Generic Soft-Error Detection and Correction for Concurrent Data Structures

Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk

Abstract—Recent studies indicate that transient memory errors (*soft errors*) have become a relevant source of system failures. This paper presents a generic software-based fault-tolerance mechanism that transparently recovers from memory errors in object-oriented program data structures. The main benefits are the flexibility to choose from an extensible toolbox of easily pluggable error detection and correction schemes, such as Hamming and CRC codes. This is achieved by a combination of aspect-oriented and generative programming techniques. Furthermore, we present a wait-free synchronization algorithm for error detection in data structures that are used concurrently by multiple threads of control. We give a formal correctness proof and show the excellent scalability of our approach in a multiprocessor environment. In a case study, we present our experiences with selectively hardening the eCos operating system and its benchmark suite. We explore the trade-off between resiliency and performance by choosing only the most vulnerable data structures for error recovery. Thereby, the total number of system failures, manifesting as silent data corruptions and crashes, is reduced by 69.14 percent at a negligible runtime overhead of 0.36 percent.

Index Terms—Fault tolerance, concurrency, aspect-oriented programming, object-oriented programming, operating systems

1 Introduction

Errors in main memory are one of the primary hardware causes of today's computer-systems failures [1], [2]. Recent studies on current DRAM technology (DDR2 and DDR3) confirm an approximate fault rate of 0.044 FIT/Mbit [3] to 0.066 FIT/Mbit [4]. For a computing cluster with terabytes of main memory—such as the "Jaguar" supercomputer at Oak Ridge, Tennessee—this fault rate "*translates to one failure approximately every six hours*" [4]. The ever increasing demand for larger computer memory worsens this reliability problem. Furthermore, as VLSI technologies move to higher chip densities and lower operating voltages, the sensitivity to electromagnetic radiation is expected to increase dramatically [5], [6].

Most DRAM faults manifest as transient errors (*soft errors*), randomly distributed over a system's lifetime [3], and cannot be resolved by device replacement. A remedy to this problem is the widespread use of *single-bit-error correcting and double-bit-error detecting* (SEC-DED) memory hardware. However, at least 17 percent of DRAM errors affect multiple bits [2], [3], [4]. *Chipkill* [7] tolerates word-wise multi-bit errors by interleaving a word on independent DRAM chips, at the cost of reduced performance and up to 30 percent higher energy consumption due to forced narrow-I/O configuration [8]. Low-cost systems, primarily addressed in this paper, cannot afford such a costly hardware mechanism.

Several studies report that a large share of memory errors is masked by the application software, for example, by overwriting a transient error [9], [10]. This behavior is highly application specific and can be exploited to selectively apply error recovery to only the *critical* memory accesses.

We propose a software-based memory-error recovery that exploits application knowledge about memory accesses, which are analyzed at compile time and hardened by compiler-generated runtime checks. A challenge is the placement of the runtime checks in the control flow of the software, requiring to analyze which program instructions work on which parts of the memory. In general, this is an undecidable problem for pointer-based programming languages. However, if we assume an object-oriented programming model, we can reason that non-public data-structure members are accessed only within member functions of the same class. Thus, data structures (*objects*) can be examined for errors by inserting a runtime check *before* each member function call. To reduce the overhead, we apply static program analyses, which yield an economic subset of call sites. Thereby, faults that occur while an object is "in use" go undetected. The research question is whether our approach keeps the risk of undetected faults at an acceptable level.

In the following sections, we describe our experiences with applying object-level error recovery to the *embedded Configurable operating system (eCos)* [11], written in object-oriented C++. Our software-based approach offers the flexibility to choose from an extensible toolbox of error-detecting and error-correcting codes, for example CRC and Hamming codes.

An inherent problem of software-based fault-tolerance mechanisms—when applied to an operating system—is to ensure correctness under concurrent execution. For instance, the eCos' scheduler function `get_current_thread()` is executed *without* acquiring a kernel lock, thereby permitting the scheduler data structure to be modified concurrently by another thread of control. Verifying the CRC code

• *The authors are with the Department of Computer Science, Technische Universität Dortmund, Dortmund 44227, Germany. E-mail: {christoph.borchert, horst.schirmeier, olaf.spinczyk}@tu-dortmund.de.*

Manuscript received 1 Oct. 2014; revised 13 Feb. 2015; accepted 29 Mar. 2015. Date of publication 29 Apr. 2015; date of current version 18 Jan. 2017. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2015.2427832

for a changing scheduler object certainly fails and must be handled carefully.

The present paper improves over our previous conference publication [12] by introducing a *wait-free* synchronization algorithm for *concurrent* memory-error detection and correction. In the following Sections 2 and 3, we summarize the generic object protection (GOP) presented in [12]. Extending that publication, we make three *new* contributions:

- We present a wait-free synchronization algorithm that improves the efficiency of software-based memory-error detection and correction (Section 4). The performance gain scales with the degree of concurrency and reaches an order of magnitude for 64 threads.
- We formally prove the correctness of our synchronization algorithm (Section 4).
- In the completely revised and extended evaluation, we demonstrate the effectiveness and efficiency of the wait-free GOP applied to the eCos kernel test suite (Section 5). Substantiated by extensive fault-injection (FI) campaigns, we explore the optimization potential opened up by fine-grained configurability and the choice among five different detection and correction variants. We propose a configuration heuristic yielding a near-optimal trade-off between runtime overhead and reliability gains, and provide measurements on real hardware.

Finally, general limitations of our approach are discussed in Section 6, followed by a comparison to related work (Section 7), after which the paper concludes (Section 8).

2 Problem Analysis

> *"... the only DRAM bit errors that cause system crashes are those that occur within the roughly 1.5% of memory that is occupied by kernel code pages." [1]*

In general, the operating-system (OS) *kernel* is the most important piece of software regarding dependability, as all other software components depend on the OS. A crash of the OS terminates *all* running applications, and recovery by a system reset leaves persistent data, such as file systems, in a defective state. Surprisingly, in spite of their impact on total system resiliency and—compared to the rest of the system—their very small memory footprint, state-of-the-art OS kernels are not equipped with countermeasures against transient memory errors: An efficient software-based fault-tolerance technique would offer an enormous potential to reduce system failures.

Our approach to efficient software-based memory-error correction is to exploit knowledge on the application's behavior and its OS usage profile. Focusing on special-purpose embedded systems, this profile can be assumed to remain largely unchanged over a system's lifetime. Our working hypothesis is that only a small *application-dependent* subset of the OS's state is actually "mission critical", and faults in other parts of memory do not affect the system's stability. Accordingly, only the critical memory space needs recovery, calling for a configurable and highly localized application of error detection (EDM) and error-recovery mechanisms (ERM).

2.1 Fault Model

To assess the validity of this working hypothesis, we examine the fault resiliency of *eCos* by fault-injection. The fault model for the FI experiments is based on uniformly distributed single-bit faults in the data memory. A large-scale study [3] from the year 2013 confirms that this fault model is valid for contemporary memory technologies.

In particular, we only inject faults into the data and BSS memory segments. We assume that read-only data and code (text) is stored in far more reliable (EEP)ROM or Flash. This assumption is reasonable for low-cost embedded systems. For systems that load read-only data and code sections into RAM, we assume that these sections are covered by other EDM/ERMs, such as software-implemented periodic error checking [13].

2.2 Baseline Assessment: eCos Fault Susceptibility

We use a set of benchmark and test programs, bundled with eCos itself, to assess the fault susceptibility of the operating system. Both the benchmark programs and the eCos kernel are implemented in object-oriented C++, and compiled for an i386 target. We use Fail* [14], our versatile FI framework based on the Bochs IA-32 (x86) emulator, to inject uniformly distributed single-bit faults into the data and BSS memory segments.

Afterwards, we observe the benchmark behavior and classify the FI results into *benign* and *failure*. We defer a further differentiation of failures into *Silent Data Corruptions (SDC)*, CPU exceptions, and timeouts to Section 5, as such a differentiation is not necessary for the baseline assessment.

Table 1 aggregates the total failures per program symbol respectively contiguous memory area, confirming that the top ten symbols that caused the thread1 benchmark to fail, amount to 97.8 percent of all observed abnormal program terminations. The mutex1 results (in the same table) display a similar address-space clustering, yet with a different distribution: As mutex1 exhibits a different OS usage profile, a different subset of the program state causes the most often failure.

This baseline assessment reveals that in the chosen set of benchmarks, the stacks and the global kernel data structures are the most susceptible.

2.3 Solution Requirements

Our analysis in this section shows that, depending on the structure of the OS and the way it is used, the memory space exhibits "neuralgic spots", i.e., data objects that are much more critical than the remaining memory regions. By applying an EDM/ERM to only these critical objects, the system's dependability would be improved significantly with only minimal overhead.

However, the approach poses some software-engineering challenges: As the set of critical objects depends on the application scenario, the mechanism has to be implemented in a generic way so that it can be reused in all possible scenarios. Ideally, the solution would be modular and completely separated from the targeted software component. This would allow developers to reuse the generic EDM/ERM in different operating systems or even on the application-software level.

TABLE 1
Quantitative Fault-Injection Results: Top Ten Fault-Susceptible Symbols (or, Contiguous Memory Areas) for the Unmodified THREAD1 and MUTEX1 Benchmarks

THREAD1				MUTEX1			
Symbol	Size	#Failures	(%)	Symbol	Size	#Failures	(%)
thread	264	9.20×10^9	(39.5%)	stack	7,632	1,687,203	(31.8%)
Cyg_RealTimeClock::rtc	52	4.29×10^9	(18.4%)	thread_obj	416	1,487,037	(28.0%)
stack	5,088	3.31×10^9	(14.2%)	cvar2	12	287,551	(5.4%)
Cyg_Scheduler::scheduler	132	8.53×10^8	(3.7%)	cvar1	8	226,016	(4.6%)
comm_channels	96	8.53×10^8	(3.7%)	m0	20	207,538	(3.9%)
pt1	4	8.53×10^8	(3.7%)	comm_channels	96	195,392	(3.7%)
Cyg_Interrupt::dsr_list_tail	4	8.53×10^8	(3.7%)	m1	20	184,180	(3.5%)
hal_interrupt_objects	896	8.53×10^8	(3.7%)	cvar0	8	177,824	(3.3%)
hal_interrupt_handlers	896	8.53×10^8	(3.7%)	Cyg_Interrupt::dsr_list	4	155,904	(2.9%)
Cyg_Scheduler_SchedLock::sched_lock	4	8.53×10^8	(3.7%)	Cyg_Scheduler::scheduler	132	148,663	(2.8%)

3 Generic Object Protection

> *"A little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault tolerance; however, excessive or inappropriately applied redundancy is pointless." [15]*

During the design of the generic EDM/ERM, special care has to be taken to minimize runtime overhead. Therefore, our solution follows two main design principles:

1) We exploit application knowledge at compile time and, thus, minimize the number of runtime checks.
2) We balance the trade-off between the cost of inserted checks and the gained error-detection rate.

3.1 Exploiting Object-Oriented Program Structure

A running program generates a sequence of *read* and *write* operations on different memory cells. While a *write* operation overwrites a preceding transient memory *fault*, a *read* "consumes" preceding *faults* and makes the program use wrong data. To avoid this, the *write* operation can store redundancy, which can be used to detect and correct bit flips by the *read* operation.

To reduce the costs of such an approach, we follow design principle 2 by identifying groups of subsequent *read* and *write* operations with temporal and spatial locality. When we find such a group, the check can be performed only once before the first operation of the group, and the redundancy for multiple memory cells can be saved once after the last operation.

This grouping is effective when there are long periods in which the memory cells are unused between one group and the next. If a *fault* occurs at a random point in time, the probability that it hits such an inter-group time frame is high. Thus, we can still detect most faults, but have a drastically reduced overhead.

The key question for the efficient implementation of the sketched mechanism is how to detect the temporal and spatial connections of *read* and *write* operations at compile time (design principle 1). Object orientation is the most natural answer: If the program was designed in an object-oriented manner, there is an implicit connection between its data objects (instances of classes) and the program code that manipulates them (methods of the class). We can thus approximate a group of related *read* and *write* operations by a method of a class that manipulates an object. This means that ...

- ... before a method of a critical object is executed, our mechanism checks whether the object suffered from a memory fault.
- ... after the execution of the method, redundancy for the object's state is stored.

Aspect-oriented programming (AOP) [16] is the implementation technique that we find most suitable for this task.

3.2 Applying Aspect-oriented Programming

The idea behind aspect-oriented programming is to provide language features that support the modular implementation of *crosscutting concerns*, i.e., concerns of the implementation that affect various different locations of the program in a systematic way. This is achieved by defining rules such as the following:

> *"In programs P, whenever condition C arises, perform action A." [17]*

As *P*, *C*, and *A* can be provided by the programmer, AOP offers a generic mechanism to instrument arbitrary programs (*P*) with error detection and correction code (*A*) whenever a member function of a critical object is executed (*C*). A tool called *aspect weaver* typically performs a code transformation at compile time and carries out the demanded adaptation of the control flow. Aspect-oriented language extensions are, for instance, available for Java (AspectJ [18]) and C++ (AspectC++ [19]). The latter is strongly focused on compile-time code adaptation and provides a compile-time *introspection* mechanism, which allows the programmer to write generic actions that depend on the target program's structure.

Fig. 1 shows a simplified version of our generic object-protection mechanism written in the AspectC++ language. The aforementioned rules are defined with the `advice` keyword, as in lines 3, 8 and 10. In AspectC++, rules (advice definitions) that implement a common concern are grouped in an *aspect*. The definition of our `GenericObjectProtection` aspect starts in line 1 with the keyword `aspect`. A benefit of aspect-oriented programming over other implementation techniques is that crosscutting concerns can be implemented in a separate module close to a natural-language description. For example, the pieces of advice in lines 8 and 10 are almost a literal translation of the two rules mentioned at the end of Section 3.1: In line 11, a function `check()` is called *before* any *call* to a member function of a *protected class*. In

```
1  aspect GenericObjectProtection {
2    pointcut protectedClasses() = "Cyg_Scheduler" || "Cyg_Thread"; // list of critical eCos classes
3    advice protectedClasses() : slice class { // generic class extension ("introduction")
4      char replica[JPTL::MemberIterator<JoinPoint, SizeOfNonPublic>::EXEC::SIZE];   // redundancy data
5      void check()  { JPTL::MemberIterator<JoinPoint, CheckReplica>::exec(this); }  // detect/handle errors
6      void update() { JPTL::MemberIterator<JoinPoint, UpdateReplica>::exec(this); } // recalculate 'replica'
7    };
8    advice call(protectedClasses()) || construction(protectedClasses()) : after() {
9      tjp->target()->update(); } // generic advice
10   advice call(protectedClasses()) : before() {
11     tjp->target()->check(); } };
```

Fig. 1. A highly simplified implementation of the generic object-protection mechanism written in AspectC++.

line 9, a function `update()` is called *after* a member function *call* or the *construction* of a protected class' instance.

The built-in pointer `tjp` (**t**his **j**oin**p**oint) can be used by advice code to access context information about the condition that triggered its execution in a generic way. `tjp->target()` yields the target object of the function call or object construction, respectively. Besides the `target()` function, the AspectC++ JoinPoint API provides much more context information, especially static type information such as the type of the calling and the called object (`JoinPoint::That` and `JoinPoint::Target`).

Advice definitions are also generic in the sense that they use the *pointcut* `protectedClasses()` to address the points of adaptation. A *pointcut* is merely an alias for a reusable part of a condition. In line 2 it is defined to match the `Cyg_Scheduler` class and the `Cyg_Thread` class. Alternatively, the wild-card character `%` can be used in pointcut expressions to match all classes, or the pointcut can be generated by an external tool (see Section 5.3).

In AspectC++ the adaptation mechanism can also insert structural extensions. The advice in line 3 shows this feature. Here the protected classes are extended by three new members: A data member `replica`, which will store the object's data redundantly, and the two member functions `check()` and `update()`. The details of the implementation can be easily replaced to support different EDM/ERMs, e.g., using a Hamming code or cyclic redundancy check (CRC). An essential language feature needed by our EDM/ERMs is, again, the `JoinPoint` type. For structural extensions, this built-in type is the interface to the introspection mechanism of AspectC++, which provides information about the target type of an extension for the inserted members. For example, it describes all data members of the target class including their type. We can exploit this information by using it as a parameter for *generative* C++ template metaprograms, such as the `JPTL::MemberIterator`. Thereby, we generate the instructions that copy or compare each data member's value to `replica`. Template metaprogramming is a powerful, *Turing-complete* [20] code-synthesis mechanism at compile time.

3.3 Implementation Challenges

3.3.1 Generic Redundancy

The redundancy for objects is directly inserted into the protected classes as additional class members (see Fig. 1, line 4). By this means, the redundancy becomes an integral part of each class instance and the C++ compiler automatically allocates the needed memory space whenever such an object is constructed.

The challenge of this approach is that—for an ERM such as the Hamming code—the amount of redundant bits depends on the protected data's size. We cannot use the built-in `sizeof` operator of C++ to determine the object size, because at the point where the redundancy member is introduced, the class type is incomplete. Then again, the redundancy member increases the object size.

Thus, we need to calculate the size of all data members *prior* to the introduction of redundancy. This is were the compile-time introspection feature of AspectC++ comes into play. With the provided information about the individual data members, a template metaprogram can apply the `sizeof` operator independently to each data member. In Fig. 1, the `SizeOfNonPublic` metaprogram sums up the individual sizes and filters out `public`[1] data members, variable-size[2] data members, and compiler-generated alignment padding. The result is a compile-time constant that defines the size of the introduced `replica` array.

3.3.2 Static Data Members

In addition to ordinary data members of objects, our approach also covers *static* data members of classes. Therefore, our complete GOP implementation inserts a `static_replica` data member accompanied by `static_check()` and `static_update()` functions. These functions implement the EDM/ERMs specifically for static data members, which are distinguishable from object members by AspectC++'s compile-time introspection API. Similar to the advice definitions shown in Fig. 1, additional advice definitions trigger the `static_check` and `static_update` functions before and after all member-function calls.

3.3.3 Object Composition

The next challenge is the composition of objects. Let the class $\mathcal{C}$ contain a class-type member $\mathcal{C}_{sub}$ plus redundancy: $\mathcal{C} = \{\mathcal{C}_{sub}, \ldots, \mathcal{R}\}$. Given this definition, the subobject—an

1. We exclude `public` data members, because they are accessible from anywhere outside of member functions. The object-oriented paradigm discourages developers to use this feature in order to restrict access and prevent unwanted modifications by other software components.

2. The C++ language supports variable-size data members for compatibility with legacy C code. Such data members are uncommon in C++ code because they break class inheritance due to unknown object size at compile time.

instance of $\mathcal{C}_{sub}$—would be protected twice, both by $\mathcal{R}$ and its own redundancy $\mathcal{R}_{sub}$. It is sufficient to cover subobjects only once, so that we decided to exclude class-type members from the GOP. We implemented the exclusion of subobjects by C++ *type traits* [21], which is a template-based technique that allows to make decisions based on types, for instance by testing whether a data member is of class type (subobject), a pointer, an integer, and so on. Additionally, this technique offers a way to tailor GOP to cover only particular data members, for instance just pointers. We have not further investigated this opportunity, yet.

3.3.4 Static Call-Site Analysis

Before a member function is executed, the GOP triggers error detection, and after return from that function, the redundancy is updated. Technically, this could be done by the *caller* or the *callee*. However, the former approach offers optimization opportunities: Consider two member functions f_1 and f_2 of the same data structure and the following call sequence: $main() \rightarrow obj.f_1() \rightarrow obj.f_2()$. In this case, a check before f_2 would immediately follow the check before f_1. Checks on such call sites can be skipped, but only if the *caller* and *callee* refer to the same object. The AspectC++ JoinPoint API (see Section 3.2) again provides the necessary information for optimization, namely the class type of the caller (`JoinPoint::That`) and the callee (`JoinPoint::Target`). These class types can be tested on equality by C++ type traits at compile time. Only if both types are identical, the actual pointers to the caller/callee objects, obtained by `tjp->that()` and `tjp->target()` respectively, need to be compared. Optimizing compilers, such as GCC, resolve the pointer comparison at compile time when a data flow from the caller to the callee is discovered, for example, on a function invocation with C++'s `this` pointer (implicitly or explicitly). The GCC provides a programming interface to its data-flow analysis by the intrinsic `__builtin_constant_p(exp)`, which evaluates whether the expression `exp` (i.e., the pointer comparison) is a compile-time constant. Only if the static data-flow analysis succeeds, the particular check/update operations are optimized out.

The call-site approach further enables the minimization of the time window between checks and redundancy updates, because outgoing function calls that leave a protected data structure can be handled additionally. As an example, consider a member function f that calls the C-library function `printf()`. Then, inside f, the call to `printf()` can be enclosed by inverted EDM/ERM-actions: $f_{update()} \rightarrow printf() \rightarrow f_{check()}$. Thus, during the execution of `printf()`, the data structure of f is safe. These additional checks/updates greatly improve error detection and correction capabilities, especially for calls that block the running process.

3.3.5 Inheritance and Polymorphism

In object orientation, *inheritance* allows a *derived* class to access data members of its *base* classes. Thus, when a derived class is used, *all* its base classes have to be verified. This is the case for the aforementioned classes `Cyg_Scheduler` and `Cyg_Thread` of eCos, which inherit from four base classes each.

The information about base classes is provided by the compile-time introspection mechanism of AspectC++: `JoinPoint::BaseClass<I>` identifies the I^{th} base class. By this means, a generative C++ template metaprogram can recursively iterate over all base classes and invoke check/update actions on each of them.

The second challenge concerns *polymorphism* by *virtual* functions, which are dispatched at runtime to the actual function's implementation, depending on the callee object's type. Hence, it is impossible to determine the type of a polymorphic object at compile time.

Polymorphism conflicts with our static call-site analysis approach. We decided to complement the static analysis by a dynamic dispatch of check/update actions. For classes with inheritance, the functions that check/update the redundancy are also declared as virtual, so that their invocation is dispatched to the most derived class. After that, the base classes are processed as described above.

In summary, data structures that are built from several base classes are treated holistically by the base-class iteration plus dynamic dispatch to allow for polymorphism.

4 Concurrent Error Detection

> *"... critical sections are poorly suited for asynchronous, fault-tolerant systems: if a faulty process is halted or delayed in a critical section, nonfaulty processes will also be unable to progress." [22]*

Operating-system developers try to minimize *critical sections*, that is, pieces of code that are only executed by one process at a time, in favor of scalable lock-free synchronization schemes. For instance, the eCos' scheduler function `get_current_thread()` is executed *without* acquiring a kernel lock, thereby permitting the scheduler data structure to be modified concurrently by another thread of control. Such nonblocking synchronization improves the scalability on multiprocessor systems.

However, the concurrent modification of shared kernel objects complicates our GOP. The two basic operations `check()` and `update()`, which examine an object for memory errors and store its redundancy as described in the previous sections, can be disturbed by concurrent execution in subtle ways. For example, verifying a checksum for a shared object, which is concurrently being modified, certainly fails. Additional critical sections, guarding the `check()` and `update()` operations, would enforce correct execution at the expense of the undesired properties of locking, such as convoying and priority inversion (see, for example, [23]). To avoid these drawbacks, we developed a wait-free synchronization algorithm for GOP that provably retains any lock-free kernel execution.

4.1 Wait-free Synchronization

"A method is *wait-free* if it guarantees that every call finishes its execution in a finite number of steps." [23] This means, that a wait-free algorithm is necessarily lock-free[3], ruling out the use of critical sections that delay other threads of

3. The *lock-free* condition only guarantees system-wide progress, and allows for individual threads to starve [23].

control. We can exploit a particular insight to address wait-freedom for GOP: If an object is—at some point in time—being modified by another thread, we can skip any further `check()` and `update()` operations on that object at the same time. The thread that *entered* the object *first* has already verified the object. The other way around, the thread that *leaves* an object *last* is committed to properly update the object's redundancy. Thus, a consensus on *"which thread was first?"* (or last, respectively) has to be found at runtime.

To identify whether an object is being used, we introduce a per-object *thread counter* into each shared class instance. This counter is atomically[4] incremented when a thread calls a member function, and decremented on function return. Hence, a zero counter indicates an unused object that needs verification before usage. Likewise, a counter value of one causes an update of the object's redundancy before returning from the current member function.

However, a running `check()` or `update()` operation can be preempted, and other threads could modify the object concurrently. To track such race conditions, we additionally introduce a *dirty flag* into each shared object. Each thread marks its presence by writing a thread-unique[5] value into the dirty flag. A preempted thread checks for a "lost" race condition by examining whether the dirty flag has been overwritten. If so, the preempted thread invalidates its checksum computation and continues without retry.

In the following, we specify the sketched algorithm more precisely by a formal model that allows us to prove its correctness.

4.2 Formal Model and Verification

We describe the wait-free synchronization algorithm for our generic object protection in *Promela* [24], a specification language targeted to abstract models of concurrent programs. This allows us to focus on the interaction and synchronization of concurrent threads, and further enables a tool-based verification of correctness properties.

Promela permits a limited set of language features in a C-like syntax. Fig. 2 shows the complete abstract model of the wait-free algorithm. Line 1 defines the thread-unique values (1...N) by using the predefined variable `_pid` that identifies each Promela process, starting with zero. Lines 3-7 describe the data structure used in our model. An instance of `CriticalClass`, the global shared `object` defined in line 8, consists of two ordinary member variables, a `checksum`, and three synchronization variables: A `dirty` flag and thread `counter` as described in Section 4.1, and a `version` tag. The checksum exemplifies the redundancy introduced by the GOP, and the synchronization variables are also introduced by an aspect. Lines 10–12 define a macro[6] for the checksum computation. Likewise, the semantic of an atomic *compare-and-swap* instruction is defined in the lines 14-18: Only if the memory `location` (first argument) contains the value `oldval`, then `newval` is written to that memory

```
1  #define thread_ID() (_pid+1) /* thread-unique value {1...N} */
2
3  typedef CriticalClass {
4    int member1 = 5, member2 = 2; /* ordinary members */
5    int checksum = 7; /* introduced by the aspect */
6    int dirty = 0, counter = 0, version = 0 /* wait-free sync */
7  }
8  CriticalClass object; /* global shared object */
9
10 inline compute_checksum(obj, chksum) {
11   chksum = obj.member1; /* non-atomic computation */
12   chksum = chksum + obj.member2     }
13
14 inline compare_and_swap(location, oldval, newval) {
15   d_step { /* one single indivisible statement */
16     if
17     :: (location == oldval) -> location = newval
18     :: else fi } }
19
20 inline enter(obj) {
21   if
22   :: (obj.counter == 0) -> /* object not in use */
23     int version = obj.version; /* remember version */
24     int checksum_tmp;
25     compute_checksum(obj, checksum_tmp)
26     if
27     :: (checksum_tmp != obj.checksum) -> /* bit error */
28       if
29       :: (obj.dirty == 0) -> /* check for race condition */
30         assert(version != obj.version) /* false positive */
31       :: else fi
32     :: else fi
33   :: else fi;
34   obj.counter = obj.counter + 1; /* atomic */
35   obj.dirty = thread_ID()   }
36
37 inline leave(obj) {
38   obj.dirty = thread_ID();
39   /* hardware memory barrier (MFENCE) needed for TSO */
40   if
41   :: (obj.counter == 1) -> /* the last thread leaving */
42      compute_checksum(obj, obj.checksum) /* update checksum */
43      obj.version = obj.version + 1;
44      compare_and_swap(obj.dirty, thread_ID(), 0) /* atomic */
45   :: else fi;
46   obj.counter = obj.counter - 1 /* atomic */   }
47
48 active[4] proctype threads() { /* start 4 threads */
49   enter(object);
50   object.member1 = object.member1 + thread_ID()*3; /* modify */
51   object.member2 = object.member2 - thread_ID();   /*  ...   */
52   leave(object) }
53
54 /* global invariant specified in linear temporal logic */
55 ltl { always ((object.dirty != 0) ||
56          (object.checksum == object.member1 + object.member2)) }
```

Fig. 2. Executable abstract model of the wait-free synchronization algorithm, specified in Promela. Correctness properties are printed on highlighted background.

location. In Promela, an arrow `(->)` denotes the *then* condition of a preceding `if` statement. The value comparison and potential exchange are implemented indivisibly.

The wait-free synchronization algorithm is split into two procedures, which are carried out *before* an object is used (`enter`) and *after* object usage (`leave`). The `enter` procedure works as follows: First, we check whether the object is already being used by testing the thread `counter`. Only if unused, we proceed with lines 23-32, which copy the object's `version` tag to a local memory location (line 23), and compute the object's checksum (lines 24-25). If the checksum mismatches, we first check for a "lost" race condition to avoid false positives, indicated by a nonzero `dirty` flag (line 29) or by a differing `version` tag (line 30). Otherwise, there would be a

4. Every multiprocessor architecture we know supports atomic *read-modify-write* instructions, such as *compare-and-swap*.

5. For example, the address of a thread's current stack frame sufficiently identifies a thread.

6. Promela does not support callable functions; reusable code fragments must be specified as `inline` macros.

hardware memory error, which is not part of the abstract model. Finally, as we either have successfully verified the object or skipped the verification due to concurrent modification, we increment the object's thread `counter` atomically and store the thread-unique value in the `dirty` flag (lines 34-35). From that point in time, the object is marked as "*in use*", and further checksum verifications are skipped. Since the `counter` and `dirty` variables are *not* written *before* the checksum verification step has been completed, concurrent attempts to verify the checksum proceed until the fastest thread has succeeded. This procedure guarantees that there is always one thread that does not skip the verification.

After object usage, the `leave` procedure is executed, which overwrites the object's `dirty` flag at first (line 38). If the current thread is the only thread using the object, indicated by a `counter` of one, the object's checksum gets updated (line 42). Further on, the `version` tag is incremented, and we try to reset the `dirty` flag to zero by the atomic `compare_and_swap` instruction (line 44). This succeeds only if the dirty flag still contains our thread-unique value stored in line 38, meaning that there had not been any concurrent modification of that particular object. Otherwise, the `compare_and_swap` fails and the `dirty` flag remains nonzero, because the object is used concurrently by another thread. Finally, the thread `counter` is atomically decremented (line 46).

The role of the `version` tag becomes evident once the `dirty` flag is reset to zero (line 44). Consider a thread that is preempted while verifying the object's checksum (line 25). In the meantime, other threads could update the object and reset the dirty flag. When the suspended thread is resumed, the pending checksum verification certainly fails, as old data, originating from before the preemption, goes into the checksum computation of the updated object. This is an instance of the "ABA" problem [23], which occurs when a shared variable switches unnoticedly from state "A" to state "B", and back to "A" again. In our case, "A" denotes an "unused object" and "B" means the opposite. The common solution is a version tag, incremented on each state transition. A sufficiently large integer variable will not wrap around during the time a thread is preempted.[7]

The remaining lines 48-56 of Fig. 2 are needed for formal verification with the *SPIN* [24] model checker. Four threads are started (lines 48-52) that concurrently invoke the `enter` procedure, modify the shared object, invoke the `leave` procedure, and exit afterwards. SPIN evaluates all possible execution sequences, which potentially interleave, of these four threads. Note that the case of multiple invocations of the procedures per thread is covered by a specific, non-interleaving execution of different threads. The primary verification property is specified in *linear temporal logic* (`ltl`), claiming that—`always`—the object's `checksum` is valid or the `dirty` flag is nonzero. Line 30 ensures that there are no false positives caused by race conditions.

7. Even if the same object was modified a billion times a second, a 64-bit version tag would overflow after 585 years.

4.3 Correctness Proof

A limitation of model checking is that only *finite* models can be verified, as the model checker exhaustively analyzes the model's state space. Therefore, we give a proof by complete induction that holds for any number of threads. The *induction basis* is already proven by model checking for one to four threads. For simplicity, we assume the thread-unique IDs to be defined as $\{1, \ldots, N\}$ for N threads and an unbounded `version` tag. In the following *inductive step*, N+1 concurrent threads execute the `enter` and `leave` procedures as in Fig. 2.

Theorem 1. *If a thread with ID X is verifying the checksum, then every checksum mismatch caused by a race condition (false positive) is detected by the* `assert` *statement in line 30.*

Proof. Assume not. A race condition is not detected by the assert statement only in the following state:

$$dirty = 0 \quad \text{and} \quad version = version_X$$

with $version_X$ being the value of `obj.version` at the time when thread X reads it for the first time (line 23). For N+1 threads that run to completion, the value of $version_X$ cannot exceed N, because thread X has not finished, yet. We differentiate between two cases:

1) $version_X \in \{1, \ldots, N\} \Rightarrow$ At least one thread has already incremented the `version` variable before thread X reads it for the first time. Such threads cannot modify the object's data members and checksum anymore, so that at most N out of N+1 threads can contribute to a race condition. Applying the *induction hypothesis*, we know that every race condition is detected for up to N threads.

2) $version_X = 0 \Rightarrow$ The initial value of the `version` variable indicates that no thread has updated the object's checksum, yet. Before the object could be modified by another thread, the `dirty` flag has to be overwritten (line 35), yielding a nonzero value. The only way to reset the `dirty` variable to zero is a prior increment of the `version` variable (lines 43–44). Hence, a modifying race condition causes a nonzero `dirty` flag or nonzero `version` variable, contradicting the proof assumption. □

Theorem 2. *If a thread with ID X updates the object's checksum and resets the dirty variable to zero (lines 42–44), then the checksum is valid. (This is equivalent to the ltl claim).*

Proof. The `dirty` variable can only be reset if the condition `obj.dirty=X` holds when thread X executes line 44 (`compare_and_swap`). Additionally, line 44 is only reached if thread X had exclusive access to the object at a previous point in time (when evaluating line 41). In between, any other thread that modifies the object overwrites the `dirty` variable with its own thread ID unequal to X (line 35). Thus, when the `compare_and_swap` succeeds, thread X had exclusive access to the object while computing the checksum (line 42). Hence, the checksum is valid. □

4.4 Relaxed Memory Consistency

We have implicitly assumed *sequential-consistent* shared memory in Section 4.2 to verify the formal model.

Sequential consistency requires that all shared memory accesses of one processor are instantly visible to the other processors, and that such memory accesses are ordered with respect to the executed programs (refer to [25] for a detailed explanation). However, most contemporary multiprocessors implement *relaxed* memory-consistency guarantees for performance optimizations [25]. This allows for a reordering of memory accesses by the hardware. For instance, a store operation could be delayed to hide memory latency. Unfortunately, such a reordering breaks our wait-free synchronization algorithm. Consider swapping lines 34 and 35 in Fig. 2—the correctness properties would be violated. Thus, every access to the synchronization variables (`dirty`, `counter`, and `version`) must appear strictly in the specified order.

The predominant memory-consistency relaxation, implemented by the x86 and SPARC architectures, is *Total Store Ordering (TSO)* [26]. The only reorderings allowed in TSO concern store instructions, which can be delayed *after* a subsequent load instruction, given that the load instruction accesses a different memory location. Other instruction pairs, such as two store operations, are never reordered. TSO is attributed to per-processor store buffers, which cache recent memory writes until they are committed to memory. The store buffer is flushed implicitly by an *atomic* CPU instruction or explicitly by an `MFENCE` instruction on x86, enforcing all pending memory operations to complete. Considering the formal model in Fig. 2, there is only one store/load instruction pair that accesses the synchronization variables (lines 38/41). This instruction pair must be explicitly serialized by an `MFENCE` CPU instruction for TSO architectures (line 39).

Another source of memory-access reordering is an optimizing compiler. Therefore, additional compiler memory barriers[8] are required around every access to the synchronization variables to locally prevent instruction reordering at compile time. Furthermore, this disables the caching of values in CPU registers, which would otherwise have an effect similar to the aforementioned store buffers.

4.5 Fault-Tolerant Synchronization

Recalling the initial motivation for the wait-free synchronization algorithm to enable concurrent detection of memory errors, the algorithm itself needs to be resilient against memory errors as well. If memory errors corrupt the three synchronization variables `dirty`, `counter`, and `version` (see Section 4.2), the running program must not fail. The two variables `dirty` and `version` get overwritten regularly and are solely used to skip or invalidate the `check()` and `update()` operations of the protected object. Bit errors affecting `dirty` and `version` are harmless and can be safely ignored.

On the other hand, a corrupted value in the `counter` variable could cause an undesired `update()` operation while the object is being used by another thread. Furthermore, as the `counter` is only incremented and decremented, bit errors are never overwritten. Therefore, we use

8. For example, the GCC implements a compiler memory barrier by an inline assembler statement: `asm volatile("":::"memory");`

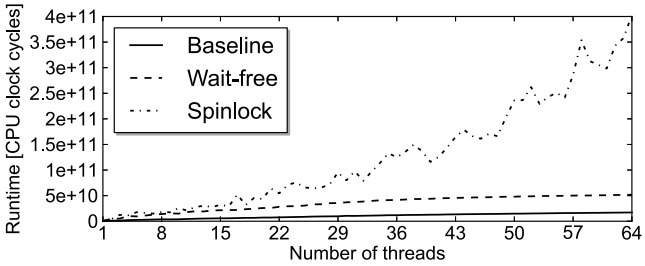


Fig. 3. Slowdown caused by synchronization: The spinlock becomes a bottleneck (less runtime is better).

arithmetic error coding [27], or rather *AN codes*, to protect the `counter` variable. Instead of incrementing the `counter` by 1, we add a large[9] odd constant value A. Thus, a valid `counter` always contains a multiple of A. Since bit errors will likely turn that value into a non-multiple of A, most errors can be detected, e.g., all single-bit flips [27]. Likewise, decrementing the `counter` variable is implemented by subtracting A.

To reduce the complexity of the abstract model in Fig. 2, only error detection is addressed. If we additionally apply error-recovery mechanisms to repair bit errors in the affected objects, it becomes evident that a short critical section, guarding the recovery instructions, is unavoidable: If an object is going to be repaired, it must not be modified concurrently by other threads. Hence, for error recovery, the synchronization algorithm remains wait-free with the exception that only in the event of a hardware error, a lock is used until the error is resolved. The recovery lock cannot introduce a deadlock, since the second *Coffman condition* (hold and "wait for" resources) [29] is not satisfied: The recovery instructions do not need to wait for additional resources but run to completion.

4.6 Scalability to Many Cores

In this section, we evaluate how the wait-free synchronization algorithm performs compared to a locking-based approach that we presented in our previous conference paper [12]. The locking-based solution uses a per-object spinlock[10] during the `check()` and `update()` operations. We implemented a micro benchmark by one shared object containing an array of 16 integers (64 bytes). A thread first computes the sum of the integer array, and then stores that sum to each array element. This procedure is repeated one million times per thread. We applied GOP, introducing a CRC-32 code into the shared object (see Section 5 for details). The micro benchmark was run on a 32-core Intel Xeon E5-4650 system, supporting 64 hardware threads by hyper-threading.

Fig. 3 shows the slowdown caused by the synchronization schemes for 1–64 threads, concurrently operating on the shared object. The *Baseline* curve denotes the runtime *without* any error detection, increasing slowly with the number of threads due to memory contention. The *Wait-free* curve shows a similar pattern despite the overhead caused

9. In this study, we chose A to be 127, as suggested by [28].

10. We use `boost::detail::spinlock` (`BOOST_SP_HAS_SYNC` variant) from the Boost C++ libraries to avoid further bus contention by giving up a thread's remaining time slice if acquiring fails too often.

TABLE 2
EDM/ERM Variants, and Their Effective Line Counts as Determined by the `cloc` Utility

Aspect/Module	Description	LOC
CRC	A CRC-32 implementation leveraging Intel's SSE4.2 instructions (EDM).	163
TMR	Triple-modular redundancy, using two copies of each data member and majority voting (EDM/ERM).	124
CRC+DMR	CRC (EDM, see above), plus one copy of each data member for additional error correction (ERM).	210
SUM+DMR	A 32-bit two's complement addition checksum (EDM), plus one copy of each data member (ERM).	198
Hamming	Software-implemented Hamming code (EDM/ERM), processing 32 bits in parallel.	355
Framework	Generic object-protection infrastructure, the basis for all concrete EDM/ERM implementations.	2,371

by frequent CRC computations. In contrast, the *Spinlock* variant slows down dramatically as the concurrent threads exceed a number of 16. For 64 threads, the runtime differs by almost an order of magnitude between the wait-free and spinlock variants. Even for a single thread, the wait-free implementation is slightly faster. This benchmark quantifies the advantage of the wait-free synchronization algorithm, which scales much better as the number of threads increase. This feature could be essential for future many-core systems, with possibly hundreds of processors.

5 Implementation and Evaluation

In the following, we describe the implementation of five concrete EDMs/ERMs based on the generic object-protection mechanism. Subsequently, we demonstrate their configurability on a set of benchmark programs bundled with eCos. We show that the mechanisms can easily be adapted to protect a specific subset of the eCos kernel data structures, e.g., only the most critical ones determined in a baseline assessment (cf. Section 2). We present fault-injection experiment results that compare the effectivity of different configurations of a single EDM, followed by a comparison of all five EDMs/ERMs. Additionally, we measure their static and dynamic overhead, and draw conclusions on the overall methodology.

5.1 EDM/ERM Variants

We implemented five EDMs and ERMs listed in Table 2 to exemplarily evaluate the generic object-protection mechanism. Especially the Hamming-code implementation has been significantly enhanced since our previous publication [12]: A template metaprogram generates an optimal Hamming code tailored for each data structure. Moreover, we applied the bit-slicing technique [13] to process 32 bits in parallel. Thereby, the Hamming-code implementation can correct multi-bit errors, in particular, all burst errors up to the length of a machine word (32 bit in our case). Besides burst errors, the CRC variants (see Table 2) cover all possible 2-bit and 3-bit errors in objects smaller than 256 MiB by the CRC-32/4 code [30].

All implementations use the wait-free synchronization algorithm from Section 4. Additionally, we integrated a check of virtual-function table pointers into the GOP, similar to [31]. Each EDM/ERM variant is implemented as a generic module and can be configured to protect any subset of the existing C++ classes of the target system.

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

5.2 Evaluation Setup

We evaluate the five EDM/ERM variants on eCos 3.0 with a subset of the benchmark and test programs that are bundled with eCos itself, namely those 19 implemented in C++ and using threads (omitting CLOCK1 and CLOCKTRUTH due to their extremely long runtime). Table 3 briefly describes each benchmark and records its number of dynamic system calls. Because eCos currently does not support x64, all binaries are compiled for i386 with the GNU C++ compiler (GCC Debian 4.7.2-5); eCos is set up with its default configuration, including GCC optimization level -O2, and *GRUB* startup. Furthermore, we disable both serial and VGA output, as the benchmarks report on success or failure before finishing, and such time-consuming output would completely mask out any EDM/ERM runtime overhead.

Again, we use the uniformly distributed transient single-bit fault model in data memory (see Section 2.1), i.e., we consider *all* program runs in which one bit in the data/BSS segments flips at some point in time.

TABLE 3
eCos Kernel Test Benchmarks

Benchmark	Description / Testing domain	Syscalls	Benchmark	Description / Testing domain	Syscalls
BIN_SEM1	Binary semaphore functionality *(2 threads)*	28	MUTEX2	Mutex release functionality *(4 threads)*	46
BIN_SEM2	Dining philosophers *(15 threads)*	659	MUTEX3	Mutex priority inheritance *(7 threads)*	55,308,146
BIN_SEM3	Binary semaphore timeout *(2 threads)*	28	RELEASE	Thread release() *(2 threads)*	117
CNT_SEM1	Counting semaphore functionality *(2 thr.)*	35	SCHED1	Basic scheduler functions *(2 threads)*	18
EXCEPT1	Exception functionality *(1 thread)*	52	SYNC2	Different locking mechanisms *(4 thr.)*	2,425
FLAG1	Flag functionality *(3 threads)*	78	SYNC3	Priorities and priority inheritance *(3 thr.)*	39
KILL	Thread kill() and reinitalize() *(3 threads)*	23	THREAD0	Thread constructors/destructors *(1 thr.)*	4
MBOX1	Message box functionality *(2 threads)*	94	THREAD1	Basic thread functions *(2 threads)*	17
MQUEUE1	Message queues *(2 threads)*	73	THREAD2	Scheduler and thread priorities *(3 thr.)*	41
MUTEX1	Basic mutex functionality *(3 threads)*	40			

The number of dynamic system calls (Syscalls) is shown in the last column.

Bochs, the IA-32 (x86) emulator back end that our FAIL* fault-injection framework [14] currently provides, is configured to simulate a modern 2.666 GHz x86 CPU. It simulates the CPU on a behavior level with a simplistic timing model of one instruction per cycle (with the exception of the HLT instruction, which spans multiple cycles until the next interrupt). Moreover, Bochs does not simulate a CPU cache. Therefore the results obtained from injecting memory errors in this simulator are pessimistic: We expect that a contemporary cache hierarchy would mask some main-memory bit flips, for example, when a cache line is written back to main memory.

5.3 Optimizing the Generic Object Protection

As described in Section 3.2, the generic object-protection mechanisms from Table 2 can be configured by specifying the classes to be protected in a pointcut expression. Either a wild-card expression selects all classes automatically, or the pointcut expression lists a subset of classes by name. In the following, we explore the trade-off between the subset of selected classes and the runtime overhead caused by the EDM/ERMs.

We cannot evaluate all possible configurations, since there are exponentially many subsets of eCos-kernel classes (the power set). Instead of that, we compile each benchmark in *all* configurations that select only a single eCos-kernel class for hardening. For these sets that contain exactly one class each, we measure their simulated runtime, and subsequently order the classes from the least to most runtime overhead individually for each benchmark. This order allows us to *cumulatively* select these classes in the next step: We compile each benchmark again with increasingly more classes being protected (from one to all classes, ordered by runtime). Fig. 4 shows the cumulative runtime of the respective selections for the CRC variant. Note that the maximum number of selected classes varies with each benchmark, as we only include classes that are used at all. The benchmarks can be divided into two categories, based on their absolute runtime:

1) *Long runtime (more than 10 million cycles).* For any subset of selected classes, the runtime overhead stays negligible (e.g., BIN_SEM2). The reason is that the long-running benchmarks spend a significant amount of time in calculations on the application level or contain idle phases.
2) *Short runtime (less than 10 million cycles).* The EDM/ERM runtime overhead notably increases with each additional class included in the selections (e.g., BIN_SEM1 benchmark). These benchmarks mainly execute kernel code.

Fig. 4b shows the results of an extensive FI campaign with FAIL* (totaling 87 million FI experiments) for the same benchmark variants. The absolute count of failed benchmark runs is broken down into *Silent Data Corruptions*, *timeouts* (the benchmark does not terminate after FI), and *CPU exceptions*. The FI results indicate that we can again apply the runtime classification obtained from Fig. 4a to find an optimal trade-off between runtime overhead and fault tolerance:

- All long-running benchmarks become *more* resilient (lower failure counts) with each additional selected class, such as the BIN_SEM2or THREAD1 benchmarks. For those benchmarks, every subset of selected classes results in less than 1 percent runtime overhead. Thus, it seems wise to protect all kernel classes to achieve an optimal resilience level.
- On the other hand, the short-running benchmarks benefit from the GOP only if the runtime overhead of the class selection stays below 1 percent (e.g., EXCEPT1 in Figs. 4a and 4b; selection of four classes). When the runtime overhead exceeds 1 percent, the additional exposure of unprotected data to faults due to runtime outweighs all gains of the EDM/ERM. For some benchmarks (e.g., BIN_SEM1), there are no classes that can be protected with less than 1 percent overhead. Those benchmarks are most resilient without GOP (see Section 5.6 for a further discussion).

It turns out that for our set of benchmarks, the following heuristic yields a good trade-off between runtime and fault tolerance: *We only select a particular class if its protection incurs less than 1 percent runtime overhead*. Using this rule of thumb can massively reduce the efforts spent on choosing a good configuration, as the runtime overhead is easily measurable without running any costly FI experiments.

5.4 Protection Effectiveness & Variant Comparison

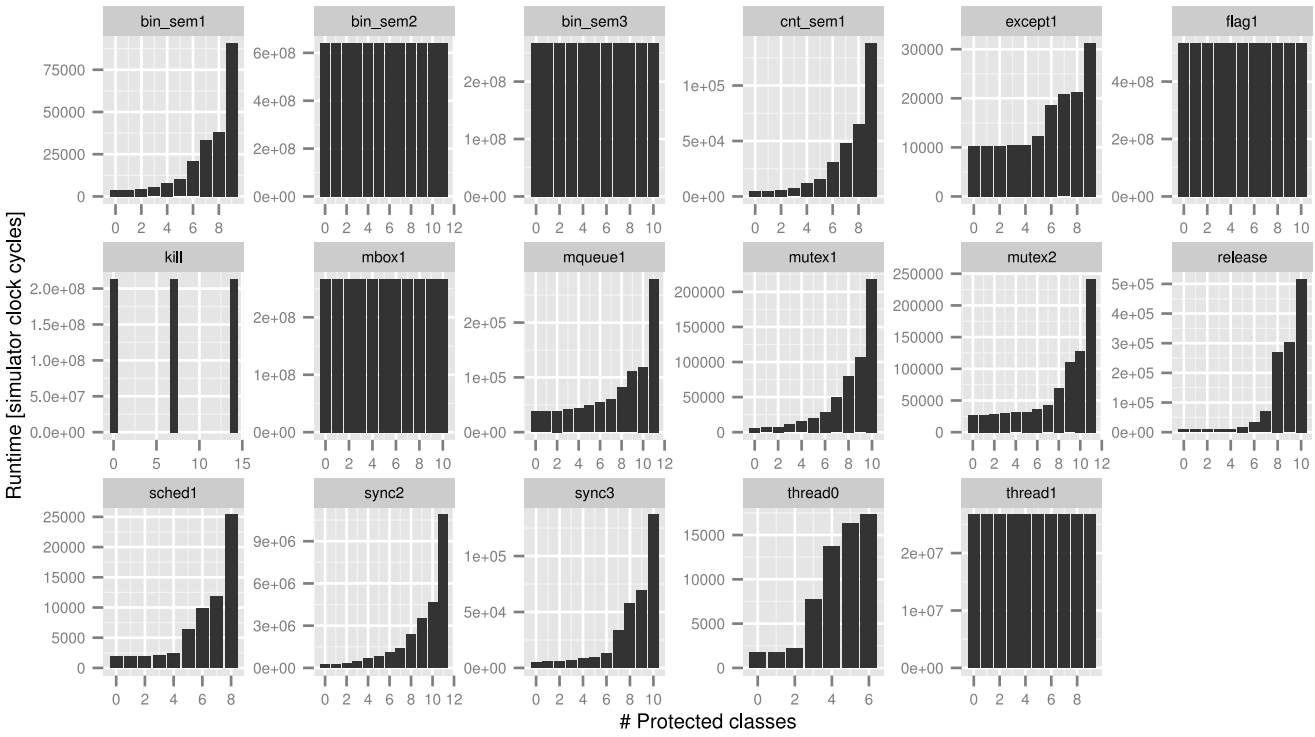
With the optimized configurations for the CRC variant from the previous section (1 percent overhead limit), we evaluate the other EDM/ERM mechanisms described in Table 2. Fig. 5 shows the FI results (additional 46 million FI experiments) that compare the different mechanisms among each other. The results indicate that the five EDM/ERMs mechanisms are similarly effective in reducing failure counts, and reduce the failure probability by up to 79 percent (MBOX1 and THREAD1, protected with CRC) compared to the baseline. The benchmarks with little improvement in the optimal case in Fig. 4b (using CRC) do not significantly improve with the other ERMs, either. The total number of system failures—compared to the baseline without GOP—is reduced by 69.14 percent (CRC error detection), and, for example, by 68.75 percent (CRC+DMR error correction).

For the FI experiments without sampling (all benchmarks except for BIN_SEM2, FLAG1, KILL and SYNC2), even 74.11 percent of the baseline's failures are prevented by the Hamming variant. Only 1.54 percent of the failures are not covered by the GOP and still originate from protected kernel data structures, whereas most remaining failures (24.35 percent) stem from unprotected data, such as the stacks of the application and the kernel.

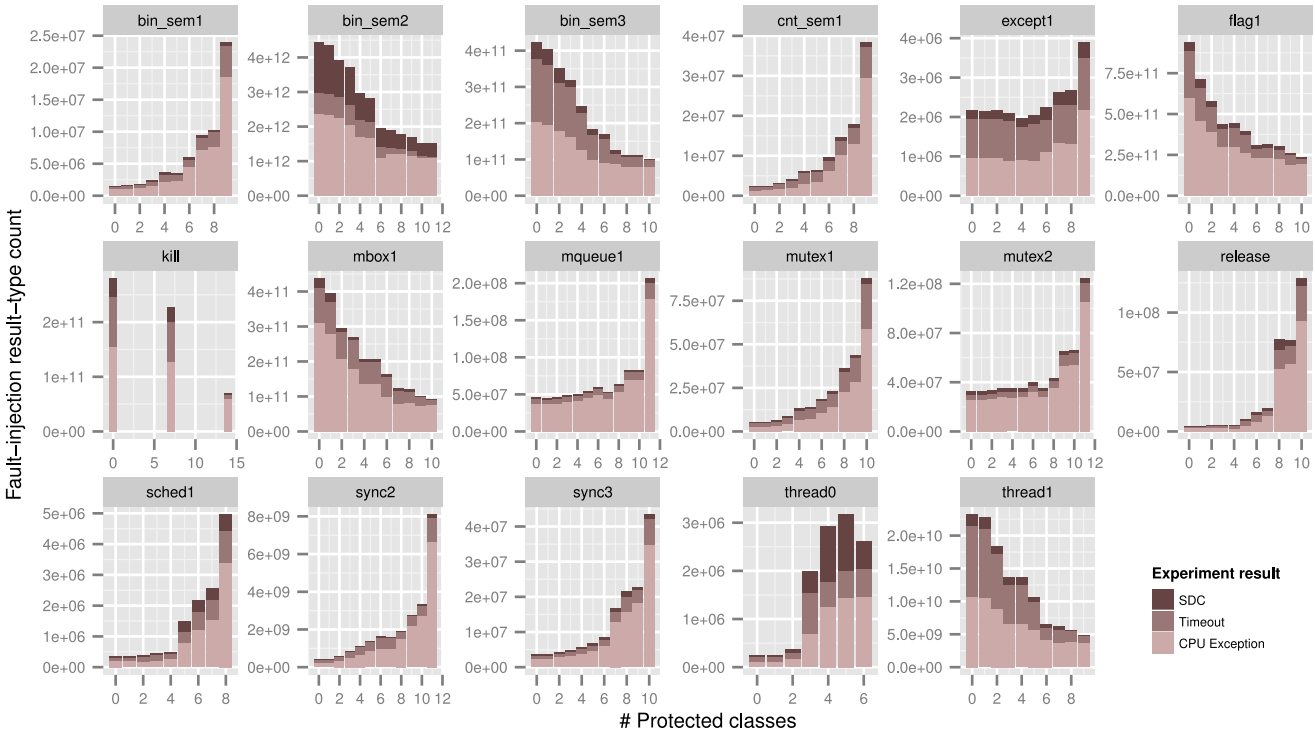
5.5 Efficiency: Static and Runtime Overhead

Although the previous sections illustrated that our protection mechanisms increase the system resiliency in many cases, they come at different static and dynamic costs.

Fig. 6 shows the static binary sizes of the benchmark variants analyzed in the previous section. The DATA sections of all baseline binaries are negligibly tiny (around 450 bytes) and increase by 5 percent (MQUEUE1, all EDM/ERM variants) up to 79 percent (KILL and THREAD2, TMR). The BSS is significantly larger (in the tens of kilobytes), and varies more



(a) The CRC runtime overhead (in CPU cycles) monotonically increases with the number of selected classes, but only amounts to a negligible increase for the long-running benchmarks (e.g., BIN_SEM2).



(b) Fault-injection results presented as failure counts: The short-running benchmarks (e.g., EXCEPT1) exhibit an optimal fault tolerance with a small or even empty subset of selected classes. However, the long-running benchmarks (e.g., BIN_SEM2) get more resilient with each additional class. Results for BIN_SEM2, FLAG1, KILL and SYNC2 are sampling estimates with a maximum relative standard error of 3.55 %.

Fig. 4. Runtime overhead and effectiveness for the CRC mechanism applied to different subsets of increasingly more eCos-kernel classes (actual names omitted). The baseline corresponds to zero selected classes. Due to their long runtime, we only measured three configurations for KILL, and omitted the MUTEX3 and THREAD2 benchmarks.

between the different benchmarks. It grows more moderately by below 1 percent (MQUEUE1, MUTEX2, RELEASE, SCHED1, THREAD0, all EDM/ERM variants) up to 15 percent (BIN_SEM2 and MBOX1, TMR). In contrast, the code size (TEXT) is even larger in the baseline (23–145 kiB), and the increases vary extremely between the different variants: While CRC increases the code by an average of 114 percent, CRC+DMR on average adds 204 percent, SUM+DMR 197 percent,

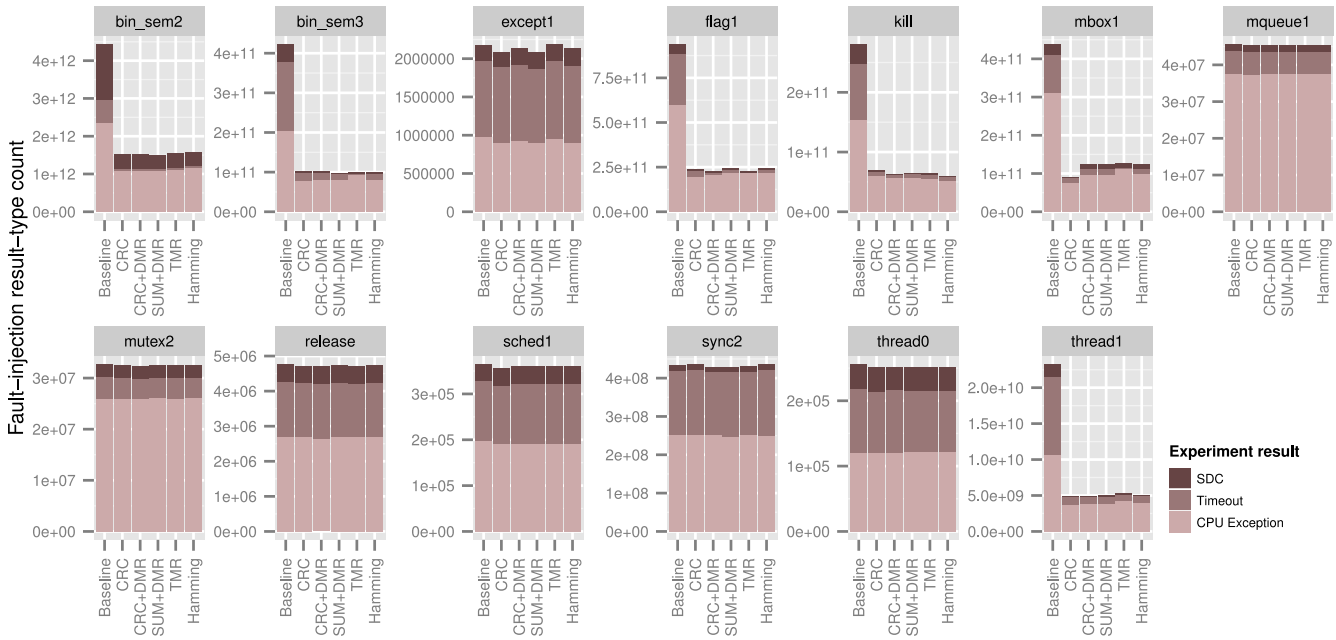


Fig. 5. Protection effectiveness for different EDM/ERM variants. Benchmarks BIN_SEM1, CNT_SEM1, MUTEX1, and SYNC3 are missing, as the analysis in Section 5.3 showed that we cannot improve the baseline with our method. Results for BIN_SEM2, FLAG1, KILL and SYNC2 are sampling estimates with a maximum relative standard error of 4.24 percent.

Hamming 200 percent, and TMR is the most expensive at an average 241 percent code size increase.

But although the static code increase may seem drastic in places, low amounts of code are actually executed at runtime, as we only protected classes that introduce less than 1 percent runtime overhead (see Section 5.3). To verify the runtime on real hardware, we deployed the benchmarks on a an Intel Core i7-M620 CPU running at 2.66 GHz, and measured their real-world timing behavior (with the RDTSCP CPU instruction). For the very short-running benchmarks, we disabled interrupts that would introduce extreme jitter (by up to a magnitude) in the measurements. Fig. 7 shows that the real-world runtime overhead totals at 0.36 percent for all variants except for TMR (0.37 percent). This total runtime corresponds accurately (99.8 percent) to the simplistic timing model of our simulation, aside from the EXCEPT1 benchmark, which triggers machine-dependent CPU exceptions that execute hundredfold slower on real hardware. The results indicate that GOP—when configured appropriately—involves negligible runtime overhead on real hardware.

5.6 Interpretation of the Results

As software-implemented error detection and correction always introduces a runtime overhead, protected variants naturally run longer than their unprotected counterparts, increasing the chance of being hit by memory bit flips (assuming them to be uniformly distributed). Consequently, there exists a break-even point between, metaphorically, quickly crossing the battlefield without protection (and a high probability that a hit is fatal), and running slower but with heavy armor (and a good probability to survive a hit). The benchmarks in Section 5.3 we identified to be *not* effectively protectable with the GOP are on the unfavorable side of this break-even point: The additional attack surface from the runtime and memory overhead outweighs the gains from being protected for all configurations.

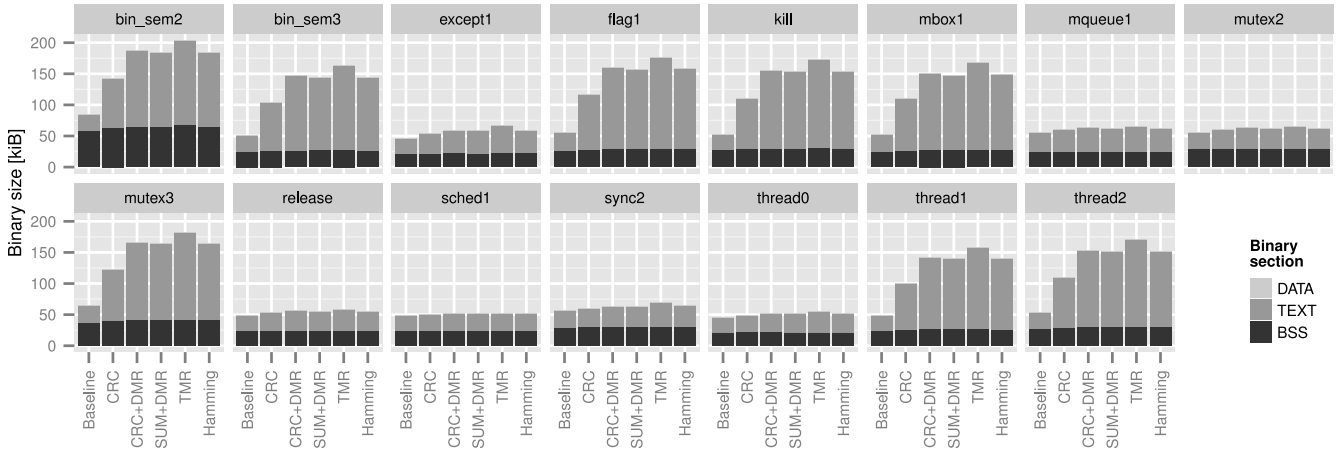


Fig. 6. Static code and data/BSS segment size of the EDM/ERM variants: The code (TEXT) segment grows due to additional CPU instructions, with CRC (detection only) being the most lightweight.

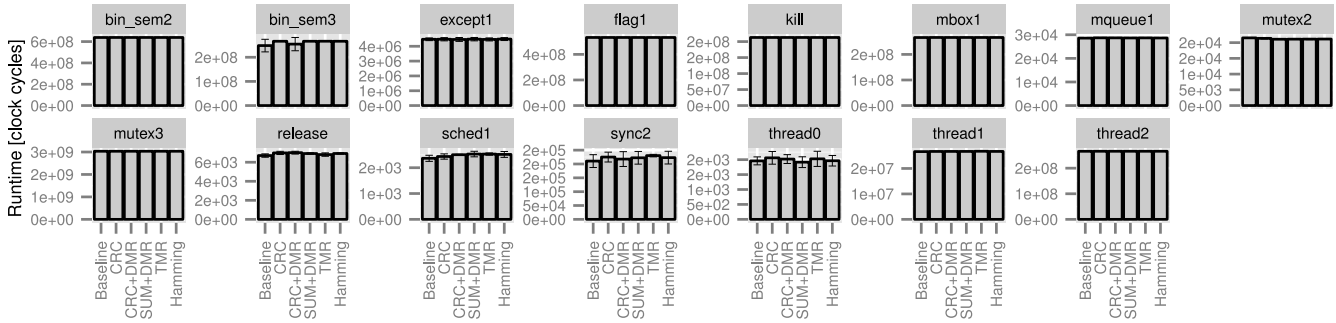


Fig. 7. Runtime on real PC hardware (with error bars showing the 95 percent confidence interval).

A more detailed analysis of what distinguishes these benchmarks from the others reveals that they actually represent the *pathologic worst case* for GOP: Unlike "normal" applications that spend a significant amount of time in calculations on the application level, or waiting for input or events from the outside, this subset of benchmarks *only* executes eCos system calls. This reduces the time frame between an `update()` after the usage of a system object, and the `check()` at the begin of the next usage (cf. Section 3.2), to a few CPU cycles. The fault resilience gains are minimal, and the increased attack surface all in all increases the fault susceptibility significantly. Nevertheless, we do not believe the kernel-usage behavior of these benchmarks is representative for most real-world applications, and do not expect this issue to invalidate our claim that GOP is a viable solution for error detection and correction in long-living data structures.

For the remaining benchmarks, the analysis in Section 5.4 shows that the EDM/ERMs mainly differ in their static overhead. CRC is clearly the best choice when detection-only suffices. For error correction, the Hamming code turns out best. The high redundancy of the DMR variants and TMR are overkill—at least unless much more destructive fault models are considered.

6 Discussion

In the previous section, we showed that the GOP mechanism significantly improves the fault tolerance of the eCos operating system. This section considers the general validity of our findings by discussing the limitations of our approach.

6.1 Safety versus Performance

By design, the GOP mechanism cannot detect memory faults that occur while an object is in use. Thus, faults that corrupt a protected object *while* a member function is executing can still lead to a failure.

Recall that our fault-injection tool does not simulate CPU caches. An actively used object would likely be stored in a CPU cache that hides memory faults while the object is cached. But even without caches, we show in Section 5.4 that only 1.54 percent of the failures originating from kernel data structures remain after applying GOP. This small fraction validates our design assumption that uniformly distributed faults likely affect objects that are not accessed while the fault occurs: At most *one* object can be accessed per CPU at a time, but *all* objects can be corrupted by a fault at any time.

However, for large data objects and long-running member functions, the probability that such an object could be corrupted during access would be higher.

The wait-free synchronization algorithm continues with the design limitation of ignoring errors during object access. While guaranteeing that every error in an idle object is detected upon the next object access, errors in constantly used objects go undetected. If multiple threads continuously invoke member functions on the same object, and there are at least two threads executing non-`const` member functions at any time, the object's redundancy never gets updated. Instead, the wait-free synchronization scheme skips concurrent checks to improve performance.

The bottom line is, that by offering a trade-off between safety and performance, the total runtime overhead of the GOP mechanism is kept at a negligible 0.36 percent while still preventing most failures.

6.2 Limitation to Object-Oriented Software

The second limitation of our approach is that only object-oriented software is addressed. Given an object-oriented programming model, we can exploit that non-public data-structure members are accessed only within member functions of the same class. Thus, data objects can be examined for memory errors by inserting a runtime check before each member function call. Otherwise, for non-object-oriented pointer-based programming languages, it is undecidable at compile time which data is accessed by the program instructions.

Hence, the GOP mechanism is not applicable to common *UNIX* and *Linux* operating systems implemented in the C programming language. However, our approach can be applied to every operating system implemented in C++. Besides eCos, we augmented the *L4/Fiasco.OC*[11] microkernel with the GOP, yielding almost identical results to those presented in this study. Furthermore, our approach can also detect and correct memory errors in application-level software, such as the *memcached* multithreaded key-value store that is evaluated in [32].

7 Related Work

Software-implemented EDMs/ERMs against memory errors provide a low-cost alternative to hardware-based memory-error recovery (see Section 1 for a discussion of hardware mechanisms). Shirvani et al. [13] evaluate several

11. http://os.inf.tu-dresden.de/fiasco/

software-implemented error-correcting codes for application in a space satellite to obviate the use of a low-performance radiation-hardened CPU and memory. Read-only data segments are periodically scrubbed to correct memory errors, whereas protected *variables* must be accessed manually via a special API to perform error correction. Similarly, *Samurai* [33] implements a C/C++ dynamic memory allocator with a dedicated API for access to replicated heap memory. Programmers have to manually invoke functions to check and update the replicated memory chunks. The latter approach exposes the heap allocator as single point of failure, which is not resilient against memory errors.

The Java virtual machine has been exercised to extend heap objects by checksums [34], [35]. These checksums are verified and updated on each data-member access, for example, on execution of `getfield` and `putfield` byte-code instructions. Additionally, object duplication has been proposed to recover from memory errors [36]. The error checking on *each* data-member access involves high overhead, in contrast to our approach that places checks around function calls, which are further reduced by the static call-site analysis (see Section 3.3.4). Moreover, a reliable heap allocator does not protect data stored in data/BSS segments and on the stack, which is common for operating systems.

Several researchers extend *compilers* for transforming non-fault-tolerant software into fault-tolerant implementations. Fetzer et al. [37] use arithmetic *AN* encoding [27] (among other methods) to detect errors by essentially doubling the storage space for the encoded programs. Compiler-implemented AN encoding plus redundant execution is proposed in [28], [38]. Tavarageri et al. [39] use checksums to detected silent data corruptions of data memory. Further code-transformation rules for source-to-source compilers have been proposed in [40], [41], [42]. These approaches are based on duplicating or even triplicating important *variables* of single-threaded user-level programs. Proof-of-concept source-to-source compilers, being far from complete, are proposed—a tedious task for the complex C/C++ language. Our work differs in that we use the general-purpose AspectC++ compiler that allows us to focus on the implementation of software-based EDM/ERMs in the OS/application layer, instead of implementing special-purpose compilers.

Fault tolerance in the OS/application layer should be separated from the "business logic" of the application to reduce complexity. This modularity problem is attacked by exercising aspect-oriented programming with AspectC++ in [43], [44], [45]. For example, Alexandersson et al. [45] implemented triple-time-redundant execution and control-flow checking as a proof of concept, which led to 300 percent runtime overhead.

Our work differs from all these related works by allowing for *configurability* to protect only the critical parts of memory, explicitly hardening an *embedded operating system*, and, most importantly, *wait-free synchronization* for concurrent error detection and correction.

8 Conclusions

Software-based dependability measures such as GOP still have to prove their practicality. Our work is an important step into this direction, as it shows that the overhead in terms of runtime and code size can be reduced to an acceptable level by exploiting the benefits of a modern software engineering approach, namely aspect-oriented programming, which facilitates application-specific tailoring of dependability measures.

Another crucial issue is the scalability of software-based dependability in the context of multicore systems. The wait-free implementation of GOP is pioneer work in this field. As it does not rely on specific OS services or exotic hardware features, GOP remains highly platform-independent. This work may be used as a guideline for other researchers or developers with similar goals. A positive side-effect of GOP, which will also gain more importance in the future, is that it not only detects hardware errors, but also unexpected object modifications caused by race conditions or other bugs in software. In fact, while applying GOP to eCos, we thereby found a kernel race condition and an access to an uninitialized object in one of the test programs.

In the future we aim to explore the possibility to offload error detection and correction to a spare CPU core, which is now possible in a scalable manner thanks to wait-free synchronization.

Acknowledgments

The authors thank their anonymous reviewers for their very helpful and encouraging comments. This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. SP 968/5-3. A preliminary version of this paper appeared in Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks, June 24-27, 2013.

References

[1] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: an empirical analysisof hardware failures on a million consumer PCs," in *Proc. ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst*, Apr. 2011, pp. 343–356.

[2] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design," in *Proc. 17th Int. Conf. Arch. Support Program. Lang. Oper. Syst.*, 2012, pp. 111–122.

[3] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng Shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2013, pp. 22:1–22:11.

[4] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Proc. Int. Conf. High Performance Comput., Netw., Storage Anal.*, 2012, pp. 76:1–76:11.

[5] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test Comput.*, vol. 22, no. 3, pp. 258–266, May 2005.

[6] S. Y. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov./Dec. 2005.

[7] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," *IBM Whitepaper*, 1997.

[8] D. H. Yoon and M. Erez, "Virtualized and flexible ECC for main memory," in *Proc. 15th Int. Conf. Arch. Support Program. Lang. Oper. Syst.*, 2010, pp. 397–408.

[9] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic, "Susceptibility of commodity systems and software to memory soft errors," *IEEE Trans. Comput.*, vol. 53, no. 12, pp. 1557–1568, Dec. 2004.

[10] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based analysis of fault and error sensitivities of dynamic memory," in *Proc. 40th IEEE/IFIP Int. Conf. Dependable Syst. Netw.*., Jun./Jul. 2010, pp. 431–436.

[11] A. Massa, *Embedded Software Development with eCos*. Englewood Cliffs, NJ, USA: Prentice Hall, 2002.
[12] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *Proc. 43rd IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2013, pp. 1–12.
[13] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey, "Software-implemented EDAC protection against SEUs," *IEEE Trans. Rel.*, vol. 49, no. 3, pp. 273–284, Sep. 2000.
[14] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th Int. Conf. Arch. Comput. Syst.*, Mar. 2012, pp. 201–210.
[15] D. J. Taylor, D. E. Morgan, and J. P. Black, "Redundancy in data structures: Improving software fault tolerance," *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 6, pp. 585–594, Nov. 1980.
[16] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proc. 11th Eur. Conf. Object-Oriented Program.*, Jun. 1997, pp. 220–242.
[17] R. E. Filman and D. P. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Proc. Workshop Adv. SoC*, Oct. 2000.
[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proc. 15th Eur. Conf. Object-Oriented Program.*, Jun. 2001, pp. 327–353.
[19] O. Spinczyk and D. Lohmann, "The design and implementation of AspectC++," *Knowl.-Based Syst., Special Issue Tech. Produce Intell. Secure Softw.*, vol. 20, no. 7, pp. 636–651, 2007.
[20] K. Czarnecki and U. W. Eisenecker, *Generative Programming. Methods, Tools and Applications*. Reading, MA, USA: Addison-Wesley, May 2000.
[21] A. Alexander, *C++ Design: Generic Programming and Design Patterns Applied*, ser. C++ In-Depth. Reading, MA, USA: Addison-Wesley, 2001.
[22] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.
[23] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2008.
[24] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, 1st ed. Boston, MA, USA: Addison-Wesley, 2003.
[25] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.
[26] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors," *Commun. ACM*, vol. 53, no. 7, pp. 89–97, Jul. 2010.
[27] D. T. Brown, "Error detecting and correcting binary codes for arithmetic operations," *IRE Trans. Electron. Comput.*, vol. EC-9, no. 3, pp. 333–337, Sep. 1960.
[28] J. Chang, G. A. Reis, and D. I. August, "Automatic instruction-level software-only recovery," in *Int. Conf. Dependable Syst. Netw.*, Jun. 2006, pp. 83–92.
[29] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, Jun. 1971.
[30] G. Castagnoli, S. Brauer, and M. Herrmann, "Optimization of cyclic redundancy-check codes with 24 and 32 parity bits," *IEEE Trans. Commun.*, vol. 41, no. 6, pp. 883–892, Jun. 1993.
[31] C. Borchert, H. Schirmeier, and O. Spinczyk, "Protecting the dynamic dispatch in C++ by dependability aspects," in *Proc. 1st GI Workshop SW-Based Methods Robust Embedded Syst.*, Sep. 2012, pp. 521–535.
[32] A. Martens, C. Borchert, T. O. Geißler, D. Lohmann, O. Spinczyk, and R. Kapitza, "Crosscheck: Hardening replicated multithreaded services," in *Proc. Workshop Dependability Clouds, Data Centers Virtual Mach. Technol.*, 2014, pp. 648–653.
[33] K. Pattabiraman, V. Grover, and B. G. Zorn, "Samurai: Protecting critical data in unsafe languages," in *Proc. 3rd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2008, pp. 219–232.
[34] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic, "JVM susceptibility to memory errors," in *Proc. Symp. JavaTM Virtual Mach. Res. Technol. Symp.*, 2001, p. 6.
[35] G. Chen, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin, "Analyzing heap error behavior in embedded JVM environments," in *Proc. HW/SW Codesign Syst. Synth.*, pp. 230–235, Sep. 2004.
[36] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Object duplication for improving reliability," in *Proc. Asia South Pacific Design Autom. Conf.*, 2006, pp. 140–145.
[37] C. Fetzer, U. Schiffel, and M. Süßkraut, "An-encoding compiler: Building safety-critical systems with commodity hardware," in *Proc. 28th Int. Conf. Comput. Safety, Rel., Security*, 2009, pp. 283–296.
[38] N. Oh, S. Mitra, and E. J. McCluskey, "Ed4i: Error detection by diverse data and duplicated instructions," *IEEE Trans. Comput.*, vol. 51, no. 2, pp. 180–199, Feb. 2002.
[39] S. Tavarageri, S. Krishnamoorthy, and P. Sadayappan, "Compiler-assisted detection of transient memory errors," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2014, pp. 204–215.
[40] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications," in *Proc. Int. Conf. Dependable Syst. Netw.*, 2000, pp. 71–78.
[41] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *Proc. 1st IEEE Int. Workshop Source Code Anal. Manipulation*, 2001, pp. 33–42.
[42] M. Leeke and A. Jhumka, "An automated wrapper-based approach to the design of dependable software," presented at the 4th Int. Conf. Dependability, Nice, France, 2011.
[43] F. Afonso, C. Silva, S. Montenegro, and A. Tavares, "Applying aspects to a real-time embedded operating system," in *Proc. 6th AOSD Workshop Aspects, Components, Patterns Infrastructure Softw.*, 2007.
[44] R. Alexandersson and P. Öhman, "Implementing fault tolerance using aspect oriented programming," in *Proc. 3rd Latin-Amer. Symp. Dependable Comput.*, 2007, vol. 4746, pp. 57–74.
[45] R. Alexandersson and J. Karlsson, "Fault injection-based assessment of aspect-oriented implementation of fault tolerance," in *Proc. 41st IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, Jun. 2011, pp. 303–314.

Christoph Borchert received the diploma from Technische Universität Dortmund, Germany, in 2010, with honors and Hans-Uhde Award. He is currently working toward the PhD degree at the Embedded System Software group at Technische Universität Dortmund. His research focus is on fault-tolerant operating systems by aspect-oriented programming.

Horst Schirmeier received the diploma from FAU in 2007. He is currently working toward the PhD degree at the Embedded System Software group at Technische Universität Dortmund. His research interests include dependability analysis, fault injection (he is a principal author of the FAIL* framework), and resilient operating-system design.

Olaf Spinczyk received the PhD degree from the University of Magdeburg, Germany, in 2002, for his research on "Operating System Construction by Aspect-Orientation." He is a professor of computer science at Technische Universität Dortmund, Germany, where he leads the Embedded System Software Group. Before moving to Dortmund, he was a post doc at the University of Erlangen-Nuremberg, Germany. His current research is focused on the construction of efficient and reliable system software for embedded systems.