the compiler needs to save on stack all registers which might be modified by the called function, as well as the function arguments.

As a solution, *d*OSEK is using inlining as much as possible. The difference between inlined and non-inlined functions is illustrated in Figure 3.3. By copying the implementation of the called function to each call site, the compiler can remove the call completely.

Storing arguments and registers values on stack is no longer necessary and any argument constraints known to the compiler, especially constant values, can be propagated through the function body. This optimized code might not only be slightly faster, but much more robust, as memory or register accesses can be removed.

Drawbacks of inlining are significantly increased code size and more difficult debugging, but these are acceptable given the enormous reliability benefits.

3.6.5 Local Variables

Local variables used in functions are usually saved on the stack as well, leading to compiler-generated indirect accesses using a (stack frame) base pointer. Storing data at fixed, absolute addresses increases reliability and is eased by the static system design. All local variables of non-reentrant functions can instead be stored at absolute addresses without changing code semantics. Due to the simple, static system design, this can be guaranteed in most cases.

3.6.6 Architectural Indirection

Some sources of indirection are introduced by the underlying hardware architecture. However, alternative approaches to common architecture-dependent mechanisms are often possible and can be used to reduce the amount of indirection added.

3.6.6.1 System Calls

Generic operating systems often save the parts of the task context (e. g. register contents) at the start of a system call. This allows the application to treat system calls like normal function calls, but requires the kernel to handle application data. *d*OSEK delegates this task to the application by informing the compiler that a system call can change the value of *every* register. This results in the compiler saving all registers whose value is still required after the system call. Register values which are no longer needed are neither saved nor restored. Saving and restoring are performed *in the application* before and after the actual system call. This way, the kernel never needs access to the application stack to save data.

The standard system call mechanism on some architectures (e. g. i386) uses software traps/interrupts to activate the kernel in supervisor mode. Usually, this results in additional indirection because task context is automatically saved on the task when a trap is triggered. *d*OSEK explores alternative ways to invoke system calls which can reduce indirection, such as specialized CPU instructions for entering and leaving the supervisor mode.

3.6.6.2 Interrupt support

Operating systems often implement interrupt handling through a generic interrupt handler routine which will determine the interrupt source and call the corresponding handler function. *d*OSEK prevents the introduction of errors in such a generic handler by generating complete, linear interrupt functions, each running interrupt entry code, the actual handler and interrupt exit code. Again, this reduction of indirection incurs the cost of increased code size.

3.7 Summary

This chapter has presented the architecture of *d*OSEK, based on a static system design to *avoid* potential errors. To *detect* remaining faults, isolation and error-detection methods are used, especially arithmetic coding. The scheduler as a critical system component is fully protected using ANB-encoding. Care must also be taken to reduce and prevent unnecessary indirections in the system implementation as these amplify potential vulnerabilities.

Chapter 4

Implementation

This chapter describes the implementation of *d*OSEK with a focus on the Intel i386 variant, which is also used for the evaluation.

First, an overview over all dynamic system structures is given, followed by a description of the chosen programming language and the implementation of encoded data structures. The used isolation mechanisms are presented in Section 4.4 while the resulting control flow is described in Section 4.5. Finally, the build system used to generate the entire operating system is presented in Section 4.6.

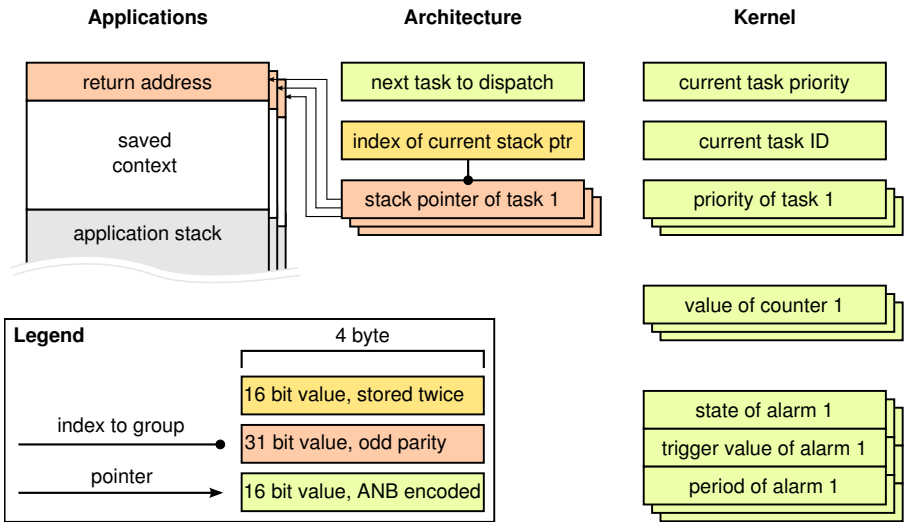


Figure 4.1 – Complete overview of *d*OSEK system structures kept in RAM.

4.1 Overview of Dynamic System Structures

An overview of *all* operating system structures kept in RAM is given in Figure 4.1. The kernel uses ANB-encoded structures for task priorities, counter values and alarm states. Additionally, the encoded ID and priority of the current task are stored. Section 4.3 describes these encoded data structures.

The architecture-specific implementation for the i386 architecture uses one ANB-encoded variable to indicate the chosen task for the dispatcher. Each task has a corresponding global variable which stores its current stack pointer, protected by using the highest bit to achieve (odd) parity. The index to the current task's stack pointer is saved as well, as this is needed to save the context before an interrupt handler is started. Section 4.5 describes interrupt and system call handling as well as the dispatcher.

Tasks which are interrupted after an asynchronous interrupt or (synchronous) system call have their context stored on the own stack. This way, the MPU can prevent (erroneous) accesses from the kernel or system calls to this context, as described in Section 4.4. The return address, stored on stack as well, is also protected through odd parity.

As indicated in Figure 4.1, *d*OSEK uses 16 bytes of RAM for internal variables plus 8 bytes per task, 4 bytes per counter and 12 bytes per alarm. Additionally, each task has its own stack plus one stack for the kernel.

4.2 Implementation Language

*d*OSEK is implemented in C++ instead of C as this allows the implementation of custom encoded data-types through classes, amongst other benefits. If run-time features like exceptions and dynamic polymorphism are avoided, the core C++ language adds no overhead compared to C, allowing it to be used for embedded (RTOS) system development.

*d*OSEK is based upon the newer C++11 standard as this adds several features which can be used to easily generate more reliable code. One important feature are *generalized constant expressions*, which allow the compiler to generate constant objects/structures at compile-time. Unlike the `const` keyword available in earlier versions of C++, this is not limited to simple types. *d*OSEK extensively uses constant expressions to store all constant structures in read-only memory, which prevents these from being corrupted at run-time. Additionally, the compiler can better optimize operations with constant expressions, reducing branches and memory accesses. This increases reliability as well.

C++11 also adds type inference features, which are useful when operating with template types. The compiler can be instructed to determine the type of a variable of function automatically. Since *d*OSEK uses template types for all encoded values (see Section 4.3), this allows the use of encoded values without manually specifying all resulting (and potentially complicated) types.

4.3 Encoded Data Structures

*d*OSEK uses encoded data structures throughout the system to detect memory and control flow corruption. Since manually implementing encoded operations is a tedious and error-prone task, a new encoded data-type is implemented as a C++ class. This encoded data-type allows construction and arithmetic operations comparable to built-in C++ integer types.

The implementation of the encoded data type is based on the code from [10, 25], but extended to use C++11 constant expressions and additional operator methods. ANBD-encoded values are represented by the encoded value v_c and the corresponding timestamp D at runtime. Additionally, the compiler needs to know the used values for A and B at compile-time to generate encoded operations.

The implementation provides a class which allows 16 bit (unsigned) integers to be encoded, resulting in a 32 bit encoded value. As the current version of *d*OSEK uses only ANB-encoded data, i. e. no additional timestamps D, this allows efficient storage and operations on a typical 32 bit architecture.

The constant values of A and B do not need to be stored in RAM and are specified as template parameters for the encoded data-type class. This way, the compiler can constant-fold all operations involving these values and derive new encoded types with correct signatures.

For example, adding two encoded numbers using the encoded addition operator in Listing 4.1 can be written as:

```
auto enc_sum = enc_a + enc_b;
```

and results in the signature value B of `enc_sum` being the sum of the static signatures of `enc_a` and `enc_b`.

```
1 template< typename T, class RET = Encoded_Static<A, B + T::B> >
2 RET Encoded_Static::operator+(const T& t) const {
3   RET r;
4   r.vc = vc + t.vc;
5   return r;
6 }
```

Listing 4.1 – Encoded Addition Operator

By defining a *constant expression* constructor for the encoded data-type, the compiler can store constant encoded values in ROM and optimize (sequences of) operations with encoded constants using constant-folding.

All encoded variables in *d*OSEK use the fixed value of 58 659 for A, which results in a minimum Hamming distance of 6 between all valid code words [10]. This allows to reliably detect up to 5 bit flips in a single variable.

4.3.1 Scheduler

The implementation of the scheduler mainly consists of a C++11 version of Algorithm 3.1 and Algorithm 3.2.

The constant values for $\mathrm{sig}_{\mathrm{prio}}$, $\mathrm{sig}_{\mathrm{id}}$, $\mathrm{sig}_{\mathrm{pos}}$ and $\mathrm{sig}_{\mathrm{neg}}$ are automatically calculated by the compiler. This is important, as a calculation at run-time would be vulnerable to errors itself. Special assertions are used to check if these variables are actually converted to constants. These assertions include an invalid string of inline assembly in the *else* branch. If the variable is not a constant (as determined by the compiler), the comparison and both branches must be compiled, resulting in a build error. When the assertion can be completely optimized away through constant folding, no error occurs.

To prevent indirections, all variables used by the scheduler (id, prio and the task IDs and priorities) are members of a global, static scheduler object and thus stored at a fixed memory address.

For future specializations, the priority queue implementation is prepared to work on a *subset* of tasks. This is achieved by passing the set of possible tasks as a template parameter and wrapping each *updateMax* step in an *if* condition. These conditions are completely optimized away through constant folding.

4.3.2 Alarms and Counters

*d*OSEK also uses encoded variables for the values of OSEK alarms and counters as described in Section 3.5.1.

For timer-based OSEK counters, the Programmable Interval Timer (PIT) is used to periodically generate an interrupt. The timer interrupt handler increments the encoded counter value and triggers alarms as necessary. If an alarm callback routine is triggered, this is executed directly in the interrupt handler. In the future, these callbacks may also be run in user-space to improve isolation.

4.4 Isolation

To provide isolation between the tasks and the operating system, *d*OSEK uses the i386 Memory Management Unit (MMU) for memory protection and CPU rings for privilege isolation.

For temporal isolation, *d*OSEK currently depends on an external watchdog timer. In the future, one of the remaining unused timers of the i386 architecture could be used for the generation of non-maskable interrupts to detect timing violations.

4.4.1 Memory Protection

Memory protection is the most important isolation mechanism for *d*OSEK, as a significant amount of errors can easily be detected by memory accesses outside of allowed regions.

On the Intel i386 architecture, memory protection features can be provided by two mechanisms: segmentation or paging.

4.4.1.1 Segmentation

Segmentation allows the usage of multiple (potentially overlapping) address spaces which are defined by their start and length in the physical RAM. Accesses beyond the length of a segment result in a CPU exception. Since this non-linear memory model is not supported by C or C++ compilers, this method was not used for *d*OSEK.

4.4.1.2 Paging

Paging allows a virtual address space to be constructed by mapping 4 KB pages from arbitrary physical addresses to virtual addresses. Each page can be setup to be read-only or only accessible from supervisor mode. The actual mapping of pages is described by a hierarchical page directory, which is stored in the physical address space itself. *d*OSEK uses identity paging, which keeps the linear physical addresses unchanged, while allowing access limits on a per-page granularity. As the compiler is completely oblivious to identity paging, no special support is necessary.

4.4.1.3 Read-Only Memory

*d*OSEK is built upon the availability of a ROM for reliable code and constant storage. While this is common for typical real-time systems (especially when a Harvard architecture is used), typical i386 computers do not use a ROM except for the BIOS. For this reason, the ROM part of *d*OSEK is stored in a contiguous section of RAM on i386, which is completely marked as read-only. This allows the detection of invalid write accesses to the ROM just as if a real ROM were used and is sufficient

for the evaluation using an instrumented i386 emulator. In the (uncommon) case that a dependable system is based on the i386 architecture, a real ROM should be used and mapped into the physical address space to attain the reliability originally assumed for the constant part of *d*OSEK.

4.4.1.4 Static Page Directories

To prevent corruption in the page directory, it is stored in ROM by *d*OSEK. This prevents changing memory protection parameters by modifying the relevant page entries at run-time, as usually done by i386 operating systems. Instead, *d*OSEK stores a complete page directory for each possible memory protection configuration in ROM and switches between these static directories at run-time. This switch is performed by writing the physical directory address into a control register in supervisor mode. Completely swapping the page directory flushes the entire CPU address translation cache, but the resulting performance degradation is acceptable given the increased robustness.

4.4.1.5 Memory Layout

Storing static page directories in ROM requires these to be available when building (linking) the final system image. A custom linker script is used to store all code and constants in a memory section starting at a fixed address. Likewise, all data which must be placed in RAM is stored starting at a different, fixed address. Additionally, code and data is grouped inside these two regions by type and task, aligned to start at possible page boundaries.

Figure 4.2 shows a simplified memory layout for a system with two tasks. On the right the contents of the generated page directories for each task and the kernel are shown. Yellow regions are read-only, green regions are writable. The orange region is readable in supervisor mode only, and is used for interrupt handler code and stack, which must always be mapped into memory when interrupts are enabled. All remaining parts of memory are unmapped and result in a CPU exception even for attempted reads.

4.4.1.6 Page Directory Generation

Since the page directory entries consist of the boundary addresses of all these memory regions, they can only be determined after the compiler and linker have placed all code and data in their final locations. For this reason, neither the compiler nor the linker can be used to produce the static page directories. Instead, a preliminary system image is built with empty page tables. Then, all page directory entries can be generated by analyzing symbol addresses in the preliminary binary. The page

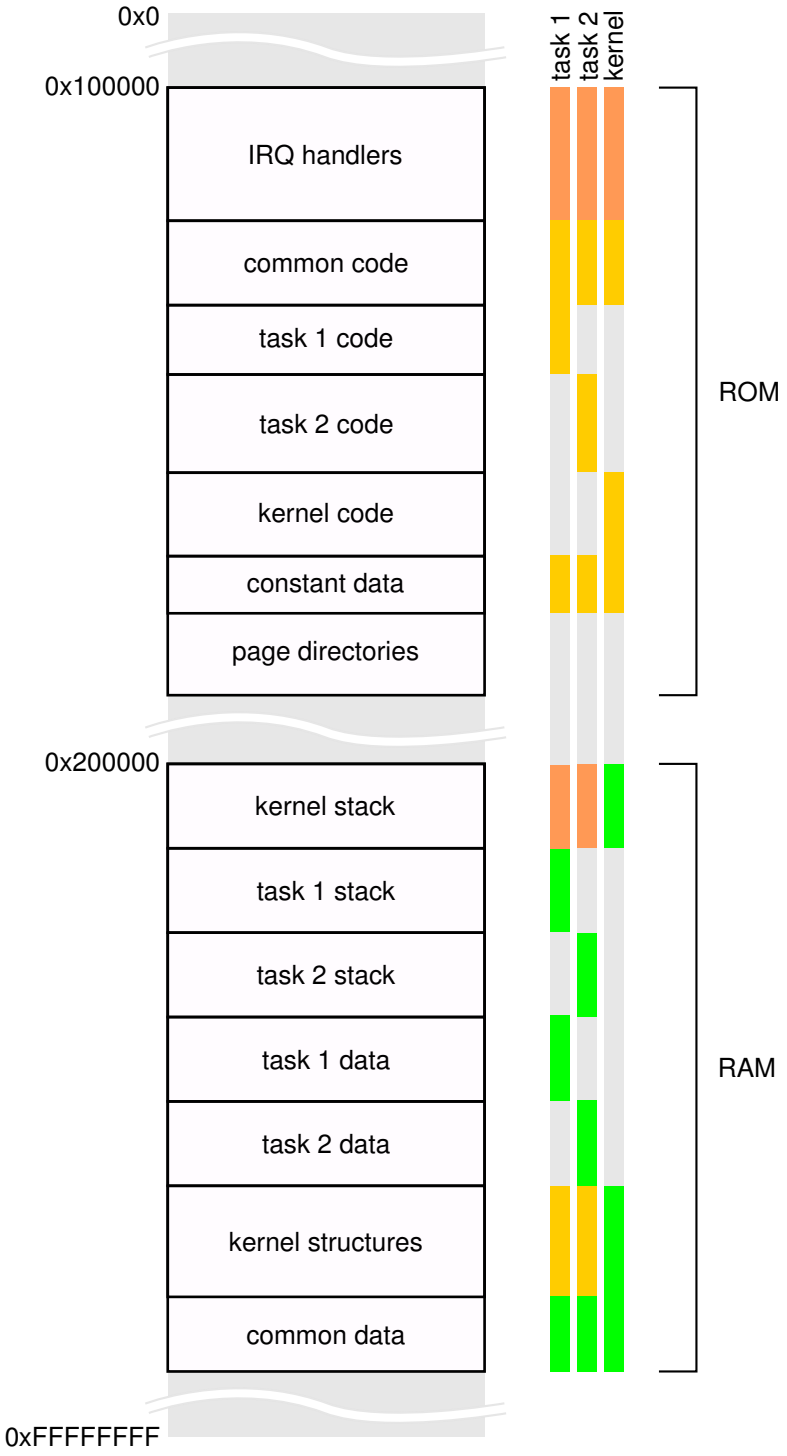


Figure 4.2 – Schematic i386 address space layout with per task/OS page directory contents. Yellow symbolizes read-only regions, green writable regions, orange supervisor-only regions. Padding and memory-mapped i386 hardware devices not shown for clarity.

directories are compiled to an object file and the entire system is linked as before, but with the actual page directories instead of empty fillers. Since the exact size of the page directories is not known before their generation, the empty directories are placed at the end of the ROM region, with a big gap before the RAM region begins. This allows the page directories to vary in size without affecting any addresses before or after them.

4.4.1.7 Padding

When aligning memory contents (to page boundaries), resulting empty space is filled with the value `0x33`, which is the encoding for the i386 `int3` instruction. This instruction always triggers a software interrupt and is used to detect invalid jumps within regions allowed by memory protection. In data sections, this value is more likely to cause a detectable failure than the default padding value of 0.

4.4.2 Privilege Separation

The Intel i386 architecture has the concept of *rings* for privilege separation, as described in Section 3.3.3.

*d*OSEK, like most i386 operating systems, only uses two of the four available rings. Ring 0 is used as supervisor mode, as all privileged operations are allowed. In ring 3, all privileged operations and accesses to supervisor memory pages result in CPU exceptions.

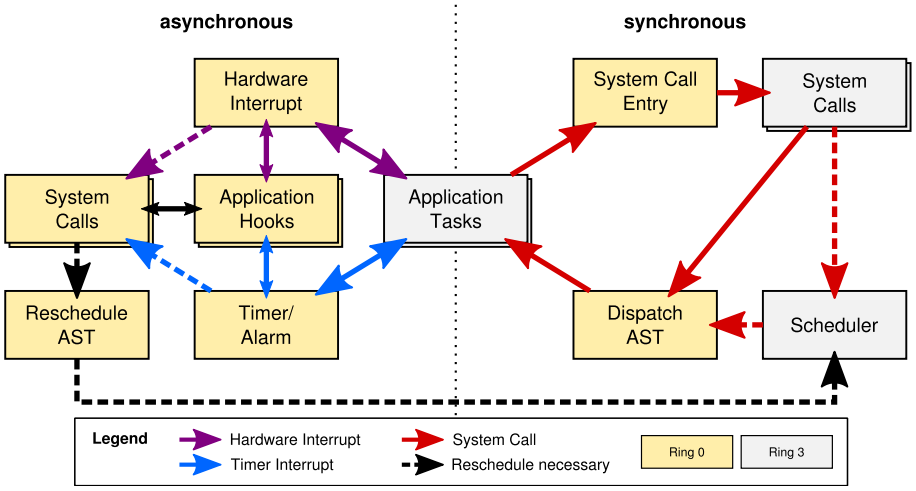


Figure 4.3 – Global spatial *d*OSEK control flow. Red arrows represent the control flow of a synchronous system call, purple arrows a hardware interrupt and blue arrows a timer/alarm update. Black arrows are possible for both interrupts and alarms. Dashed control flow occurs if changed task priorities require a rescheduling.

Tasks in *d*OSEK run in ring 3 to prevent them from performing actions affecting other parts of the system. Additionally, all (non-trivial) system calls are executed in ring 3 as well, to gain isolation benefits for them as well. When a system call is finished, the dispatcher (trap) is invoked from user-space to return to the application.

As a result, only the system call entry function, the dispatcher and interrupt handlers run in supervisor mode. The dispatcher and system entry function must be executed in ring 0 to allow them to change page directories while performing the context switch.

Interrupt handlers are always entered on ring 0, but as future work they may switch to ring 3 while executing application handler code. It has to be evaluated if isolation benefits outweigh the additional overhead.

4.5 Control Flow

A large part of the OSEK specification is concerned with system control flow handling, making this the main responsibility of *d*OSEK.

Figure 4.3 shows an overview of the spatial control flow between system and application components, as described in the next sections. System calls (red arrows) are synchronous to a task's execution while interrupts (purple and blue arrows) can occur asynchronously at any time during task execution. The operating system needs to coordinate the shown transitions to prevent corruption due to a lack of synchronization.

Figure 4.4 shows the temporal control flow between two tasks (upper half) and a hardware interrupt (lower half). Privilege isolation, as indicated by the dotted lines, has been described in Section 4.4.2. The system call mechanism (green arrows) is presented in Section 4.5.2 while the dispatcher is described in Section 4.5.3. The next section describes how interrupts are handled in *d*OSEK, as used for Asynchronous System Traps (ASTs) and hardware requests (red and blue arrows).

4.5.1 Interrupt Handling

Interrupts are an important aspect of the operating system control flow, especially in embedded systems. As interrupts can occur asynchronously at almost any point in time, the system must ensure that no synchronization problems arise and the interruption is completely transparent to the application.

4.5.1.1 Unrolled Handlers

*d*OSEK creates a linear interrupt handler function for each possible interrupt, in contrast to most operating systems, which often use a small assembly function to

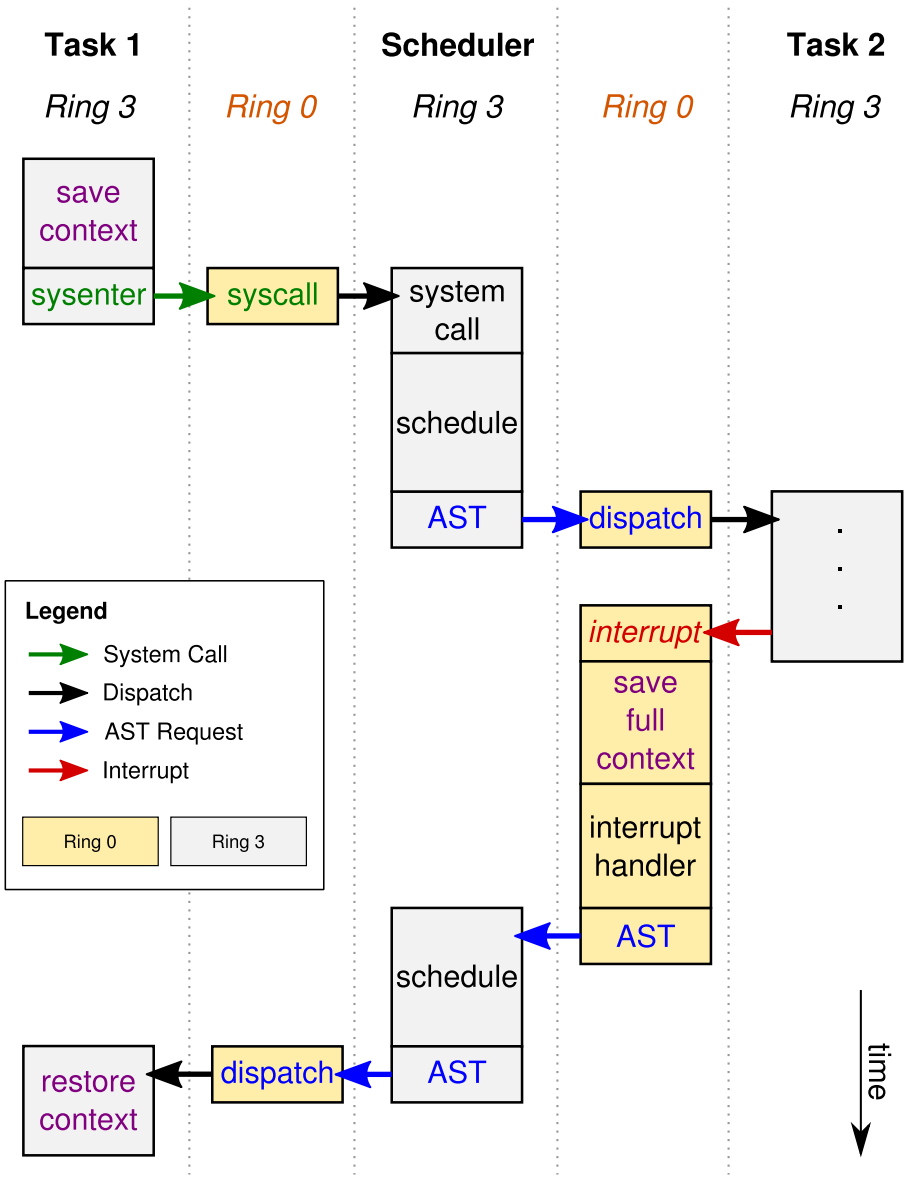


Figure 4.4 – Temporal control flow between two tasks and one hardware interrupt. While the scheduler is running in ring 3 (center), memory accesses to application data are prevented by the MMU. The red arrow symbolizes hardware interrupt entry, green arrows system entry instructions, black arrows system exit instructions and blue arrows AST activations.

perform a call to a generic handler written in a higher language. This function then calls the specific handler as determined by interrupt number, which is passed as an argument. While this is a flexible and more architecture-independent solution, this indirection suffers from the previously mentioned problems of function calls and stack usage as well.

*d*OSEK creates "unrolled" interrupt handlers by combining the entry code (written in assembly) and the interrupt handler function (written in C++) to a linear function. While this increases code size through duplication, it does not require any function calls.

As the common exit code does not depend on the interrupt source, it is shared by all handlers, which jump to it at the end. Unlike a function call, this unconditional jump to a fixed address never returns and no corruptible return address is used. The exit code does however need to know the original page directory and CPU context stack address. This information is stored in fixed registers by the entry code. Using inline assembly at the start and end of the C++ handler, the compiler is instructed to save and restore these registers if they are temporarily overwritten. This way, these values are only stored on stack if the interrupt handler actually needs the registers for its execution, in contrast to the standard approach of always storing all these variables on stack.

4.5.1.2 Context Checksum

When an interrupt occurs, the CPU automatically stores the original stack and instruction pointer as well as the flags register on stack. Since this saved context must be retained to resume the preempted task, it is very sensitive to bit errors. At the same time it is unsuitable for protection using arithmetic coding as these are 32 Bit values and must be accessed in assembly. Thus, a simple checksum is used as a more basic error detection method. All (32 bit) values of the context are combined using XOR and the result is kept in memory. Before the interrupt returns, the context is checked for corruption by performing the same XOR sequence again, but starting with the result value. The properties of the XOR operation ensure that the result will always be zero if the context and the saved checksum are unchanged while any single bit-flip is detected. If an even count of bit-flips occurs in the same bit of different checked values, the XOR operation will not detect this. Also, the introduction of additional zero values cannot be detected. Despite these shortcomings, this methods allows to detect a large portion of possible errors while only using one value (register) for storage *and computation*.

A possible alternative to the XOR operation is the CRC32 cyclic redundancy check, which detects more potential errors, especially additional zero values. The

Intel SSE 4.2 instruction set adds new CPU instructions for this algorithm, allowing it to be used as efficiently as XOR.

4.5.1.3 Static Interrupt Table

Similar to other architectures, the Intel i386 architecture maps interrupts to entry functions using an Interrupt Descriptor Table (IDT), which is loaded from memory. Since all interrupt mappings are fixed in *d*OSEK, this table is generated at build-time and stored in ROM. Like the static page directories (see Section 4.4.1.4), this prevents corruption due to bit errors.

Currently, entry functions for all 256 possible i386 interrupts are included in the system image. In the future, only handlers which are actually used by the system may be generated to reduce ROM size. Unlike the present solution, this requires an additional generation step.

4.5.1.4 Context Saving

The preemption of the running task by an interrupt is the only case where the entire context of a task needs to be saved. Unlike a system call, this interruption is asynchronous and no cooperation from the task (or compiler) is possible.

A global stack pointer index variable (`save_sp`) is used to determine if an application task is interrupted. In this case, the application context (including the instruction pointer) is saved on the task's stack and the (modified) stack pointer is saved to the corresponding task-specific global variable, derived from the value of `save_sp`.

If `save_sp` is zero, a system call was interrupted and its register contents are saved on the interrupt (system) stack. As only ISR1s, which may not interfere with system state, can interrupt a system call, execution will always resume at the point of interruption.

To protect the `save_sp` variable against bit flips, the actual value is stored twice inside the 32 bit memory location, supporting up to 2^{16} tasks. While this method is less robust compared to ANB-encoding, it prevents the significant overhead of arithmetic coding in interrupt entry code, especially since no arithmetic operations are performed with this value at all.

4.5.1.5 Suspending Interrupts

The OSEK specification allows to suspend all *ISR2* interrupts, which is also done automatically while a system call is executed. *d*OSEK enforces that the priorities of all ISR2 interrupts are lower than any ISR1 interrupt, as recommended by the OSEK specification. As a result, ISR2s can be blocked easily by raising the processor

priority above the maximum ISR2 priority using the Local Advanced Programmable Interrupt Controller (LAPIC). This disables servicing for all interrupts with lower priority, i. e. ISR2s. ISR1 interrupts remain enabled during system call execution as they are not allowed to use any system service and thus unable to interfere with system state. When ISR2s are re-enabled by resetting the processor priority all queued interrupts are serviced according to their priority.

To suspend *all* interrupts, the corresponding OSEK system calls are transformed to i386 interrupt disable/enable instructions (`cli` and `sti`, respectively). Both operations are allowed in user-space by setting the I/O privilege level of the CPU to ring 3. This completely prevents the usual overhead of a system call.

4.5.2 System Calls

While interrupts occur asynchronously to a task's execution, system calls are used to synchronously invoke operating system services from tasks running in non-privileged mode.

4.5.2.1 Software Interrupts

The original Intel architecture can switch from user-space to supervisor mode only by using software interrupts. Interrupts which are configured to be callable from ring 3 can be triggered using the `int` CPU instruction and behave just like hardware interrupts. Specifically, they switch to the interrupt handler function and stack in ring 0 and store the return address, stack pointer and flags register on the system stack. When the system call is finished, the `iret` interrupt return instruction will pop these values from the stack and restore the original user-space control flow.

This method stores critical values on stack, resulting in all of the problems described in Section 3.6.3. To prevent this indirection, the next section presents an alternative system call mechanism used in *d*OSEK.

4.5.2.2 Specialized Entry/Exit Instructions

Since the Intel i586 architecture, specialized system entry (`sysenter`) and exit (`sysexit`) CPU instructions are available. This pair of instructions uses registers instead of the stack and performs only the minimum functionality needed. For example, steps like saving the return address and flags register must be implemented in software.

While originally introduced because of speed gains compared to expensive software interrupts, their minimal functionality is also advantageous for robustness.

In this method, the return address is saved by the application task on its own stack before `sysenter` is invoked, thus catching errors in this operation by all

user-space isolation mechanisms. The normal method of saving the return address in supervisor mode is not safeguarded by these isolation mechanisms.

4.5.2.3 Unrolled System Calls

Each system call invocation is specialized in the application analysis step as described in Section 3.2.1. The OS code generator creates both the tailored user-space invocation function and the corresponding kernel implementation.

In the present implementation, the invocation function passes the unique, fixed address of the generated system call in a register, which is then jumped to by the system entry handler. As this address is susceptible to corruption like other pointers, it could be replaced by an encoded index/offset to a handler jump table stored in ROM in the future.

4.5.2.4 Argument Passing

Some operating systems emulate the behavior of a standard function call by passing system call arguments on the stack. *d*OSEK uses fixed registers instead, which limits the maximum number of arguments, but prevents the additional indirection of stack usage.

Although support for up to three arguments is implemented, standard OSEK system services never use more than two arguments. Most arguments of the unrolled and specialized system calls are constant with only two system calls passing run-time arguments at all (see Section 3.2.1).

4.5.2.5 Context saving

As described in Section 3.6.6.1, the application is responsible for saving all parts of the CPU context still required after the system call. The compiler does this automatically when specifying that the system call instruction (`int` or `sysenter`) "clobbers" (potentially overwrites) *all* registers. This generates code to push the necessary registers to the task's stack before the system call and to pop them afterwards.

4.5.2.6 Isolation

In order to limit the effects of errors in system calls, they are isolated like tasks. The actual system call implementation is executed in user-space while memory accesses to application data are prevented. To return to the application, the dispatcher is invoked at the end of the system call. This requires two transitions from user-space to supervisor mode and back for one system call.

For some small systems calls the resulting relative overhead is large and might add more vulnerabilities through context switching than the isolation mechanisms prevent. For these cases, *d*OSEK (optionally) allows system calls to run directly in supervisor mode.

4.5.3 Dispatcher

The dispatcher is responsible to perform the actual task switch, as described in Section 2.1.2.

4.5.3.1 Task Context

To switch tasks the dispatcher has to load the next task's CPU context, as shown in Figure 4.5. Other CPU (configuration) registers are not changed between tasks and need not be considered by the dispatcher.

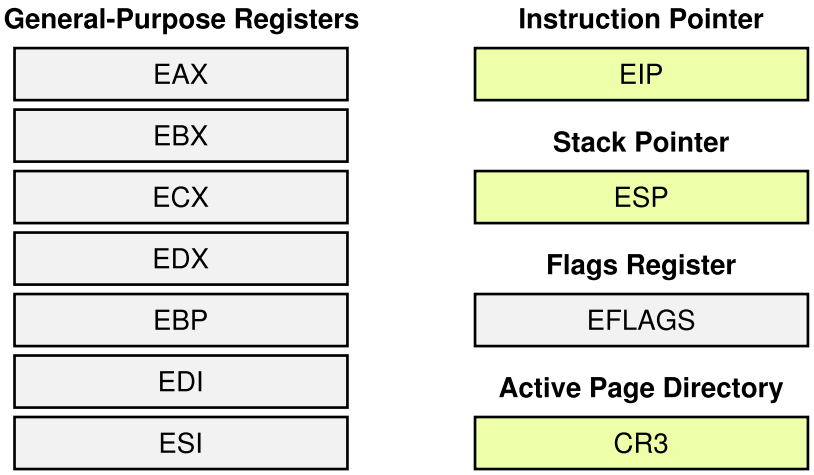


Figure 4.5 – Overview of the task context on the Intel i386 architecture. All registers are 32 bit wide. Green registers are *restored* by the *d*OSEK dispatcher, others are *cleared*.

4.5.3.2 Software Interrupt

*d*OSEK dispatches using software interrupts, i. e. the dispatcher is implemented as an interrupt handler. This ensures the dispatcher is executed in supervisor mode (ring 0), which is required to switch page directories.

Since interrupts can be the cause of a reschedule/dispatch decision, multiple concurrent interrupts (which are processed non-preemptively) can activate different tasks. If the dispatcher were activated synchronously by each handler using a

software interrupt, each dispatch would be performed followed by the processing of the next queued interrupt. As all but the last schedule/dispatch execution are unnecessary, this would waste time and reduce reliability by additional data accesses.

For this reason, the dispatcher interrupt is an Asynchronous System Trap (AST) with lowest priority in *d*OSEK. This means that all normal interrupt handlers are executed before the dispatcher is started.

An AST on i386 is not different from a normal interrupt handler, but it must be triggered using the LAPIC. By writing to a LAPIC control register, the interrupt is handled as if it were caused by a hardware device. As a result, it is processed after all higher-priority interrupts, unlike software interrupts which are executed immediately. Interrupt priorities can be adjusted as required using the Input/Output Programmable Interrupt Controller (IOAPIC).

Before the AST is requested, the ID of the dispatched task is written to the global, ANB-encoded variable `dispatch_task`. This variable can be updated by subsequent (queued) interrupts before the AST is started.

4.5.3.3 AST Implementation

The implementation of the dispatcher AST handler performs these operations (in this order):

- Decode the ANB-encoded task ID stored in `dispatch_task`.
- Update `save_sp` to this ID. The interrupt entry code uses this to determine where to save the stack pointer of the interrupted task.
- Switch to the task's page directory.
- Determine if the task has run before by checking the saved stack pointer. Only the stack pointer of a non-running task points at the top of the stack, while in all other cases, context is stored on stack.
  - If resuming, obtain the stored instruction pointer from stack.
  - If not resuming, set the instruction pointer to the task entry function.
- Clear general-purpose registers to ensure a clean working state for the task.
- Exit the interrupt handler (supervisor mode) and jump to task with interrupts disabled.

Notably, the dispatcher does not actually restore the entire task context on its own, only the green-colored registers shown in Figure 4.5. This is done to reduce unnecessary memory accesses performed in supervisor mode. The next section describes how the remaining task context is restored (if necessary).

4.5.3.4 Context Restoration

In case a task is started (not resumed), the dispatcher AST ensures a clean state without traces left from the execution of previous tasks and no context needs to be restored.

If a task is resumed after a system call, the compiler has added instructions to restore necessary parts of the context from the task stack, as described in Section 3.6.6.1.

When an interrupt occurs during task execution, the complete task context needs to be saved. As seen in Figure 4.6, the interrupted instruction pointer, all general-purpose registers and the flags register are saved on the task's stack in this case. Additionally, the return address used by the dispatcher is substituted by the address of a small helper function (`irq_resume`). The dispatcher will then start this function (shown in Listing 4.2), which restores all flags and registers before continuing the original task flow by performing a return using the instruction pointer stored on stack. For future work, the saved context might be protected by computing a checksum over its contents, which is also stored on the stack and checked by `irq_resume`.

The dispatcher always starts and resumes tasks with interrupts disabled, but with the privilege to enable and disable them in userspace. Preemptive tasks enable

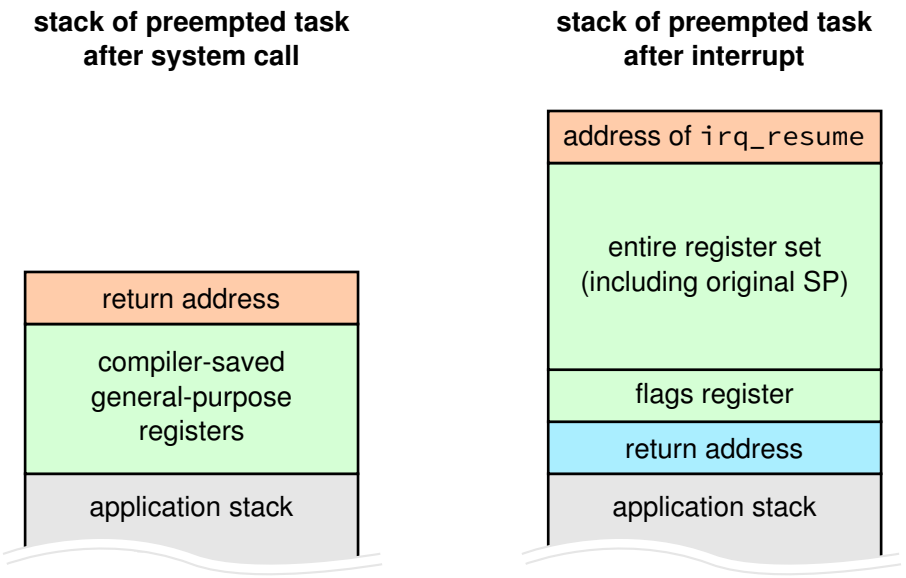


Figure 4.6 – Stack contents of preempted tasks. Region used directly by application code colored in gray, saved application context in green. The return address used by the dispatcher is shown with orange background. The actual application return address in case of an interrupt shown in blue.

```
1 irq_resume:
2   popa    # restore and pop all saved general purpose registers
3   popf    # restore and pop saved flags register
4   ret     # pop and jump back to saved instruction pointer
```

Listing 4.2 – Implementation of the `irq_resume` helper function

interrupts at the entry of the task's function. When resuming a task, the `irq_resume` function restores the interrupt enable flag as part of the flags register.

The only difference between starting and resuming a task for the dispatcher is how the task instruction pointer is determined. A task has to be resumed if the task's stack pointer is not equal to its initial value since a previously running task must have saved context on the stack.

If the task has not run before, the task is started at its fixed entry function with the stack pointer set to the top of the stack. If the task was interrupted, the task's stack pointer points to the saved instruction pointer. In this case, the task is resumed by using the saved instruction and stack pointer.

4.6 Build System

Building *d*OSEK for a specific application is more involved than simply compiling source files. The build system combines all analysis and generation steps to create the final system image.

4.6.1 Toolchain

The overall build process (shown in Figure 4.7) is coordinated using *CMake*[6], which generates Makefiles from its own higher-level description language. Using macros and variables, the complex dependencies between build steps are expressed more easily.

The first build step is whole system analysis. Unmodified application sources, which use the OSEK API defined in a common header file, are compiled using the RTSC. Additionally, the RTSC reads in the OSEK system description XML file. After this step, the application is compiled to LLVM Intermediate Representation (IR) and the analysis results are stored as XML.

Then, a generator framework, written in Python, generates the *d*OSEK kernel C++ sources from the analysis results and template files. Most importantly, kernel data structures and system call implementations are generated at this stage.

[6] `http://www.cmake.org/`

The generated kernel and the sources for the architecture-dependent layer of *d*OSEK are compiled to LLVM IR using the *clang* compiler of LLVM 3.4. Then, the LLVM optimizer and linker combines the application, kernel and architecture-dependent parts, each supplied in form of IR, into a single system image. The memory layout is determined by a linker script generated in the previous step.

For the Intel i386 architecture, the page directories are now generated and the final system image is built, as described in Section 4.4.1.6.

4.6.2 Testing

*d*OSEK contains a large number of test cases to automatically detect regressions.

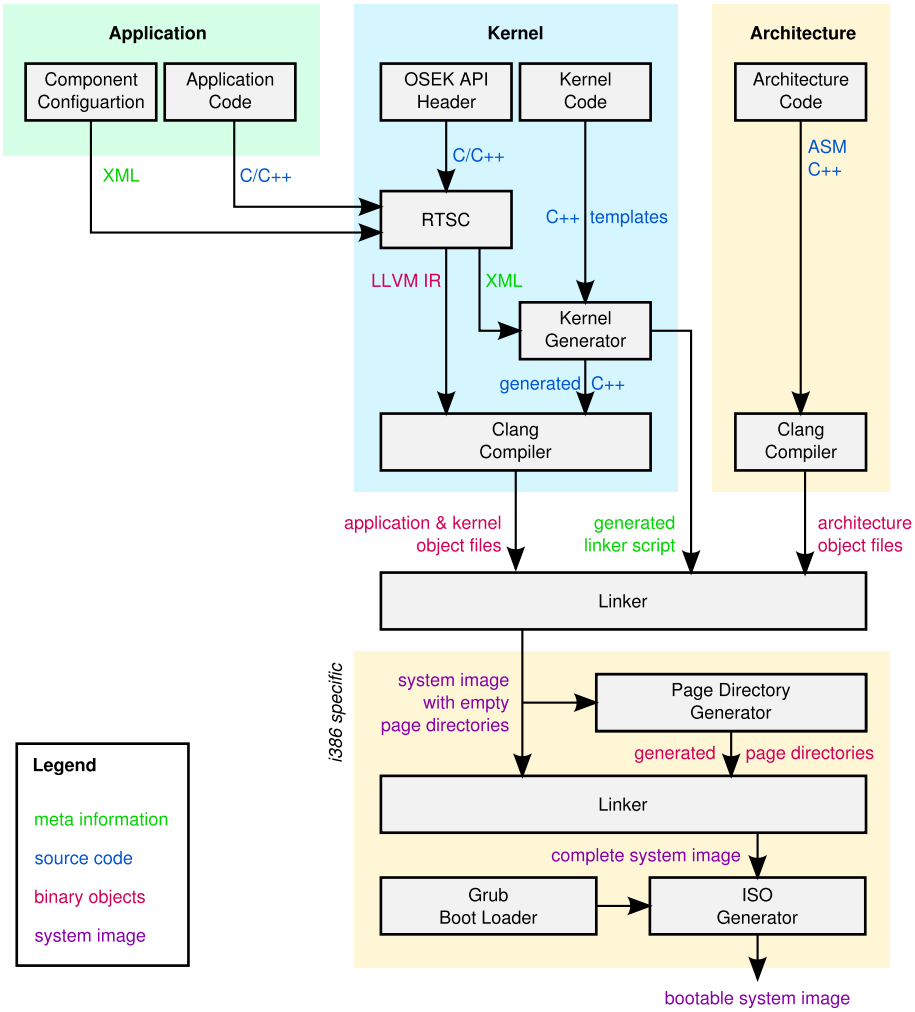


Figure 4.7 – *d*OSEK build process for the Intel i386 architecture.

Each test case is a simple yet complete *d*OSEK application which uses a predefined test library to output test results. These test functions print the results on the (emulated) screen or serial console, depending on build parameters. The result is a combination of unit and integration test, as each test uses/integrates the complete operating system but is only built to test one aspect.

A bootable ISO image with the *Grub*[7] boot loader is built with the test binary and then run in the *Bochs*[8] or *QEMU*[9] emulator by *CTest* with the generated output deciding if the test succeeds or fails.

For faster testing and easier debugging, as well as detection of undesired architecture-dependence in the kernel, a POSIX variant of *d*OSEK has been developed as well.

4.6.2.1 Continuous Integration

In order to detect regressions as early as possible, this test suite is also used for continuous integration. *d*OSEK uses *Gerrit*[10] to track and review all code changes. For every new change, an automatic *Jenkins*[11] job is triggered, which checks that the system still builds and passes all tests for all variants.

4.6.2.2 Automated Fault Injection

While the tests described in Section 4.6.2 check the correctness of the system in the absence of soft errors, automated fault injection experiments are necessary to check the robustness of the system.

The build system supports running complete fault injection campaigns using FAIL* (see Section 5.1.2) for each application and test. After running the simulations, the raw data is saved and a result report is generated automatically.

4.7 Summary

This chapter has discussed the implementation of *d*OSEK for the Intel i386 architecture. Using C++11 as the implementation language, encoded and/or static data structures can be used without sacrificing robustness or code clarity.

The i386 MMU and CPU ring functionality has been used to provide isolation while keeping the resulting indirections to a minimum. Indirections have also been reduced in all parts of the system control flow, specifically the dispatcher, interrupt handlers and system calls.

[7] `http://www.gnu.org/software/grub/`
[8] `http://bochs.sourceforge.net/`
[9] `http://www.qemu.org/`
[10] `http://code.google.com/p/gerrit/`
[11] `http://jenkins-ci.org/`

Chapter 5

Analysis

In this chapter, the robustness of *d*OSEK against soft errors is evaluated using fault injection experiments for several benchmarks and system variants. *d*OSEK is also compared to the open-source *ERIKA Enterprise* RTOS. Finally, the overhead of the *d*OSEK approach is evaluated and possible optimizations are presented.

5.1 Fault Injection

The effects of all possible soft errors are simulated in automated fault-injection campaigns. Using pruning methods, complete (100 %) fault space coverage can be achieved within reasonable simulation time, removing any uncertainties resulting from randomized testing.

5.1.1 Fault model

Single bit errors on the Instruction Set Architecture (ISA) level are assumed for the evaluation, i. e. the value of any memory location, any register or the program counter can be changed in one bit before any instruction. Not simulated are errors below the ISA level, for example in the computational logic of the arithmetic unit, which can lead to results which differ at multiple locations in time and space.

Single bit errors, as used in this evaluation, are shown to amount to over 95 % of the overall soft-error rate [26, 27]. The used fault model assumes no errors occur in ROM, as these memories are inherently more robust than volatile memory [28].

The time and space diagram in Figure 5.1 shows all bits in memory over all (discrete) time steps. In the time between a bit is written and read again, a soft error can occur, resulting in an incorrect read. This interval is called an *equivalence class*, as the consequences for the system are the same no matter at which time the

bit is flipped. For the evaluation, all equivalence classes which lead to silent data corruption are determined.

Each point in the time/space diagram is then called an *occurrence* of silent data corruption. The amount of occurrences per bit is proportional to the time during which a bit flip can lead to SDC. Likewise, the amount of occurrences per time tick is proportional to the number of vulnerable bits at this point. The robustness of different variants *performing the same task* can thus be compared using the total count of SDC occurrences.

5.1.2 FAIL* Framework

The FAIL* [29] fault injection framework is used to perform the experiment campaigns.

The first step of a campaign is to record (*trace*) the system without any errors using an instrumented simulator. This *golden run* is then analyzed to extract all memory accesses and the control flow.

From this analysis, the equivalence classes for memory, register or instruction pointer errors can be determined. In a pruning step, all equivalence classes are

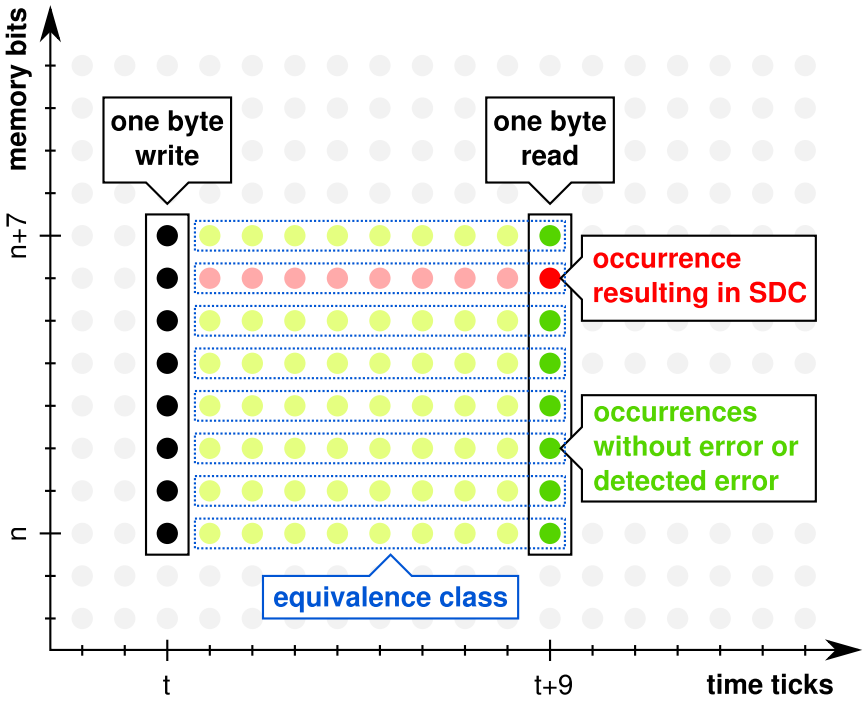


Figure 5.1 – Schematic illustration of data corruption occurrences. One byte is written to memory and read after 9 time ticks. If one corrupted bit at the read results in SDC this affects 9 occurrences.

condensed to a single representative experiment, resulting in a much smaller set of experiments which need to be run.

Each experiment is run in a simulator instrumented to inject the specific fault and record the consequences. FAIL* automates this task of running experiments in parallel while collecting the results.

5.1.3 Methodology

For the evaluation of *d*OSEK, correctness of control flow and integrity of application data is used to determine SDC.

The benchmarks used for the evaluation are *d*OSEK applications containing control-flow *checkpoints*. When a checkpoint is reached in the application control-flow, a unique value is written to a fixed, special memory location. This write is detected by the instrumented simulator and used to record the correct control flow during tracing. During the experiments, these checkpoints are also detected and checked against the control flow of the golden run. This way, incorrect behavior of the operating system after an injection can be detected.

An error in the operating system might corrupt application data even though the control remains correct. To detect this, all application data is checksummed at each checkpoint by the FAIL* experiment and compared against the golden run. The application data consists of each task's data region as well as the *used* parts of the task's stack, i. e. the region from the top of the stack to the current stack (pointer) position.

Three different types of soft errors are evaluated:

- Any used memory location is injected to simulate errors corrupting values stored in RAM.
- All used registers, including the stack pointer and flags register, are injected to simulate errors in the CPU memory.
- For each executed instruction, all possible one bit errors in the instruction pointer are injected to simulate control flow errors due to bit flips in the CPU.

All these injections are performed after the system has booted, as the (unprotected) architecture-specific hardware initialization routines are only run for a short time at startup.

The benchmark *application code* is exempt from all injections to evaluate only the operating system itself. This also prevents unintended side-effects due to corruption in the checkpointing process. Additionally, in a real system, the application tasks might use some form of redundancy to protect against errors in the application, while the operating system must work correctly to support this.

5.2 Benchmarks

The benchmarks used for evaluating *d*OSEK can be divided into micro-benchmarks, which test a single aspect of the system, and a sample application, which simulates an entire complex, real-world system.

5.2.1 Task Dispatch Micro-Benchmark

To evaluate the basic control flow handling between application tasks, a micro-benchmark tests the activation of three tasks.

Figure 5.2 shows the interactions between the tasks and the trace checkpoints. The auto-started task activates two other tasks, one with lower and one with higher priority. The higher-priority task is dispatched immediately while the lower priority task must wait in the *ready* state. After all tasks terminate, the idle task is activated.

The injection starts with the execution of the *StartOS* routine, which is called after architecture-specific initialization. According to the OSEK specification, this activates all auto-started alarms and tasks. Errors in this start-up routine are detected at the first checkpoint.

5.2.2 Interrupt and Alarm Micro-Benchmark

Asynchronous events like interrupts and OSEK alarms are an important yet challenging aspect of real-time operating systems. For the evaluation, a micro-benchmark using both an alarm and an interrupt (ISR2) hook is tested.

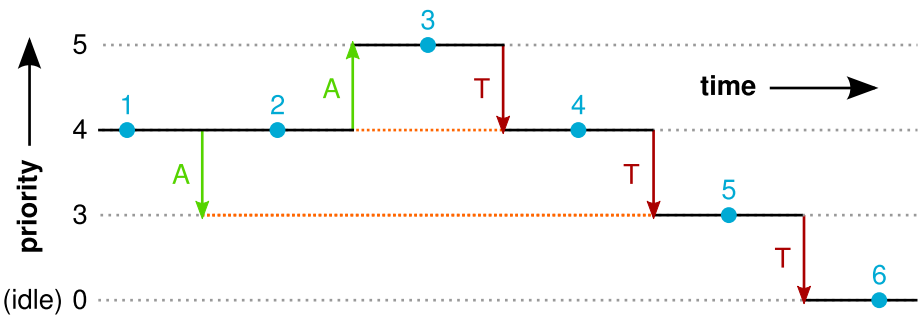


Figure 5.2 – Control flow of the Task Dispatch micro-benchmark. Running task shown by solid black line, ready tasks dotted orange. Green arrows are *ActivateTask* system calls, red arrows *TerminateTask* calls. Blue dots are checkpoints for the evaluation.

As seen in Figure 5.3, the alarm is triggered (by the timer interrupt) from the idle task. The alarm activates the lower-priority task and then executes its alarm callback routine, which triggers the ISR2 interrupt and exits.

After servicing the timer interrupt, the ISR2 hook is executed, which activates the higher-priority task. This task is dispatched when the ISR2 is finished, followed by the lower-priority task and the idle task after each termination.

5.2.3 Copter Sample application

As a realistic testcase, a larger benchmark was derived from a quadcopter flight control application. Illustrated in Figure 5.4, this sample application uses eleven tasks, three alarms and one (timer) counter, resource and hardware interrupt.

Each alarm directly activates one specific task while the hardware interrupt triggers a callback routine activating a task. The resource is shared by three tasks and two tasks are non-preemptable.

During the benchmark execution, which is significantly longer compared to the micro-benchmarks, the timer interrupt occurs frequently. Alarms and tasks are activated periodically after several timer ticks, but finish their execution quickly compared to the entire benchmark run-time.

The error-free benchmark runs for three hyper-periods of the system and results in 172 checkpoints.

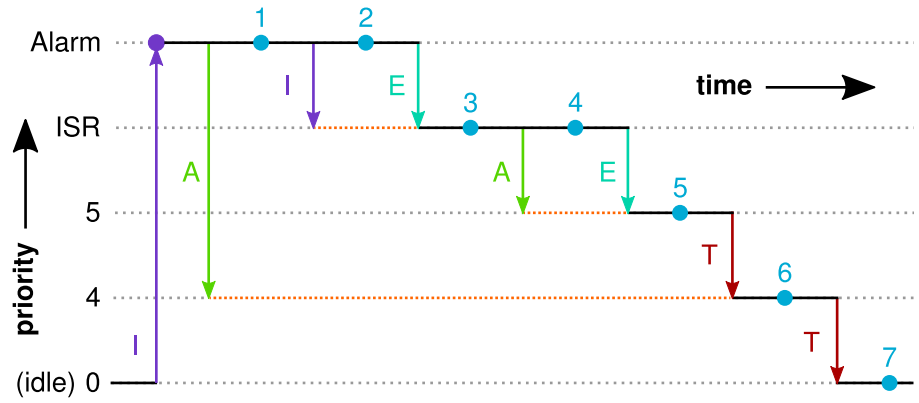


Figure 5.3 – Control flow of the Interrupt/Alarm micro-benchmark. Running activity shown by solid black line, ready activities dotted orange. Purple arrows indicate hardware interrupts, cyan arrows show the end of interrupt handling. Green arrows are task activations, red arrows *TerminateTask* system calls. Blue dots are checkpoints for the evaluation.

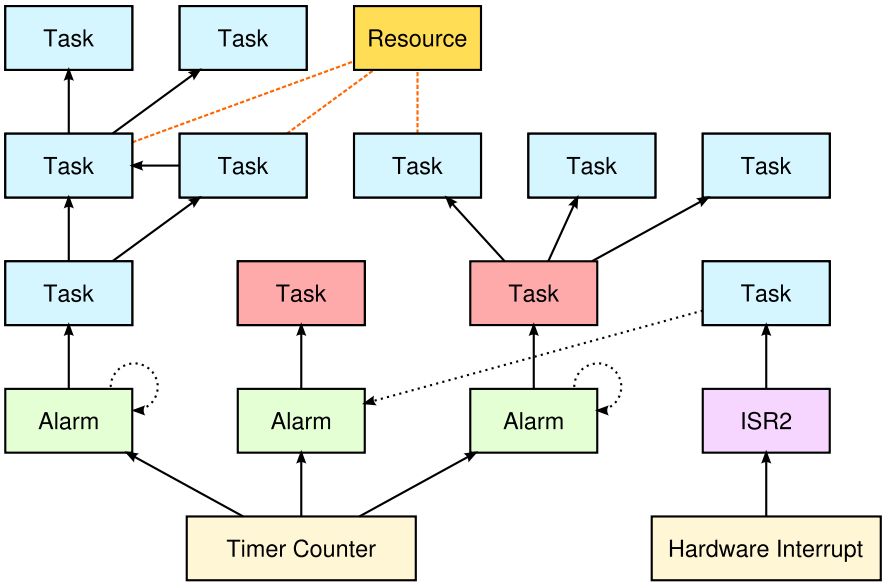


Figure 5.4 – System structure of the Copter sample application. Solid arrows show possible control flows, beginning from hardware events at the bottom. Dotted arrows indicate arming of alarms (periodic or through system calls). Orange dashed lines connect a resource to all tasks which can claim it. Blue boxes represent preemtable tasks, red boxes non-preemtable tasks.

5.3 Measurement Results

Fault injection experiments covering the entire fault space are performed for each benchmark and several variants of *d*OSEK. Four result types are possible for each injected fault:

- The experiment finishes correctly with no effects on application data, as if the fault did not occur.

  If a memory location is read without using the returned value, the resulting equivalence class will never result in an error. As this can be used to artificially raise the number of such benign faults, their number is not meaningful for the evaluation.

- The fault is detected by the operating system through software or hardware measures. This includes failed assertions and traps due to invalid memory accesses (when using memory protection) or jumps to invalid code.

  As these faults are detected, the operating system can stop or reset the system to prevent the system from working incorrectly (*fail-stop*).

- The fault causes the system to hang, which is detected by the (simulated) external watchdog. Similar to hardware traps, this allows the system to stop or reset, preventing further malfunctions.

- The fault is *not* detected, causing errors in application data or wrong control flow decisions (silent data corruption). These faults are dangerous as the system continues working in a corrupted state.

As only the remaining silent data corruptions are a threat to the system, these faults are used to evaluate the robustness of the benchmarks. Note that absolute SDC numbers between *different* benchmarks cannot be compared as they are dependent on execution time, among other factors.

Additionally, the results for injections in the instruction pointer are valid for the specific tested system/binary only. Even minimal changes to the system's code or a different compiler version can offset the addresses of large parts of the binary. As bit flips in the Instruction Pointer (IP) only cause jumps by powers of two, this can lead to completely different errors.

5.3.1 Task Dispatch Micro-Benchmark

The amounts of silent data corruption for the Task Dispatch micro-benchmark resulting from the fault injection experiments are shown in Figure 5.5 (by injection type), Figure 5.6 (by injection code function), Figure 5.7 (by injection location) and Figure 5.8 (by checkpoint).

Comparing encoded and unencoded variants in Figure 5.5 and Table 5.1, it can be seen that errors in RAM are completely detected through the use of arithmetic coding. The effectiveness of memory protection is shown as well, especially for the instruction pointer as 73 percent of the original IP errors are detected. Compared to the unencoded variant without MPU protection, the fully encoded and protected variant reduces the amount of SDC occurrences by 82 percent.

The distribution of remaining SDC occurrences by injection function in Figure 5.6 shows that the unencoded variant is vulnerable in the system call implementations,

Task Dispatch		SDC occurrences			
Encoded	MPU	Memory	Registers	IP	Total
yes	yes	0	2395	195	2590
yes	no	0	3184	710	3894
no	yes	10 468	1988	284	12 740
no	no	10 494	3240	1049	14 783

Table 5.1 – SDC occurrences for the Task Dispatch micro-benchmark.

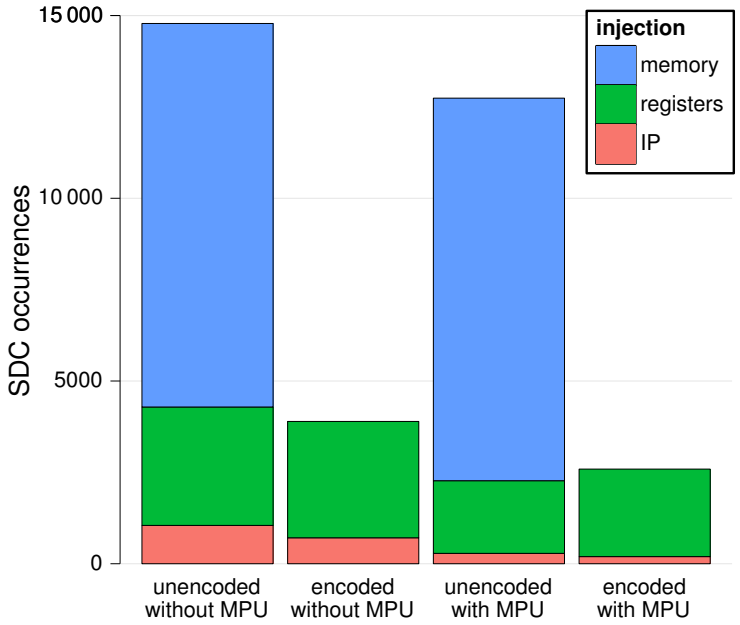


Figure 5.5 – SDC occurrences for the Task Dispatch micro-benchmark with and without MPU and encoded system structures. Results cover the entire fault space and are colored by injection type.

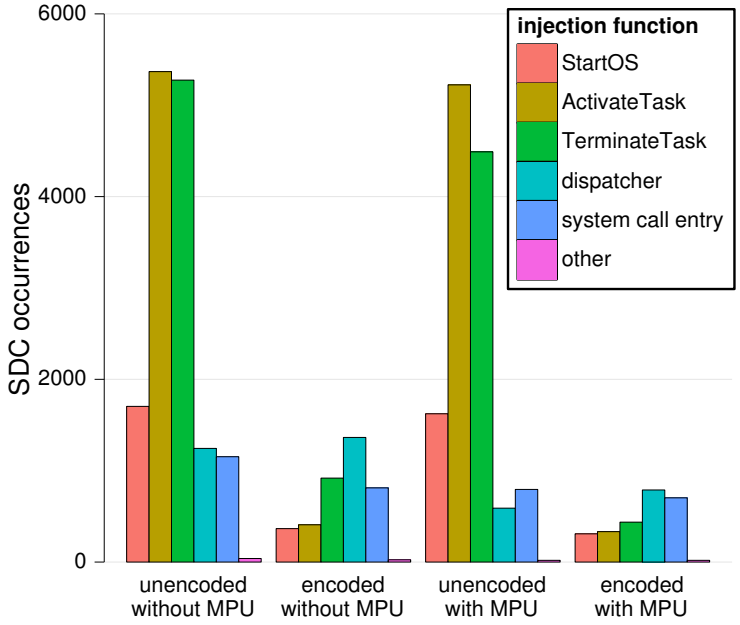


Figure 5.6 – SDC occurrences for the Task Dispatch micro-benchmark with and without MPU and encoded system structures. Results are colored by injection function (multiple instances of the same function are grouped).

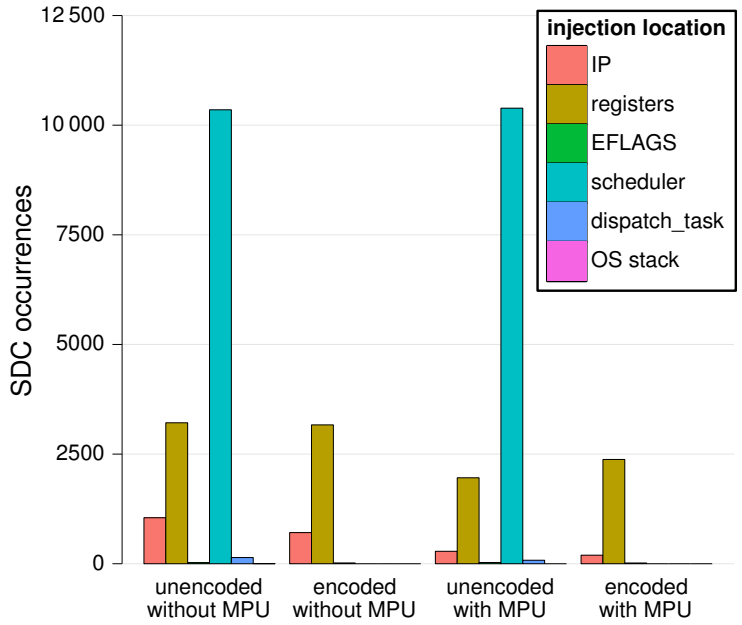


Figure 5.7 – SDC occurrences for the Task Dispatch micro-benchmark with and without MPU and encoded system structures. Results are colored by memory location in which the error is injected.

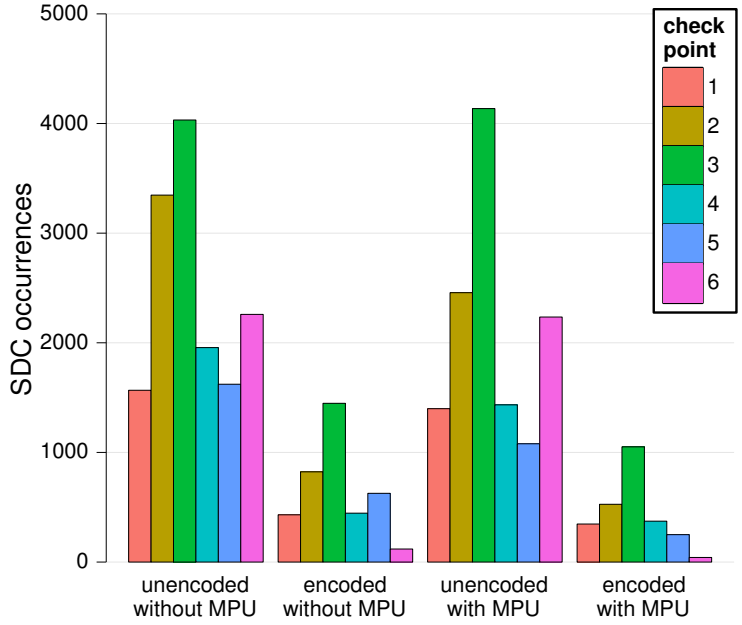


Figure 5.8 – SDC occurrences for the Task Dispatch micro-benchmark with and without MPU and encoded system structures. Results are colored by the experiment checkpoint at which the error is detected.

which include the (unprotected) scheduler. In contrast, the encoded variant is significantly less vulnerable in the system calls, with the majority of silent data corruption appearing in the dispatcher and system call entry code. These remaining vulnerabilities occur when errors are introduced after decoding or checking an encoded value but before this (briefly unprotected) value is written to the necessary hardware registers/locations. Hardware support to process protected values might be a way to further improve robustness in this area.

Grouping the SDC occurrences by injection location, it can be seen that most of the remaining corruptions in the encoded variant result from errors in register values. The unencoded variant shows that the scheduler without arithmetic coding is a major source of errors. Register error levels remain relatively stable between the variants since the added data and control flow checks detect many register errors while the higher code complexity increases likelihood of such errors.

Comparing SDC occurrences by checkpoint in Figure 5.8 between the encoded and unencoded variant, it can be seen that the distribution between checkpoints remains mostly the same. However, checkpoint 6 is often reached erroneously in the unencoded variant. As shown in Figure 5.2, this corresponds to an activation of the idle task. In the unencoded task list implementation, a single bit flip in the priority value can change a ready task to suspended. If this is the only ready task, the bit flip would result in an erroneous dispatching of the idle task. As seen in Figure 5.8, such errors in the encoded priorities are reliably detected by the encoded scheduler.

In conclusion, the encoded data structures reduce 74 percent of silent data corruptions despite the added complexity. Futher enabling MPU protection detects 33 percent of the remaining SDC occurrences.

5.3.2 Interrupt and Alarm Micro-Benchmark

The amounts of silent data corruption for the Interrupt/Alarm micro-benchmark are shown in Figure 5.9 (by injection type), Figure 5.10 (by injection code function), Figure 5.11 (by injection location) and Figure 5.12 (by checkpoint).

Interrupt/Alarm		SDC occurrences			
Encoded	MPU	Memory	Registers	IP	Total
yes	yes	0	896	406	1302
yes	no	0	1210	844	2054
no	yes	5294	562	403	6259
no	no	4943	899	745	6587

Table 5.2 – SDC occurrences for the Interrupt/Alarm micro-benchmark.

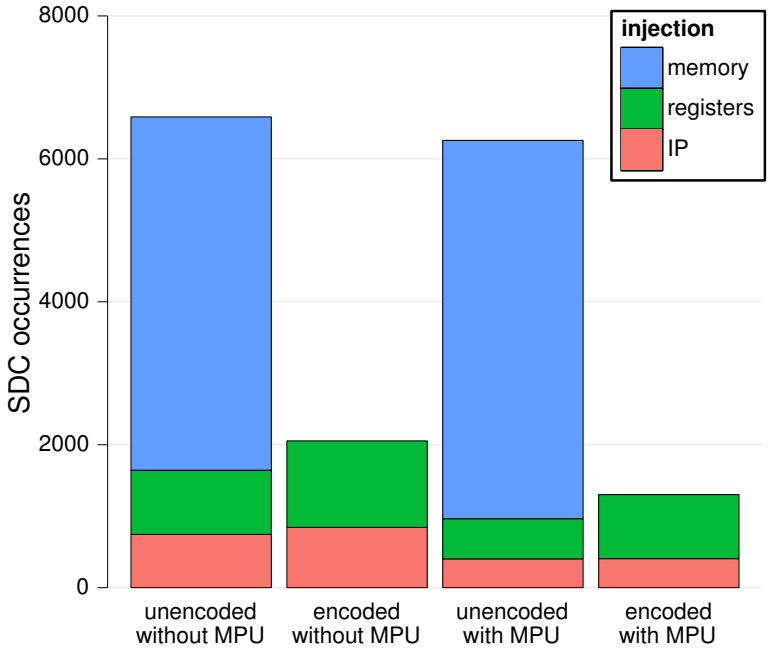


Figure 5.9 – SDC occurrences for the Interrupt/Alarm micro-benchmark with and without MPU and encoded system structures. Results cover the entire fault space and are colored by injection type.

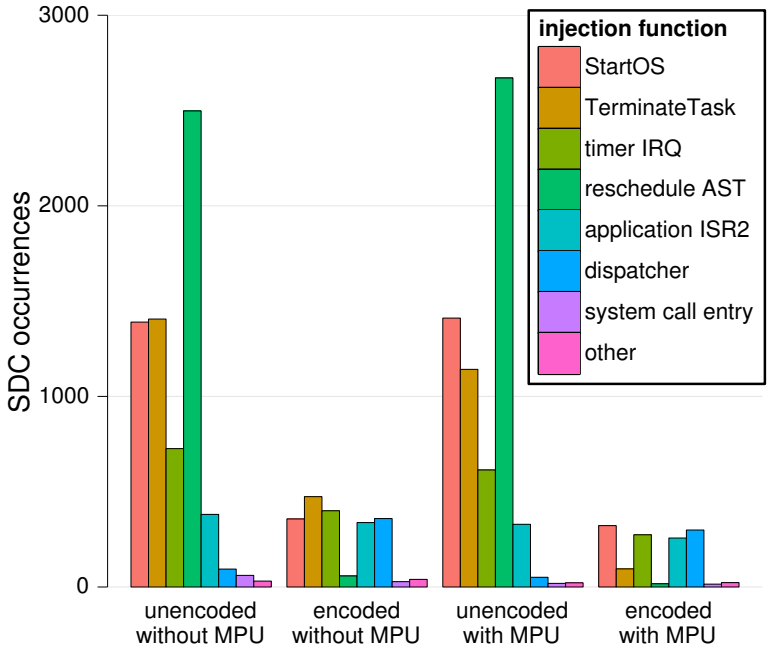


Figure 5.10 – SDC occurrences for the Interrupt/Alarm micro-benchmark with and without MPU and encoded system structures. Results are colored by injection function (multiple instances of the same function are grouped).

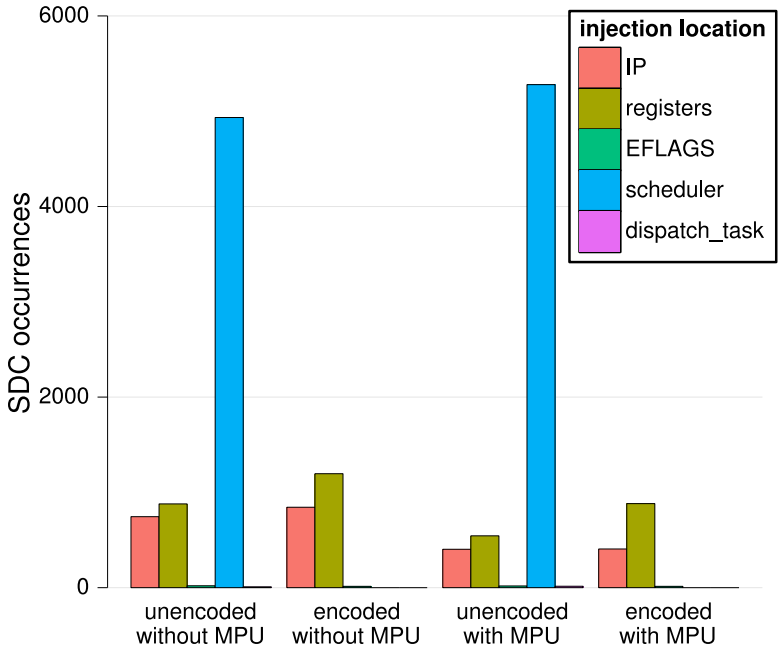


Figure 5.11 – SDC occurrences for the Interrupt/Alarm micro-benchmark with and without MPU and encoded system structures. Results are colored by memory location in which the error is injected.

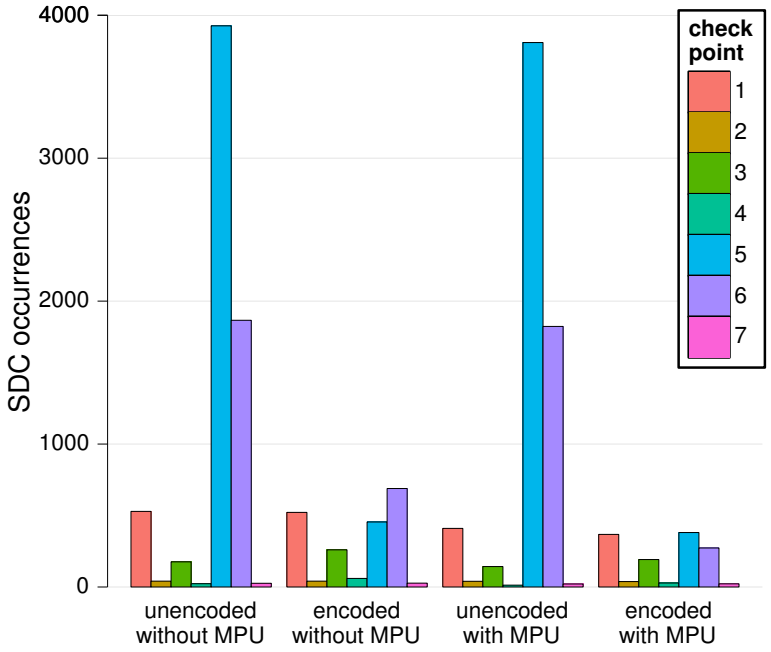


Figure 5.12 – SDC occurrences for the Interrupt/Alarm micro-benchmark with and without MPU and encoded system structures. Results are colored by the experiment checkpoint at which the error is detected.

As in the Task Dispatch micro-benchmark, Figure 5.9 and Table 5.2 show that all errors in RAM are detected by arithmetic coding. Similarly, memory protection detects 52 percent of the IP errors remaining in the encoded variant. The encoded variant with MPU protection reduces the amount of SDC occurrences by 80 percent compared to the unencoded, unprotected variant.

The SDC occurrences by injection function (Figure 5.10) show that the encoded variant eliminates vulnerabilities in all functions using the scheduler (*TerminateTask* system call and the reschedule AST). Remaining corruption is mainly caused by low-level functions like the dispatcher and the timer interrupt. In contrast to the task dispatch benchmark, the system call entry function causes few errors, as fewer explicit system calls are used in this benchmark. Errors in the application ISR2 handler remain constant as these callback handlers are not protected in the current implementation of *d*OSEK. The slightly higher absolute error value for the reschedule AST in the unencoded variant *with* MPU protection might be caused by subtle differences between the generated systems, such as the ordering of tasks and system calls in ROM or the resulting absolute code addresses and padding.

The same effect can also be seen when grouping by injection location (Figure 5.11) in the scheduler data structure, which is used by the reschedule AST. The encoded variant shows no remaining scheduler errors. When MPU protection is enabled, 69 percent of remaining errors are caused by bit flips in registers while the rest result from faults in the instruction pointer.

Figure 5.12 shows SDC occurrences by checkpoint. The two large values for checkpoints 5 and 6 in the unprotected variant indicate erroneous activations of tasks (compare Figure 5.3). This is caused by the fact that at the start of the benchmark, all tasks are suspended and thus any bit flip in the unprotected task priority values will change them to ready. The encoded variant does not suffer from this problem as such bit flips are detected.

The Interrupt/Alarm micro-benchmark shows that encoded data structures prevent 69 percent of silent data corruptions even though this benchmark depends more on the underlying hardware architecture, which cannot be easily protected. With additional MPU protection, 37 percent of the remaining errors are detected.

5.3.3 Copter Sample Application

The amounts of silent data corruption for the Copter sample application by injection type are presented in Table 5.3 and Figure 5.13. The results are shown on a logarithmic scale due to the wide range of values. The gray bars represent the sum of the individual error types, which are shown by the smaller, colored bars. Due to the logarithmic scale, the height of a gray sum bar is not the sum of the heights of the individual error type bars.

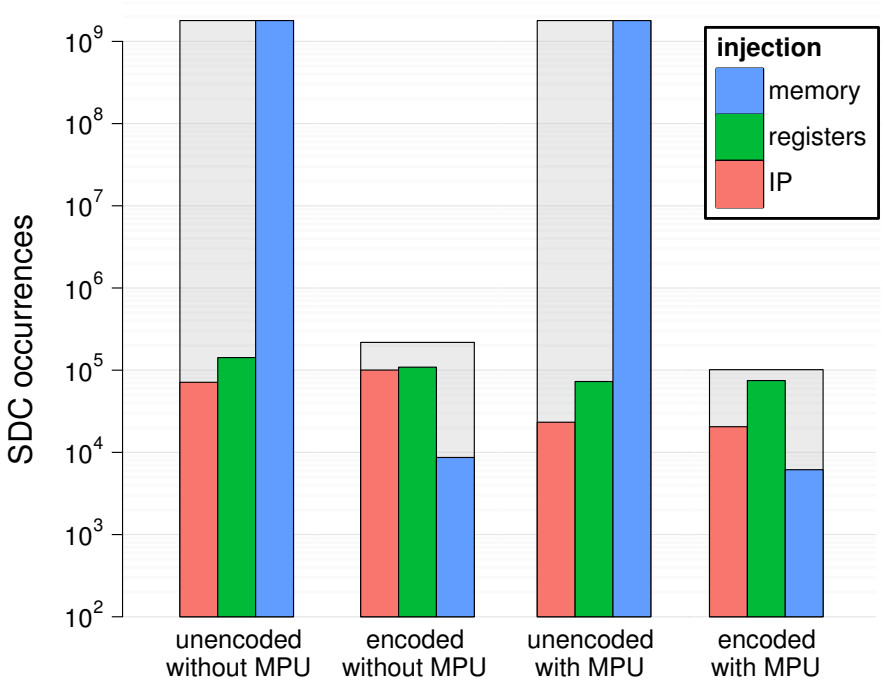


Figure 5.13 – SDC occurrences for the Copter sample application with and without MPU and encoded system structures. Occurrence counts are shown on a **logarithmic scale** because of the wide range of values. Figure 5.14 shows the smaller values on a linear scale. Results cover the entire fault space and are colored by injection type. Gray, wide bars behind the individual injection types show the sum of all types (not equal to the sum of bar heights on a logarithmic scale).

Copter Sample		SDC occurrences			
Encoded	MPU	Memory	Registers	IP	Total
yes	yes	6156	74 779	20 525	101 460
yes	no	8671	108 821	100 343	217 835
no	yes	1 792 564 837	72 846	23 284	1 792 660 967
no	no	1 793 022 678	142 299	71 242	1 793 236 219

Table 5.3 – SDC occurrences for the Copter sample application.

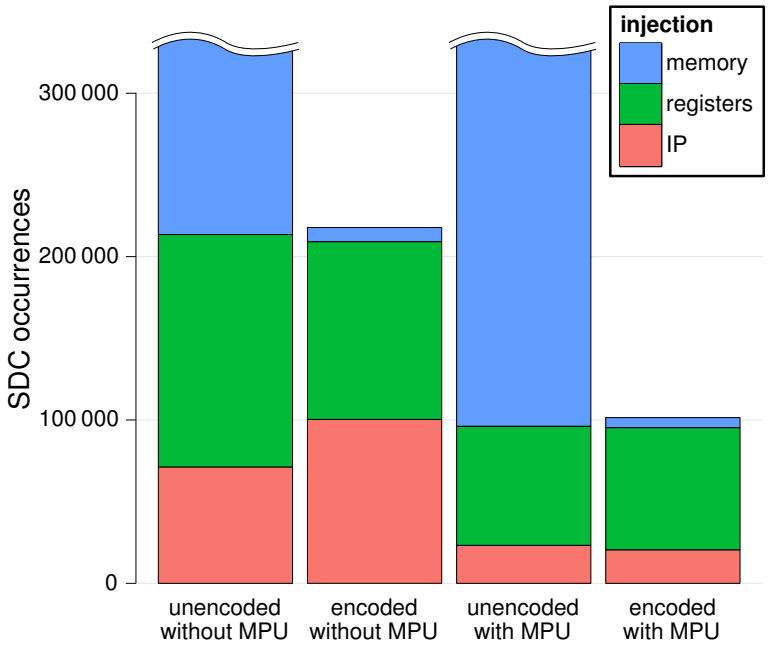


Figure 5.14 – SDC occurrences for the Copter sample application with and without MPU and encoded system structures. Figure 5.13 shows the same data on a logarithmic scale while this graph is **cut off** to show the smaller values. Results cover the entire fault space and are colored by injection type.

It can be seen that the unencoded variants have **four orders of magnitude** more SDC occurrences than the encoded variants. While almost all corruptions result from memory errors in the unprotected systems, the opposite is the case for the protected variants, where memory errors are almost non-existent. This can be seen in Figure 5.14, which shows the smaller values on a linear scale.

Enabling memory protection removes over half of the remaining SDC occurrences in the encoded variant, which is mostly due to instruction pointer errors being reduced by 80 percent.

As shown in Figure 5.15, one reason for the high amount of corruptions resulting from memory errors is the extremely vulnerable implementation of the scheduler. The unencoded scheduler can easily be corrupted as it uses 8 bits for each task priority with no checks at run-time. This vulnerability, which is also clearly seen in the micro-benchmarks, is aggravated by the much longer run-time of the sample application.

The other major vulnerability of the unencoded variants shown in Figure 5.15 are the counter and alarm values. Since the counter values are used and updated in each counter tick, errors will propagate until wrong system behavior results eventually. Similarly, the alarm state values remain unprotected in memory for the entire benchmark duration and can result in wrong activations if corrupted.

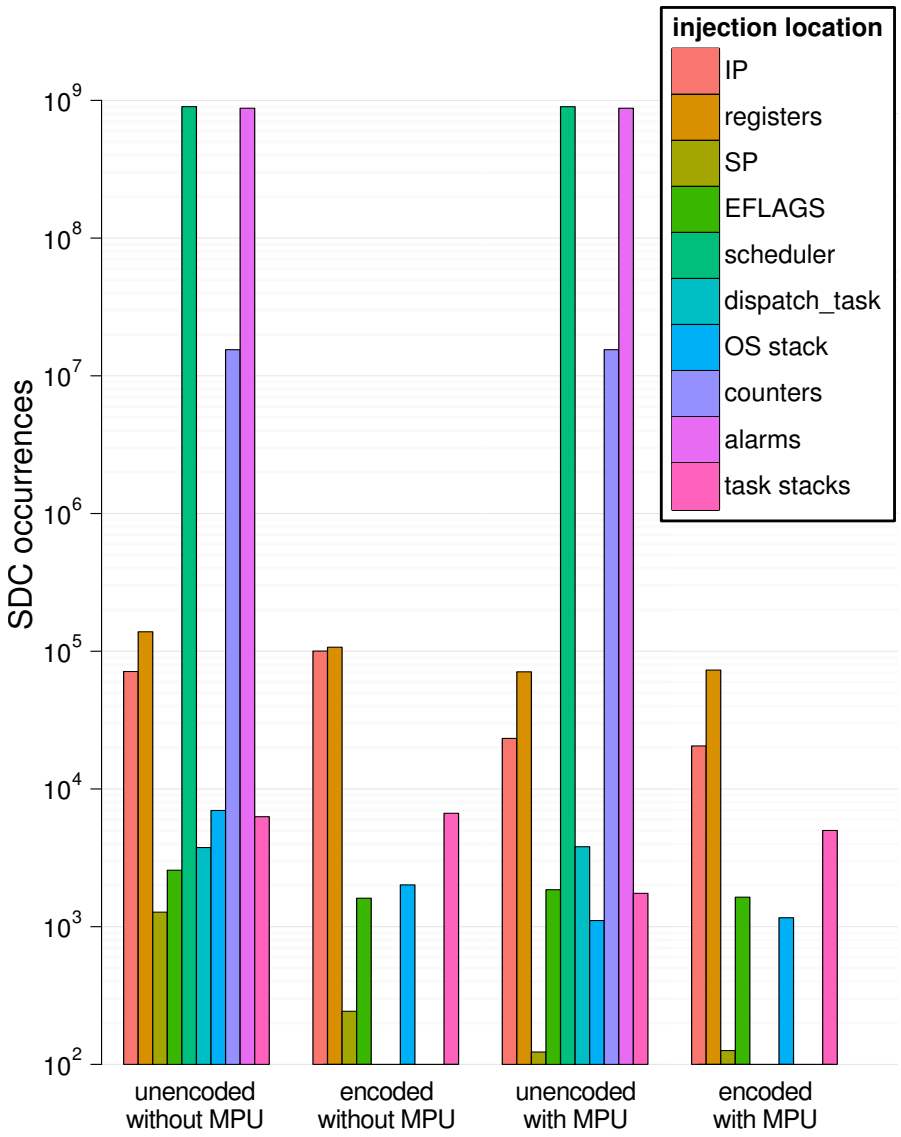


Figure 5.15 – SDC occurrences for the Copter sample application with and without MPU and encoded system structures. Occurrence counts are shown on a **logarithmic scale** because of the wide range of values. Figure 5.16 shows the smaller values on a linear scale. Results cover the entire fault space and are colored by injection location.

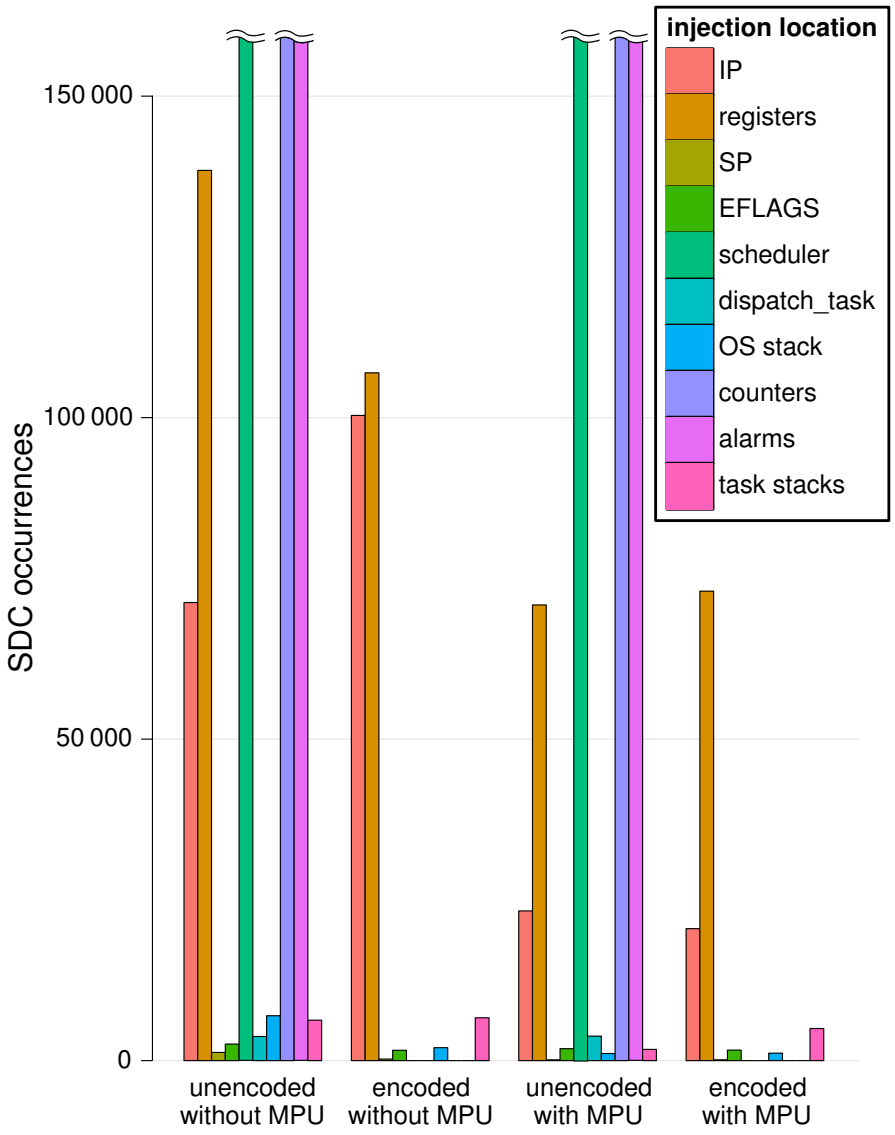


Figure 5.16 – SDC occurrences for the Copter sample application with and without MPU and encoded system structures. Figure 5.15 shows the same data on a logarithmic scale while this graph is **cut off** to show the smaller values. Results cover the entire fault space and are colored by injection location.

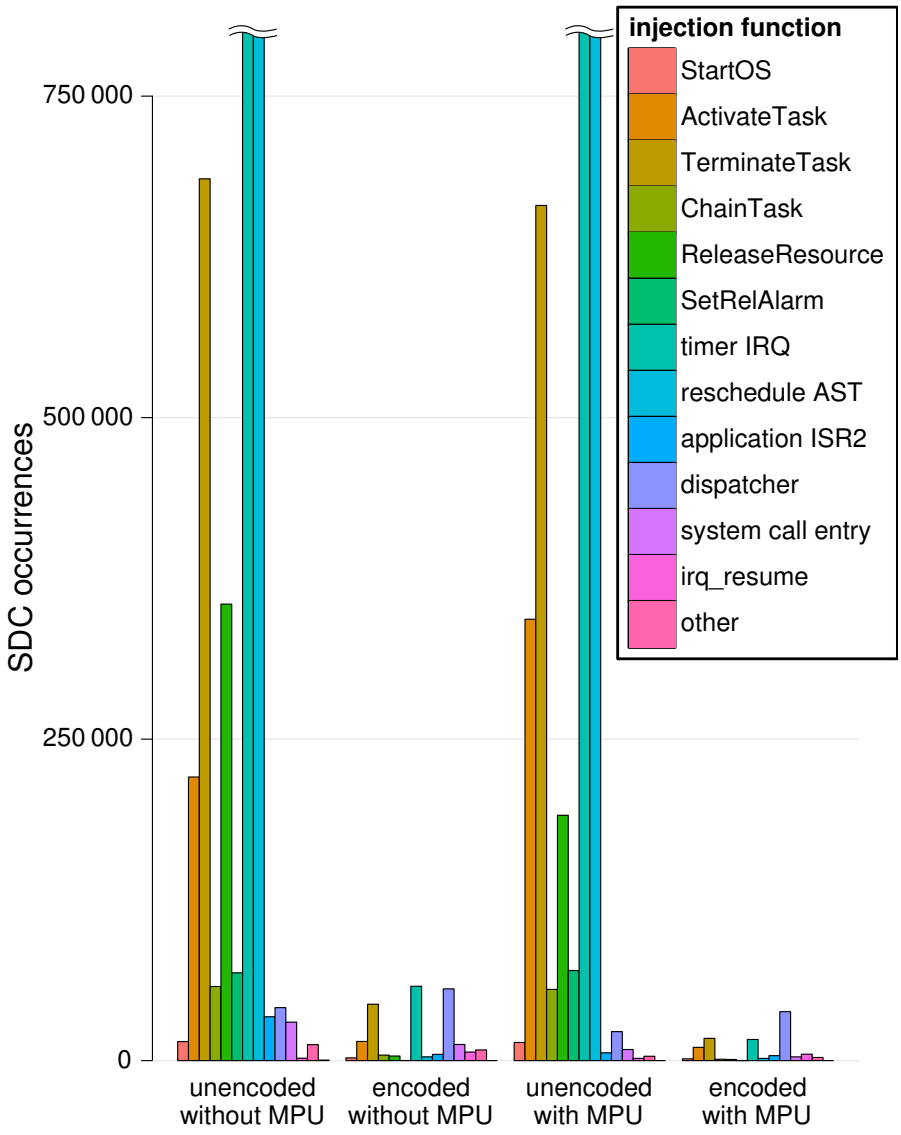


Figure 5.17 – SDC occurrences for the Copter sample application with and without MPU and encoded system structures. This graph is **cut off** to only show the smaller values. Results cover the entire fault space and are colored by injection function.

The encoded variants prevent errors from corruption in the scheduler and counter/alarm values. Remaining errors, as shown on a linear scale in Figure 5.16, are mostly caused by register corruption. The majority of the small amount of memory errors is due to corruption in the stacks of suspended tasks. In this case, the context saved on stack (see Figure 4.6) can be corrupted by memory errors. Adding a checksum over this context could prevent such errors.

Figure 5.17 groups SDC occurrences by injection function. Again, only the smaller result values are shown as both the timer interrupt handler and the reschedule AST are extremely vulnerable in the unencoded variants. This corresponds to the vulnerabilities in the counter/alarm values and the scheduler, respectively.

The results for the other injected functions are similar to the micro-benchmarks. Unencoded variants are mainly vulnerable in system calls due to the unprotected scheduler. These errors are prevented in the encoded system with the dispatcher being the single biggest source of silent data corruptions.

The Copter sample benchmark shows that encoded data structures are very effective at reducing silent data corruption with four orders of magnitude less SDC occurrences. If only memory errors are considered, the difference spans more than five orders of magnitude, as these errors are almost completely eliminated. Using memory protection, the remaining SDC occurrences are further reduced by 53 percent.

5.3.4 Architecture-specific Error Reduction

Beside the architecture-independent protection mechanisms, such as arithmetic coding, the Intel i386 variant of *d*OSEK uses several architecture-specific methods to avoid and detect errors. Figure 5.18 and Table 5.4 compare the robustness of three such methods in various combinations.

The first variant (A) is the Copter sample application with encoding and memory protection, but using software interrupts for system calls (see Section 4.5.2.1). Removing architectural indirection using the `sysenter` and `sysexit` instructions

Copter Sample	SDC occurrences			
	Memory	Registers	IP	Total
Variant A	156 369	79 232	23 221	258 822
Variant B	58 813	88 506	24 944	172 263
Variant C	10 278	91 726	19 802	121 806
Variant D	6156	74 779	20 525	101 460

Table 5.5 – SDC occurrences of the copter sample application for the *d*OSEK variants with varying architecture-specific error reduction methods listed in Table 5.4.

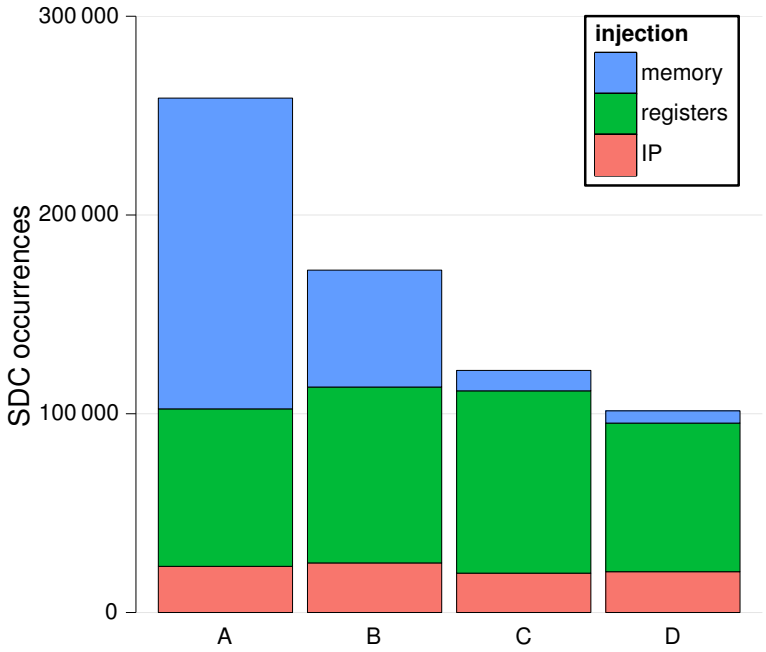


Figure 5.18 – SDC occurrences of the Copter sample application for the *d*OSEK variants with varying architecture-specific error reduction methods listed in Table 5.4. All shown variants are built with encoded data structures and memory protection. Results cover the entire fault space and are colored by injection type.

	A	B	C	D
sysenter/sysexit system calls and dispatch		•	•	•
odd parity checks for stack/instruction pointers			•	•
XOR checksum over IRQ context				•

Table 5.4 – Enabled architectural improvements for the *d*OSEK variants shown in Figure 5.18.

as described in Section 4.5.2.2 results in a significant reduction of SDC occurrences by one third (variant B). By using registers instead of the stack for system calls and dispatching, memory errors are almost halved while register errors only increase by 10 percent.

At this point, the architecture-specific stack and instruction pointers were identified as vulnerable values. Testing their integrity by ensuring and checking for odd parity (see Section 3.4.1) further reduces silent data corruption (variant C).

Finally, corruption from errors in the stored context of interrupt handlers was observed and prevented using a XOR checksum in variant D. This is the *d*OSEK variant used for all other evaluations.

Combining these protection mechanisms eliminated 60 percent of the SDC occurrences remaining after encoding. As the mechanisms mainly avoid or detect errors occurring in memory, register and instruction pointer error levels remain stable while 96 percent of memory errors are prevented.

5.3.5 Comparison with ERIKA Enterprise

In order to compare the robustness of *d*OSEK, fault injection campaigns were also performed for an existing real-time operating system: *ERIKA Enterprise*[12], the first open-source RTOS certified to be fully OSEK compliant, was chosen for this comparison. It is in active use by various companies in the automotive industry.

The free availability of ERIKA enabled a port to the Intel i386 architecture, which is used for the evaluation, but was not supported until now. In ERIKA, only a small hardware abstraction layer interacts with the hardware while most parts of the system are architecture-independent. For this abstraction layer, the low-level hardware initialization and interrupt handling code was adapted from *d*OSEK. The actual kernel of ERIKA is then used without modifications and the port can easily be tested with the provided ERIKA test-suite.

The Copter sample application is used for the comparison between *d*OSEK and ERIKA. To prevent vulnerabilities due to unnecessary features, ERIKA is configured with the minimal feature set required for the application.

Two ERIKA variants are evaluated, one without error detection and one with the OSEK *ErrorHook* enabled. This hook is called whenever a system call would return an error code, thus allowing the system to detect some inconsistencies.

As a baseline for the comparison to ERIKA, an “unprotected” *d*OSEK variant without memory protection, encoded data structures or architecture-specific protection methods is evaluated (variant B in the previous section). Finally, two *d*OSEK variants which use all these protection methods are tested, one with and one without memory protection (variant D in the previous section).

[12] `http://erika.tuxfamily.org/`

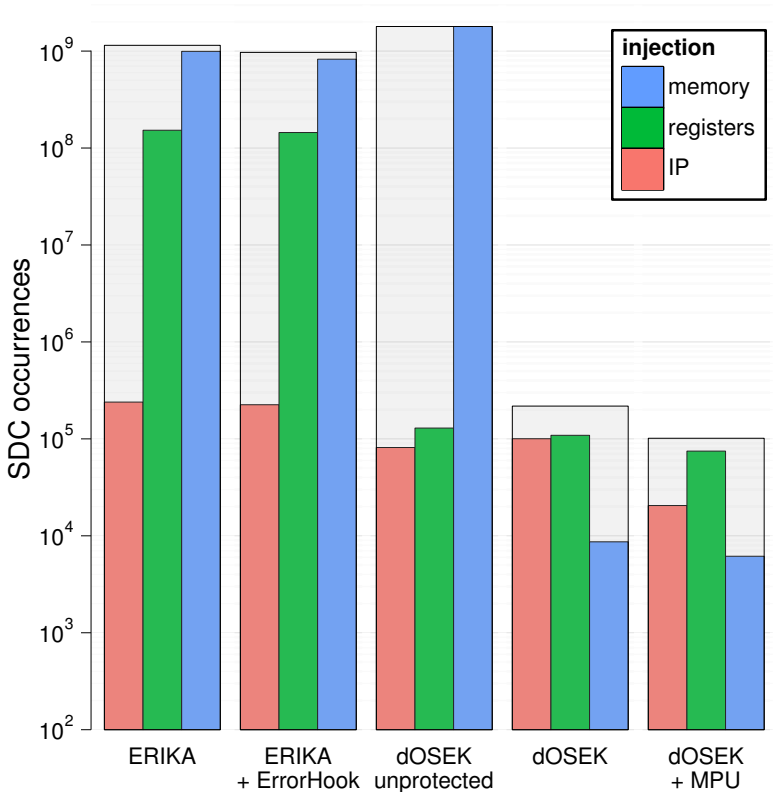


Figure 5.19 – SDC occurrences of the Copter sample application for ERIKA and *d*OSEK variants. Occurrence counts are shown on a **logarithmic scale** because of the wide range of values. Results cover the entire fault space and are colored by injection type. Gray, wide bars behind the individual injection types show the sum of all types (not equal to the sum of bar heights on a logarithmic scale).

Table 5.6 and Figure 5.19 show the results of the fault injection campaigns.

In the ERIKA variants, memory errors cause the majority of SDC occurrences. Enabling the ErrorHook can detect only 15 percent of these occurrences as many errors result in a system state which is different yet valid according to the OSEK specification. Also, checks are only performed by system calls and cover only the part of the system affected by the specific call.

The fully unprotected *d*OSEK variant is less robust than the standard ERIKA system, with 56 percent more SDC occurrences. As described in Section 5.3.3, memory errors occur mainly in the scheduler and the counter/alarm values. The unencoded *d*OSEK scheduler always uses 8 bits for each task priority while ERIKA uses more compact bit-fields to store the scheduler state. Since every additional unprotected bit leads to more corruptions, this *d*OSEK variants shows more memory errors than ERIKA.

Unlike ERIKA, this unprotected *d*OSEK variant is also using privilege separation with the resulting overhead of context switching for system calls. While this variant is very vulnerable to memory errors, it has three orders of magnitude less SDC occurrences resulting from register and instruction pointer errors. This improvement can be attributed to the reduction of indirection throughout the system.

Finally, the protected *d*OSEK variants are significantly more robust than ERIKA as corruption from memory errors is reduced almost completely. The encoded *d*OSEK variant with memory protection is **four orders of magnitude** more robust than both ERIKA variants. If only memory errors are considered, the reduction even exceeds **five orders of magnitude**. While the variant *without* memory protection shows more than twice the number of SDC occurrences, this is still significantly better than all unencoded variants.

5.4 Overhead

While effectively preventing silent data corruption, the error avoidance and detection methods of *d*OSEK introduce overhead in code size, data size and execution time. This section analyzes the overhead of the different system variants.

5.4.1 Code Size Overhead

Minimal code size has never been a goal for *d*OSEK, which is designed for maximum robustness. For this reason, system calls are duplicated and specialized for each call site despite the resulting size increase. This means the code size grows with the number of call sites for each system call, which is not the case for other operating systems like ERIKA. However, both systems increase in code size when system objects such as tasks and alarms are added.

As the kernel code consists mostly of system calls, Figure 5.20 presents the *average* size of each system call for the evaluated systems. In the case of *d*OSEK this can be measured directly, as each system call instance is generated into a separate

Copter Sample	SDC occurrences			
	Memory	Registers	IP	Total
ERIKA	993 884 279	152 619 027	239 582	1 146 742 888
ERIKA + ErrorHook	824 910 242	144 524 959	224 978	969 660 179
*d*OSEK unprotected	1 793 060 211	129 268	81 462	1 793 270 941
*d*OSEK	8671	108 821	100 343	217 835
*d*OSEK + MPU	6156	74 779	20 525	101 460

Table 5.6 – SDC occurrences of the Copter sample application for ERIKA and *d*OSEK variants by injection type.

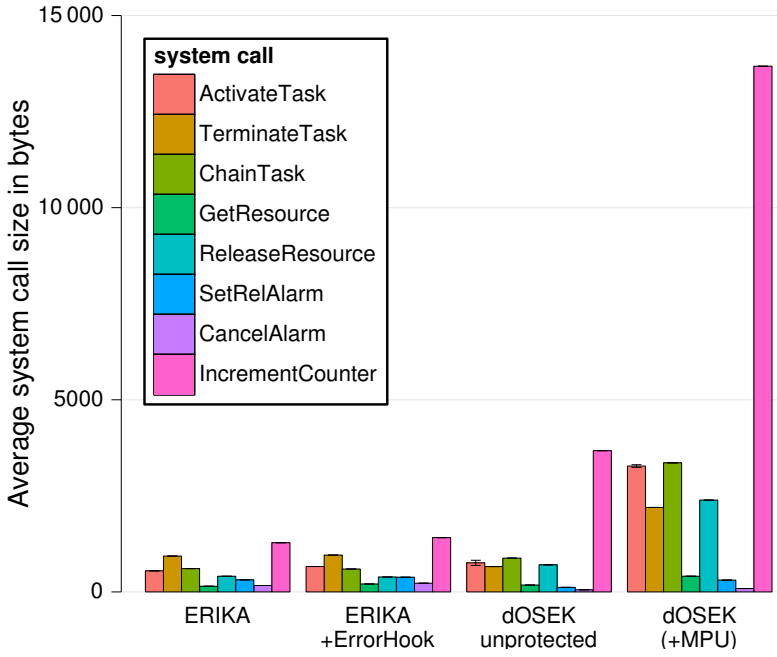


Figure 5.20 – Average code size of system calls in bytes by *d*OSEK and ERIKA variant. Standard deviation (σ) shown by error bars.

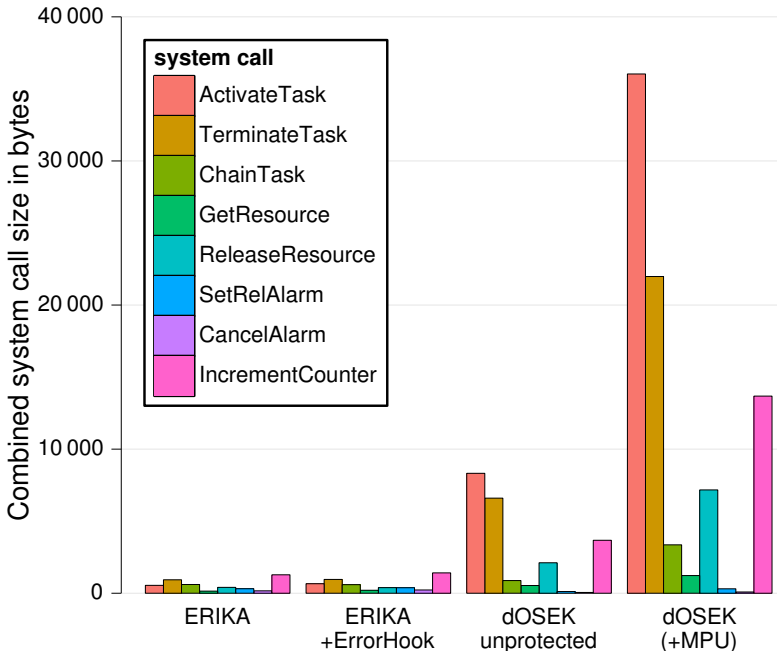


Figure 5.21 – Combined code size of system calls in bytes by *d*OSEK and ERIKA variant.

Copter Sample	ROM Size		RAM Size	
	Kernel	Total	Kernel	Total
ERIKA	3782	25 306	512	53 852
ERIKA + ErrorHook	4447	26 034	529	53 872
*d*OSEK unprotected	16 579	154 040	111	61 464
*d*OSEK	60 028	209 476	172	61 464
*d*OSEK + MPU	60 097	209 476	172	61 464

Table 5.7 – ROM and RAM requirements of the Copter sample application for ERIKA and *d*OSEK variants.

linear block of code. For a comparison with ERIKA, which uses calls to shared functions, the size of all visited functions during a system call is summed up. For completeness, the *IncrementCounter* bar shows the size of the counter increment operation although it is not an actual system call. It is called by the timer interrupt and can activate tasks when alarms reach their trigger value. Since enabling memory protection has no effect on system call overhead the two protected *d*OSEK variants are combined.

The results show that the unprotected *d*OSEK variant has system call code sizes comparable to both ERIKA variants. Only the counter operation increases by a larger factor of 2.9, caused by inlined scheduler operations for all possible alarm task activations.

The protected *d*OSEK variants show significantly increased code sizes for all system calls involving scheduler operations, such as *ActivateTask* and *ReleaseResource*. This is caused by the usage of arithmetic coding, as each operation performed on encoded values is composed of many individual arithmetic operations. Compared to ERIKA, the maximum size increase of these calls is by a factor of 6 while other system calls only show an insignificant absolute overhead. The counter increment implementation is also much bigger because the inlined scheduler operations are performed using arithmetic encoding.

Since kernel code size grows with every call site of a system call, Figure 5.21 shows the *combined* code size for each system call. As this does not apply for ERIKA, its values are identical to Figure 5.20. In *d*OSEK, the majority of the kernel ROM space is used for the *ActivateTask* and *TerminateTask* system calls as they are called frequently. This illustrates the fact that the *d*OSEK kernel grows with each additional system call. In the worst case (*ActivateTask*), the combined code size is 65.8× as big as for ERIKA.

The resulting kernel sizes and total ROM and RAM requirements of the systems evaluated in the previous section are shown in Table 5.7. Due to the duplicated system calls, the kernel code size of the unprotected *d*OSEK variant is 4.4 times as

big as the unprotected ERIKA kernel. The resulting total ROM size for this variant is 6.1× the size of ERIKA.

Through the added protection, the kernel code size of encoded *d*OSEK variants is 3.6× the size of the unencoded kernel. However, since the kernel is small compared to the total system size in *d*OSEK, the total ROM requirement for the encoded variants is only 1.4 times bigger. Compared to the unprotected ERIKA system, the ROM size of the fully protected *d*OSEK is 8.3 times as large.

5.4.2 Data Size Overhead

As an ANB-encoded value uses 32 bits for a 16 bit original value, arithmetic coding bears a data size overhead of 100 percent. However, compared to the kernel or application stack sizes this increase is negligible. This can also be seen in Table 5.7, where the total RAM needed by all *d*OSEK variants remains constant despite arithmetic coding increasing the kernel RAM size by 35 percent.

Since *d*OSEK avoids potential errors by minimizing the dynamic system state, the (encoded) *d*OSEK kernel requires only about one third the amount of RAM used by the ERIKA kernel. When comparing the total system RAM requirement, the overhead of *d*OSEK is only 14 percent.

5.4.3 Run-Time Overhead

The run-time of system operations is of interest for real-time operating systems as (ISR2) interrupts are delayed during their execution. As this influences the maximum system response time, the run-time overhead of *d*OSEK is evaluated.

The comparison in Figure 5.22 shows the executed instruction count for different system calls, as derived from the sample application's error-free trace. Average run-times are presented since the run-time of system calls varies by parameter values, call site and run-time system state.

The run-times for the unprotected *d*OSEK variant are similar to the results for ERIKA. While most system calls execute slighty longer, the *SetRelAlarm* call and the timer interrupt are faster in *d*OSEK. As expected, system calls which include a scheduling operation, such as *ActivateTask* and *TerminateTask*, show that the *d*OSEK scheduler implementation is not optimized for execution time. The counter update operation exhibits the biggest variation in run-time for both systems, as it runs considerably longer if an alarm triggers and tasks must be activated.

The protected *d*OSEK variants show significantly increased run-times for most system calls due to arithmetic coding. System calls which require a rescheduling execute 3.8–4.1× as long as in the unprotected variant. While substantial, the relative robustness improvement is much greater than this relative run-time overhead. If the scheduler is not involved, system call run-time increases only by between 56 percent

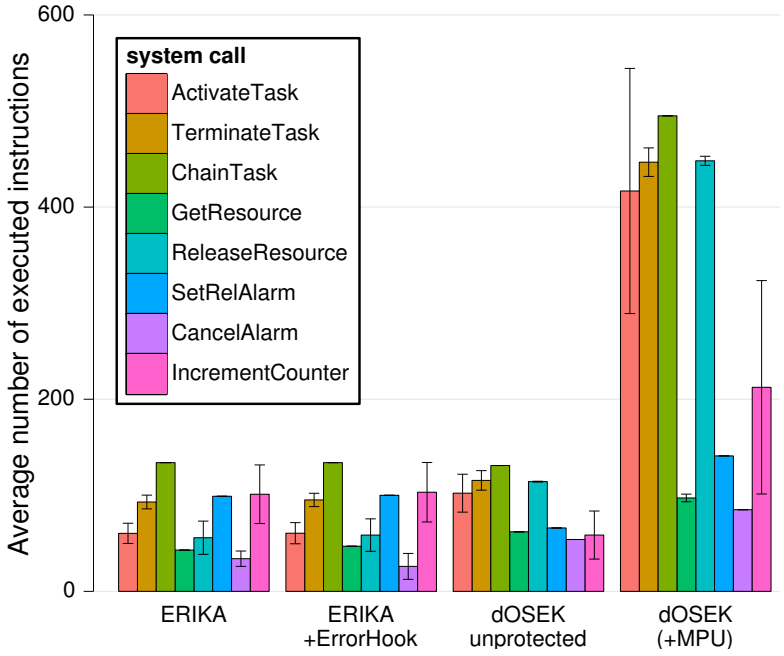


Figure 5.22 – Average run-time of system calls in executed instructions by *d*OSEK and ERIKA variant. Standard deviation (σ) shown by error bars.

and 114 percent as fewer encoded operations are needed. Compared to the unprotected ERIKA variant, the system call with the largest overhead is *ReleaseResource*, which executes longer by a factor of 8.

Note that due to the static, loop-free design of *d*OSEK all system calls execute in $\mathcal{O}(1)$, i. e. a constant upper bound can be determined before run-time. Additionally, the actual calculation of such worst-case execution times is eased by the loop-free control flow and absence of indirect memory accesses.

5.5 Possible Optimizations

Several optimizations and alternative implementations are possible based on the evaluation results and current design choices.

The comparison of the unprotected *d*OSEK system shows that the overhead of performing system calls with privilege isolation negates the benefits of reduction of indirection in other aspects. In the future, a variant with memory protection, but without privilege isolation can be evaluated. This would imply to always run in supervisor mode, vastly simplifying the system call entry and exit control flow. Inlining of this system call entry and exit code might then be possible, further reducing indirections.

Alternatively, it might be beneficial to further isolate the application from the operating system. For this, all interrupt and alarm handlers could be run in user-mode instead of supervisor mode.

As a variant to the current approach, the desired system call could be passed as an encoded index to a table of system call entry addresses stored in ROM. This would replace directly specifying a code address to start, which can be corrupted easily.

To detect lost updates, the ANB-coding of *d*OSEK can be extended to include dynamic signatures (D timestamp values). This requires a robust way to keep track of the current timestamps and to update the encoded values whenever the associated timestamp changes.

The usage of checksums can also be extended to the saved context of preempted tasks to detect corruption. Additionally, other checksum algorithms like *CRC32* could be used.

Each of these optimizations needs to be evaluated through fault injection campaigns to verify the actual gains or losses in robustness are as expected.

5.6 Summary

The robustness of the *d*OSEK operating system has been evaluated in this chapter through extensive fault injection campaigns. Single bit flips (on the ISA level) have been injected in micro-benchmarks as well as a sample application to obtain resulting SDC quantities.

The results show that potential vulnerabilities in the encoded *d*OSEK system are reduced by up to four orders of magnitude compared to unprotected systems while the space and time overhead increases by less than one order of magnitude. The comparison against the open-source ERIKA Enterprise RTOS has shown similar gains in overall robustness.

Chapter 6

Conclusion and Future Work

In this work, the *d*OSEK operating system is introduced to serve as a reliable computing base in the presence of soft-errors. The system is designed for maximum robustness through error avoidance and detection methods.

To prevent potential vulnerabilities, large portions of the system are stored in reliable read-only memory. The static OSEK design fits this approach by not including unnecessary dynamism. Furthermore, several causes of indirection have been identified and reduced to prevent their resulting vulnerabilities.

To detect remaining errors, isolation techniques like memory protection and protected data structures are used. Arithmetic coding is employed for the majority of the kernel structures to detect multiple bit faults and control flow errors. This is used to implement a reliable scheduler based on a fully encoded task queue.

To evaluate *d*OSEK, the Intel i386 variant is tested using fault injection campaigns. The experiments, which cover the entire fault space for single bit flips on the ISA level, show a reduction of silent data corruption by four orders of magnitude compared to the established ERIKA Enterprise OSEK system and unprotected *d*OSEK variants. The main trade-offs are increased code-size and execution time, which increase by less than one order of magnitude compared to the unprotected systems.

In the future, remaining weaknesses can be systematically and iteratively identified and eliminated. The ANB-coding used in *d*OSEK can be extended to include dynamic signatures (timestamps) to protect against lost updates. To further increase robustness and decrease overhead, results from a static analysis of the entire application and OS can be leveraged to tailor the generated system. The underlying hardware architecture could also be extended to support the robustness of the system, for example by providing protected mechanisms to dispatch tasks, which has been shown to be a remaining weakness. Furthermore, the system can be extended to provide reliability services to the application, such as redundant execution with automatic voting when leaving the sphere of replication.

List of Acronyms

OSEK	Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen
RCB	Reliable Computing Base
ECC	Error-Correcting Code
TMR	Triple Modular Redundancy
RTOS	Real-Time Operating System
API	Application Programming Interface
SDC	Silent Data Corruption
IP	Instruction Pointer
AST	Asynchronous System Trap
MPU	Memory Protection Unit
MMU	Memory Management Unit
ROM	Read-only Memory
RAM	Random Access Memory
ISA	Instruction Set Architecture
LLVM	Low-Level Virtual Machine
IR	Intermediate Representation
RTSC	Real-Time Systems Compiler
ABB	Atomic Base Block
LAPIC	Local Advanced Programmable Interrupt Controller
IOAPIC	Input/Output Programmable Interrupt Controller
IDT	Interrupt Descriptor Table

List of Figures

List of Tables

Bibliography

[1] J.-C. Laprie, "Dependable computing: Concepts, limits, challenges," in *FTCS-25, the 25th IEEE International Symposium on Fault-Tolerant Computing-Special Issue*, 1995, pp. 42–54.

[2] A. Avižienis, J.-C. Laprie, and B. Randell, "Dependability and its threats: a taxonomy," in *Building the Information Society*. Springer, 2004, pp. 91–120.

[3] ISO 26262-9, *ISO 26262-9:2011: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*. Geneva, Switzerland: International Organization for Standardization, 2011.

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

[5] S. Mukherjee, *Architecture Design for Soft Errors*, San Francisco, CA, USA, 2008.

[6] P. Lee and T. Anderson, "Fault tolerance," in *Fault Tolerance*, ser. Dependable Computing and Fault-Tolerant Systems. Springer Vienna, 1990, vol. 3, pp. 51–77. [Online]. Available: http://dx.doi.org/10.1007/978-3-7091-8990-0_3

[7] M. Engel and B. Döbel, "The reliable computing base-a paradigm for software-based reliability." in *GI-Jahrestagung*, 2012, pp. 480–493.

[8] 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.

[9] P. Forin, "Vital coded microprocessor principles and application for various transit systems." in *IFAC/IFIP/IFORS Symposium*, 1989, pp. 79–84.

[10] M. Hoffmann, P. Ulbrich, C. Dietrich, H. Schirmeier, D. Lohmann, and W. Schröder-Preikschat, "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)*. Miami, Florida, USA: IEEE Computer Society Press, Jan. 2014.

[11] U. Schiffel, "Hardware error detection using AN-codes." Ph.D. dissertation, Dresden University of Technology, 2011.

[12] N. Ignat, B. Nicolescu, Y. Savaria, and G. Nicolescu, "Soft-error classification and impact analysis on real-time operating systems," in *Design, Automation and Test in Europe, 2006. DATE'06. Proceedings*, vol. 1. IEEE, 2006, pp. 6–pp.

[13] D. D. Thaker, D. Franklin, J. Oliver, S. Biswas, D. Lockhart, T. Metodi, and F. T. Chong, "Characterization of error-tolerant applications when protecting control data," in *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 142–149.

[14] O. Goloubeva, M. Rebaudengo, M. Reorda, and M. Violante, "Soft-error detection using control flow assertions," in *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, Nov 2003, pp. 581–588.

[15] J. Arlat, J.-C. Fabre, and M. Rodríguez, "Dependability of COTS microkernel-based systems," *Computers, IEEE Transactions on*, vol. 51, no. 2, pp. 138–163, 2002.

[16] J. Aidemark, P. Folkesson, and J. Karlsson, "Experimental dependability evaluation of the artk68-ft real-time kernel," in *International Conference on Real-Time and Embedded Computing Systems and Applications*, 2004.

[17] P. Axer, R. Ernst, B. Döbel, and H. Härtig, "Designing an analyzable and resilient embedded operating system." in *GI-Jahrestagung*, 2012, pp. 506–520.

[18] C. M. Jeffery and R. J. Figueiredo, "A flexible approach to improving system reliability with virtual lockstep," *Dependable and Secure Computing, IEEE Transactions on*, vol. 9, no. 1, pp. 2–15, 2012.

[19] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ser. VEE '14. New York, NY, USA: ACM, 2014, pp. 201–212. [Online]. Available: http://doi.acm.org/10.1145/2576195.2576206

[20] J. Song, J. Wittrock, and G. Parmer, "Predictable, efficient system-level fault tolerance in C^3," in *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013, pp. 21–32.

[21] B. Döbel and H. Härtig, "Who watches the watchmen? protecting operating system reliability mechanisms," in *8th Workshop on Hot Topics in System Dependability (HotDep'12)*, 2012.

[22] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on*, May 2010, pp. 34–41.

[23] M. Hoffmann, C. Borchert, C. Dietrich, H. Schirmeier, R. Kapitza, O. Spinczyk, and D. Lohmann, "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)*. IEEE Computer Society Press, Jun. 2014, to appear.

[24] M. Hoffmann, C. Dietrich, and D. Lohmann, "Failure by design: Influence of the RTOS interface on memory fault resilience," in *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, ser. Lecture Notes in Informatics. German Society of Informatics, Sep. 2013.

[25] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schröder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*. Washington, DC, USA: IEEE Computer Society Press, May 2012, pp. 49–60.

[26] X. Li, K. Shen, M. C. Huang, and L. Chu, "A memory soft error measurement on production systems," 2007, pp. 1–14.

[27] J. Maiz, S. Hareland, K. Zhang, and P. Armstrong, "Characterization of multi-bit soft error events in advanced SRAMs." New York, NY, USA: IEEE Press, 2003, pp. 21.4.1–21.4.4.

[28] F. Irom and D. Nguyen, "Single event effect characterization of high density commercial NAND and NOR nonvolatile flash memories," *IEEE Transactions on Nuclear Science*, vol. 54, no. 6, pp. 2547–2553, Dec 2007.

[29] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, ser. Lecture Notes in Informatics, G. Mühl, J. Richling, and A. Herkersdorf, Eds., vol. 200. German Society of Informatics, Mar. 2012, pp. 201–210.

Lebenslauf

Florian Lukas, geboren am 22. Juni 1988, erhielt 2007 sein Abitur am Kepler-Gymnasium Weiden und begann anschließend das Studium der Informations– und Kommunikationstechnik an der Friedrich–Alexander–Univerisät Erlangen–Nürnberg. Nach einem Fachrichtungs-Wechsel 2008 erlangte er 2012 den Bachelor of Science der Informatik (1,3). Diese Arbeit schließt sein konsekutives Master-Studium der Informatik ab.