Trading Fault Tolerance for Performance in AN Encoding

Norman A. Rink
Center for Advancing Electronics Dresden
Technische Universität Dresden, Germany
norman.rink@tu-dresden.de

Jeronimo Castrillon
Center for Advancing Electronics Dresden
Technische Universität Dresden, Germany
jeronimo.castrillon@tu-dresden.de

ABSTRACT

Increasing rates of transient hardware faults pose a problem for computing applications. Current and future trends are likely to exacerbate this problem. When a transient fault occurs during program execution, data in the output can become corrupted. The severity of output corruptions depends on the application domain. Hence, different applications require different levels of fault tolerance. We present an LLVM-based *AN encoder* that can equip programs with an error detection mechanism at configurable levels of rigor. Based on our AN encoder, the trade-off between fault tolerance and runtime overhead is analyzed. It is found that, by suitably configuring our AN encoder, the runtime overhead can be reduced from 9.9× to 2.1×. At the same time, however, the probability that a hardware fault in the CPU will result in silent data corruption rises from 0.007 to over 0.022. The same probability for memory faults increases from 0.009 to over 0.032. It is further demonstrated, by applying different configurations of our AN encoder to the components of an arithmetic expression interpreter, that having fine-grained control over levels of fault tolerance can be beneficial.

CCS CONCEPTS

• **Software and its engineering → Software reliability**; **Software fault tolerance**; **Compilers**; • **Computer systems organization** → *Reliability*;

KEYWORDS

transient hardware faults, soft errors, resilience, error detection, code generation, LLVM, fault injection

ACM Reference format:
Norman A. Rink and Jeronimo Castrillon. 2017. Trading Fault Tolerance for Performance in AN Encoding. In *Proceedings of CF'17, Siena, Italy, May 15-17, 2017,* 8 pages.
DOI: http://dx.doi.org/10.1145/3075564.3075565

1 INTRODUCTION

Transient hardware faults are a widely occurring problem [20, 30, 41] and can have as malicious effects as entire system outages [1]. Transient faults, also referred to as *soft errors*, are commonly attributed to charge generation by cosmic radiation [3], and attention has been drawn to increasing fault rates for some time [3, 6, 8, 43]. As feature sizes continue to shrink, devices are becoming more vulnerable to variations in supply voltage and temperature [6, 42]. Therefore, in the future, fault rates will likely increase further.

The current trend toward reducing the energy consumption of chips exacerbates the problem of faults. For example, operating devices at near-threshold voltage lowers reliability, cf. *dim* and *dark silicon* [12, 42, 44], and reducing the power supplied to memory modules lowers their capability to retain data [13, 28, 46]. In the context of more disruptive trends, e.g. quantum computing, faults are an even more prominent problem.

In the presence of hardware faults, software that is intended to satisfy high safety and reliability requirements must be made fault-tolerant. This can be achieved by adding integrity checks to programs: when a check fails, an error has been detected and suitable measures can be taken to recover from it. The present work focuses on error detection.

It is convenient to equip software with error detection measures by automatic program transformations, e.g. source transformations [24, 26, 34] or code transformations facilitated by the compiler [33, 35, 37, 40]. Popular schemes detect errors by duplicating the data flow of programs [11, 14, 33, 35, 47, 48]: faults lead to disagreement between duplicated copies of data, and errors are detected when disagreement is found. This approach is referred to as *dual modular redundancy* (DMR).

DMR-based error detection schemes typically make the assumption that memory is protected against faults by hardware measures, e.g. ECC. This assumption is problematic for two reasons. First, cost and area considerations may rule out using ECC memory at all levels of the memory hierarchy, especially in on-chip caches and load-store queues [11]. Second, it has been found that the widely used *single error correcting, double error detecting* (SECDED) codes are incapable of handling large fractions of error patterns that occur in practice [23]. Simply extending DMR to memory, i.e. by duplicating memory accesses, is not an option since this may lead to race conditions in multi-threaded applications [11].

An alternative approach to error detection is based on *encoded processing* [39]: data words are encoded, and programs operate exclusively on code words. Errors are detected when non-code words are encountered. The advantage of encoding-based error detection is that entire systems, including CPUs, memories, and communication buses, are automatically protected against faults [17]. Moreover, no issues arise for multi-threaded applications since memory accesses need not be duplicated. However, without dedicated hardware support, encoding-based error detection schemes incur large runtime overheads.

In the present work, we study the simple, yet effective, *AN encoding* scheme [7, 17]. In AN encoding, integer values are encoded by multiplying with a fixed constant A. Hence, the valid code words

CF'17, Siena, Italy

978-1-4503-4487-6/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3075564.3075565

are precisely the multiples of A. This implies that $\lfloor 2^k/A \rfloor$ encoded values can be represented by k bits. AN encoding can be used to detect errors in the data flow of programs that process integer values. This introduces two major sources of runtime overhead: (i) error checks require expensive modulo operations; (ii) some native integer operations must be modified to ensure that the results are again multiples of A. Typical overheads due to AN encoding are several 10× [37, 40], noticeably higher than for DMR-based error detection schemes. Our work identifies code generation strategies that improve the performance of AN encoding. The price for this is a reduction in the rigor of error detection.

While safety-critical applications, e.g. in the automotive or aerospace domain, may require that practically all errors resulting from hardware faults be detected and handled appropriately, other applications have less strict demands, e.g. applications with frequent or continuous user interaction [14], or image processing and machine learning applications [28, 38]. For such applications, less rigorous error detection can still ensure operation within reliability requirements in an environment that is prone to hardware faults; and applying error detection too eagerly would lead to unnecessarily big runtime overhead. This calls for error detection and fault tolerance schemes that can be adjusted and can thus suitably trade rigor for performance. In this work, we make the following contributions toward turning the basic AN encoding scheme into such an adjustable error detection scheme:

(1) We present a configurable AN encoder that can apply error detection measures at different levels of rigor, trading fault tolerance for performance.[1] The encoder is based on the LLVM compiler infrastructure [27].
(2) When code generation strategies that are specific to AN encoding are applied to programs that have already been optimized, the average overhead can be reduced from 9.9× to 3.6×. The corresponding degradation in the capability to detect errors is analyzed.
(3) When pointers are not encoded, the runtime overhead can be further reduced to 2.1×.
(4) Using an interpreter as an example application, we show that it can be beneficial to utilize different trade-offs for different application components.

The structure of this paper is as follows. The AN encoder and its configurations are introduced in Section 2. Section 3 defines the fault model used to evaluate the AN encoder in Section 4. Section 5 discusses related work. Section 6 summarizes our work and gives directions for future research.

2 AN ENCODING

Error detection schemes rely on invariant conditions that are satisfied by valid data, and the violation of an invariant condition indicates the presence of an error. In encoding-based error detection, the invariant condition is that all data words are in fact valid code words. Errors can be detected effectively if the set of code words is only a small subset of all possible data words since, in this situation, a fault that causes a bit flip in a code word is unlikely to produce another valid code word. In AN encoding, the code words are precisely the multiples of a fixed integer constant A. Thus, errors are detected by checking whether an integer value n satisfies

$$n \bmod A = 0. \tag{1}$$

If (1) does not hold, an error must have occurred. Checking for erroneous values enables AN encoding to detect errors in a program's data flow. AN encoding does not detect erroneous control flow.

In processing AN-encoded data, there is never a need to operate on values that are not multiples of A. Nevertheless, when implementing AN encoding on real hardware, an interface must be defined between AN-encoded and conventional processing of data. Specifically, an arbitrary integer m must be *encoded* before being used in AN-encoded processing:

$$m_{enc} = \mathsf{encode}(m) \equiv m \cdot A. \tag{2}$$

Similarly, when passing a value from AN-encoded processing to conventional processing, it must be *decoded*:

$$m = \mathsf{decode}(m_{enc}) \equiv m_{enc}/A. \tag{3}$$

The right members of (2) and (3) should be read as definitions of the operations encode and decode respectively.

AN encoding introduces redundancy because additional bits are required to represent multiples of A: if m is represented by k_m bits and the constant A has k_A bits, then $k_m + k_A$ bits must be afforded to m_{enc}. Therefore, in theory, the memory overhead introduced by AN encoding is k_A bits per data word. In practice, however, the memory overhead is typically lower for the following reason: if the machine data word is k bits wide, then the binary representation of m leaves $k - k_m$ bits unused; thus, provided $k - k_m > k_A$, no additional memory is needed to represent encoded values. Note also that $\lfloor 2^k/A \rfloor$ code words can be represented by k bits.

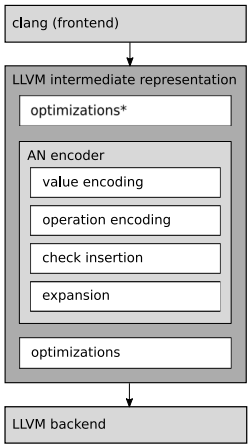

Figure 1: Compilation flow for AN encoding.

2.1 AN Encoding of Operations

AN encoding produces *arithmetic* codes, meaning that certain arithmetic operations, namely addition and subtraction, preserve code words, e.g.

$$m \cdot A + n \cdot A = (m + n) \cdot A\,. \tag{4}$$

[1] https://github.com/normanrink/an-encoder

Table 1: Configurations of transformation passes.

check insertion	A	B
$v_{\text{enc}} = \texttt{load}_{enc}\ a_{\text{enc}}$	$v_{\text{enc}} = \texttt{load}_{enc}\ a_{\text{enc}}$ $\texttt{check}(v_{\text{enc}})$	$v_{\text{enc}} = \texttt{load}_{enc}\ a_{\text{enc}}$
expansion	A	B
$\texttt{check}(v_{\text{enc}})$	$\texttt{an\_assert}(v_{\text{enc}}\ \%\ A)$	$\texttt{an\_accumulate}(v_{\text{enc}})$

Table 2: Encoding variants.

variant	load	check	pre-opt.
1	A	A	✗
2	B	A	✗
3	B	B	✗
po.1	A	A	✓
po.2	B	A	✓
po.3	B	B	✓

However, not all operations preserve multiples of A. For example, the result of a division is no longer a multiple of A:

$$m_{enc}/n_{enc} = (m \cdot A)/(n \cdot A) = m/n\,. \tag{5}$$

To remedy this, an encoded version of division must be used, i.e.

$$m_{enc}\ /_{enc}\ n_{enc} \equiv (A \cdot m_{enc})/n_{enc} = (m/n) \cdot A\,. \tag{6}$$

Some operations require that arguments be decoded, e.g.

$$n_{enc}\ \&_{enc}\ m_{enc} \equiv (n\ \&\ m) \cdot A \tag{7}$$

for the bitwise *and*-operation, denoted by the symbol &. This limits the error detection capability of AN encoding since errors in the non-encoded values m, n, or $m\ \&\ n$ cannot be detected, cf. [36]. Since address buses cannot handle encoded addresses, memory operations also require that pointer arguments be decoded, i.e.

$$\texttt{load}_{enc}\ a_{enc} \equiv \texttt{load}\ a\,, \tag{8}$$

$$\texttt{store}_{enc}\ n_{enc}, a_{enc} \equiv \texttt{store}\ n_{enc}, a\,, \tag{9}$$

where a is a pointer and $a_{enc} = \texttt{encode}(a)$. More details on encoded operations can be found in [39].

2.2 AN Encoding of Programs

Error detection can be added to a program by transforming the program's processing of integers into AN-encoded processing. The resulting program is *AN-encoded*. Transforming a program to its *AN-encoded* version is achieved by the following steps, cf. Figure 1.

(1) *Value encoding* replaces every integer constant m in the program with m_{enc}.
(2) *Operation encoding* replaces all operations with their corresponding encoded versions. E.g., the operation & is replaced with $\&_{enc}$.
(3) *Check insertion* decides which values are checked for errors. An abstract operation check is introduced to represent this.
(4) *Expansion* turns encoded operations and checks into native machine operations. The $\&_{enc}$-operation is expanded as on the right-hand side of (7); expansions of memory operations are defined by (8), (9); and checks can be implemented using (1).

The transformations (1)–(4) have been implemented based on the LLVM framework [27], and each of them corresponds to a transformation pass that operates on LLVM intermediate representation (IR). Optimizations at level -O3 are applied before and after AN encoding.[2] However, optimizations before AN encoding are optional, as indicated by the * in Figure 1, and we refer to these as *pre-optimizations*. Since AN encoding adds complexity to a program's data flow, it may be difficult for a compiler to identify opportunities for optimization in an AN-encoded program. Hence, it is reasonable to optimize programs prior to encoding.

2.3 Configurable AN Encoding

The check insertion pass and the expansion pass can be configured to generate different variants of AN-encoded programs. For the discussion of configurations it is important to note that AN-encoded programs produce output in one of only two ways: either an encoded value is written to memory, or it is passed to a library function, in which case the value must be decoded since library functions may not work with encoded data. Therefore, to detect errors in program output, one must check for errors every time a value is written to memory or decoded.

Based on this observation, the check insertion pass applies the following rule to transform store operations,

$$\texttt{store}_{enc}\ n_{enc}, a_{enc} \longrightarrow \begin{array}{l}\texttt{check}(n_{enc})\\ \texttt{store}_{enc}\ n_{enc}, a_{enc}\end{array}\,.$$

Whether checks are also inserted after load operations depends on which of the configurations from Table 1 is chosen. Configuration A can immediately detect if an error in memory has corrupted a value. However, checking immediately after a load operation may not be necessary due to the *error propagation property* of AN encoding: whenever an error corrupts a value so that it is no longer a multiple of A, it is unlikely that this will be rectified by subsequent computations.[3] Hence, the corruption can be detected by a check at a later point in time, i.e. at the next store or decode operation. Omitting checks which are deemed unnecessary, as in configuration B of the check insertion pass (see Table 1), is a common approach to reducing the overhead of error detection, cf. [35, 47].

The expansion pass is responsible for checking every time a value is decoded. The transformation rule used for this is

$$v = \texttt{decode}(v_{\text{enc}}) \longrightarrow \begin{array}{l}v = v_{\text{enc}}/A\\ \texttt{an\_assert}(v_{\text{enc}} - v \cdot A)\end{array}\,,$$

where an_assert is defined in Listing 1. Additionally, the expansion pass can transform checks in one of the two ways defined in Table 1 (see Listing 1 for the definition of an_accumulate). Configuration B should lead to lower runtime overhead since the function an_accumulate essentially performs an addition instead of the expensive modulo operation in configuration A. Configuration B can therefore be regarded as AN-encoding-specific strength reduction.

In an_accumulate, values to be checked are added to an accumulator acc. Each AN-encoded function maintains its own accumulator acc in a local variable. Since addition preserves multiples

[2] The sequence of optimization passes performed can be obtained by executing `llvm-as < /dev/null | opt -O3 -disable-output -debug-pass=Arguments`.

[3] If multiple errors occur, it is also unlikely that later errors will restore values to multiples of A when they have previously become corrupted by earlier errors.

of A, it suffices to check only once per function whether `acc` is a multiple of A. This is done immediately before the AN-encoded function returns. The fact that `acc` may overflow complicates the structure of `an_accumulate`, but this may be acceptable as long as overflow happens rarely.

Listing 1. Assertion and accumulation in AN encoding.

```
an_assert(x) {
  if (x) exit(AN_ERROR_CODE);
}

an_accumulate(x) {
  _acc = acc;
  acc += x;
  if (/* overflow has occured */) {
    an_assert(_acc % A);
    acc = x;
  }
}
```

Table 2 defines encoding variants obtained by combining different configurations of the check insertion and expansion passes. Variants *po.1–po.3* include the optional pre-optimizations from Figure 1. In our AN encoder, the constant A can also be configured. For the present work, it has been set to $A = 58659$, cf. [22].

3 FAULT MODEL

Encoded processing can protect entire computing systems against faults, including on-chip components, such as functional units and register files, as well as the full memory hierarchy. We therefore consider faults in both the CPU and in memory to assess the error detection capabilities of the encoding variants. The rarity of faults in practice justifies a single-event upset (*SEU*) model [6]; i.e. it is assumed that at most a single hardware fault occurs during the execution of a program.

In this work, error detection is assessed by *symptom-based fault injection* [4, 26, 39] This means that instead of simulating a fault at the circuit level, its effect on the executing program, i.e. the symptom of the fault, is modeled. It is common practice to evaluate error detection schemes by injecting single bit flips, cf. [11, 14, 47]. Therefore, a fault in the CPU is modeled by flipping a bit in a register. This covers faults in the register file and in functional units since the results of operations are generally written to registers. A fault in the memory system is modeled by a load operation that returns a value with one bit flipped. This subsumes faults in on-chip memories, i.e. load-store queues and caches, as well as in main memory.

Fault injection experiments are performed on *x86* binaries, and symptoms are injected by running a program binary under the control of the Pin dynamic instrumentation tool [29]. In a first *golden run*, the dynamically executed instructions are recorded. Based on this, all possible fault symptoms are determined. By simple combinatorics, the spaces of all possible symptoms are generally quite large, and we therefore perform statistical sampling. To inject a symptom into the CPU, an instruction in the targeted binary is chosen randomly from all dynamic instructions that write to registers. If the instruction writes to multiple registers, one of the output registers is randomly selected for injecting the symptom. The bit flip is injected into the selected register immediately after the instruction has executed. To inject a memory fault, one of the dynamically executed load operations is randomly selected, and a bit is flipped in the value accessed by the load.

Table 3: Suite of test programs.

	description
	bubblesort
A	inputs in ascending order
B	inputs in random order
C	inputs in descending order
D	cyclic redundancy checker (CRC-32)
E	DES encryption algorithm
F	Dijkstra's algorithm (shortest path)
	arithmetic expression interpreter
G	recursive expression tree evaluation
H	token lexer for arithmetic expressions
I	parser
J	Fibonacci numbers
K	matrix multiplication
L	array copy
	quicksort
M	inputs in ascending order
N	inputs in random order
O	inputs in descending order

4 EVALUATION

To obtain statistically significant results, program responses to a large number of fault injection experiments must be sampled, following the procedure described at the end of Section 3. To achieve this in manageable time, a suite of relatively small test programs, listed in Table 3, is used. Some of the test programs (D, F, M–O) appear in the MiBench suite [21], and the programs generally represent typical algorithmic tasks, such as sorting (A–C, M–O), manipulation of bit patterns (D, E), graph and tree traversal (F, G, I), and linear algebra (K). Assessing error detection schemes on small programs is meaningful since larger applications are composed of these algorithmic tasks. For example, the structure of a signal processing algorithm may be similar to D and E, and a full compiler will be made up of components like H and I. Sorting (A–C and M–O) and copying data (L) appear in numerous applications.

The number of fault injection experiments carried out for each test case has been hand-tuned so that (a) each executed instruction can, in principle, be targeted by at least one experiment, and (b) enough statistical significance is reached to distinguish between the encoding variants. For faults in the CPU, and for each encoding variant, between 9,600 and 230,000 fault injection experiments have been performed per test program, depending on the size of the program. For faults in memory, the number of experiments per encoding variant ranges from 2,400 to 9,600 per test program. Memory has been targeted by noticeably fewer fault injection experiments since only a small fraction of instructions in the test programs access memory.

Program responses to injected fault symptoms are classified into five categories.

1. *correct*: Despite the fault, the program has terminated normally and produced correct output.
2. *detected*: The error has been detected by AN encoding. As a result, the program has exited with the exit code `AN_ERROR_CODE`, cf. Listing 1.
3. *hang*: If the program runs for longer than 10× its normal execution time, it is deemed to hang and hence terminated. In practice, e.g., in safety-critical embedded applications, a hardware watchdog may terminate and restart long-running programs.

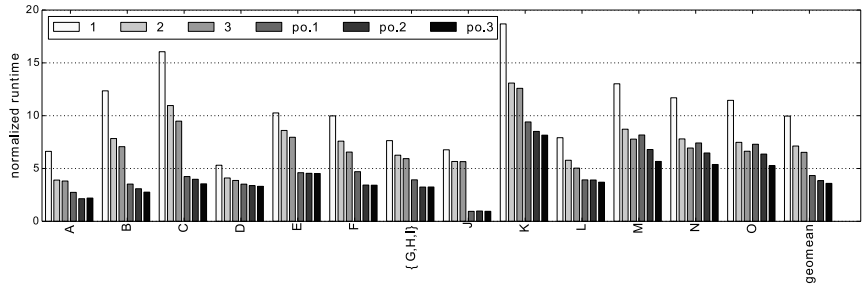

Figure 2: Runtime overheads of AN encoding.

Table 4: Runtime overheads (geometric means).

variant	overhead
1	9.9
2	7.2
3	6.5
po.1	4.3
po.2	3.8
po.3	3.6

4. *crash*: The program has terminated abnormally. Either the operating system has terminated the program, e.g., due to a segmentation fault, or the program itself has exited prematurely due to an error condition caused by invalid data.
5. *sdc*: Silent data corruption occurs when the program has terminated normally but has produced incorrect output.

In the following, we analyze how different encoding variants control the probability of *sdc*, namely

$$p_{sdc} = P(\text{“silent data corruption”} \mid \text{“hardware fault”}), \qquad (10)$$

i.e. the conditional probability of *sdc* given that program execution is affected by a hardware fault. The probability p_{sdc} can be estimated based on observed frequencies of program responses. Assessment of fault tolerance schemes based on these frequencies is commonly done in the literature [9, 25, 26, 33, 35, 47]. The values of p_{sdc} must be contrasted with the runtime overheads introduced by the different encoding variants. Figure 2 gives the runtimes of the AN-encoded test programs, normalized to the runtimes of the *plain* programs, i.e. without any form of AN encoding, compiled at optimization level -O3. Test cases G, H, I combine to form a larger application, and only the runtime overhead of the full application is reported. The last set of bars in Figure 2 gives the geometric means across the suite of test programs. The numerical values of these means are listed in Table 4.

4.1 Processor Faults

Relative frequencies of program responses to faults in the CPU are summarized by Figure 3. These frequencies are the result of our sampled fault experiments, and analogous results have been obtained for memory faults. To demonstrate that AN encoding is effective at reducing the frequency of *sdc*, the first bar in each group shows the responses of the *plain* test program.

Given the limitations of AN encoding mentioned in Section 2.1, it is not surprising that even in variant *1*, which is the most eager at error detection, non-zero *sdc* frequencies occur. It should also be noted that AN encoding may increase the frequencies of *correct* responses. This is because the instructions that AN encoding adds to a program do not affect the program's output. Errors in these instructions, even if not detected, cannot modify program output.

Since the quantities of interest are the *sdc* frequencies, they are reproduced in Figure 4 for better visibility. This time, however, frequencies are treated as estimates for p_{sdc}, and the error bars depict confidence intervals at the 0.95%-level. Results for *plain* programs have been omitted since they would set too large a scale for the plot. Numerical values for the mean p_{sdc}, including for *plain* programs, are listed in Table 5. Figure 4 shows that the encoding variants behave as expected: variants with lower runtime overhead lead to higher p_{sdc}. For many test programs there is no statistically significant difference in p_{sdc} between variants *2* and *3*. Despite this, variant *3* incurs slightly lower overhead than variant *2*. The same statements hold for variants *po.2* and *po.3*. Note that only variant *1* succeeds at keeping $p_{sdc} < 0.01$.

Table 5: Mean $p_{sdc} \times 1e3$.

fault type	*plain*	*1*	*2*	*3*	*po.1*	*po.2*	*po.3*
CPU	161.2	6.8	10.7	11.2	18.7	20.2	22.1
memory	480.0	8.8	26.8	32.9	19.4	20.5	32.3

4.2 Memory Faults

Estimates of p_{sdc} based on the injection of memory faults are depicted in Figure 5. The mean values listed in Table 5 show that memory faults are generally more likely to lead to *sdc* than faults in the processor. Moreover, according to Figure 5, the dependency of p_{sdc} on the level of optimizations performed is less clear than for faults in the CPU. This is because memory accessing instructions are relatively rare, and hence a single vulnerable memory access can have a strong influence on p_{sdc}.

The stark differences between the encoding variants in Figure 5 are caused by faults that affect values on the stack. Stack accesses are added to programs by the compiler backend to handle local variables, callee-saved registers, and register spills. Since the AN encoder operates at the level of IR, memory accesses that handle callee-saved registers or register spills cannot be protected against faults. Local variables, on the other hand, are present at the IR level, and hence are protected by AN encoding. However, variants other than *1* and *po.1* do not perform checks immediately after loading values from memory, and may therefore fail to detect errors that affect local variables. For example, the *sdc* occurring for variant *2* of *bubblesort* (B, C) is caused by a corrupted local variable on the stack. After the corrupted value has been loaded from the stack, incorrect control flow ensues, leading to *sdc* since protecting control flow is outside the scope of AN encoding. For variants *po.1* and *po.2*, the pre-optimizations promote local variables to registers, which is why there is no *sdc*. Optimizations applied after AN encoding, as in variants *2* and *3*, fail to promote local variables because of the

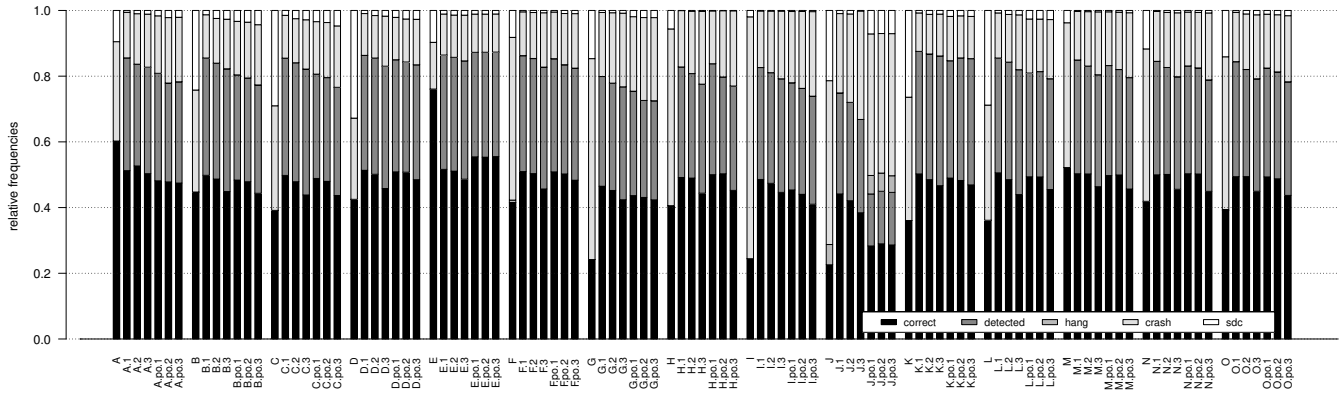

Figure 3: Program responses to faults in the CPU.

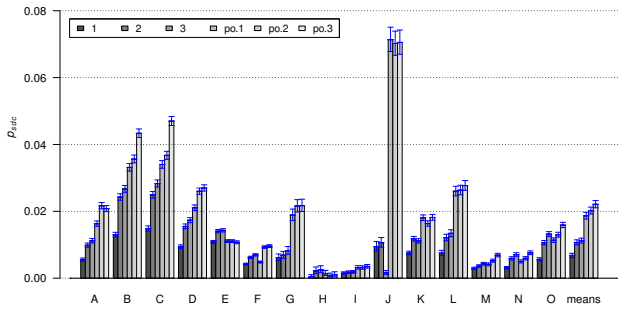

Figure 4: p_{sdc} for faults in the CPU.

complexity that AN encoding introduces into the program IR. In variants *3* and *po.3* a register spill occurs, which can be attributed to the long life range of the accumulator from Listing 1. Restoring a corrupted value to the spilled register again leads to incorrect control flow, and hence *sdc*. Similar behavior is seen for D and L.

Practically all *sdc* in variants *1–3* and *po.3* of the *lexer* (H) results from a corrupted value that is restored to a callee-saved register. The pre-optimizations in variants *po.1* and *po.2* produce code that

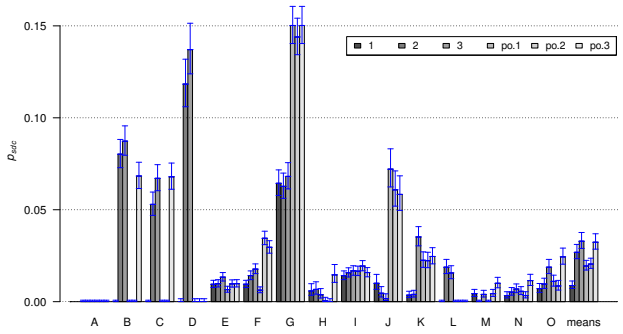

Figure 5: p_{sdc} for faults in memory.

does not make use of this register. In variant *po.3* the register is used again because of the long life range of the accumulator.

After pre-optimizations, the IR of the *Fibonacci* test case (J) that is passed into the AN encoder performs all computations in registers. Therefore, no checks are inserted by the AN encoder except when the final result is decoded. Hence, the same final code is generated for variants *po.1–po.3*. This explains why the runtime overheads are the same for these variants, cf. Figure 2, and why the confidence intervals for p_{sdc} overlap substantially.

4.3 Leaving Pointers Unprotected

When pointers are protected by AN encoding, they must be decoded every time memory is accessed. Since this requires an integer division, accompanied by a check, memory accesses are expensive in AN-encoded programs. If address buses could handle encoded addresses, there would be no need for decoding pointers, and thus the runtime overhead of AN encoding would be lower. To study the extent to which the overhead can be reduced, the encoding variants *npp.3* and *po.npp.3* have been derived from variants *3* and *po.3* respectively by *not protecting pointers (npp)*.

Table 6 summarizes the measured properties of variants *npp.3* and *po.npp.3*. The mean runtime overhead of variant *po.npp.3* is 2.1×, which is considerably lower than the typical several 10× that have been reported for AN encoding [40]. Moreover, the overhead of 2.1× is comparable with DMR-based schemes [33].

Table 6 also lists the p_{sdc} for variants *npp.3* and *po.npp.3*. Comparing these with Table 5 shows that not protecting pointers does not necessarily increase p_{sdc} noticeably. This can be explained by higher *crash* frequencies due to leaving pointers unprotected.

4.4 Mixed Encoding

Test cases G, H, I are taken from an arithmetic expression interpreter, which processes input that consists of integers and arithmetic operators in prefix notation. The input is broken up into tokens by the *lexer* (H). Valid tokens are integer constants, operator symbols, and parentheses. The *parser* (I) transforms the token sequence into a tree that represents an arithmetic expression. By traversing this expression tree, the *evaluator* (G) computes the integer value of the input expression.

Table 6: $p_{sdc} \times 1e3$. Confidence intervals at the 95%-level in parentheses. Geometric means of runtime overheads.

	CPU	memory	ohd.
npp.3	19.1 (17.9, 20.5)	33.1 (28.5, 38.9)	2.6
po.npp.3	23.9 (22.5, 25.4)	25.1 (21.7, 29.7)	2.1

To enable more fine-grained trade-offs between fault tolerance and performance, different encoding variants can be applied to the interpreter components, i.e. to *lexer*, *parser*, and *evaluator*. According to Figures 4 and 5, the *evaluator* (G) is the most vulnerable part of the interpreter application. We therefore apply the following *mixed* encoding: variant *3* is applied to the *evaluator* only, and the *lexer* and *parser* are treated with variant *po.3*.

Table 7 lists p_{sdc} estimates and measured runtime overheads for encoding variants *3*, *mixed*, and *po.3*. Less eager error checking gives rise to higher values of p_{sdc}, but also improves performance. However, the difference in p_{sdc} for CPU faults between variants *3* and *mixed* is not significant at the 95%-level. One could use Table 7 as follows. If, say, $p_{sdc} < 0.05$ is required for memory faults, variants *3* and *mixed* can be applied, and by using the *mixed* variant the runtime overhead is reduced by 26%, compared to *3*. This reduction would not be possible without the ability to apply different encoding variants to different application components.

5 RELATED WORK

A summary of software-based fault tolerance schemes appeared in [19]. Error detection by source code transformation appeared early [34] and has recently received renewed attention [24, 26]. Compiler-based implementations typically add error detection and recovery mechanisms to an intermediate or low-level representation of programs, e.g., [9, 11, 16, 25, 33, 35, 37, 40].

The DMR scheme EDDI [33] inserts duplication at the level of machine instructions, incurring a runtime overhead of about 2×. The SWIFT scheme [35] assumes that memory is protected by ECC and achieves a runtime overhead of less than 1.5× by removing redundant error checks. Similarly, the ESoftCheck scheme [47] reports an average overhead of about 1.5×. The Shoestring scheme acknowledges that not all applications require equally high levels of fault tolerance [14]. DMR-based error detection has recently been implemented using SIMD extensions [10].

AN encoding was introduced in [7] and studied in detail, among other *arithmetic error codes*, in [2, 18]. Protecting processors by AN encoding was suggested in [17], where the ANB and ANBD schemes were also introduced. While large runtime overheads have been reported for AN encoding and the derived ANB and ANBD-mem schemes in [40], the rates of *sdc* are very low at the same time. The ED4I [31] and the Δ-encoding [26] schemes combine DMR with AN encoding. Δ-encoding comes in different variants, the fastest of which leads to overheads between 2× and 4×. The present work has borrowed the concept of accumulating values to be checked from [26].

Schemes for detecting erroneous control flow typically assign signatures to basic blocks. Signatures are then checked dynamically to detect errors [32, 45]. The impact of compiler optimizations on control flow protection has been studied in [15].

Table 7: Interpreter application, $p_{sdc} \times 1e3$.

	CPU	memory	ohd.
3	4.6 (4.2, 5.1)	24.9 (22.6, 27.4)	5.9
mixed	5.1 (4.7, 5.7)	36.3 (33.5, 39.2)	4.7
po.3	7.3 (6.7, 7.9)	49.1 (45.9, 52.4)	3.3

Trade-offs between fault tolerance and hardware performance measures, i.e. area and power, were analyzed for two hardware-implemented schemes in [5]. All of the mentioned DMR schemes come in different variants that allow trading levels of fault tolerance for performance. To the best of our knowledge, the present work is the first comprehensive analysis of trade-offs in the AN encoding scheme.

6 SUMMARY AND OUTLOOK

We have presented a configurable AN encoder that adds error detection to programs in LLVM IR form. The studied variants of AN encoding produce program binaries with varying levels of fault tolerance and runtime overheads, and it has been found that the overhead of AN encoding can be reduced from 9.9× down to 3.6×, considerably less than previously reported in the literature [40]. While techniques specific to AN encoding have played a role in this, more performance can be gained if programs are optimized before applying AN encoding.

As the runtime overhead of AN encoding is reduced, the probability that a fault in the CPU leads to silent data corruption rises from 0.007 to over 0.022. However, this increase is gradual, implying that performance and tolerance of CPU faults can be adjusted according to an application's needs. The probability that a memory fault results in silent data corruption rises from 0.009 to over 0.032. More interestingly, the vulnerability of an application to faults in memory varies greatly for the studied encoding variants. The ways in which an application uses the stack has been identified as the reason for this variation. It would therefore make sense to pair our AN encoder with error detection measures that are inserted during a later compilation stage and that specifically target the stack. This would enable the protection of local variables, register spills, and callee-saved registers against faults in memory.

By not protecting pointers, the overhead of AN encoding can be reduced further to 2.1×, which is comparable with early DMR-based schemes [33]. This should motivate future work to aim for a reduction to below 1.5×, which is seen in more sophisticated and selective DMR schemes [14, 35, 47]. Alternatively, one might want to look into hardware support for AN encoding: if address buses could handle encoded addresses, the observed overhead of 2.1× could be achieved without losing the protection of pointers.

An arithmetic expression interpreter has been used to show that it can be beneficial to apply different encoding variants to different application components, based on how vulnerable individual components are to hardware faults. The decision which encoding variants to apply was based on the results of long-running fault injection experiments, which is not very practical. Hence, research is required into statical analyses that can estimate an application's vulnerability to hardware faults, ideally also depending on the chosen code generation and optimization methods.

Finally, future work should look into carrying the presented analysis of trade-offs over to other encoding-based error detection schemes, cf. [17, 22, 26, 40].

ACKNOWLEDGMENTS

This work was funded by the German Research Council (DFG) through the Cluster of Excellence 'Center for Advancing Electronics Dresden' (cfaed). The authors thank Sven Karol and Pramod Bhatotia for useful discussions and for proof-reading.

REFERENCES

[1] 2008. Amazon S3 availability event: July 20, 2008. (2008). http://status.aws.amazon.com/s3-20080720.html
[2] A Avizienis. 1971. Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. *IEEE Trans. on Computers* C-20, 11 (November 1971), 1322–1331.
[3] R Baumann. 2005. Soft Errors in Advanced Computer Systems. *IEEE Design & Test of Computers* 22, 3 (2005), 258–266.
[4] D Behrens, M Serafini, S Arnautov, F P Junqueira, and C Fetzer. 2015. Scalable Error Isolation for Distributed Systems. In *Proc. 12th USENIX Conf. Networked Systems Design and Implementation (NSDI'15)*. 605–620.
[5] J A Blome, S Gupta, S Feng, and S Mahlke. 2006. Cost-efficient soft error protection for embedded microprocessors. In *Proc. Int'l Conf. Compilers, Architecture and Synthesis for Embedded Systems (CASES'06)*. 421–431.
[6] S Borkar. 2005. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro* 25, 6 (2005), 10–16.
[7] D T Brown. 1960. Error Detecting and Correcting Binary Error Detecting and Correcting Binary Codes for Arithmetic Operations. *IRE Trans. Electronic Computers* (1960), 333–337.
[8] V Chandra and R Aitken. 2008. Impact of Technology and Voltage Scaling on the Soft Error Susceptibility in Nanoscale CMOS. In *Proc. Int'l Symp. Defect and Fault Tolerance of VLSI Systems (DFTVS'08)*. 114–122.
[9] J Chang, G A Reis, and D I August. 2006. Automatic Instruction-Level Software-Only Recovery. In *Int'l Conf. Dependable Systems and Networks (DSN'06)*. 83–92.
[10] Z Chen, A Nicolau, and A V Veidenbaum. 2016. SIMD-based Soft Error Detection. In *Proc. Int'l Conf. Computing Frontiers (CF'16)*. 45–54.
[11] M Didehban and A Shrivastava. 2016. nZDC: A compiler technique for near Zero Silent Data Corruption. In *Proc. Design Automation Conf. (DAC'16)*.
[12] H Esmaeilzadeh, E Blem, R St. Amant, K Sankaralingam, and D Burger. 2011. Dark silicon and the end of multicore scaling. In *Proc. 38th Int'l Symp. Computer Architecture (ISCA'11)*. ACM, New York, NY, USA, 365–376.
[13] H Esmaeilzadeh, A Sampson, L Ceze, and D Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proc. 17th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 301–312.
[14] S Feng, S Gupta, A Ansari, and S Mahlke. 2010. Shoestring: Probabilistic Soft Error Reliability on the Cheap. In *Proc. 15th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'10)*. 385–396.
[15] R R Ferreira, R B Parizi, L Carro, and A F Moreira. 2013. Compiler Optimizations Impact the Reliability of the Control-Flow of Radiation Hardened Software. *J. Aerosp. Technol. Manag.* 5, 3 (Jul.-Sep. 2013), 323–334.
[16] C Fetzer, U Schiffel, and M Süßkraut. 2009. AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. In *Proc. 28th Int'l Conf. Computer Safety, Reliability, and Security (SAFECOMP'09)*. 283–296.
[17] P Forin. 1989. Vital Coded Microprocessor Principles and Applications for Various Transit Systems. In *Control, Computers, Communications in Transportation: Selected Papers from the IFAC/IFIP/IFORS Symp.* 79–84.
[18] H L Garner. 1966. Error Codes for Arithmetic Operations. *IEEE Trans. Electronic Computers* EC-15, 5 (1966), 763–770.
[19] O Goloubeva, M Rebaudengo, M Sonza Reorda, and M Violante. 2006. *Software-implemented hardware fault tolerance.* Springer.
[20] H S Gunawi, M Hao, T Leesatapornwongsa, T Patana-anake, T Do, J Adityatama, K J Eliazar, A Laksono, J F Lukman, V Martin, and A D Satria. 2014. What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems. In *Proc. Symp. Cloud Computing (SOCC'14)*.
[21] M R Guthaus, J S Ringenberg, D Ernst, T M Austin, T Mudge, and R B Brown. 2001. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. Int'l Symp. Workload Characterization (IISWC'01)*. 3–14.
[22] M Hoffmann, P Ulbrich, C Dietrich, H Schirmeier, D Lohmann, and W Schröder-Preikschat. 2014. A Practitioner's Guide To Software-based Soft-Error Mitigation Using AN-Codes. In *Proc. 15th Int'l Symp. High-Assurance Systems Engineering*.
[23] A A Hwang, I A Stefanovici, and B Schroeder. 2012. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *Proc. 7th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. 111–122.
[24] S Karol, N A Rink, B Gyapjas, and J Castrillon. 2016. Fault Tolerance with Aspects: A Feasibility Study. In *Proc. 15th Int'l Conf. Modularity*.
[25] D Kuvaiskii, R Faqeh, P Bhatotia, P Felber, and C Fetzer. 2016. HAFT: hardware-assisted fault tolerance. In *Proc. 11th European Conf. Computer Systems (EuroSys'16)*.
[26] D Kuvaiskii and C Fetzer. 2015. Δ-encoding: Practical Encoded Processing. In *Proc. 45th Ann. Int'l Conf. Dependable Systems and Networks (DSN'15)*.
[27] C Lattner and V Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Int'l Symp. Code Generation and Optimization (CGO'04)*. 75.
[28] S Liu, K Pattabiraman, T Moscibroda, and B G Zorn. 2011. Flikker: Saving DRAM Refresh-power through Critical Data Partitioning. In *Proc. 16th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS'11)*. 213–224.
[29] C-K Luk, R Cohn, R Muth, H Patil, A Klauser, G Lowney, S S Wallace, V J Reddi, and K Hazelwood. 2005. PIN: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. Conf. Programming Language Design and Implementation (PLDI'05)*. 190–200.
[30] E B Nightingale, J R Douceur, and V Orgovan. 2011. Cycles, Cells and Platters: An Empirical Analysis of Hardware Failures on a Million Consumer PCs. In *Proc. 6th European Conf. Computer Systems (EuroSys'11)*. 343–356.
[31] N Oh, S Mitra, and E J McCluskey. 2002. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Trans. Computers* 51, 2 (2002), 180–199.
[32] N Oh, P P Shirvani, and E J McCluskey. 2002. Control-Flow Checking by Software Signatures. *IEEE Trans. on Reliability* 51, 2 (2002), 111–122.
[33] N Oh, P P Shirvani, and E J McCluskey. 2002. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Trans. Reliability* 51, 1 (2002), 63–75.
[34] M Rebaudengo, M Sonza Reorda, M Torchiano, and M Violante. 1999. Soft-error Detection through Software Fault-Tolerance techniques. In *Int'l Symp. Defect and Fault Tolerance in VLSI Systems (DFT'99)*. 210–218.
[35] G A Reis, J Chang, N Vachharajani, R Rangan, and D I August. 2005. SWIFT: Software Implemented Fault Tolerance. In *Int'l Symp. Code Generation and Optimization (CGO'05)*. 243–254.
[36] N A Rink and J Castrillon. 2015. Improving Code Generation for Software-Based Error Detection. In *Proc. 1st Int'l Workshop Resiliency in Embedded Electronic Systems (REES '15)*.
[37] N A Rink, D Kuvaiskii, J Castrillon, and C Fetzer. 2015. Compiling for Resilience: The Performance Gap. In *Proc. Mini-Symp. Energy and Resilience in Parallel Programming (ERPP'15)*.
[38] A Sampson, W Dietl, E Fortuna, D Gnanapragasam, L Ceze, and D Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Proc. 32nd Conf. Programming Language Design and Implementation (PLDI'11)*. 164–174.
[39] U Schiffel. 2011. *Hardware Error Detection Using AN-Codes.* Ph.D. Dissertation. Technische Universität Dresden.
[40] U Schiffel, A Schmitt, M Süßkraut, and C Fetzer. 2010. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *Proc. 29th Int'l Conf. Computer Safety, Reliability, and Security (SAFECOMP'10)*. 169–182.
[41] B Schroeder, E Pinheiro, and W-D Weber. 2009. DRAM Errors in the Wild: A Large-scale Field Study. In *Proc. 11th Int'l joint Conf. Measurement and Modeling of Computer Systems (SIGMETRICS'09)*. 193–204.
[42] M Shafique, S Garg, J Henkel, and D Marculescu. 2014. The EDA Challenges in the Dark Silicon Era: Temperature, Reliability, and Variability Perspectives. In *Proc. 51st Design Automation Conf. (DAC'14)*. 1–6.
[43] P Shivakumar, M Kistler, S W Keckler, D Burger, and L Alvisi. 2002. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proc. Int'l Conf. Dependable Systems and Networks (DSN'02)*. IEEE, 389–398.
[44] M B Taylor. 2012. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Proc. 49th Design Automation Conf. (DAC'12)*. 1131–1136.
[45] R Vermu, S Gurumurthy, and J A Abraham. 2007. ACCE: Automatic Correction of Control-Flow Errors. In *Proc. Int'l Test Conf.* 1010.
[46] C Weis, M Jung, P Ehses, C Santos, P Vivet, S Goossens, M Koedam, and N Wehn. 2015. Retention Time Measurements and Modelling of Bit Error Rates of WIDE I/O DRAM in MPSoCs. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE'15)*. 495–500.
[47] J Yu, M J Garzarán, and M Snir. 2009. ESoftCheck: Removal of Non-vital Checks for Fault Tolerance. In *Proc. 7th Int'l Symp. Code Generation and Optimization (CGO'09)*. 35–46.
[48] Y Zhang, J W Lee, N P Johnson, and D I August. 2010. DAFT: Decoupled Acyclic Fault Tolerance. In *Proc. 19th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'10)*. 87–98.