# Fault Tolerance with Aspects: A Feasibility Study

Sven Karol Norman A. Rink Bálint Gyapjas Jeronimo Castrillon
Chair for Compiler Construction, cfaed, TU Dresden, Dresden, Germany
forename.surname@tu-dresden.de

## Abstract

To enable correct program execution on unreliable hardware, software can be made fault-tolerant by adding program statements or machine instructions for fault detection and recovery. Manually modifying programs does not scale, and extending compilers to emit additional machine instructions lacks flexibility. However, since software-implemented hardware fault tolerance (*SIHFT*) can be understood as a cross-cutting concern, we propose aspect-oriented programming as a suitable implementation technique. We prove this proposition by implementing an *AN* encoder based on *AspectC++*. In terms of performance and fault coverage, we achieve comparable results to existing compiler-based solutions.

***Categories and Subject Descriptors*** D.4.5 [*Reliability*]: Fault-tolerance; D.3.3 [*Language Constructs and Features*]

***Keywords*** Aspect-oriented programming, generative programming, arithmetic codes, *AN* encoding, *SIHFT*

## 1. Introduction

Shrinking transistors for performance is no longer the ultimate goal in microchip production. With the rise of smart embedded devices, energy efficiency and manufacturing costs become equally important. This motivates the reduction of gate voltages or even the use of partially defective processors. Both approaches lead to less reliable hardware and hence faulty program execution, as reported in [9, 18]. To enable correct and reliable execution of applications on unreliable hardware, programs can be augmented with instructions for fault detection and recovery, which is referred to as *software-implemented hardware fault tolerance* (*SIHFT*) [4]. *SIHFT* is an attractive and cheap alternative to hardware-based solutions, as it can be flexibly applied on existing commodity hardware.



*MODULARITY'16*, March 14–17, 2016, Málaga, Spain
ACM. 978-1-4503-3995-7/16/03...$15.00
http://dx.doi.org/10.1145/2889443.2889453

An obvious way of providing *SIHFT* is to manually rewrite the program by adding statements for fault detection [11] and recovery. While this approach offers maximum flexibility, it has to be repeated for each new program. Furthermore, this leads to a *tangling* of *SIHFT*-related statements with statements that implement the program's core functionality—as is typical of *cross-cutting concerns* [20]. Hence, in applying *SIHFT*, automated approaches are desirable, and compile-time approaches are typically used. For instance, *SIHFT*-related statements can be added by transforming a program's abstract syntax tree (AST) [7, 12]. Transformations can be implemented at more low-level program representations by extending compilers [3, 13–15].

While automated approaches avoid tangling and are applicable to arbitrary programs, some drawbacks remain. First, developing these *use-case-specific weavers* based on existing tools typically involves writing boilerplate code, e.g., for AST traversal. Second, these weavers have limited flexibility when it comes to modifying the *SIHFT* scheme or selecting the scope in which *SIHFT* is to be applied.

In this paper, we investigate and evaluate aspect-oriented programming (AOP) [5] as an approach to *SIHFT*. This has several advantages over existing approaches. First, *SIHFT*-related code is cleanly separated from the program code. Second, AST traversal and transformation are handled by an aspect weaver. Third, the *SIHFT* scheme is implemented more declaratively. Finally, *AOP* makes it easy to modify, combine or create new *SIHFT* schemes. For our investigation we focus on the simple, yet powerful fault detection scheme known as *AN* encoding [1, 3], and we implement an aspect-based *AN* encoder (*ABAN*).

## 2. Related Work

The authors in [2] describe a generic software-based approach to memory-error correction for memories without error-correcting codes. Similar to the present work, *AOP* is used to encapsulate the encoding. However, in [2] only object-oriented data structures in a specific embedded operating system are considered. Our work is more general since it can be applied to arbitrary data structures. Moreover, *AN* encoding can detect faults in CPUs *and* memory.

The *dual modular redundancy* (DMR) scheme detects faults by duplicating instructions and comparing their re-

sults. Although DMR was originally developed as a hardware approach, it can be implemented in software too. Software-based DMR was implemented using AST transformations [12] and compilers [13].

## 3. *AN* Encoding

*AN* encoding enables the detection of hardware faults by operating on encoded data. Only correctly encoded data is valid. If a fault occurs, the encoding is destroyed, which can be detected by an appropriate check. Specifically, any integer value $n$ is encoded by multiplying it with a fixed constant $A$:

$$n_e = n * A. \tag{1}$$

Fault detection and recovery schemes usually rely on redundancies in the representation of data. In *AN* encoding redundancy is introduced in the form of extra bits that must be afforded to correctly represent the encoded value $n_e$.

Decoding of $n_e$ is achieved by dividing it by $A$, yielding the original $n$. Validity of encoded values can be checked at any point in time by evaluating the boolean expression:

$$0 = n_e \ \% \ A. \tag{2}$$

In an *AN* encoded program, since arithmetic operations do not necessarily preserve the encoding, they must be replaced with encoded versions. The encoded versions of addition and subtraction are identical to the usual ones. For multiplication and division the situation is more involved (cf. [17, p. 27]):

$$n_e *_e m_e \ = \ (n_e * m_e)/A, \tag{3}$$

$$n_e /_e m_e \ = \ (A * n_e)/m_e. \tag{4}$$

Observe that the order of computation steps matters: if, for multiplication, one of the operands was first divided by $A$, an un-encoded intermediate value, vulnerable to hardware faults, would result. Other operations, e.g., bitwise operation, need special treatment too, for which refer to [17].

## 4. Aspect-Based *AN* Encoding (*ABAN*)

In *AOP*, a program is made up of four components: basic compilation units containing class declarations and their enclosed definitions (*core concerns*), aspects (*cross-cutting concerns*), *pointcuts*, and *advice functions* (cf. [5, 6]). Core concerns are the functional and algorithmic core of the program, which is typically self-contained and functionally independent of any aspect. Aspects define additional compilation units that implement cross-cutting functionality. Advice functions and pointcuts define how aspects cross-cut the functional core of a program. A pointcut defines a set of *joinpoints* in a program's control flow where associated advice functionality may augment the core functionality. To compile a program with aspects, a *weaver* is required. Weavers take as input the compilation units of the core program and aspects. From this a modified version of the core program is produced that includes the aspects' advice code at the respective joinpoints.

### 4.1 *AN* Encoding as a Cross-Cutting Concern

To use *AOP*, places where *AN* encoding cross-cuts core programs must be identified:

- Constant values must be encoded at object instantiation time. Hence, an aspect must be provided that applies an encoding advice at the joinpoints corresponding to object constructors.
- Values must be decoded when a program generates output. Hence, an aspect must be provided that applies a decoding advice at "output" joinpoints.
- To replace operations with their encoded versions (cf. Sec. 3), an aspect must be provided consisting of joinpoints which match un-encoded operators and of advice functions which implement the encoded operators.

While the set of aspects required for a basic *AN* encoder is considerably small, we are currently not aware of any *AOP* tool that allows us to implement *ABAN* directly. As a prominent example, *AspectJ* [6] does not support joinpoints which can address and modify operators. The tool that comes closest to meeting our requirements is *AspectC++* [19], which supports operator-execution joinpoints. However, at the time of writing, *AspectC++* does not support matching and advising built-in operators. This can be overcome by implementing wrapper classes for integer values. Joinpoints can then be defined at operators which the wrappers overload. We therefore decided to use this *AspectC++* for our implementation.

### 4.2 Implementation in *AspectC++*

We define the wrapper classes `Int`, `u_Int`, and `Bool` that encapsulate values of type `int64_t`, `uint64_t`, and `bool` respectively. Furthermore, the wrapper classes overload the built-in operators for their encapsulated types.

Our implementation of *ABAN* is comprised of five aspect declarations. `an_encoding` defines the constant $A$ that is shared across all aspects. The aspects `an_Int`, `an_uInt`, and `an_Bool` specify encoding and decoding operations. They also supply advices that lift the basic operators to encoded ones. Listing 1 exemplifies how the initial encoding step for integer values is realized using *AspectC++*. The advice is applied before the `construction` joinpoint of any `Int` object and performs the encoding by delegating to the `encode` method. For the other types this works similarly.

```
1 advice construction("Int") : before() {
2   int n_args = JoinPoint::args();
3   if( n_args == 1 && JoinPoint::argtype(0) == "1") {
4     Int* p_Int = (Int*)tjp->target();
5     encode(p_Int);
6 }}
```

**Listing 1:** Encoding `Int` on initialization (in `an_Int`).

As explained in Sec. 3, some arithmetic operations must be adjusted. As an example, we discuss the division operator. According to Eq. 4, it should suffice to encode the dividend a second time. However, since integer division is performed, a correction term must be introduced to handle remainders. This is shown in Listing 2. Before any division, the advice first computes the remainder using the modulo operation (Line 9–10), subtracts it from the dividend (Line 12), and encodes the modified dividend a second time (Line 13).

```
1  // matching division on Int
2  pointcut division() = "% Int::operator /=(const Int&)";
3
4  // adding advice before division execution
5  advice execution(division()) : before(){
6    Int* dividend = (Int*)tjp->that();
7    Int* divisor  = (Int*)tjp->arg(0);
8
9    int64_t rem = ((dividend->getN())/A)
10           % ((divisor->getN())/A);
11
12   dividend->n -= rem * A; // subtracting the remainder
13   encode(dividend); // encoding the dividend
14 }
```

**Listing 2:** Encoding of division operator (from `an_Int`).

The aspect `an_compchk` is responsible for applying a simple checking strategy: wherever an operation is executed, operands are first checked for valid encoding. This is shown in Listing 3. The first advice (Line 3–4) adds a check before unary operators on `Int` objects (pointcut `op_unary_Int`). The second advice (Line 7–10) does the same for binary infix operators on `Int`s (pointcut `op_infix_binary_to_Int`).

```
1  aspect an_compchk : an_Int , an_uInt , an_Bool {
2    // check unary operators
3    advice execution(op_unary_Int()) : before(){
4      an_Int::check_validity( (Int*)tjp->that() ); }
5
6    // check binary operators
7    advice execution(op_infix_binary_to_Int()): before(){
8      an_Int::check_validity( (Int*)tjp->that() );
9      an_Int::check_validity( (Int*)tjp->arg(0) );
10   }...
```

**Listing 3:** Checking `Int` operands (in `an_compchk`).

## 5. Evaluation

The *ABAN* encoder is evaluated on two benchmark algorithms: **Matrix-Vector Multiplication (MV)** and **Quicksort (QS)**. MV features many arithmetic operations and should be well-protected against faults by *AN* encoding. QS, however, has dynamic control-flow and few arithmetic operations. It is therefore not clear how well *AN* encoding in general can protect QS. These benchmarks were also chosen to enable comparisons of *ABAN* with previous implementations of *AN* encoding. For each of the benchmark algorithms three binaries are generated: the *native* binary, without any modifications, a binary using the *wrapper* classes, and the *ABAN-encoded* binary. All binaries are compiled with `clang` at optimization level `O2`.

Two metrics are commonly used to evaluate *SIHFT* schemes: *fault coverage* and *performance penalty*. Fault coverage is the frequency of faults that are successfully detected or do not affect program output. Note that faults can be detected outside *SIHFT* schemes: for example, if a fault causes a segmentation violation, this will be detected by the operating system. Performance penalty means the run-time overhead incurred due to the program transformations that are introduced by the *SIHFT* scheme. In the following we report measurements for both metrics that were obtained on an Intel Core i7 CPU running at 3.6GHz with 32GB RAM.

### 5.1 Fault Injection Experiments and Fault Coverage

For each binary 16,000 fault injection experiments are conducted. Using the Pin tool [8], a random single fault, consisting of one or multiple bit-flips, is injected during program run-time into one of the following places: registers, memory, the address bus, or instruction opcodes. Injecting a fault into a program results in one of five events. *CORRECT*: the fault does not affect the program output; *DETECTED*: the fault is detected by the *SIHFT* scheme; *OSCRASH*: the fault causes the operating system to terminate the program; *HANG*: the fault causes the program to hang; *SDC*: silent data corruption occurs when the program terminates normally but produces incorrect output. Based on the given definition, fault coverage equals the frequency of non-SDC events.

The results of our fault injection experiments are shown in Fig. 1. The native binaries already show good fault coverage, which is raised by *ABAN* to $0.90$ (MV, encoded) and $0.97$ (QS, encoded). The results for QS are comparable to the ones in [3, 14]. For encoded MV the fault coverage is noticeably better than in [15]. This is unsurprising given that fault coverage for native MV is already as good as for encoded MV in [15]. It is unclear why the wrapper classes introduce additional vulnerabilities for MV but not for QS.

### 5.2 Performance Penalty

Performance penalties are shown in Fig. 2 as ratios of execution times. An optimizing compiler would be expected to remove any overhead due to the wrapper classes. The bars *wrapper/native* show that this is too optimistic an expectation. However, the overhead that *ABAN* introduces on top of the wrapper classes (cf. *encoded/wrapper*) is remarkably low. This may be due to the fact that encoding on top of the wrapper classes adds opportunities for exploiting instruction level parallelism [10], but this conjecture remains to be verified. The *encoded/native* ratios are comparable to overheads of *AN* encoding that have been reported elsewhere [14, 15]. When comparing with [14], one should bear in mind that only a minimal number of checking instructions is introduced by their *AN* encoder. Nonetheless the performance penalties are of the same order of magnitude.

### 5.3 Suitability of AOP

Our experiments show that *AOP* is a viable and convenient approach to *SIHFT*. The *AN*-related aspect code is fully decoupled from program code, and the amount of boilerplate code is minimal. This strongly suggests that *ABAN* has fewer lines of code than previous approaches. *ABAN* is also more portable than compiler-based approaches since it can be used whenever the weaver is available.

*AOP* enables easy experimenting with different *SIHFT* strategies by modifying advice functions and pointcuts, e.g., using control-flow-specific pointcuts, *SIHFT* can be applied selectively at run-time. *AOP* is thus well-suited for prototyping new *SIHFT* strategies. The performance impact of the

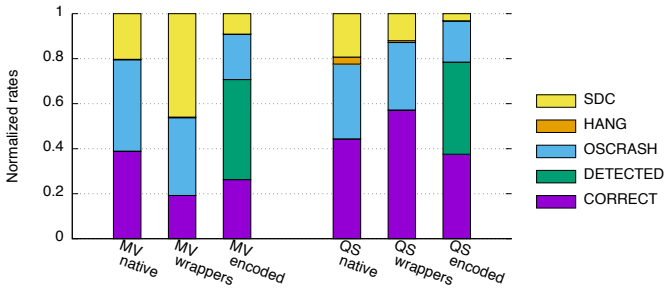


**Figure 1:** Fault coverages for different binaries.

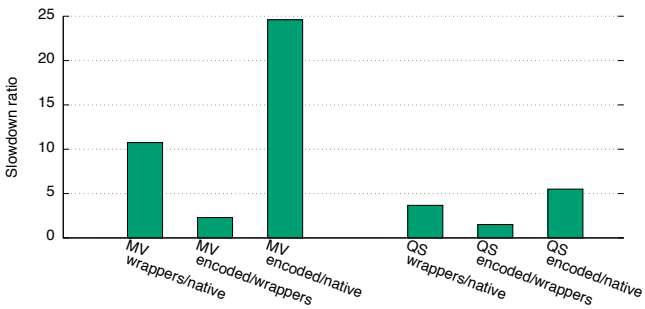


**Figure 2:** Performance penalties due to wrappers and *ABAN*.

wrapper classes may vanish in the future since pointcuts at built-in operators are planned in *AspectC++* [16].

One of the downsides of *ABAN* is that the joinpoint models of existing *AOP* tools might be too coarse to realize extensions of *AN* (cf. [17]) which require adapting if-statements. Another drawback is the lack of control over how the weaving affects generated code. However, compiler-based approaches suffer from this too unless all stages of the compilation process are suitably modified.

## 6. Conclusion and Outlook

We have introduced *ABAN* as an alternative approach to implementing *AN* encoding, discussed its fault-coverage and performance and compared it to other implementation approaches with reasonable results. In the future, we intend to investigate selective encoding strategies and more sophisticated encodings such as $\Delta$-encoding [7]. Also, the impact of the wrapper classes on performance and compilation should be investigated more thoroughly.

## Acknowledgments

This work was supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed).

## References

[1] A. Avizienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Trans. on Computers*, C-20(11):1322–1331, 1971. ISSN 0018-9340.

[2] C. Borchert, H. Schirmeier, and O. Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proc. of DSN'13*, pages 1–12. IEEE, 2013.

[3] C. Fetzer, U. Schiffel, and M. Süßkraut. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *Proc. of SAFECOMP'09*, LNCS/5775. Springer, 2009.

[4] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, 2006. ISBN 0387260609.

[5] G. Kiczales, A. Mendhekar, J. Lamping, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP'97*, LNCS/1241. Springer, June 1997.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of ECOOP'01*, LNCS/2072, pages 327–353. Springer, 2001.

[7] D. Kuvaiskii and C. Fetzer. $\Delta$-encoding: Practical encoded processing. In *Proc. of DSN'15*. IEEE, June 2015.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. of PLDI'05*, pages 190–200. ACM, 2005.

[9] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proc. EuroSys'11*. ACM, 2011.

[10] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Trans. on Reliability*, 51(1):63–75, 2002.

[11] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Proc. of DFT'99*, pages 210–218. IEEE, 1999.

[12] M. Rebaudengo, M. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Proc. of SCAM'01*, pages 33–42. IEEE, 2001. .

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. of CGO'05*, pages 243–254. IEEE, 2005.

[14] N. A. Rink and J. Castrillon. Improving code generation for software-based error detection. In *Proc. of REES'15*. To appear, 2015.

[15] N. A. Rink, D. Kuvaiskii, J. Castrillon, and C. Fetzer. Compiling for resilience: the performance gap. In *Proc. of ERPP'15*. Edacentrum, 2015.

[16] Roadmap of AspectC++. URL `http://www.aspectc.org/Roadmap.php`. Visited: 2015-11-06.

[17] U. Schiffel. *Hardware Error Detection Using AN-Codes*. PhD thesis, TU Dresden, Germany, 2011.

[18] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM errors in the wild: A large-scale field study. In *Proc. of SIGMETRICS'09*, pages 193–204. ACM, 2009.

[19] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to the C++ programming language. In *Proc. of CRPIT'02*. Aus. Com. Soc., 2002.

[20] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. of ISCE'99*, pages 107–119. IEEE, 1999.