Software-Implemented Hardware Error Detection: Costs and Gains

Ute Schiffel  Andre Schmitt  Martin Suesskraut  Christof Fetzer
Faculty of Computer Science
Technische Universität Dresden
Dresden, Germany
{ute.schiffel, andre.schmitt, martin.suesskraut, christof.fetzer}@tu-dresden.de

Abstract—Commercial off-the-shelf (COTS) hardware is becoming less and less reliable because of the continuously decreasing feature sizes of integrated circuits. But due to economic constraints, more and more critical systems will be based on basically unreliable COTS hardware. Usually in such systems redundant execution is used to detect erroneous executions. However, arithmetic codes promise much higher error detection rates. Yet, they are generally assumed to generate very large slowdowns. In this paper, we assess and compare the runtime overhead and error detection capabilities of redundancy and several arithmetic codes. Our results demonstrate a clear trade-off between runtime costs and gained safety. However, unexpectedly the runtime costs for arithmetic codes compared to redundancy increase only linearly, while the gained safety increases exponentially.

Keywords—safety-critical systems, hardware error detection, arithmetic codes

I. INTRODUCTION

It is expected that decreasing feature sizes of integrated circuits and increased complexity of hardware architectures will lead to less reliable hardware [1]. New field studies show that current hardware has already relatively high error rates. For example, the authors of [2] analyzed memory errors detected in Google’s server fleet. They observed higher than expected error rates with 25,000 to 70,000 errors per billion device hours per Mbit. Up to now, the assumption was that soft errors dominate memory errors. However, the results presented in [2] show also relatively high hard error rates, i.e., high rates of permanent errors.

While ECC detects most memory errors in hardware, the data and address bus to the memory and registers of the processor are mostly not well protected. According to [3], the error rate of logical circuits has actually overtaken error rates in memory. Especially, COTS systems provide no detection for other error types than memory errors.

In the best case, hardware errors have no impact or they lead to easy to detect crashes. But they can also cause silent data corruptions (SDC), that is, generate incorrect output that is not recognized as such. SDCs may have enormous impact: In 2008, a bit flip in a message caused several hours of downtime for Amazon’s servers in the US and the EU [4].

Traditionally, safety-critical systems used custom hardware that is, for example, radiation hardened to prevent environment induced execution errors or uses hardware redundancy. The most important arguments against hardware solutions are the high development costs and the restricted market. Thus, the trend for critical and safety-critical systems is to use COTS systems instead of custom hardware and to implement error detection and toleration in software. Approaches to detect hardware faults in software provide much more flexibility. They are easier to apply, cheaper and faster to develop, and allow to use most recent, i.e., more powerful, hardware.

Software-implemented hardware error detection mechanism provide failure virtualization, i.e., the transformation of arbitrary errors including SDCs into benign fail-stop failures without the need for expensive custom hardware. This paper’s objective is to compare the following software-implemented hardware error detection mechanisms:

- SWIFT [5] and SWIFT ECF [6] that implement double modular redundancy for registers; ECF adds enhanced control flow checking to SWIFT (see Sec. III),
- Software encoding with different arithmetic codes such as AN-, ANB-, and ANBD-code (see Sec. IV).

We will shortly introduce the approaches compared. However, we do not discuss any further related work with respect to software-implemented hardware error detection. Readers interested in software-implemented error detection, may want to read our previous publication [7].

After introducing our application scenario in Sec. II and the different error detection approaches in the sections III and IV, we present our evaluations that consider runtime costs and detection capabilities in Sec. V.

Note that our symptom-based error injection surpasses previous bit flip based approaches that were mainly used in recent research papers for the evaluation of error detection mechanisms. In particular, previous evaluations of SWIFT and SWIFT ECF presented in [5], [6] only applied bit flips. In contrast, we inject various data modifications, faulty operation executions, and data and control flow errors as both transient and permanent errors.

AN-encoding was previously evaluated by Chang et. al in [5] and in our previous work [7]. But the AN-encoding evaluated in [5] was incomplete, i.e., only applied to easily encodable instructions such as additions or subtractions. In contrast, the AN-encoding evaluated in this paper is complete, i.e., is applied to the whole application. Further-
more, only bit flips were applied during the error injections presented in [5]. Our own evaluation in [7] used completely encoded programs but a less elaborate error model. For the AN-encoding implemented by ED4I [8] no error injection results or runtime measurements were presented at all.

Forin, who introduced ANBD-codes in [9], neither presented runtime measurements nor error injection results. Our previous evaluation of the ANBD-code in [10] is also much less elaborate in terms of error model and completeness of encoding. To the best of our knowledge no evaluation of an ANB-code (compared to an ANBD-code and other techniques) exists at all.

To fill these gaps we compare SWIFT and the AN-code variants with respect to their detection capabilities and runtime overhead. Our evaluation results are especially important since some of the mechanisms evaluated are prescribed by functional safety standards such as the upcoming ISO 26262. This prescription should be, among others, based on error injection results. Our contributions are:

Comparison of arithmetic codes and redundancy. To the best of our knowledge this is the first comparison of programs completely encoded using different arithmetic codes to programs protected by SWIFT or SWIFT ECF.

Superiority of arithmetic codes wrt. error detection. We show the superiority of arithmetic codes compared to redundancy with respect to error detection, especially for permanent errors. Other purely redundancy-based approaches, e.g., [11], [12] or triple modular redundancy, can be expected to be similarly susceptible to permanent errors.

Unexpected result: safety costs less than expected. As expected, increased safety as increased security does not come for free. Arithmetic codes provide higher error detection and generate higher slowdowns. Also between different codes this rule applies. However, while the gained safety increases exponentially, the runtime costs increase only linearly.

II. Application Scenario

Usually, some parts of an application are more safety-critical than others [13]. For example, for detecting a soft error that disturbs a message transmission, it is sufficient to protect the message with a safely computed end-to-end checksum that checks that the message was delivered to the intended receiver unmodified. It is not required to provide error detection for the complete network protocol stack. Another application example are event processing frameworks that support the development of distributed applications. While the framework handles, for example, message distribution, the so-called operators implement the business logic. The framework itself can remain unprotected while the operators have to be sufficiently equipped to detect erroneous executions that may result in SDCs.

Thus, for our comparisons, we used several algorithms that could be expected in safety-critical and event processing systems and executed them within a networked environment. While the network stack is unprotected, the safety-critical algorithms are equipped with error detection. However, we assume that the execution of the complete application – including safety-critical algorithms and network protocol – can be modified by transient and permanent errors.

III. SWIFT and SWIFT ECF

Software implemented fault tolerance (SWIFT) [5] duplicates all instructions and registers apart from memory accesses and control flow instructions. SWIFT ECF [6] extends SWIFT with enhanced control flow checking. For SWIFT and SWIFT ECF any failed check results in the abort of the application. Since no implementation of SWIFT and SWIFT ECF was available to us, we reimplemented the approaches using the LLVM compiler framework [14].

**SWIFT:** SWIFT does not duplicate memory because it assumes that memory is protected by other means such as parities. This can lead to undetected lost stores because store instructions are not duplicated. Loads from memory are also not duplicated because they might be uncachable [5]. Instead values loaded from memory are copied. This approach opens a window of vulnerability. For example, data modifications on the bus may remain undetected.

Function calls are not duplicated. For a function defined in an external library, we do not know if it is idempotent and, thus, can be called twice. Instructions within internal functions are already duplicated by SWIFT and, thus, executed two times without duplicating the calls. Parameters of an internal function are duplicated at the start of the function. The return parameter of any function is duplicated when the call returns to the callee. These duplications can lead to undetected data modifications.

The equality of duplicates is checked before their externalization, before values are stored to memory or are used as address in a load or store instruction, or before the values influence control flow. Errors occurring in the vulnerable window between check and use might remain undetected. Checks in SWIFT are not easily protectable because they are strongly interleaved with the original program.

As already discussed SWIFT contains several windows of vulnerabilities, i.e., SWIFT does not provide end-to-end detection of hardware faults. Additionally, if both duplicates are affected by the same error, for example a permanently faulty operation, we expect that SWIFT will not detect this error. Furthermore, SWIFT is susceptible to control flow errors such as taking the wrong branch of a conditional jump.

**SWIFT ECF:** The authors of [6] added enhanced control flow checking (ECF) to SWIFT. This facilitates the detection of control flow errors. Therefore, a unique signature is assigned to every basic block. A dedicated register named **GSR** keeps track of the signature of the basic block that is supposed to execute currently. Before leaving a basic block an adaptation value for **GSR** is written to a
second dedicated register RTS. For unconditional jumps, the value assigned to RTS is a constant. For conditional jumps, it is one of two constants depending on the jump destination. Which of the two constants is assigned to RTS is chosen using the duplicate of the jump condition. The actual jump uses the original. Thus, if only duplicate or only original were modified, control flow errors are detectable. At the beginning of a basic block GSR is adapted to its new expected value by xor-ing RTS to it. Afterwards it is checked that GSR contains the expected signature for this basic block. The expected signature and the adaptation values are hard-coded into the binary because they are constant.

IV. ARITHMETIC CODES

Arithmetic codes – as SWIFT – add redundancy to all data words. In contrast to SWIFT, the redundancy is also applied to memory. Important is that arithmetic codes also check the execution of arithmetic operations. A correctly encoded operation preserves the code, i.e., it takes valid code words as input and outputs a valid code word. But a faulty arithmetic operation destroys the code with high probability, i.e., produces a result which is an invalid code word [15]. Thus, arithmetic codes facilitate the detection of errors made during storage, transport, and processing of data. When an application is encoded using an arithmetic code, it will solely process encoded data. All inputs have to be encoded and all computations use and produce encoded data.

We implemented complete encoding of applications at compile time with either the AN-, ANB-, or ANBD-code. For a description of the AN-encoding version of our compiler see [7]. For details on ANB- and ANBD-encoding of operations see [16]. For completeness we briefly introduce the three evaluated arithmetic codes: AN-, ANB-, and ANBD-code in the following.

AN-Code: For an AN-code the encoded version \( x_c \) of variable \( x \) is obtained by multiplying its original functional value \( x_f \) with a constant \( A \). Code checking is done by computing the modulus of \( x_c \) with \( A \), which is zero for a valid code word. Consider the following unencoded C code:

```c
int f(int x, int y, int z) {
    int u = x + y;
    int v = u + z;
    return v;
}
```

Its AN-encoded version\(^1\) uses solely AN-encoded data. The comments depict the variable content in the error-free case:

```c
int_c f(int_c xc, int_c yc, int_c zc) {
    int_c uc = xc + yc; // uc = A*x_c+y_c
    int_c vc = uc + zc; // vc = A*(x_c+y_c+z_c)
    return vc; // expected: vc mod A == 0
}
```

\(^1\)The presented pseudo code is simplified and ignores the over- and underflow issues described in [16].

Note that all variables should contain multiples of \( A \).

An AN-code can detect faulty operations and modified operands. If, for example, one of the additions is faulty or \( x_c \) is hit by a bit flip, this will be detected because the result will not be a multiple of \( A \) with high probability. The probability that an error results in a valid code word in this case is approximately \( \frac{1}{2} \) [9]. Yet, when a bit flip happens on the (unencoded) address bus, a wrong memory word will be accessed. For example, variable \( y_c \) might be exchanged by another encoded variable \( a = A * a_f \) (exchanged operand). An AN-code will not detect this because the result still will be a multiple of \( A \). A bit flip in the instruction unit of a CPU might cause the execution of a wrong operation (operator error) that might not be detected by an AN-code.

ANB-Code: For in [9] introduced static signatures (which he referred to as “B’s”). The resulting ANB-code can additionally detect operator and exchanged operand errors. The encoding of a variable \( x \) in ANB-code is defined as \( x_c = A * x_f + B_x \) with \( 0 < B_x < A - 1 \). To check the code of \( x_c \), \( x_c \)'s modulus with \( A \) is computed. The result has to be equal to the assigned or precomputed signature \( B_x \). The ANB-encoded version of the above example looks as follows:

```c
int_c f(int_c xc, int_c yc, int_c zc) {
    int_c uc = xc + yc; // uc = A*x_c+y_c
    int_c vc = uc + zc; // vc = A*(x_c+y_c+z_c)
    return vc; // expected: vc mod A == B_x+B_y+B_z
}
```

When encoding the program \( f \), we assign static signatures to the input variables \( x \), \( y \), and \( z \). Knowing the program, we can precompute the result’s expected signature \( B_x = B_x + B_y + B_z \). Note that for implementing dynamically allocated memory, we introduced dynamic signatures in [10] that are assigned at runtime. If an error would now exchange the variable \( y_c \) with another encoded variable \( u_c = A * u_f + B_u \), the result’s computed signature \( v_c \) mod \( A \) would be \( (B_x + B_u + B_z) \) instead of the expected \( (B_x + B_y + B_z) \). If the addition were to be replaced erroneously by a subtraction, the resulting computed signature would be \( (B_x-B_y+B_z) \) instead of \( (B_x+B_y+B_z) \). Thus, an ANB-code can detect exchanged operands and operators additional to faulty operations and modified operands. However, now consider that there is a bit flip on the address bus when storing variable \( y_c \). Thus, we have a lost update on \( y_c \) because \( y_c \) is stored in a wrong memory location. When reading \( y_c \) the next time, the old version of \( y_c \) is read — which is correctly ANB-encoded but outdated.

ANBD-Code: To detect the use of outdated operands, i.e., lost updates, Forin introduced a timestamp \( D \) that counts variable updates [9]. In the resulting ANBD-code, the encoded version of \( x \) is \( x_c = A * x_f + B_x + D \). The code checker also has to have access to a matching counter \( D \) to facilitate checking the validity of code words.
Currently, our ANBD-code implementation does only apply timestamps to memory. Thus we denote it as ANBDMem-code in the following.

V. Evaluation

Our evaluation focuses on the error detection capabilities and the performance overhead of the five approaches introduced. For our evaluation we use the following benchmark algorithms that can be expected in safety-critical or event processing systems:

- **md5** calculates the md5 hash of a string.
- **tcas** is an open-source implementation of the traffic alert and collision avoidance system [17] which is mandatory for airplanes.
- **pid** is a Proportional-Integral-Derivative controller [18].
- **abs** implements an antilock braking system.
- **topK** finds the k most frequent items in a data stream [19].
- **primes** implements the Sieve of Eratosthenes.

Although, **primes** is not a typical example of our targeted application domain, we included it because it uses a large array and thus is vulnerable to data modifications.

For our runtime measurements we evaluate these algorithms within a networked environment. A client sends requests to a server application that executes one of our benchmark algorithms. The benchmark algorithms represent the safety-critical part of the system. Hence, they are completely protected by one of the approaches presented in sections III and IV. The client, the part of the server that receives the messages, and the network protocol stack are completely unprotected. For the approaches presented in Sec. IV we transfer encoded data from client to server and vice versa to enable end-to-end detection of hardware faults.

The error injection experiments inject errors only in the protected benchmark algorithms. Therefore, the algorithms are not running within the networked environment.

Our server test machine has two Intel Xeon processors (in total 8 cores) and runs a 64-Bit Fedora 10. The client machine has the same OS as the server, but runs on an AMD Opteron processor (in total 16 cores). Client and server are connected by a 10-MBit half-duplex network.

A. Error Injection

Our error injector uses symptom-based error injection [20]. Instead of injecting errors directly into the hardware either physically or using fault injecting hardware, it injects the software-level symptoms of possible hardware failures. Directly injecting at software-level reduces masking and thus makes the error injection more efficient.

We injected the following kinds of errors:

- **Faulty operation**: The result of an operation is modified by bit flips. We inject multiple as well as single bit flips. The number of flipped bits is chosen using a Poisson distribution, that is, single bit flips are more probable than multiple bit flips.
- **Lost store**: A store operation is omitted.
- **Modified operand**: An operand used by an instruction is modified by a single or a multiple bit flip. While faulty operation influences every read of the result, modified operand only influences one read of a register.

Further errors can be represented by combinations of these symptoms. For example, the replacement of a complete instruction comprised of operator and operands with a different instruction can be emulated using the symptoms exchanged operator and exchanged operand. Control flow errors can be emulated by combinations of instruction replacements.

This error model is based on the assumption that every hardware error that is not masked influences the execution of a program in some way and that all possible influences can be emulated by these basic symptoms.

We applied those errors in three different modes:

- **Deterministic (Det)**: In this mode per run exactly one error is triggered. We execute approximately 50,000 runs for each benchmark and protection mechanism: 10,000 for each symptom. In each run another error of the same symptom is triggered. For each symptom the injection points are distributed equally over the possible injection points available in the program execution used. This tests the ability of a detection mechanism to cope with seldom occurring errors.
- **Probabilistic (Prob)**: Here, all error symptoms are injected with the same probability and they all might be injected in one run. We use the same error probability for all error detection mechanisms evaluated. At each possible point where an error (of any symptom) could be triggered an error is injected with the given probability. Thus, one execution might be hit by several different errors. This mode allows to mirror the fact that for an error detection mechanism which increases code size, the protected program version is more probable to collect errors than the program version without error detection. With this mode we executed 6,000 runs per benchmark and per detection mechanism.
- **Permanent errors (Per)**: In this mode we inject permanent faulty operation errors simulating permanent logic errors in the processor. Depending on the input values of an instruction, its result is modified. If a specific bit within the input values of a specific operation is set, a bit of the result is flipped. For one instruction run, the targeted operation, the bit which has to be set for triggering the error, and the bit flipped in the result remain the same. Permanent errors are only applied to arithmetic integer operations, and loads and stores of integer values. We are injecting approximately 1,700 different permanent errors per benchmark, per detection mechanism – one error only per run.
All example applications are of similar size. Hence, with our fixed number of fault injection runs we achieve similar coverages for all applications. We chose the number of fault injection runs so that the experiments complete in a feasible time.

We compared the results of injection runs to the results of an error-free run to determine if the error injected resulted in a silent data corruption (SDC). Figure 1 presents the results of the described error injection experiments. It depicts the share of injection runs that resulted in an SDC, i.e., a failure of the error detection mechanism. Note the logarithmic scale.

We make the following observations:

- We see that the amount of SDCs in the native, i.e., unprotected, case depends on the benchmark. There are programs that are more robust, e.g., tcas, than others, e.g., md5.
- We see also large differences between transient and permanent errors. While md5 is very susceptible to transient errors (Det and Prob), it is not to permanent ones (Per). For topK, for example, it is the other way round.
- Furthermore, benchmarks that show already high silent data corruption rates for the native case, show also rather high rates for SWIFT, SWIFT ECF, or the AN-code.
- We clearly see the superiority of all the AN-codes compared to SWIFT and SWIFT ECF with respect to permanent errors. For transient errors (Det and Prob) the AN-code has for some benchmarks higher detection rates (i.e., lower SDC rates) than SWIFT and SWIFT ECF and for some not. However, the ANB- and ANBDmem-codes always have an order of magnitude better detection rates than SWIFT and SWIFT ECF. Between ANB- and ANBDmem-codes the differences are small.
- In some cases error detection mechanism with additional protection perform worse than their supposedly weaker counter parts. For example, in 50% of the cases SWIFT ECF detects less permanent errors than SWIFT (for topK, pid, md5). Another example are the deterministic injections (det) into tcas. In one case (abs in mode det) ANB detects more injected errors than ANBDmem. Increased complexity seems to introduce new vulnerabilities that lead to undetected silent data corruptions. A more detailed analysis of ANBDmem-encoded runs indeed revealed such vulnerabilities for our ANBDmem-code implementation.

B. Performance

To evaluate the performance we measured the throughput of our client server scenario. The client sends a request to the server and waits for the reply. The server feeds the request as input to the benchmark implementation that is protected by one of the detection techniques and sends back the output as reply to the client. After receiving the reply the client sends the next request to the server. For AN, ANB, and ANBDmem all data sent through the network
stack is encoded. We measured the throughput in requests per second. All results presented are the trimmed arithmetic mean of at least 8 measurements. The deviation is negligible. Hence, we omit it in our diagrams.

The throughput values shown in Fig. 2 are relative to the throughput of the native and unprotected execution of the respective server benchmark. The throughput of SWIFT and SWIFT ECF is within the measurement error of the throughput of the native execution. As we expected, the arithmetic codes come at a higher cost. AN degrades the throughput much less than ANB. ANB degrades the throughput not as much as ANBDmem. However, the overhead varies with the application. For instance, abs has even for ANBDmem 70% of the throughput of the native execution, whereas topK protected by ANBDmem only reaches 4% of the throughput of the native execution. The reason is that some encoded operations are more expensive than others and topK uses more of the expensive ones.

While in Fig. 2 the overhead is CPU-bound, Fig. 3 depicts the effects of a network-bound setting. Therefore, we ran eight servers and clients in parallel on the server and the client machine, respectively. Because both computers have at least eight cores, that did not introduce additional limits on the CPU. However, the network traffic increased by a factor of eight. For this setting, we observe reduced throughput for all detection techniques. But the observed throughput degradation that comes with increased safety is in general much lower. For topK with ANBDmem the throughput is still only 12% of the native execution. But abs with ANBDmem degrades the throughput only down to 85% of the throughput of the native execution.

C. Error Detection vs. Performance

Figure 4 summarizes the engineering trade-offs when choosing an error detection technique. This graph relates the costs of the presented detection techniques to the achieved gain when using them. Our cost model (y-axis) is the throughput of the protected execution relative to the throughput of the native execution, i.e., \textit{the higher the throughput the smaller are the costs}. The used costs for the left graph are taken from the non-parallel experiment depicted in Fig. 2 and for the right graph from parallel, i.e., network-bound, experiment shown in Fig. 3. For both graphs, the gain (x-axis) is the share of SDCs relative to the share of SDCs in the native execution (Fig. 1), i.e., \textit{the smaller this remaining rate of undetected errors is the better}. We averaged the number of undetected errors for all three injection modes: Det, Prob, and Per. Note that the x-axis is log-scale. Every point is the mean of the measurements of all our benchmark applications. The native execution as expected has no additional costs, but also no gain. Thus, the native execution is in the upper right corner.

SWIFT’s and SWIFT ECF’s runtime overhead is negligible. But using them about 18% of the undetected errors of the native execution remain undetected. ANB and ANBDmem have a negligible rate of undetected errors, but introduce high costs. AN has neither a negligible error detection rate nor a negligible performance overhead.

From comparing Fig. 4 left and right we conclude that limits of the environment (such as limited bandwidth) can hide some of the performance costs of the detection techniques used. Overall we conclude from Fig. 4: When choosing one of the detection mechanism with higher detection rate, the performance degradation is \textit{linear}. But the gain, i.e., reduction of the rate of undetected silent corruptions, grows \textit{exponentially}. 

Figure 2. Throughput of our five detection techniques normalized against the native execution.

Figure 3. Same setting as in Fig. 2 but network-bound because of eight parallel running servers and clients on each computer.

Figure 4. Comparison of the cost and the gain of our five detection techniques.
VI. Conclusions

To the best of our knowledge we are the first to compare the error detection capabilities and runtime overhead of applications protected by SWIFT or SWIFT ECF or application encoded completely with any of the following arithmetic codes: AN-, ANB-, or ANBDmem-code.

The arithmetic codes – especially ANB and ANBDmem – provide better error detection than SWIFT and SWIFT ECF, which are based on redundant execution. Furthermore, the results demonstrate that arithmetic codes are indeed better in detecting permanent errors than SWIFT and SWIFT ECF.

It is not surprising that the arithmetic codes lead to a higher reduction in performance in terms of reduced throughput. Still, we are suprised by the cost-effectiveness of the ANB- and the ANBDmem-code. Because by using the ANB- or the ANBDmem-code instead of the AN-code or SWIFT the error detection rate grows exponentially, but the performance decreases only linearly.

We derive the following recommendations. For performance critical applications where undetected errors have only non-catastrophic effects SWIFT or SWIFT ECF can be chosen. If, however, it is critical to achieve a negligible rate of undetected errors, the ANB-code or better the ANBDmem-code should be used. Using the AN-code is a good compromise between performance degradation and error detection rate.

REFERENCES


