Abstract
Transient hardware faults in computer systems have become widespread as shrinking structures and low supply voltages reduce the amount of energy needed to trigger a fault. This paper describes the latest improvements of a software-based fault-tolerance mechanism called Generic Object Protection (GOP). It is based on Aspect-Oriented Programming in AspectC++ and has been used in a case study to harden the L4/Fiasco.OC microkernel. As a result, the improved GOP avoids 60% of kernel failures at an acceptable overhead of 19% code size and less than 1% runtime. The GOP improvements use static whole-program analysis and have been implemented in a prototypical manner. As an outlook, the paper presents envisioned language extensions providing whole-program control-flow and data-flow analyses in future AspectC++ versions.

Categories and Subject Descriptors D.4.5 [Operating Systems]: Reliability—Fault-tolerance; D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Reliability, Languages, Performance

1. Introduction
Recent studies [16, 34, 35] show that transient hardware faults in computer systems have become widespread. These faults do not cause permanent damage to the physical devices, but their consequences are system crashes and data corruption in the memory subsystem. Therefore, the observable effects of transient hardware faults are termed soft errors. The root cause of soft errors is electronic noise coming from either inside the system, such as crosstalk or clock jitter, or noise coming from natural background radiation that contains high-energy terrestrial neutrons.

As a remedy to this problem, modern computer hardware implements error-correcting codes (ECC) in the memory subsystem and the CPU [17]. The prevalent code corrects single-bit errors and detects double-bit errors, however, recent large-scale studies [16, 34] point out that such a code is insufficient and that an unacceptable high rate of data corruption remains. To tolerate multi-bit errors, more expensive techniques such as Chipkill [11] for main memory must be used, which sacrifices throughput and consumes up to 30% more energy due to forced narrow-I/O configuration [37].

Many researchers investigated software-implemented error detection and correction [15, 30, 32] to cope with unreliable hardware.

Software approaches increase the development costs when implemented manually, and therefore several compiler-based solutions exist that automatically apply instruction-level error detection and correction [8, 12, 22, 23, 27]. These approaches have not been applied to an operating system (OS), because compiler solutions are inflexible when only a subset of data needs protection, and when high-level source code is intertwined with assembler code. Yet, the OS is the most important piece of software regarding dependability, as all applications depend on it.

In this paper, we propose soft-error correction by aspect-oriented programming [18] in the AspectC++ programming language [33]. Thereby, we implemented a library of generic fault-tolerance mechanisms [4–6]. Extending our prior work, we describe the Generic Object Protection (GOP) that transparently detects and corrects errors in kernel data structures by software-implemented ECC.

In summary, the contributions of this paper are:

• We present the new get/set advice programming-language features of AspectC++, used by our GOP (Section 3). Furthermore, we describe whole-program data-flow and control-flow analyses (Section 4) that reduce the GOP’s runtime overhead.

• In a case study, we apply our techniques to the L4/Fiasco.OC microkernel [20]. By extensive fault-injection experiments (Section 5), we show that our approach reduces the total number of OS failures by 60% at a negligible runtime overhead.

• Finally, we discuss future programming-language support for whole-program analysis in AspectC++ (Section 6).

2. Related Work
Software-implemented detection and correction of soft errors provides a low-cost alternative to hardware circuits with ECC. Moreover, software can easily implement stronger ECCs for multi-bit error correction. With that motivation, Shirvani et al. [30] evaluate the performance of several ECCs for application in a space satellite. 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 [24] implements a C/C++ dynamic memory allocator with a dedicated API for access to replicated heap memory. The use of such APIs for variable access is tedious and error-prone for the programmer.

Reducant multi-threading [26, 36] and process-level redundancy [31] are transparent OS techniques that execute identical copies of the same program simultaneously and compare the outputs. Unfortunately, such approaches cannot be applied to the OS itself, because they depend on the OS to schedule and isolate the redundant program copies.

Most of the related work on software-implemented error correction does not cover the OS. Only our prior work [14] on hardening two embedded operating systems stands out, which led to the manual implementation of soft-error detection in an automotive OS [15].
Dedicated programming-language constructs for soft-error handling have been proposed [7, 10, 25]. However, such approaches require existing source code to be rewritten. Afonso et al. [2] and Alexandersson et al. [3] pioneered the idea that aspect-oriented programming with AspectC++ helps to modularize fault-tolerance mechanisms from other software components. Yet, their recovery mechanisms are based on redundant execution, which is not applicable to OS kernels. Our work proceeds with that and proposes generic soft-error detection and correction for kernel data structures.

3. Generic Object Protection with AspectC++

Our experience with the embedded operating system eCos shows that OS kernel data structures are highly susceptible to soft errors in main memory [6]. Several kernel data structures, such as the process scheduler, persist during the whole OS uptime, which increases the chance of being hit by a random soft error.

As a countermeasure, OS kernel data structures can contain redundancy, for example a separated Hamming code [13, 30]. Before an instance of such a data structure – an object in object-oriented jargon – is used, the object can be examined for errors. Then, after object usage, the Hamming code can be updated to reflect modifications of the object.

Manually implementing such a protection scheme in an object-oriented programming language is a tedious error-prone task, because every program statement that operates on such an object needs careful manipulation. Therefore, we propose to integrate object checking into existing source code by aspect-oriented programming (AOP) [18]. Over the last fourteen years, we have developed the general-purpose AspectC++ programming language and compiler [33] that extends C++ by AOP features. Recently, we have extended AspectC++ by new language features that allow for a completely modular implementation of the sketched object protection scheme, termed Generic Object Protection (GOP). In the following, we describe these new programming-language features taking the example of GOP.

3.1 Generic Introductions by Compile-Time Introspection

Figure 1 shows the source code for a highly simplified implementation of the GOP. The keyword aspect in the first line declares an entity similar to a C++ class that additionally encompasses pointcut expressions and pieces of advice. A pointcut expression is a reusable alias for names defined in the program. For example, the pointcut critical() in line 2 lists two classes, namely “Cpu” and “Timeout.q”, from the L4/Fiasco.OC microkernel. This pointcut is used by the following line that defines advice that those two classes get extended by a slice introduction, which inserts an additional member into these classes. The inserted member “code” is an instance of the template class HammingCode<typeName>, whose template argument is bound to the built-in type JoinPoint. This type is only available in the body of advice code and offers an interface to a compile-time introspection API.

Table 1 summarizes the introspection API that provides the programmer with information on the class type that is being extended by the slice introduction. We use this information within the template class HammingCode to instantiate a generative C++ template meta-program [9] that compiles to a tailored Hamming code for each class. In particular, we use the number of existing data members (MEMBERS) prior to the slice introduction, their types (Member<>::Type) to obtain the size of each member, and a typed pointer (Member<>::pointer(T *obj)) to each data member to compute the actual Hamming code. Furthermore, for classes with inheritance relationships, we recursively iterate over all base classes that are exposed by the introspection API (see Table 1). To simplify the iteration over this API, we implemented a JoinPoint Template Library (JPTL) that offers compile-time iterators for each API entry.

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

3.2 Advice for Control Flow and Data Access

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

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

The advice in line 16 invokes the check() routine on the Hamming-code sub-object based on the trigger_check() pointcut, that is, whenever a member function is called, or a member variable is read or written. Similarly, the advice in line 20 invokes the update() function after member-function calls or writing to a member variable. Both pieces of advice invoke these routines only if the caller object (tjp->that()) and the callee object (tjp->target()) are not identical. This is an optimization that avoids unnecessary checking when an already verified object invokes a function on itself.
we separate the redundancy for the vptr from the aforementioned AspectC++'s introspection API. Our prior work shows that such there a two efficiency problems, yet:

Recalling the GOP implementation described in the previous section, 4. Whole-Program Analysis
checks are inserted by pieces of generic advice in AspectC++. Hamming code for efficiency. Again, the redundancy and runtime
pointers are nevertheless highly susceptible to soft errors [4].

Most C++ compilers allocate a hidden data member – the virtual-
function pointer (vptr) – in polymorphic objects, that is, instances
of classes that contain virtual functions. The hidden vptr is not
specified by the C++ language standard, and therefore not part of
AspectC++'s introspection API. Our prior work shows that such
 pointers are nevertheless highly susceptible to soft errors [4].

Hence, the GOP can optionally cover the vptr as well. Because
an object’s vptr remains constant until object destruction, we need
only to check for errors when a virtual function is called. Thus,
we separate the redundancy for the vptr from the aforementioned
Hamming code for efficiency. Again, the redundancy and runtime
checks are inserted by pieces of generic advice in AspectC++.

4. Whole-Program Analysis
Recalling the GOP implementation described in the previous
section, we might argue that a smart programmer would have manually
implemented the same mechanism more efficiently. In particular,
there are two efficiency problems, yet:

1. Short-running functions: The two hindmost advice definitions
in Figure 1 compute a Hamming code whenever a function call
leaves a critical object, as specified by within(member(critical())), and when the caller
object is not identical to the callee object. When the function returns,
the Hamming code gets checked by the advice in line 30.

By defining such generic pieces of advice, AspectC++ enables
a modular implementation of the GOP mechanism, completely
separated from the remaining source code. More advice definitions
exist in the complete GOP implementation, for instance, covering
static data members, and non-blocking synchronization [6].

3.3 Virtual-Function Pointer Protection
Most C++ compilers allocate a hidden data member – the virtual-
function pointer (vptr) – in polymorphic objects, that is, instances
of classes that contain virtual functions. The hidden vptr is not
specified by the C++ language standard, and therefore not part of
AspectC++’s introspection API. Our prior work shows that such
 pointers are nevertheless highly susceptible to soft errors [4].

Hence, the GOP can optionally cover the vptr as well. Because
an object’s vptr remains constant until object destruction, we need
only to check for errors when a virtual function is called. Thus,
we separate the redundancy for the vptr from the aforementioned
Hamming code for efficiency. Again, the redundancy and runtime
checks are inserted by pieces of generic advice in AspectC++.

2. Call sequences on the same object: The following excerpt from
the L4/Fiasco.OC source code shows two consecutive invocations on the same object.

```c++
void Context::schedule()
{
    /* some lines of code omitted here ... */
    sc->set(p);
    sc->replenish();
}
```

Instead of updating the Hamming code right after the first call
(sc->set(p)), and immediately verifying the same code before
the second call (sc->replenish()), it would be more efficient
to verify the code only once before the call sequence and to
update afterwards.

Both efficiency problems can be solved by leaving out check and
update operations on recently used objects. We propose static
source-code analysis to automatically identify respective source-
code locations. Therefore, we extended the AspectC++ compiler by
static control-flow and data-flow analyses.

Internally, the AspectC++ compiler builds a project model that
aggregates all results from the static analyses and thereby abstracts
from the syntax tree. Figure 2 shows an excerpt from the project
metamodel. Each depicted class gets instantiated according to the
source code being compiled, e.g., an object from the class Call
corresponds to one function call present in the source code. The
callee, an instance of the class Function, is linked via the target
association. Taking the example of a function call, the base class
Access holds the following additional results:

- `lid`: A local identifier describing the order of each function call.
- `target_object_lid`: A local object identifier that is assigned
to each object used as a call target. When two function calls have
an identical target_object_lid, then the same object is used
with certainty. Different IDs indicate either different objects
or that the static pointer analysis cannot prove their identity.
- `cfg_block_lid`: A local identifier describing the basic block
that the call belongs to. A basic block is a linear sequence of program
statements that are always executed in order.

The AspectC++ compiler currently processes one translation unit
at a time, and, thus, the project model only contains partial information
on all files during compilation. However, the partial project model
from each translation unit is serialized into a shared XML file
(the project repository). Thus, after all translation units have been
compiled once, the project repository contains the complete model
for whole-program analysis.

In this paper, we use the XQuery language [21] to extract
the whole-program information from the XML project repository.
Therewith, we generate a new header file that contains a pointcut
describing the source-code locations where the computation of
the Hamming code can be optimized out. Using this new header file, we
compile each translation unit once more to apply the optimization
pointcut. Our XQuery is organized as follows:

- The programmer first has to specify “long-lasting” functions,
  for example, functions that block the current process, such as
  “void Context::schedule()” from L4/Fiasco.OC. Exploiting
  knowledge on the whole program’s call graph, we identify the
  set of functions that transitively call such long-lasting functions.
The complement set yields the short-running functions needed
to solve the first efficiency problem.
- Call sequences on the same object can be obtained by com-
  paring the target_object_lid of calls that belong to the same
  cfg_block_lid. Our XQuery selects the longest call sequence,
  which additionally is allowed to contain calls to other short-
  running functions. This sequence is ordered by the lid entry.
5. Case Study: Hardening L4/Fiasco.OC

In this section, we apply the GOP mechanism to the L4/Fiasco.OC [20] microkernel OS, a real-time kernel written in C++. We use its x86 implementation and compare four kernel variants:

1. **Baseline**: The unmodified kernel serves as a reference.
2. **VPtr**: We configured the GOP to only protect virtual-function pointers (see Section 3.3) and applied it to 26 kernel classes.
3. **GOP**: In addition to the VPtr variant, we applied the GOP to protect all data members of 23 kernel classes.
4. **GOP-S**: Same as the GOP variant, but we applied the static whole-program analysis (see Section 4) to reduce the overhead.

Both GOP variants implement a Hamming code that is automatically tailored by an introspective template metaprogram for each data structure (see Section 3.1). Moreover, we applied the bit-slicing technique [30] 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).

To evaluate the microkernel itself, we use seven benchmark and example programs from the L4 runtime environment that is bundled with L4/Fiasco.OC. These benchmarks implement functional tests of the essential microkernel features, that is, thread scheduling, inter-process communication (ipc), interrupt requests (irq), shared-memory management, and capability-based access control. Each benchmark reports its status on the serial line. We disable the timing of the serial-device driver, because waiting for fixed baud rate would completely mask out any GOP runtime overhead.

### 5.1 Fault Model and Fault Injection

Our first evaluation metric is the **fault tolerance** of the four kernel variants. We conduct extensive fault-injection (FI) experiments with the FAIL* [29] framework based on the Bochs IA-32 (x86) emulator, configured to simulate a 1 GHz CPU. The fault model of our FI experiments is based on uniformly distributed single-bit faults in the data memory. A large-scale study [35] from the year 2013 confirms that this fault distribution is valid for contemporary memory technologies. In particular, we inject faults into the entire kernel address space from \(0x00000000\) to \(0xFFFFFFFF\), except for the read-only text section and I/O devices mapped there. The kernel already checks the text section for errors with a checksum before booting. Thus, we assume that read-only code is covered by other mechanisms, such as periodic error checking [30].

For each benchmark and kernel variant, we conduct \(N_{sampled} = 100,000\) program runs and randomly inject one bit flip in each run. Afterwards, we observe the benchmark behavior and count the number of failed program runs, manifesting as **Silent Data Corruption (SDC)** of the serial output, **timeouts** (the benchmark does not terminate after FI), and **CPU exceptions**. Since the hardened kernel variants run slower than the baseline for the same workload, the raw failure counts must not be compared directly [28]. Taking into account that a slower program is vulnerable to transient hardware faults for a longer period of time, we extrapolate the counted number of failed program runs \(F_{sampled}\) to the product of the benchmark runtime \(\Delta t\) and the amount of data memory \(\Delta m\):

\[
F_{extrapolated} = \Delta t \cdot \Delta m \cdot \frac{F_{sampled}}{N_{sampled}}
\]

Figure 3 shows the extrapolated failure counts for each benchmark and variant combination. The Baseline variant is the most susceptible to soft errors and exhibits a total of \(11.45 \times 10^{12}\) failures for all benchmarks. By protecting only virtual-function pointers (VPtr), the total failures are reduced to \(10.07 \times 10^{13}\). The GOP-S variant is the most effective and transparently avoids \(60\%\) of the baseline’s failures. The remaining \(4.99 \times 10^{13}\) failures stem from unprotected data that has not been hardened by GOP yet. In summary, the fault tolerance of the L4/Fiasco.OC microkernel is significantly improved by GOP for all benchmarks.

### 5.2 Memory Footprint and Runtime Overhead

The hardened kernel variants include redundancy to transparently correct soft errors. Hence, the memory usage of the kernel image increases, as shown in Table 2. The GOP-S variants adds 80 KiB of code, an increase of \(19\%\) compared to the text section of the baseline. On the other hand, the RAM usage – the sum of the Data and BSS sections – increases only by less than \(1\%\).

<table>
<thead>
<tr>
<th>Memory Usage per Variant</th>
<th>Text Data BSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>430,993</td>
</tr>
<tr>
<td>VPtr</td>
<td>456,453</td>
</tr>
<tr>
<td>GOP</td>
<td>519,757</td>
</tr>
<tr>
<td>GOP-S</td>
<td>513,159</td>
</tr>
</tbody>
</table>

Table 2. Static binary size of the L4/Fiasco.OC kernel image.

The growth of the text section is also reflected by the number of CPU instructions executed at runtime, plotted in Figure 4. The GOP variant executes 3.5 times the instructions of the baseline, whereas our static analyses (GOP-S) efficiently reduce the instruction-count overhead to 2.3 times. However, the real execution time of four benchmarks (ipc, map_irc, shared_ds, and urq) is each three orders of magnitude higher than their pure instruction counts. These benchmarks have slack time in their schedules and thus contain idle phases, whereas the remaining three benchmarks spend their primary execution time in the kernel. The total runtime overhead, aggregated for all benchmarks, is therefore dominated by the long execution time of those four benchmarks. For the GOP-S variant, the total runtime increases by only \(0.01\%\) compared to the baseline. This negligible runtime overhead estimates the slowdown for real applications, which spend most of their time in the microkernel.

6. Whole-Program Analysis in AspectC++

Our whole-program optimization of GOP has been implemented in the XQuery language. However, having to leave the AspectC++ lan-
language domain increases development costs. We therefore envision two groups of general-purpose AspectC++ language extensions.

The first group of new language elements shall support queries on the whole program’s call graph. AspectC++ already has the `cflow(<pointcut-expr>)` pointcut function that matches all joinpoints in the control-flow graph under any of the joinpoints matched by `<pointcut-expr>`, e.g., `cflow(execution("% A::%()"))` matches all joinpoints that are reachable from class A’s member functions. The condition, which decides whether advice code has to be executed, was or was not met in the past. Therefore, if the static analysis cannot prove that a particular function must have been executed before, runtime checks are being added by the AspectC++ compiler. A complementary new language feature is `cflowto(<pointcut-expr>)`, which shall trigger advice if the control flow will reach any of the joinpoints matched by the argument in the future of the program execution. Runtime checks do not compensate for the imprecision of a static analysis, which is either sound or complete. Soundness means that we guarantee that the joinpoint will be reached, while completeness means that it is possible (and provably not impossible). We intend to support both and even configurations in between by allowing programmers to specify the requested level of confidence that the target will be reached (100% soundness, 0% completeness). In the GOP implementation that could be used to find functions that directly or indirectly might lead to a context switch: `cflowto<>(execution("void Context::schedule()"))`.

The second group of language features is related to sequences of events in the control flow combined with conditions on accessed objects. For the GOP use case it will be necessary to check whether two functions are invoked on the same object with no other calls in between that might reach a context switch. In the pointcut expression, `T` is a free type variable and `obj` is a free instance of type `T`. The new pointcut functions `first()` and `last()` select the first resp. last event in an event sequence, i.e., the first or last call on `obj` in the example (lines 7 and 10), which suffice to compute the Hamming code.

7. Conclusions

Reducing the overhead of software-based fault-tolerance measures is crucial for their success. The presented whole-program optimizations of the GOP not only reduce the code size and number of executed instructions notably, they even improve the rate of corrected errors. To achieve this, we exploited the AspectC++ “project repository” to gather information on all C++ translation units. It would even be possible to describe the optimizations in the AspectC++ language itself with a few extensions that we regard as simple and programmer compatible.

AOP-based fault-tolerance measures are especially well-suited for hardening OS kernels, because on this level other mechanisms, such as process-level redundant execution, are not applicable. The case study proves that the L4 microkernel benefits from our approach, but it is generally applicable to all C++-based OS implementations. AspectC++ has also been successfully applied for the implementation of many other concerns in OS code. It is a powerful tool that each system software developer should consider to use.

This work was partly supported by the German Research Foundation (DFG) under priority program SPP1500 grant no. SP 968/5-3, and SFB876/A4.


