# Protecting the Dynamic Dispatch in C++ by Dependability Aspects *

Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk

Department of Computer Science 12, TU Dortmund, Germany
e-mail: firstname.lastname@tu-dortmund.de

**Abstract:** Computer systems, especially devices with highly-miniaturized feature sizes, are unreliable. Data memory is susceptible to a number of physical effects that cause faults, which can be observed as spontaneous bit flips. Although in many application scenarios corrupt data is harmless ("almost" correct result often suffices), control-flow transitions are very sensitive to faults. Indirect jumps, such as the dynamic dispatch of virtual functions in C++, often crash the system in case of a single bit flip. This paper describes a suitable software-based fault-tolerance mechanism, which can be applied to arbitrary C++ software by source-to-source compilation. The overall cost for this mechanism is below 10 % for both runtime and memory overhead. Our evaluation results show that this approach eliminates 67.1 % of all irregular program terminations in a case study using an embedded weather-station software, whose entire data memory is corrupted by single-bit flips.

## 1 Introduction

From a user's perspective, fault tolerance has two dimensions: *data consistency*, which means that the system produces correct output, and *availability*, which means that the system will neither halt nor crash and, thus, always remains responsive. Often both dimensions are equally important. However, recent studies show that commodity hardware is becoming more and more unreliable [BADAD+08]. This is due to shrinking on-chip structure sizes but also a result of the trend towards low-power hardware [DYdS+10]. For instance, probabilistic hardware even trades fault probability for energy savings [Bor05]. As a consequence, fault-tolerance issues become relevant in consumer products. This is a broad spectrum of application areas and in many of these fields availability is more important than data consistency. Therefore, there are several new directions in fault-tolerance research, which all focus on *making software systems survive* hardware faults such as failure-oblivious computing [RCD+04] and software rejuvenation [VT05].

The work presented in this paper targets the same goal. Based on our unique FAIL* fault-injection framework [SHK+12] we quantitatively analyze the reasons for errors that occur due to bit flips in volatile memory of a simple weather-station software. The software

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

is written in an object-oriented manner in C++. In this software most errors that let the system die are caused by virtual-function calls with an invalid target address. However, we show that most of these situations can either be detected or even be corrected with a simple software-based solution at a low cost in terms of runtime and memory overhead.

This approach is based on our aspect-oriented programming language AspectC++ [SL07]. By means of aspect-oriented software development [KLM+97] we can implement *Dependability Aspects* that protect the dynamic-dispatch mechanism. Even though aspects affect various points in the control flow of a program, e.g., *every* virtual-function call, the implementation is (on the source-code level) modular and completely separated from the protected system components. This feature of aspect orientation facilitates an *application-specific* selection and configuration of dependability aspects based on the actual dependability requirements. This means in other words that the error probability and the costs of fault-tolerance mechanisms can be adjusted easily.

As we have analyzed only a single simple application the work presented here can only be regarded as a proof of concept. Nevertheless, it shows that *Dependability Aspects* have the potential to become a relevant technique in the space of software-based fault-tolerance solutions.

This article is organized as follows: After a brief discussion of related work in Section 2 we will present the weather-station application and the results of an initial dependability assessment of the unmodified system in Section 3. After this motivation, Section 4 will describe the C++ dynamic-dispatch mechanism as necessary background information for the following sections. Section 5 will explain the *Dependability Aspects* and is followed by an evaluation of the approach in Section 6. The paper closes with a discussion of our results and the conclusions in Section 7 and 8.

## 2 Related Work

Software-based fault tolerance is a wide research area with a long history. Even though software-based protection of data structures has been proposed earlier [DR03], we are not aware of any study that especially covers a fault-tolerance mechanism for protecting dynamic dispatch in C++ programs. Kuhn and Binkley have proposed to transform the pointer-based dynamic dispatch into a semantically equivalent switch/case statement with statically bound function calls [KB96]. We regard this approach as complimentary, because in contrast to our approach it does not contain a repair mechanism. A drawback is that the transformation requires a whole-program analysis. Furthermore, it is possible that the compiler implements the switch/case statement by a jump table, which means that a bit flip in the table index could also lead to undefined behavior.

There is also some experience on using AspectC++ for the implementation of other fault-tolerance measures. For instance, Afonso has described several aspect-oriented idioms for improving fault-tolerance in embedded OS code [ASMT07]. Alexandersson et al. have implemented time-redundant execution and control-flow checking with AspectC++ [AÖK10].

The fault-injection framework FAIL*, which we used for this study, has been presented earlier [SHK+12]. It has a unique set of features, as it provides back-end flexibility, facilitates experiment reuse, and supports deep target-state access. It combines the strengths of generalist fault-injection frameworks, such as GOOFI [AVFK01] and QInject [DCCC08], with the advantages of specialized frameworks, such as FAUmachine [SPS09].

## 3 Scenario and Motivation

In this section we outline the application scenario and the fault model used in this article, and motivate our approach by a baseline dependability assessment.

### 3.1 Weather-Station Application Scenario

As a realistic analysis and evaluation scenario, throughout this article we use the object-oriented C++ implementation of an *embedded weather-station* software we already described in an earlier publication [LSSP06]. In essence, the weather station consists of several *sensors* to gain environmental information, and *actors* to process the collected weather data. The software iteratively activates all sensors and actors, and pauses for a predefined amount of time between each iteration. Central elements of the C++ implementation are the classes `Weather` and `Sink`, which aggregate all sensors and actors, respectively. Internally, `Sensor` and `Actor` (both abstract base classes, each defining a *virtual method*; cf. Section 4) objects are chained in singly-linked lists for each base type. The application's main loop processes these lists in each iteration.

As our FAIL* experimentation framework [SHK+12] currently only provides an x86 emulator back end, and the actual sensor inputs make no difference for the purpose of this article, we replaced the original sensor and actor driver implementations with variants that generate synthetic data, and a simple CGA text-display driver.

### 3.2 Fault Model

The fault model covers uniformly-distributed single-bit flips in data memory (data and BSS segments), i.e., we consider program runs in which a single bit in data/BSS flips at some point in time. This model seems reasonable for low-cost embedded systems where read-only data and code (text) is stored in far less susceptible (EEP)ROM or Flash, and global objects and the program stack is kept in non-ECC RAM.

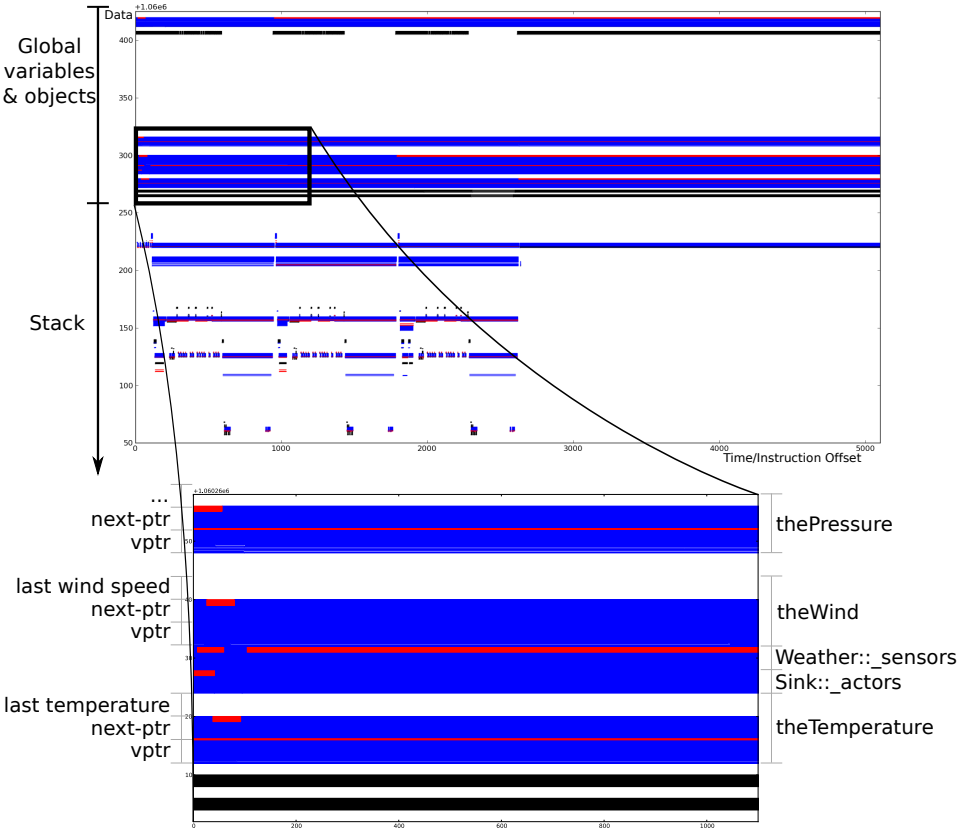



Figure 1: Excerpt from fault-injection experiment campaign with unmodified weather-station software: Bit flips in virtual-function table pointers (vptr: 32-bit word at object start, cf. Section 4) and pointers to objects (linked-list head pointers, and "next"-link pointers embedded in sensor/actor objects) result in a large fraction of observed program crashes.

### 3.3 Dependability Assessment

In order to assess the weather station's baseline susceptibility to single-bit flips in data memory, we conducted a fault-injection (FI) campaign with FAIL*. Each single experiment is an (always identical) deterministic run of the weather-station software, starting at the entry of its `main` function, and ending after executing 30,871 instructions. In the course of this run, a single bit in memory is flipped at a single point in time (within the first 20,559 instructions from program start).

A particularly interesting excerpt from the FI campaign's result data is shown in Figure 1. The larger diagram covers a single loop iteration in its horizontal dimension (ca. 5,000 instructions out of the 30,871 total), and the used part of the stack (bottom half of the diagram) and all global variables/objects (top half) in the vertical dimension (ca. 420 out

of 6,432 bytes covering all of the data and BSS segments, plotted on bit-level granularity). The experiment results are color-coded:

- *White* data points denote experiments where the injected fault did not propagate to a user-observable change in program behavior. In order to automate this classification, we categorize all experiments in which the weather-station software *continues to run*: The main loop body is visited the same number of times as in the golden run (six times), and none of the other conditions (see below) is met.
- *Blue* marks experiments in which the FI (at some point in time after the injection) led to a jump outside the text segment.
- Experiments that at some point triggered a CPU exception (division by zero, illegal instruction, or a MMU general protection violation) are colored in *red*.

Many crashes in the stack area result from corrupted function return addresses, which is outside this article's scope. The magnified subplot shows a few of the global objects and the two linked-list entry pointers in detail: The lowest and the second-lowest 32-bit words in the three `Sensor` objects in the close-up (`theTemperature`, `theWind`, and `thePressure` – two more, `theDisplay` and `kout`, are not shown), just as the two linked-list head pointers (`_actors`/`_sensors`) seem to be highly sensitive to bit flips, almost in every case resulting in an application crash (misplaced jump, or exception). It turns out that the first 32-bit words – the *vptr*, as we will describe in the next section – and the linked-list head and chaining pointers (`next`, embedded in each object as the second machine word) contribute to 26.4 % respectively 40.5 % of all observed irregular program terminations (i.e., non-white data points in the diagram). Thus, the following chapters outline an approach and implementation to protect these entities against memory errors.

## 4 Background: Object-Oriented Programming in C++

The embedded weather-station software, which we have analyzed in the previous section, was implemented by *object-oriented programming* in C++. Classes encapsulate functions (called *methods* in object-oriented jargon) and provide advantages in code reuse by *inheritance*. Inherited methods can be selectively overridden by derived classes (for further information see [Lip96]).

Inheritance features the principle of *substitution*: An object of a derived class *is* also an object of its super class, and pointers and references of the super type can refer to it. However, invoking a method on such a pointer leads to methods of the super-class (the pointer's type). Yet a common view on object-oriented programming is *polymorphism*: A call to a super-class pointer should yield a method invocation depending on the actual type of the object rather than the pointer's type. This behavior is inherent in the Java programming language – but not the default case in C++. Instead, C++ programmers have to declare a function `virtual` in order achieve polymorphism. Thus, `virtual` methods are dynamically bound to object's type at runtime (late binding), whereas calls to other methods are statically determined at compile time.

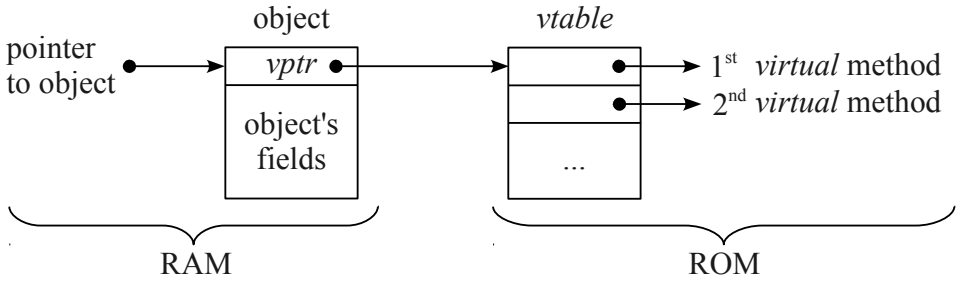



Figure 2: Memory layout of objects, vptrs and vtables used by common C++ compilers

A `virtual`-method call has to be dynamically dispatched at runtime to one of several implementations, depending on the object's type. The C++ standard [Ins03] does not explicitly specify how this has to be performed, but in general, compilers implement the *dynamic dispatch* by fast lookup tables. The predominant way, used by every C++ compiler we are aware of, is the *virtual method table (vtable)*. For each class that contains a `virtual` method, a lookup table is generated, whose entries refer to the method implementations for this particular type. Each object of such a class contains a pointer to its vtable, known as the *virtual method pointer (vptr)*. Thus, a `virtual` method call is carried out in two steps: First, the object's vptr is dereferenced to index the corresponding vtable. Second, the vtable's entry for this particular method, which points to the desired implementation, is called. Figure 2 illustrates this procedure. Since the vtable is indexed via a pointer, the final program can be separately compiled in multiple compilation units as in plain C. This is essential when the complete source code is not available, for example when libraries are used.

## 5 A Dependability Aspect for the Dynamic Dispatch

The dynamic dispatch relies on an object's vptr referring to a well-defined vtable for method lookup. Section 3 already revealed that the vptr's correctness is crucial for a program being executed. Faults in a vptr lead almost inevitably to software failures caused by illegal control-flow transitions. Therefore, we propose a mechanism to protect vptrs from unwanted changes, so that once an object is entirely constructed in the RAM, its vptr remains fixed even in the presence of transient hardware defects. As opposed to the vptr, vtables are statically generated by the compiler and can be stored in read-only memory locations, such as non-volatile flash memory. In the following section, we briefly introduce *aspect-oriented programming*, which we used to implement a software-based fault-tolerance mechanism for vptrs, as described in Section 5.2.

### 5.1 Aspect-oriented Programming

Modularity is the concept behind *aspect-oriented programming (AOP) [KLM+97]*, which supports the separation of *"what"* should be executed from *"where"*. This is done by *implicit invocation*. In AOP, an *aspect* is an entity that encompasses one or more pieces of *advice*. An advice intercepts the control flow or extends existing classes by additional fields and methods. The locations, where an advice takes effect, are described via *pointcut expressions* in a textual form. Thus, the coupling between aspects and classes is inherently loose – classes are oblivious of aspects. Crosscutting functionality, such as dependability, can be implemented in a modular way.

Our group has been developing AspectC++ [SL07] over the last ten years in order to extend C++ by AOP mechanisms. AspectC++ performs a source-to-source transformation by compiling aspects into the source code at the relevant locations specified by pointcut expressions. The advice code becomes inlined into the existing source code and, thus, is free of any overhead compared to a manual source-code instrumentation [LST+06]. This makes it an excellent tool for efficient fault-tolerance mechanisms.

### 5.2 Triple-modular Redundant Vptrs

Virtual-method pointers are a threat to safety-critical systems. Our idea is to apply the well-known *triple-modular redundancy (TMR)* concept to the dynamic dispatch of virtual methods. An object's vptr is replicated twice, so that, finally, each object contains three vptrs – all pointing to the same vtable. On dynamic dispatch, a voting over these vptrs is performed:

1. If all *three vptrs are identical*, the dynamic dispatch is performed ordinarily.
2. If and only if exactly *one vptr differs* from the other two, the dispatch is performed using the two identical ones. The deviating vptr is repaired.
3. If all *three vptrs differ* pairwise, no dispatch is performed. Instead, an error is signaled to recover safely from that situation, for instance by rebooting the device.

This way, defects in a single vptr are not harmful anymore, because there are still two valid replica around for recovery. This is a very generic mechanism that can be applied to arbitrary C++ classes. Therefore, *vptr protection* is a crosscutting concern that ought to be implemented separately, apart from the classes *"where"* it is applied to, and this is where AspectC++ comes into play.

Figure 3 shows a simplified implementation of a modular vptr-protection mechanism in AspectC++. The algorithm is encapsulated inside an aspect. The pointcut definition `critical()` describes those classes of the weather-station software targeted by the aspect, namely `"Actor"` and `"Sensor"`. This is *where* the aspect is applied. These classes are extended by two additional fields, `redundant_vptr1` and `redundant_vptr2`, and methods to perform their initialization (`init_vptr()`) and voting (`check_vptr()`).

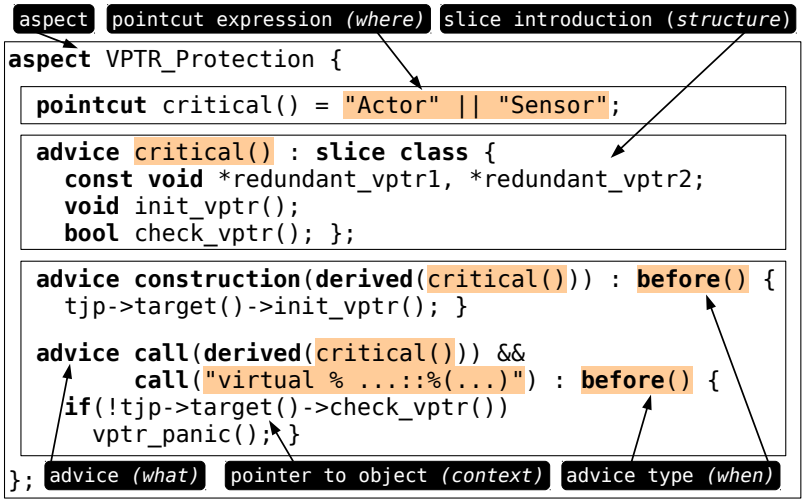



Figure 3: Protection of vptrs by TMR implemented in AspectC++

This extension is achieved by *slice introduction*, which adds members behind all existing members. Thus, spatial distance between the original vptr and the redundant ones inside a class is maximized to compensate locality of memory faults. Beside slice introduction of redundancy and dedicated methods, the given aspect contains two pieces of advice that control runtime behavior. The first one binds the method `init_vptr()` to the `construction` of objects, that is, whenever a C++ constructor of such an object is executed, the method `init_vptr()` will be implicitly invoked. `tjp` (this join point) provides access to context information, and `tjp->target()` yields a typed pointer to the object targeted by the advice. The pointcut expression `derived(critical())` computes all classes that are either listed in the pointcut `critical()` itself or inherit from at least one of them. In summary, this advice ensures that the two `redundant_vptr`s are correctly initialized. The last advice is responsible for performing the voting algorithm properly on virtual-method calls. This is specified by `call("virtual % ...::%(...)")`, which matches each virtual-method call by using the wildcard expressions `"%"` for any result type and method name, as well as `"..."` for arbitrary class names (plus namespaces) and arbitrary method arguments. Again, the expression `derived(critical())` identifies relevant locations *where* the method `check_vptr()` has to be invoked: `before` calls to virtual methods in the `Actor` and `Sensor` class hierarchies. In short, the last advice ensures that the voting part of triple-modular redundancy is performed timely.

The implementation of both introduced methods is straightforward: `init_vptr()` creates the initial backup of the vptr in the two `redundant_vptr`s. The second method `check_vptr()` is shown in Listing 1: The value of the original vptr is obtained through C++'s `this` pointer[1], as indicated by the method `get_vptr()`. At last, an optional *acceptance test* is performed after voting: The result is checked for plausibility, that is,

[1]The `this` pointer always refers to the vptr, given the object layout produced by a recent GNU g++. If compilers with different object layouts are used, compiler intrinsics, such as `_vptr` for g++, should be used.

```
1 inline const void* const get_vptr() {
2   return *((const void**)this); } // 'this' points to the vptr
3
4 inline bool acceptance_test() {
5   return (get_vptr() >= &__VTABLES_START__) &&
6          (get_vptr() <= &__VTABLES_END__); } // provided by linker
7
8 bool check_vptr() {
9   if(redundant_vptr1 == get_vptr()) {
10    if(redundant_vptr1 != redundant_vptr2) {
11     redundant_vptr2 = redundant_vptr1; } // fix redundant_vptr2
12  } else if(redundant_vptr2 == get_vptr()) {
13    redundant_vptr1 = redundant_vptr2; // fix redundant_vptr2
14  } else if(redundant_vptr1 == redundant_vptr2) { // fix real vptr
15    memcpy((void*)this, &redundant_vptr1, sizeof(redundant_vptr1));
16  } else { return false; } // all three vptrs differ
17  return acceptance_test(); } // perform acceptance test, at last
```

Listing 1: The complete implementation of `check_vptr()`. These methods are introduced into each class with vptr protection by AspectC++'s *slicing* feature (see Figure 3).

whether the vptr actually points to the memory area where vtables are placed by the linker (between `__VTABLES_START__` and `__VTABLES_END__`). If this acceptance test fails, an error is raised for safe recovery.

## 6 Evaluation

We evaluated the two vptr-protection variants – with and without the additional acceptance test – under both qualitative and quantitative aspects[2]. By repeating the weather-station FI experiment campaigns we already used for the baseline assessment (cf. Section 3), we examined both effectiveness and efficiency of the protection. Additionally, we applied static code and data size metrics to compare the additional cost each variant introduces.

### 6.1 Effectiveness: Error Correction & Detection, and Interesting Ways to Fail

The FI campaign described in Section 3 was reused with slightly-differing target memory areas (the data/BSS size changed from one variant to the other, cf. Table 1); this difference was factored out in all quantitative comparisons throughout this article. FAIL*'s fault-space pruning techniques helped reducing the number of necessary experiments from $\sim 3.18 \times 10^9$ (i.e., one for each space/time coordinate in the fault-space outlined in Section 3) for all

[2] Pun intended.

three variants to 413,024. With parallel execution, experiment runtime was kept within an order of minutes.

Figures 4a and 4b highlight major improvements from the *unprotected* to the *vptr-protection* variant of the weather station: Without the vptr-protection aspect, 77.5 % of all bit flips in vptrs lead to a crash, while the protected variant completely eliminates this fault susceptibility[3].

In spite of this noticeable improvement, the object pointers[4] are still highly susceptible: Without the guard, 99.0 % of all bit flips in these pointers result in a crash; with the protection, still 78.7 % leave the weather station in an unusable state, while now 20.3 % are detected as uncorrectable errors. The additional result color code *green* (visible in Figure 4b in several bit positions of the object pointers) marks experiments that triggers the *three vptrs differ* (cf. Section 5.2) path in the vptr-protection aspect – the new object pointer refers to a memory area where the vptr and both redundant copies differ. This is still a positive outcome, as the error handler can subsequently reboot the device, which continues being usable after a short service interruption. Unfortunately, in most cases this detection is not triggered, as at least two of the three vptr copies are identical – in many (but not all) cases simply *zero*, if the corrupted object pointer refers to, e.g., unused areas of the stack.

For this reason, the additional *acceptance test* (Figure 4c) eliminates the vast majority of the remaining cases: Only 2.6 % of all object-pointer corruptions result in a crash, while now 96.4 % are detected (*green*). The remaining failures resulting from object-pointer corruptions seem to be specific to particular objects; e.g., the least-significant bits of `theTemperature`'s `next` pointer still crash the weather station with a trap. We currently do not have any further insights on these corner cases.

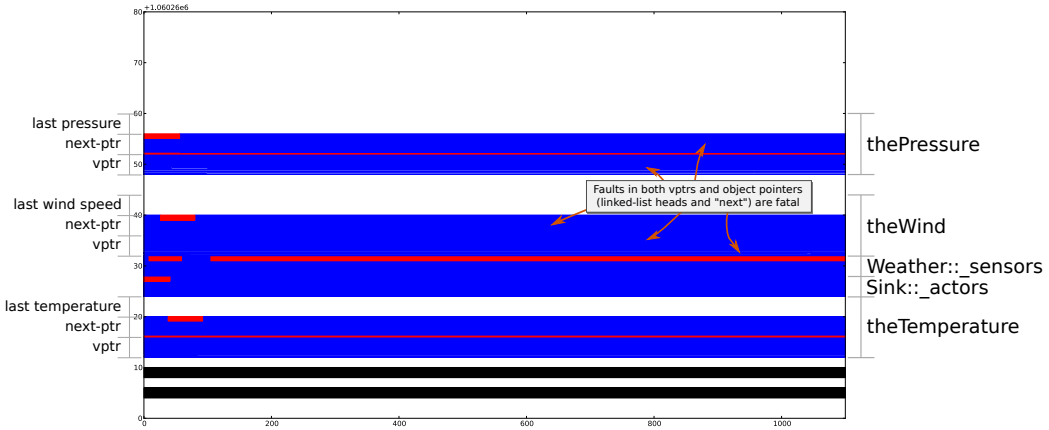
A less surprising result is that FI in the two `redundant_vptr` copies causes no crash at all; by construction, our protection aspect repairs the faulty pointer in all cases. Indeed surprising was to observe the weather station's main loop to *exceed* the golden run's six iterations (by up to another six) in some experiments: For instance, clearing a skillfully selected bit in the `_actors` list-head pointer stops all display output, resulting in far less instructions to be executed per loop iteration, but does not cause a crash.

### 6.2 Efficiency: Static and Runtime Overhead

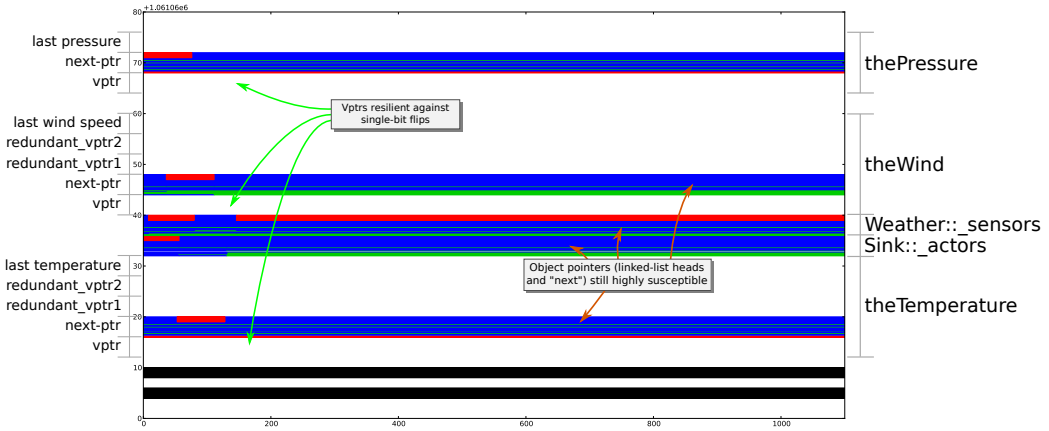
While the vptr-protection aspect – especially the variant with the additional acceptance test – was shown to be highly effective in the last subsection, like most software-based methods for fault tolerance it comes at a cost. Table 1 details on the static image size overhead for the weather-station scenario: In total, 6.8 % (another 2.6 % for the acceptance test) was added to the image size, which could be considered very reasonable for a scenario with

[3] For the evaluation of this paper, we used the GNU g++ 4.7.0 compiler with optimization level `-O3`. The optimizer of older g++ versions (prior to 4.6.0) causes a reload of the vptr after it was already validated, which leads to a short time frame where vptr corruptions break the application (0.34 % for g++ 4.4.5).

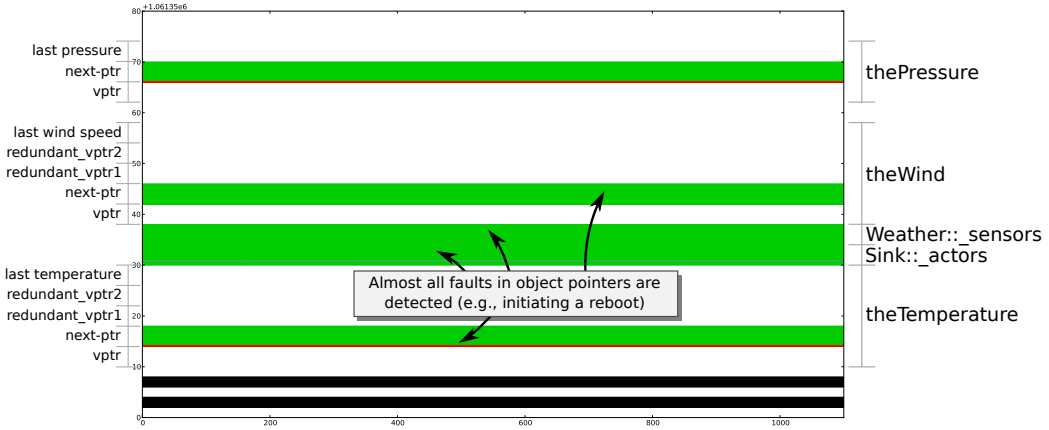
[4] We remember from Section 3 that they contribute to almost 41 % of all irregular program terminations in the baseline variant!




(a) No protection (cf. Figure 1)




(b) Vptr protection




(c) Vptr protection w/acceptance test

Figure 4: Fault-susceptibility comparison for unmodified and vptr-protected variants of the weather-station software.

| Scenario | Total | ROM | | RAM | | Runtime |
|---|---|---|---|---|---|---|
| | | *text* | *rodata* | *data* | *bss* | *(main loop)* |
| No Protection | 11,831 | 5,023 | 440 | 2,104 | 4,264 | 5,136 |
| Vptr Protection | 12,633 | 5,807 | 440 | 2,104 | 4,312 | 5,457 |
| | +6.8 % | +15.6 % | +0 % | +0 % | +1.1 % | +6.3 % |
| Vptr Protection with | 12,947 | 6,091 | 440 | 2,104 | 4,312 | 5,532 |
| Acceptance Test | +9.4 % | +21.3 % | +0 % | +0 % | +1.1 % | +7.7 % |

Table 1: Static and runtime overhead for vptr protection. Runtime is measured in number of instructions for one main loop iteration in the (fault-less) "golden run". Static memory consumption is counted in bytes.

unrealistically many virtual-function calls. At linker-section level, the text (code) segment is responsible for most of the increase: The 15.6 % (acceptance test: additional 5.7 %) increase originates in – depending on compiler optimizations and disabled/enabled acceptance test – 64–96 bytes overhead per virtual-function call site. The 1.1 % increase in the BSS segment (no additional cost for the acceptance-test variant) are solely to be attributed to the 8 bytes for the two redundant vptrs per protected object (plus/minus possible alignment changes).

With an increase of 6.3 % (acceptance: plus 1.4 %), the additional runtime penalty (in this article only measured in executed instructions, not wall-clock time on a real CPU) for the protection aspect is lower than the static text increase: Only the fast-path checks need to be executed in most cases, while the voting, correction and detection code is reserved for the rare fault case.

## 7 Discussion

The main goal of this work was to achieve a fail-safe dynamic dispatch of virtual methods in C++ with regard to transient hardware defects. As shown in the previous section, our approach yields an almost perfect coverage of control-flow failures caused by the dynamic dispatch. The overhead of our vptr-protection mechanism is small in terms of memory consumption and additionally executed instructions. Compared to in-circuit fault tolerance, such as watchdog timers and MMU-based memory protection, we provide *forward error correction* to minimize device resets. ECC-protected memory would be overkill, since only a small amount of memory locations are control-flow critical (baseline assessment: 2.4 %).

However, the evaluation covers only a single case study, so that the results cannot be generalized to all kinds of software. It is worth taking into consideration that there are reasonable doubts on using C++ in mission-critical software at all [DG06], mostly grounded on the dynamic dispatch. We think that a resilient dispatch mechanism, as provided by modular dependability aspects, is a step forward in that direction. But there are two remaining technical issues that impair the usability of our current implementation, which

we plan to resolve by extending the AspectC++ language, as described in the following two sections.

### 7.1 Pointcut Determination

The determination, which classes ought to be protected by our mechanism, is not fully automated yet. The pointcut expression given in Figure 3 lists only abstract base classes of the weather-station software – derived classes automatically *inherit* vptr-protection functionality. This knowledge, which classes are on top-level to *introduce* vptr protection, and which classes inherit this functionality, has to be provided manually. In general, those classes that have virtual methods but no super classes with virtual methods (transitively), have to be factored into the pointcut expression. This information can be obtained by static source-code analysis. We plan to integrate such an analysis into AspectC++ to simplify the usage of our vptr-protection aspect.

### 7.2 Multiple Inheritance

A class that inherits from more than one top-level class with virtual functions produces objects with multiple vptrs inside [Lip96]. On dynamic dispatch, there are consequently several candidates for the vptr that is going to be used for the dispatch. Our current implementation cannot statically determine which one is actually used, so that we have to check them all. We implemented this by C++ template meta programming [CE00] to generate a set of runtime checks. However, this could be more efficient, because there is essentially only need for a single check on the actually used vptr. Again, an analysis on the AspectC++ language level could reveal this information by providing the class type that belongs to a virtual method's initial declaration.

A side-effect of multiple inheritance is, that a class can inherit from the *same* super class multiple times. In C++, this can be explicitly avoided by *virtual inheritance*. For virtual derived classes, even field access is performed by a vtable lookup. At the moment, our mechanism does not recover field-access faults in virtual inheritance, but in general, this kind of fault results in corrupt data rather than control-flow failures.

## 8 Conclusions and Future Work

Safety-critical systems have to be resilient to transient hardware defects, such as spontaneous bit flips in the main memory. Especially dynamic data, which cannot be stored in nonvolatile flash memory, is subjected to hardware faults. This calls for software-based countermeasures based on redundancy and acceptance tests.

We presented a vptr-protection mechanism that recovers transparently from failures caused

by the dynamic dispatch in C++, which is a major cause of system crashes. Our implementation is modular, which was achieved by aspect-oriented programming, and can therefore be applied to arbitrary C++ software easily. We evaluated the effectivity of this mechanism empirically in a case study using an embedded weather-station software. The results are promising: All bit flips in vptrs are instantly recovered, and 96.4 % of all bit flips in pointers to objects with virtual methods are detected. Thus, 67.1 % of all system crashes are avoided at a cost of less than 10 % runtime and memory overhead.

Our fault assessment revealed that, beside vptrs and object pointers, the program stack is also a frequent point of failure (baseline assessment: 17.4 % of all irregular program terminations). We plan to develop dedicated dependability aspects for this type of fault in order to further improve the reliability of C++ software.

## References


[AÖK10] Ruben Alexandersson, Peter Öhman, and Johan Karlsson. Aspect-oriented implementation of fault tolerance: an assessment of overhead. In *Proceedings of the 29th international conference on Computer safety, reliability, and security*, SAFECOMP'10, pages 466–479, Berlin, Heidelberg, 2010. Springer-Verlag.

[ASMT07] Francisco Afonso, Carlos Silva, Sergio Montenegro, and Adriano Tavares. Applying aspects to a real-time embedded operating system. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, ACP4IS '07, New York, NY, USA, 2007. ACM.

[AVFK01] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. GOOFI: Generic Object-Oriented Fault Injection Tool. In *Proceedings of the 31st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '01)*, pages 83–88, Los Alamitos, CA, USA, 2001. IEEE Computer Society Press.

[BADAD+08] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. *Trans. Storage*, 4(3):1–28, 2008.

[Bor05] Shekhar Y. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[DCCC08] Francis M. David, Ellick Chan, Jeffrey Carlyle, and Roy H. Campbell. Qinject: A virtual-machine based fault injection framework. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, 2008. (Poster Presentation).

[DG06] Robert Dewar and Franco Gasperoni. Safety and OOP. In *System Safety, 2006. The 1st Institution of Engineering and Technology International Conference on*, pages 146 –157, jun. 2006.

[DR03] Brian Demsky and Martin Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 78–95, New York, NY, USA, 2003. ACM.

[DYdS+10] Marc Duranton, Sami Yehia, Bjorn de Sutter, Koen de Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. The HiPEAC Vision. Technical report, Network of Excellence on High Performance and Embedded Architecture and Compilation, 2010.

[Ins03] The British Standards Institute. *The C++ Standard (Incorporating Technical Corrigendum No. 1 )*. John Wiley & Sons, Inc., second edition, 2003. Printed version of the ISO/IEC 14882:2003 standard.

[KB96] Bradley M. Kuhn and David W. Binkley. An enabling optimization for C++ virtual functions. In *Proceedings of the 1996 ACM symposium on Applied Computing*, SAC '96, pages 420–428, New York, NY, USA, 1996. ACM.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[Lip96] Stanley B. Lippman. *Inside the C++ object model*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[LSSP06] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and Efficient System Software Product Lines: Where Aspects Beat Objects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 227–255. Springer-Verlag, 2006.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the EuroSys 2006 Conference (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

[RCD+04] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. In *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI '04)*, pages 303–316, Berkeley, CA, USA, 2004. USENIX Association.

[SHK+12] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *25th International Conference on Architecture of Computing Systems (ARCS '12), Workshop Proceedings*, volume 200 of *Lecture Notes in Informatics*, pages 201–210. German Society of Informatics, March 2012.

[SL07] Olaf Spinczyk and Daniel Lohmann. The Design and Implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[SPS09] Matthias Sand, Stefan Potyra, and Volkmar Sieh. Deterministic high-speed simulation of complex systems including fault-injection. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '09)*, pages 211–216. IEEE Computer Society Press, July 2009.

[VT05] Kalyanaraman Vaidyanathan and Kishor S. Trivedi. A Comprehensive Model for Software Rejuvenation. *IEEE Transactions Dependable and Secure Computing*, 2(2):124–137, April 2005.