# Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors

Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk
Department of Computer Science 12
Technische Universität Dortmund, Germany
e-mail: {horst.schirmeier, christoph.borchert, olaf.spinczyk}@tu-dortmund.de

***Abstract*—Since the first identification of physical causes for soft errors in memory circuits, fault injection (FI) has grown into a standard methodology to assess the fault resilience of computer systems. A variety of FI techniques trying to mimic these physical causes has been developed to measure and compare program susceptibility to soft errors.**

**In this paper, we analyze the process of evaluating programs, which are hardened by software-based hardware fault-tolerance mechanisms, under a uniformly distributed soft-error model. We identify three pitfalls in FI result interpretation widespread in the literature, even published in renowned conference proceedings. Using a simple machine model and transient single-bit faults in memory, we find counterexamples that reveal the unfitness of common practices in the field, and substantiate our findings with real-world examples. In particular, we demonstrate that the *fault coverage* metric must be abolished for comparing programs. Instead, we propose to use extrapolated absolute failure counts as a valid comparison metric.**

## I. Introduction

Since the identification of physical causes for transient hardware errors *(soft errors)* in the 1970s [1], [2], computer systems have been hardened on different hardware and software levels to reduce the probability of system failures [3]. This paper focuses on *software-implemented* hardware fault-tolerance for soft-error mitigation, such as [4], [5], [6], [7], [8].

Software-based hardware fault-tolerance mechanisms need to be tested, measured and *compared* in order to assess their effectiveness in the particular use case. In the simplest case, an unmodified *baseline* version of a benchmark program (with some pre-defined input) is supposed to compete with a *hardened* version of the same program, and the latter is expected to exhibit increased fault resilience.

However, soft errors per bit are very rare in reality [9], [10], [11], and consequently hardened systems cannot simply be deployed in their target environment under normal conditions to observe their fault handling capabilities: Even if soft errors occur, this approach does not yield statistically authoritative evidence that proves a fault-susceptibility reduction of the hardened program.

For at least two decades, the common solution to this problem in the field has been fault injection (FI) [12], [13], [14], [15], [16]. FI is used to mimic the effects of the original causes for soft errors to a certain degree, but with extremely increased occurrence probability to trigger the fault-tolerance mechanisms often enough for sufficient evidence of their effectiveness. Note that, besides for effectiveness measurements, FI is also used for other purposes that are beyond the scope of this paper, such as the injection of software bugs, or functional testing of fault-tolerance measures.

Due to closeness to the root causes of soft errors in reality, early hardware-based FI solutions were based on experiments with radiation sources [17]. Although triggering realistic fault scenarios, the main disadvantage of this approach is that FI experiments have a very low *controllability* (where and when to inject a fault). As a consequence, they are not deterministically *repeatable* (the ability to inject a specific fault, and to obtain the same experiment result) [16]. These properties are important for reproducing bugs in the fault-tolerance implementation. Moreover, they are required for systematic so-called "pruning" techniques to reduce the amount of experiments to conduct. Additionally, experiments with radiation sources are extremely expensive (both money and time-wise), and handling of radioactive material is delicate. Pin-level FI, and experiments under the influence of electromagnetic interference, share these disadvantages to a certain degree [18].

### A. Benchmark Comparison with Hardware-based FI

Besides the costs and the implementation intricacies, the procedure of comparing a baseline and a hardened version of a benchmark with hardware-based FI is quite straightforward. After exposing the system-under-test to the influence of a radiation source, e.g., a Californium-252 isotope [18], the baseline version is executed on the target system $N$ consecutive times (with proper system resets in between). In each run, the system's output is observed, e.g., by recording the data printed on a serial interface. Each time it behaves differently from a previously defined correct behavior, the failure count $F$ is incremented. With a sufficiently large $N$, the failure probability for a benchmark run *under increased radiation conditions* can be approximated by using the relative frequency of observed failures, $P(\text{Failure})_{\text{baseline}} \approx F_{\text{baseline}}/N_{\text{baseline}}$. (Arlat et al. [12] define a reliability metric $R = 1 - P(\text{Failure})$, but this distinction has no practical relevance for this paper).

The same failure probability approximation is calculated for the hardened version, yielding $P(\text{Failure})_{\text{hardened}}$. *For normal radiation conditions*, both failure probabilities are over-approximated, because the fault rate has been massively increased by the radiation source. Compared to systems running under normal conditions, this over-approximation constitutes a linear and constant factor, which cancels out when calculating the comparison ratio $r$,

$$r = \frac{P(\text{Failure})_{\text{hardened}}}{P(\text{Failure})_{\text{baseline}}}.$$

The hardened version improves over the baseline iff $r < 1$.

### B. Software-implemented Fault Injection

Due to the disadvantages of hardware-based FI, a widely adopted alternative is software-implemented FI. Here, transient hardware faults are emulated by corrupting the state of a simulated machine [19], [20], or by injecting faults into development hardware via a debugger interface [21]. Thereby, faults can be injected only into parts of the machine that are visible to the FI implementation, such as memory and CPU registers.

In this paper, we dissect current practices in interpreting software-implemented FI results for transient memory errors from the literature. We identify three common pitfalls that can skew or even completely invalidate the analysis, and lead to wrong conclusions when comparing the effectiveness of software-based hardware fault-tolerance solutions applied to benchmark programs. We support each pitfall with a concrete example, and propose an alternative metric that can be used for benchmark comparison.

In particular, the main contributions of this paper are:

- We *dissect current practices* in software-implemented FI with regard to transient memory errors, including FI experiment-reduction techniques (Section III). Our findings – quantitatively substantiated by a real-world data set – are that special care has to be taken to *avoid distorted results* by such techniques.
- We show that the widely used *fault coverage* metric is *unsound* for comparing different programs (Section IV). Specifically, this metric is defective for the evaluation of software-based hardware fault-tolerance mechanisms applied to a benchmark program.
- As a remedy, we *construct an objective comparison metric* based on extrapolated absolute failure counts, and introduce the mathematical foundation supporting this proposition (Section V).

The following section describes the fault and machine model used throughout this paper. Section VI discusses possible generalizations and implications of our findings, and, after reviewing related work (Section VII), the paper concludes in Section VIII.

## II. Setting the Stage: Definitions, and Machine, Fault and Failure Model

In this section, we first define the semantics of fundamental terms used throughout this paper. We then establish the fault and machine model, and describe the assumed repeatable, deterministic FI experiment execution. Subsequently, we describe the benchmark data we used, and describe the possible failure modes these programs can exhibit.

### A. Terms and Definitions

In this paper, we use the terms *fault*, *error* and *failure* in their classical meaning [22] from a software-level fault-tolerance perspective. A failure is specified by a deviation of the software system's behavior, primarily its *output*, from its correct behavior. The *failure mode* differentiates between different forms of failure. An *error*, a deviation of the system's internal state from the norm, may lead to a failure. The root cause of an error is a *fault* that turns into an error if it is *activated*. Otherwise, the fault stays *dormant*. [22]

The term "soft error" – a transient corruption of machine state, such as bits in main memory – was originally devised from a hardware perspective. From our software-level perspective, a "soft error" is actually a *fault*, and forms the root cause for errors and failures. Nevertheless, we will use "soft error" throughout this paper, and actually mean transient faults.

### B. Fault Injection (FI)

Fault injection [12], [13], [14], [15], [16] started out many years ago as a *testing* technique for dependability validation. A common use case involves uncovering design and implementation weaknesses, for example, by providing faults in the program input. Here, the representativeness of the injected faults is irrelevant for the goal of revealing program bugs.

Since then, software-implemented FI has also been applied to benchmark programs for *quantitative evaluation* of software-based hardware fault-tolerance mechanisms, as in [4], [5], [6], [7], [8]. This use case is completely different from testing. For the assessment of fault-tolerance effectiveness, the injected faults have to closely represent real hardware faults: A realistic spatial and temporal distribution of the injected faults is crucial.

This paper only concerns software-implemented FI for quantitative evaluation. In the following, we refer to FI as software-implemented technique to obtain *statistics* for the comparison of program susceptibility to soft errors. Other goals of FI, such as testing or the injection of program bugs, are beyond the scope of this study.

### C. Fault and Machine Model

To focus on the core findings in this paper, we use a simplistic machine model. Abstracting from CPU specifics, we assume a simple RISC CPU with classic in-order execution, without any cache levels on the way to a wait-free main memory, and with a timing of one cycle per CPU instruction. The CPU executes programs from read-only memory. Section VI-B discusses possible generalizations from this simple model.

On this machine, benchmark runs can be carried out deterministically, i.e., the same program with an identical start configuration (program input and machine state) is exposed to a pre-defined sequence of external events (timer interrupts or other input at runtime), and leads to an exactly identical program run.[1] Additionally, the machine can be paused at an arbitrary cycle during the run (e.g., to inject a fault by changing the machine state) and resumed afterwards. In practice, this can be achieved by, e.g., using a hardware simulator.

As the basic fault model throughout this paper, we use a classic soft-error model widely used in the literature: uniformly distributed, independent and transient single-bit flips in main memory, modeled as originating from *direct* influences of ionizing radiation. We pick this fault model primarily due to its simplicity, easing the illustration of the issues presented, but a large-scale study from the year 2013 [11] confirms that

[1] Note that *deterministic* does not mean that system reactions on external events, such as asynchronous device interrupts, cannot be analyzed. In deterministic benchmark runs, such events are replayed at the exact same point in time during each run.

the model is still valid for contemporary memory technology. As memory cells dominate the chip area of modern CPUs [23], our findings may possibly also apply to SRAM-based on-chip caches; Section VI-B discusses this, and other possible implications. We quantize the time with a granule of CPU cycles, restricting faults to be only injected between the execution of one instruction and the next. Additionally, we assume the ROM, holding the program instructions, to be immune to faults.

We are aware that in practice, other parts of the machine are also susceptible, and that errors may also propagate from the CPU logic. Some of our findings *may* apply for other fault models, too, but this is beyond the scope of this paper, and material for future work.

### D. Failure Model and Benchmark Setup

As already mentioned in the introduction, the effectiveness of a software-based hardware fault-tolerance mechanism can be assessed by applying it to a set of benchmarks, and comparing the failure probability of these *hardened* benchmark variants to their *baseline* counterparts. Thus, the primary ingredients for this undertaking are benchmark programs, a fault-tolerance mechanism, and a definition of *failure* that fits the benchmarks' original purpose. Additionally, the benchmark inputs must be chosen based on a fair sample of the "operational profile", i.e., they must be representative of what to expect in real operation.

As a real-world example, we use benchmarks, fault-tolerance mechanisms and result data from an earlier publication [8] throughout this paper. In this publication, we developed a library of software-based fault-tolerance mechanisms, and aimed at protecting "critical" data with long lifetimes. These mechanisms were applied to a set of run-to-completion test programs with known output, belonging to the eCos operating system [24]. From the raw data, we pick the results for the BIN_SEM2 and SYNC2 benchmarks in both their *baseline* variant, and a variant *hardened* by checksums and data duplication (termed "SUM+DMR"). Figure 2g shows the runtime (in CPU cycles) and the memory usage of these benchmark variants.

In the previous publication [8], we ran extensive FI campaigns with our FAIL* tool [25] and observed the benchmarks' behavior after the injection. We examined the benchmark output on the serial interface for silent data corruption, and monitored the system for CPU exceptions and timeouts. Overall, we differentiated between eight experiment-outcome types, of which two – "No Effect" and "[Error] Detected & Corrected" – can be interpreted as a benign behavior that has no visible effect from the outside. We coalesce these two result types into "No Effect", and the remaining six failure modes into a subsuming "Failure" type, as the detailed differentiation is of no relevance for this paper.

## III. Fault-Space Scanning and Pruning

In this section, we discuss the statistics behind improbable independent faults, and thereby motivate that injecting a single fault per experiment suffices. We show that even with this simplification the number of necessary FI experiments to cover the whole fault space is practically infeasible. Consequently, we describe two widely used experiment-reduction techniques, namely *sampling* and *def/use pruning*. By examining common practices in applying these techniques, we identify our *first pitfall,* and present a means to avoid it.

### A. Improbable Independent Faults

With a fault model of uniformly distributed, independent and transient single-bit flips in main memory (cf. Section II-C), *a single run* of a simple run-to-completion benchmark can theoretically be hit by *any* number of independent faults. Multiple faults can occur at arbitrary points in time, and affect different bits in memory.

In Figure 1a, each black dot denotes a possible time[2] (CPU cycle) and space (memory bit) coordinate where a fault can hit the benchmark run and flip a bit in memory, affecting the stored value throughout the subsequent cycles until it gets eventually overwritten. Without considering the problem in more depth, trying to run one FI experiment each *for every subset* of these hit coordinates is infeasible, as the cardinality of the power set grows exponentially.

In reality, though, the probability for a single-bit flip occurring at one bit in main memory within the time span of one CPU cycle – e.g., the probability for flipping bit #3 exactly in the time frame between cycle 4 and 5 in Figure 1a – is extremely low. Recent large-scale studies on DRAM (DDR-2 and DDR-3) memory technology report soft-error rates of $0.061\,FIT$[3] per $Mbit$ [9], $0.066\,FIT/Mbit$ [10] and $0.044\,FIT/Mbit$ [11]. Even though different DRAM vendors were tested, the resulting soft-error rates are quite similar. Using the mean of these three error rates, for a single bit, the soft-error rate per nanosecond (assuming a clock rate of 1 GHz, or one cycle per *ns*, for our simplistic CPU from Section II-C) is:

$$\begin{aligned} g &= 0.057\frac{FIT}{Mbit} = \frac{0.057}{10^9 h \cdot 10^6 bit} \\ &= \frac{0.057}{10^9 \cdot 3600 \cdot 10^9 ns \cdot 10^6 bit} \approx 1.6 \cdot 10^{-29}\frac{1}{ns \cdot bit} \end{aligned}$$

The probability of *one* benchmark run being hit by $k = 0$, 1, 2, or more independent faults can be calculated using the binomial distribution. Nevertheless, for such an extremely low fault probability, it can be well approximated using the Poisson distribution (assuming the occurrence of faults is a Poisson process) [26]:

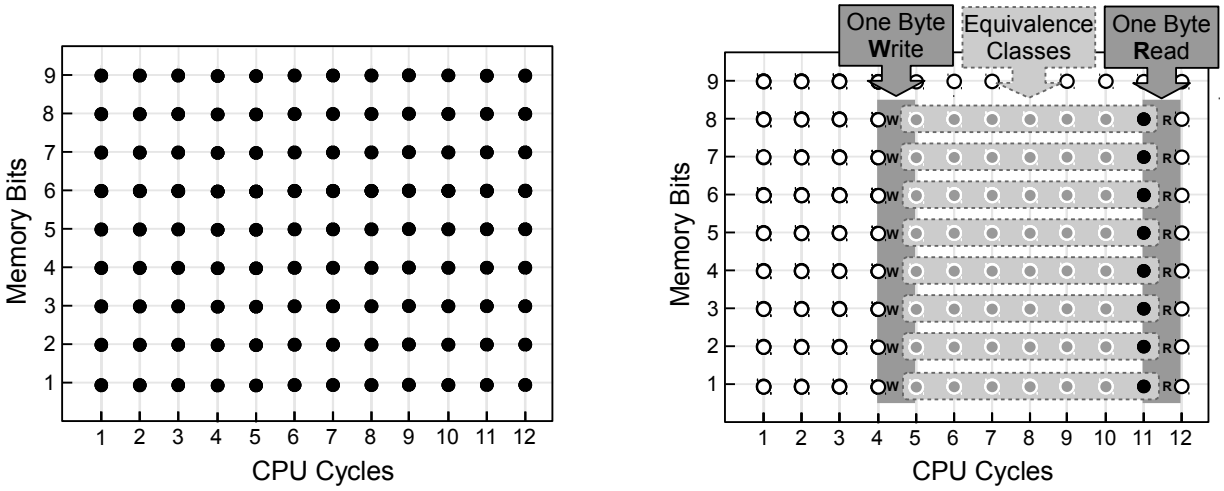
$$P_\lambda(k) = \frac{\lambda^k}{k!}e^{-\lambda} \tag{1}$$

The Poisson parameter $\lambda = gw$ is calculated using the aforementioned soft-error rate $g$, and the fault-space size $w = \Delta t \cdot \Delta m$ characterizing the benchmark by its runtime in CPU cycles $\Delta t$, and the amount of main memory in bits $\Delta m$ it uses. Using concrete values for the benchmark runtime and memory usage, e.g., $\Delta t = 1s$ (corresponding to $10^9$ cycles in our simplistic CPU model) and $\Delta m = 1\,MiB = 2^{23} bit$, yields the probabilities for $k$ faults hitting one benchmark run listed in Table I.

Unsurprisingly, for the magnitude of the parameter values, the probability that a benchmark run is *not hit at all* is extremely high. This zero-fault case naturally does not require any FI

[2]Note that time is quantized in granules of CPU cycles (cf. Section II-C).

[3]The FIT (Failures In Time) rate measures the number of failures to be expected per $10^9$ hours of operation.




(a) Single-bit flip fault space for a run-to-completion benchmark. Each dot represents a possible FI space/time coordinate. The benchmark proceeds until and stops at a specific point in time, the corresponding memory bit is flipped, and the benchmarks resumes.

(b) Single-bit flip fault space and def/use equivalence classes extracted from a program trace, reducing the FI experiments (dots) that need to be conducted. Fault injections at white coordinates (non-filled circles) can be omitted, as a fault there is overwritten or never read (dormant faults). A black dot represents a class of equivalent faults (light-gray coordinates) between the write and subsequent read instruction.

Fig. 1. The fault space spanned by *CPU Cycles* × *Memory Bits*. Every discrete (cycle, bit) coordinate denotes an event where a memory bit can flip during the depicted twelve CPU cycles.

TABLE I. POISSON PROBABILITIES FOR $k = 0,\ 1,\ 2,$ OR MORE INDEPENDENT FAULTS HITTING ONE BENCHMARK RUN.

| $k$ | $P_\lambda(k\,\text{Faults})$ | $k$ | $P_\lambda(k\,\text{Faults})$ |
|---|---|---|---|
| 0 | 0.999999999999867 | 3 | $3.905 \cdot 10^{-40}$ |
| 1 | $1.328 \cdot 10^{-13}$ | 4 | $1.297 \cdot 10^{-53}$ |
| 2 | $8.821 \cdot 10^{-27}$ | … | … |

experiments. But, even more noteworthy, the probability for two or more hits is so much lower than for one fault hitting the benchmark's used memory $\Delta m$ that these cases can be considered negligible.[4] Hence, at current (and tomorrow's) fault rates, and for short benchmark runtimes, it suffices to inject one fault per benchmark run.

Nevertheless, applying the numbers from our hypothetical benchmark example to the fault-space diagram in Figure 1a (the *CPU cycles* axis now spans from 0 to $\Delta t = 10^9$ cycles, the *memory bits* axis from bit #0 to #$2^{23}$) clarifies that, for a *full* fault-space scan, even with only one fault per run $w = \Delta t{\cdot}\Delta m \approx 8.4{\cdot}10^{15}$ FI experiments would have to be conducted. Even assuming we can simulate our simple CPU in real-time, this procedure would take about 266 million CPU years.

### B. Reducing Experiment Efforts: Fault Sampling

One widespread solution to this fault-space explosion problem is *fault sampling* [12], [27], [28]. Since the distribution of faults in the fault space is assumed uniform (Section II-C), FI experiments are picked *uniformly* from this space (Figure 1a). Consequently, the results can be used to estimate the *fault coverage factor* (or short, fault coverage) [29], *"defined as the probability of system recovery given that a fault exists"* [27].

[4]Even at a hypothetical fault rate of $g = 10^{-20}$, nine orders of magnitude higher than in the example, the distance between $P_\lambda(1\,\text{Fault})$ and $P_\lambda(2\,\text{Faults})$ is still more than $10^4$.

The fault coverage $c$, or – formalizing the citation from [27] – $P(\text{No Effect}|1\,\text{Fault})$ or $1\text{-}P(\text{Failure}|1\,\text{Fault})$, can be calculated after randomly picking $N$ (time, space) coordinates from the fault space, and running an FI experiment for each of them. In each experiment, the benchmark program is run from the beginning until the CPU cycle for the FI (the *time* component of the randomly picked coordinate from Figure 1a) has been reached. The machine is then paused, the fault gets injected by flipping the bit in memory corresponding to the *space* component of the coordinate, and the machine is resumed. As described in Section II-D, then the experiment outcome is observed, turning out either as "No Effect", or as "Failure". In the latter case, the failure counter $F$ is incremented. ("No Effect" results are implicitly counted as $N - F$).

The fault coverage $c$ can subsequently be calculated as

$$c = 1 - P(\text{Failure}|1\,\text{Fault}) = 1 - \frac{F}{N}. \qquad (2)$$

We will see in Section IV and V that the *fault coverage* metric, originally only devised for the assessment of hardware systems [29], is flawed for comparing software programs. The sampling process itself is unproblematic, as long as a sufficiently large number of samples is taken for statistically authoritative results. This topic is outside the scope of this paper, and we, thus, refer to the literature covering this matter [12], [27], [28].

### C. Reducing Experiment Efforts: Def/use Pruning

Recent FI techniques proceed with more sophistication than randomly sampling locations in the fault space, and are based on instruction and memory-access traces. These traces are created during a so-called *"golden run"*, which exercises the target software without injecting faults (and, thus, serves as a reference for the expected program behavior).

Figure 1b exemplarily shows the memory-access information recorded during the golden run. The dynamic instruction

starting in CPU cycle 4, a *store*, writes ("W") eight bits to main memory, and the data is read ("R") back into the CPU in cycle 11, executing a *load* instruction.

Based on this kind of memory-access trace information, Smith et al. [30] and Güthoff and Sieh [31] are among the first concisely describing the classical *def/use analysis* for experiment reduction (termed "operational-profile-based fault injection" in the latter paper). It is so fundamental that it was subsequently reinvented several times, e.g., by Benso et al. [32], [15], Berrojo et al. [33] ("Workload Dependent Fault Collapsing"), Barbosa et al. [34] ("inject-on-read"), and recently by Grinschgl et al. [35] and Hari et al. [19].

The basic insight is that all fault locations between a *def* (a write or *store*, "W" in Figure 1b) or *use* (a read or *load*, "R" in the figure) of data in memory, and a subsequent *use*, are equivalent: regardless of when exactly in this time frame a fault is injected there, the earliest point where it will be activated is when the corrupted data is read. Instead of conducting one experiment for every point within this time frame, it suffices to conduct a *single* experiment (for example at the latest possible time directly before the *load*, black dot in Figure 1b), and assume the same outcome for all remaining coordinates within that *def/use equivalence class* (light-gray frames in Figure 1b).

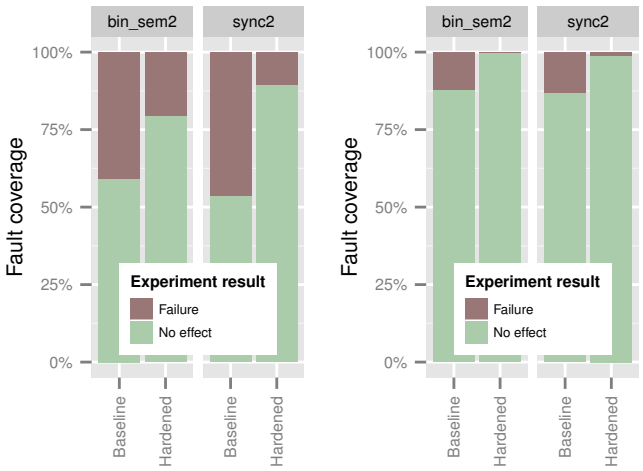
Similarly, all points in time between a *load* or *store* and a subsequent *store* without a *load* in between (light-gray dots in the remaining fault-space coordinates left, right and above the marked equivalence classes in Figure 1b) are known to result in "No Effect" without having to run experiments at all: injected faults will be overwritten by the next *store* in all cases.

The result of the def/use pruning process is a partitioning of the fault space into equivalence classes, some of which a single experiment needs to be conducted for (those ending with a *load*, "R"), and some with a priori known experiment outcome. From the $12 \cdot 9 = 108$ experiments in the illustrative example of Figure 1a, only 8 remain after def/use pruning in Figure 1b.

In real-world examples, def/use pruning (especially when applied to faults in memory) is extremely effective. For example, the *baseline* variant of the SYNC2 benchmark (cf. Section II-D) is reduced from a raw fault-space size of $w \approx 1.5 \cdot 10^8$ to merely 19,553 experiments. Thus, a full fault-space scan is feasible even on a single machine within a reasonable time frame, and without any loss of precision regarding the result information on any point in the fault space.

### D. Def/use Pruning and the Weighting Pitfall

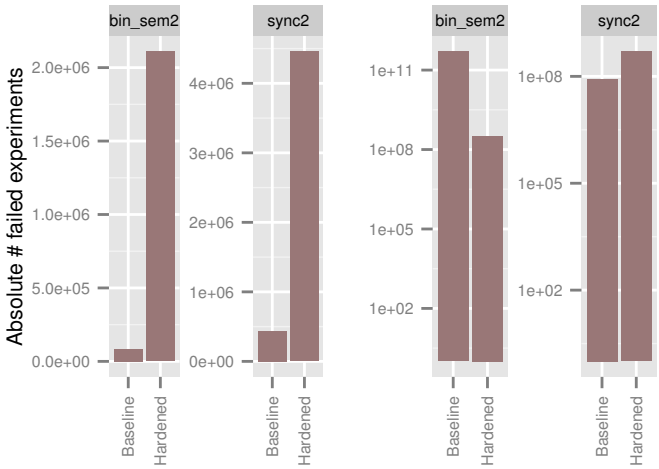
Since its inception, def/use pruning was meant as an effort-reducing, conservative *optimization* for the theoretical full fault-space scanning model [34]. If, assuming the experiment numbers from Figure 1b, four of the eight (black-dotted) actually conducted experiments turned out as "Failure" (and, inversely, the remaining four as "No Effect"), this number must *not* be used in Equation 2 for fault-coverage calculation without any post-processing, yielding a *wrongly calculated* coverage of $c = 1 - \frac{4}{8} = 50\%$. Instead, the previously collapsed equivalence classes must be expanded to their original size again, *weighting each FI-obtained result with the corresponding equivalence-class size*. Güthoff et al. also state this explicitly: *"Each result obtained by [...] fault injection experiments must be weighted with the corresponding relative data life-cycle length." [31]*




(a) Coverage, without result weighting (b) Coverage, with result weighting

| Fault Coverage | BIN_SEM2 | | SYNC2 | |
|---|---|---|---|---|
| | w/o | Weighted | w/o | Weighted |
| Baseline | 59.09 % | 87.97 % | 53.83 % | 87.07 % |
| Hardened | 79.64 % | 99.99 % | 89.64 % | 98.76 % |

(c) Fault coverage, raw data. The percentages without (w/o) weighting are off by 9–33 percent points compared to the weighted coverages.




(d) Absolute failure result counts, *without* result weighting (e) Absolute failure result counts, with result weighting (**log. scale**)

| Failure Count | BIN_SEM2 | | SYNC2 | |
|---|---|---|---|---|
| | w/o | Weighted | w/o | Weighted |
| Baseline | 85,636 | $5.08 \times 10^{11}$ | 435,371 | $8.55 \times 10^{7}$ |
| Hardened | 2,110,356 | $3.15 \times 10^{8}$ | 4,459,345 | $5.02 \times 10^{8}$ |

(f) Absolute failure counts, raw data. Without (w/o) weighting, the failure counts are underestimated by several orders of magnitude.

| Characteristics | BIN_SEM2 | | SYNC2 | |
|---|---|---|---|---|
| | Cycles | Mem. | Cycles | Mem. |
| Baseline | 559,868,647 | 942 | 313,132 | 264 |
| Hardened | 559,905,321 | 1,134 | 10,170,140 | 496 |

(g) Benchmark characteristics: Runtime in CPU cycles, and memory usage (data addresses read or written during the run) in bytes.

Fig. 2. FI result interpretation with and without avoidance of Pitfalls 1 and 3.

Going beyond their statement, the literature presents no plausible argument why fault-space coordinates we do not conduct any experiment for (the white dots in Figure 1b, known a priori to yield "No Effect") should be omitted in the result calculation. Section IV and V will shed more light on this debate; for now, we assume that *all* coordinates – the example fault-space size $N$ is $12 \cdot 9 = 108$ – should be included in the fault coverage calculation, which now correctly (with a weight of 7, the size of each light-gray equivalence class in Figure 1b, for each of the four "Failure" results) calculates as $c = 1 - \frac{F}{N} = 1 - \frac{4 \cdot 7}{108} \approx 74.1\%$.

Another explanation why this weighting is necessary can be derived from intuition: the longer data lives in a memory cell, the more probable a soft error will affect it. If no weighting is applied, the same fault coverage is calculated regardless of seven (Figure 1b) or seven million cycles between the *store* and the subsequent *load* of the data. Then, the pruning technique is not only a methodology to reduce FI experiment efforts, but has a severe impact on the result: The *fault model* unintentionally has degenerated from "uniform transient single-bit flips in main memory" to "uniform transient single-bit flips in main memory *while a memory read operation is in progress*" (modeling something similar to single-bit flips on the memory bus). Hence, the results are extremely skewed depending on the amount of memory accesses the benchmark executes, and the variance in memory-data lifetimes.

Now that we know def/use equivalence classes should be weighted in theory, does this have an impact on real-world examples? Figure 2a and 2b show fault coverages for the *baseline* and *hardened* variants of the BIN_SEM2 and SYNC2 benchmarks (cf. Section II-D), calculated without and with weighting. The difference is directly visible: in the unweighted case, the fault coverages of all benchmark variants are severely underestimated compared to the weighted case. The coverage values are off by 9.1 (SYNC2 *hardened*) up to 33.2 percent points (SYNC2 *baseline*).

The reason for the bias in the two example benchmarks is a correlation between def/use equivalence class size and experiment outcome. For the four benchmark variants we use, the only positive aspect is that the *trend* from the baseline to the hardened variants is the same in all cases, so that no dangerously wrong design decisions would have been made using the unweighted coverage results.

> *Pitfall 1: Unweighted Result Accounting*
> Summarizing this section, our first pitfall is the *unweighted result accounting* when using def/use pruning. Fault-space pruning is an *optimization*, and, thus, must not have any influence on the resulting numbers. When a technique such as the common *def/use* pruning changes the fault model's uniform distribution into a distribution that is strongly biased by the program execution, **each result must be weighted with the corresponding data lifetime** to compensate.

In the literature, a lack of result weighting is in most cases hidden behind fault-coverage factor percentages that do not reveal whether weights were applied. One example where the additionally provided data indicates that no weights were used is from Hoffmann et al. [36], who compare the fault susceptibility of two embedded operating systems using unweighted experiment numbers. Barbosa et al. [34] recognize that weighting is needed to compensate for the effects of pruning, but conclude that the difference is small for the benchmarks they used. In contrast, our benchmark examples in Figure 2 serve as a warning that this is not always the case. Alexandersson and Karlsson [37] realize that def/use pruning affects the comparability of their results to those of other studies. Other def/use pruning descriptions simply omit the relevant detail whether weighting is used, e.g., Berrojo et al. [33]. Additionally, tools clearly designed for the purpose of testing (where weighting is not necessary), such as Relyzer from Hari et al. [19], may be misused for comparison purposes. Correct metrics should be integrated.

### E. Combining Def/use Pruning and Sampling

The conclusions from the previous section are based on def/use pruning of a *full* fault-space scan. However, often a prohibitive number of FI experiments still remain after def/use pruning. Pruning and sampling can be combined to further reduce the experiment count. Clearly, the combination of both techniques must yield the same results as pure sampling, but with reduced effort.

Therefore, samples (fault-space coordinates) have to be drawn uniformly from the raw, unpruned fault space to get a representative sample of the entire fault-space population. The def/use pruning is then carefully applied in a second step: We only need to conduct a single FI experiment for fault-space coordinates in the sample that belong to the same def/use equivalence class. Nevertheless, we still need to count the results of *all sampled* fault-space coordinates to properly calculate the estimate for the entire fault-space population. Thereby, even coordinates known to result in "No Effect" (because their def/use equivalence class ends with a *store*, cf. Section III-C) must be included, although we will lift that requirement in Section V-C.

The other way around, applying def/use pruning first and then drawing samples uniformly from the already-pruned fault space, i.e., picking def/use equivalence classes with the same probability, leads to a *biased* estimate. A fault-space coordinate that belongs to a small def/use equivalence class would be included in the sample with a higher probability than for uniform sampling of the raw fault space. The reason is that the *weight* of each equivalence class biases the selection probability of its fault-space coordinates.

> *Pitfall 2: Biased Sampling*
> Hence, our second pitfall is *biased sampling*. If def/use pruning and sampling are combined, the sampling process must **pick samples from the raw, unpruned fault space**. If several samples belong to the same def/use equivalence class, only a single FI experiment needs to be conducted for them, but **all samples count in the estimate**.

## IV. Fooling Fault Coverage: A Gedankenexperiment

In this section, we will conduct a Gedankenexperiment with an apparently ineffective software-based hardware fault-tolerance mechanism protecting a simple benchmark program, and miraculously improving its fault coverage. Subsequently,

we will revisit the fault-coverage numbers for our BIN_SEM2 and SYNC2 benchmarks, and wonder how effective the used SUM+DMR hardening mechanism really is.

### A. Hi, A Simple Benchmark

Figure 3 shows the C-like source code for a tiny benchmark program that initializes a local character array, and subsequently communicates both character values to the outside world via the serial interface. The corresponding machine code consists of eight machine instructions consisting of four *load* and four *store* instructions. Figure 3a shows these loads and stores ("R" respectively "W"), similarly as in Figure 1b, in the complete fault space of this benchmark, spanning 16 bits on the memory axis, and eight cycles on the time axis.

If we run a full fault-space scan, i.e., run one independent FI experiment for every discrete coordinate in the fault space, and observe whether the benchmark's output is identical to the golden run (it is supposed to say "Hi"), the black-dotted experiments in Figure 3a will turn out as a "Failure", and the other ones as "No Effect". In the "Failure" cases, the fault hits `msg[0]` at address `0x1` while the letter `'H'` is stored there but not yet read back (this happens in CPU instruction #5), or analogously `msg[1]` at address `0x2` while the datum `'i'` lives there. The "No Effect" cases are FI experiments where the fault is subsequently overwritten (before the store cycle #1 respectively #3), or it is not activated anymore because the program terminates (after the load cycle #4 respectively #6).

The fault coverage $c_{\text{baseline}}$ can easily be calculated (cf. Section III-B and III-D) by counting the number of experiments $N_{\text{baseline}} = 8 \cdot 16 = 128$ and the number of "Failure" outcomes $F_{\text{baseline}} = 3 \cdot 8 \cdot 2 = 48$, and inserting them into Equation 2:

$$c_{\text{baseline}} = 1 - \frac{F_{\text{baseline}}}{N_{\text{baseline}}} = 62.5\%$$

### B. The Fault-Space Dilution Delusion

Now, we apply a hypothetical software-based fault-tolerance method – we call it "Dilution Fault Tolerance", or short DFT – on the baseline's machine code by conducting a program transformation. It works by prepending four NOP instructions (**no** o**p**eration, performing no real work for one cycle each) to the machine code, increasing the benchmark's runtime from eight to twelve CPU cycles. Figure 3b shows the modified fault-space diagram for the DFT-hardened benchmark: The loads and stores have shifted four cycles to the right, and the newly added experiment dots are all "No Effect", as no live data is stored in memory before the original beginning of the benchmark.

Again calculating the fault coverage $c_{\text{hardened}}$ with $N_{\text{hardened}} = 12 \cdot 16 = 192$ and the number of "Failure" outcomes $F_{\text{hardened}} = 3 \cdot 8 \cdot 2 = 48$ yields:

$$c_{\text{hardened}} = 1 - \frac{F_{\text{hardened}}}{N_{\text{hardened}}} = 75.0\%$$

Interestingly, by applying a seemingly ineffective "fault tolerance" program transformation, we increased the fault coverage by 12.5 percent points. In fact, we could arbitrarily increase the coverage to any $c_{\text{hardened}} < 100\%$ by inserting more NOPs!

Now, the attentive reader may point to the literature, and cite, e.g., Barbosa et al. [34], who argue (in the context of their def/use fault-space pruning technique) that never activated faults – a priori known "No Effect" results – should not be included in the coverage calculation. The newly added experiment dots in Figure 3b, that contributed to $N_{\text{hardened}}$ in the above calculation would disappear again (though for no authoritative reason), and yield $c_{\text{hardened}} = c_{\text{baseline}}$.

As a reaction to such disgraceful attempts to make our DFT mechanism look bad, we would devise DFT', which replaces the NOPs by memory reads. For instance, alternatingly executing `ld r1 ← 0x1` and `ld r1 ← 0x2` instructions to read those two memory locations. Now, all newly added experiment dots would represent faults that actually are "activated" – by being loaded into a CPU register (and subsequently discarded). DFT' would be back at $c_{\text{hardened}} = 75.0\%$, this time *with* the restriction from Barbosa et al. [34].

The central problem with this restriction is, that in a black-box technique such as FI the "activation" of a fault, turning it into an error, is an extremely vague business in itself. It strongly depends on the sophistication of the used fault-space pruning technique, e.g., not injecting into *unused* memory, not injecting into state that is *known to be overwritten* afterwards, not injecting into state that is *known to be masked by subsequent arithmetic operations* [19], ... – this list could be continued for a while. In our opinion, this vague distinction between "activation" and "no activation" should therefore have no place in the context of an objective fault-tolerance benchmarking metric. Though, we will see in Section V that benchmark comparison can take place without having to decide this question at all.

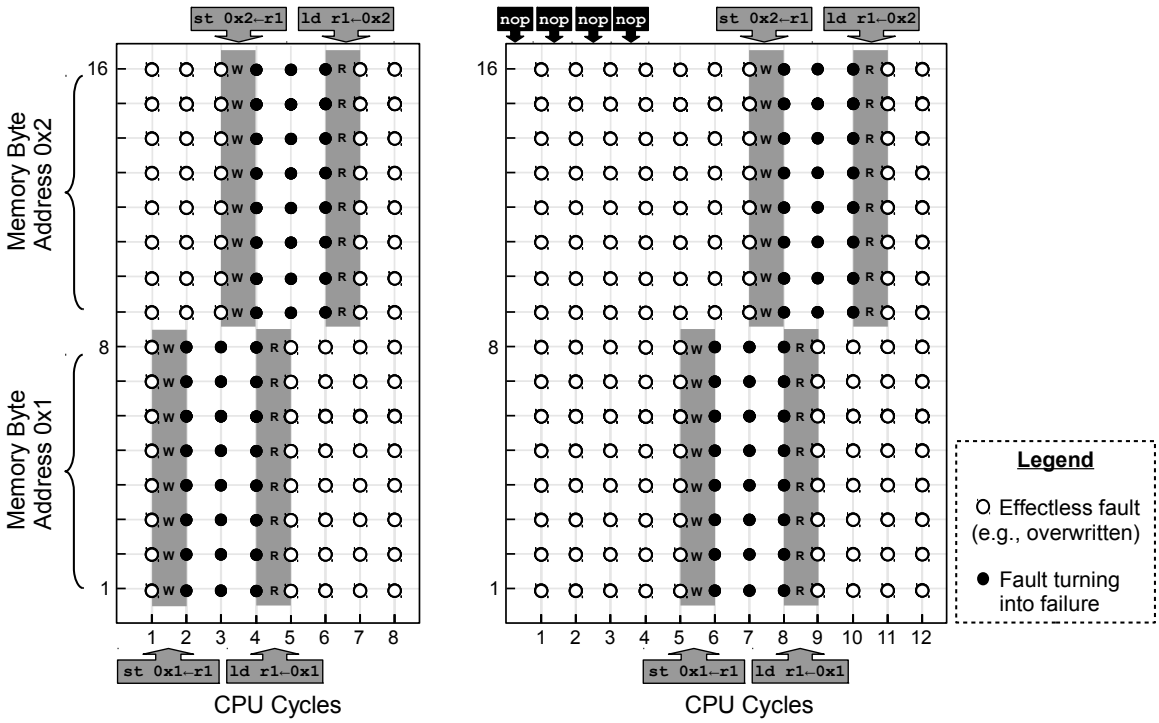
To summarize, a seemingly unsuspicious – but entirely artificial – program transformation can severely skew the fault-coverage metric.

### C. Analyzing the Root Cause

The remaining question is: *Why* is the fault-coverage metric unfit to discover the obvious ineffectiveness of the Dilution Fault-Tolerance mechanism – and does this metric skewing not only happen in an artificial example, but also occur in real-world examples?

The fact that we used a (hypothetical) full fault-space scan in the example, and not sampling, as most works in the community, is not the culprit. A sufficient number of samples taken from both fault spaces in Figure 3 would yield estimates close to the exact numbers for $c_{\text{baseline}}$ and $c_{\text{hardened}}$ from the previous section.

A closer look at the definition of fault coverage (Equation 2) suggests that *the metric itself is unfit* for benchmark comparison: The calculated percentages are not relative to the same base – a common $N$ for both benchmarks – as the divisor directly depends on the benchmark's runtime, and also its memory usage. (The DFT could also simply have used more memory for no particular purpose instead of prolonging the benchmark's runtime). Due to the usual overhead in space and time for most software-based fault-tolerance mechanisms, a different fault-space size for baseline and hardened variants must be considered the norm rather than the exception.




(a) Fault-space diagram for the baseline version, finishing after eight CPU cycles.

(b) Fault-space diagram after applying the *Dilution Fault Tolerance (DFT)* mechanism. Four no-operation instructions (`nop`) are prepended to the baseline version, resulting in an offset of four CPU cycles.

```
volatile char msg[2];          0  ld r1 <- 'H'
msg[0] = 'H';                  1  st 0x1 <- r1
msg[1] = 'i';                  2  ld r1 <- 'i'
serial_put_char(msg[0]);       3  st 0x2 <- r1
serial_put_char(msg[1]);       4  ld r1 <- 0x1
                               5  st $SERIAL <- r1
                               6  ld r1 <- 0x2
/* C program */                7  st $SERIAL <- r1
```

Fig. 3. Gedankenexperiment with C-like source code for the baseline version to the left, and corresponding machine instructions (and CPU cycle numbers) to the right.

If a simple benchmarking cheat – the DFT is, of course, nothing more – can improve the fault coverage from 62.5 % to 75 %, how about the *real* fault-tolerance mechanism used on the BIN_SEM2 and SYNC2 benchmarks in Figure 2b? Does the SUM+DMR mechanism *really* improve the hardware fault-tolerance of these programs, or is this also a (dilution) delusion?

The next section will try and construct an objective metric that can be used for benchmark comparison, and then revisit these benchmarks again.

## V. Constructing an Objective Comparison Metric

Arrived at the suspicion from the previous section that the fault-coverage factor may be unfit for the comparison of software-based fault-tolerance mechanisms, we construct an *objective comparison metric* in this section. Subsequently, we answer the question whether the SUM+DMR mechanism really improves both the BIN_SEM2 and SYNC2 benchmarks.

### A. Back to the Roots: Failure Probability

In Section I, we stated that the absolute probability for the benchmark's failure $P(Failure)$ represents the *ground truth* for comparing different variants of a benchmark. It can be calculated by decomposing it using the law of total probability:

$$\begin{aligned} & P(\text{Failure}) \\ = \; & P(\text{Failure}|0\,\text{Faults} \vee 1\,\text{Fault} \vee 2\,\text{F.} \vee 3\,\text{F.} \vee \ldots) \\ = \; & P(\text{Failure}|0\,\text{Faults}) \cdot P(0\,\text{Faults}) + \\ & P(\text{Failure}|1\,\text{Fault}) \cdot P(1\,\text{Fault}) + \\ & P(\text{Failure}|2\,\text{Faults}) \cdot P(2\,\text{Faults}) + \\ & P(\text{Failure}|3\,\text{Faults}) \cdot P(3\,\text{Faults}) + \ldots \end{aligned}$$

$P(\text{Failure}|0\,\text{Faults})$ is known to be zero, and from Section III-A we know $P(\text{k Faults})$ is negligibly small for $k \geq 2$ for real-world soft-error rates and sufficiently short benchmark runs. Hence:

$$P(\text{Failure}) \approx P(\text{Failure}|1\,\text{Fault}) \cdot P(1\,\text{Fault}) \quad (3)$$

In Equation 3, $P(\text{Failure}|1\,\text{Fault})$ can be directly calculated from the FI results collected by a complete fault-space scan (cf. Section III-A), using the number of failed experiments $F$,

and the fault-space size $w$ (cf. Section III-A):

$$P(\text{Failure}|1\,\text{Fault}) \quad = \quad \frac{F}{w} \tag{4}$$

In Equation 3, $P(1\,\text{Fault})$ can be calculated using the Poisson probability $P_\lambda(k=1)$ from Equation 1 in Section I. Inserting Eqn. 1 and 4 in Equation 3 yields:

$$\begin{aligned} P(\text{Failure}) \quad &\approx \quad \frac{F}{w} \cdot \frac{\lambda^1}{1!} e^{-\lambda} = \frac{F}{w} \cdot g \cdot w \cdot e^{-gw} \\ &= \quad F \cdot g \cdot e^{-gw} \end{aligned} \tag{5}$$

$g$ is constant for different benchmark runs, and may not even be exactly known (but is expected to be very small). $-gw$ is negative and depends on the fault-space size, but with the order of magnitude of the parameter values also so small (taking the example numbers from Section III-A gives $-gw \approx 1.3 \cdot 10^{-13}$) that assuming $e^{-gw} \approx 1$ yields an error of $1 - e^{-gw} < 10^{-12}$. Hence, the failure probability – the metric identified in Section I as the ground truth – can be approximated to be directly proportional to the absolute number of failed experiments $F$:

$$P(\text{Failure}) \quad \propto \quad F \tag{6}$$

Using this proportionality, we also can calculate the comparison ratio $r$, knowing that $r < 1$ denotes an improvement of the *hardened* variant over the *baseline*:

$$r \quad = \quad \frac{P(\text{Failure})_{\text{hardened}}}{P(\text{Failure})_{\text{baseline}}} = \frac{F_{\text{hardened}}}{F_{\text{baseline}}}$$

To conclude, the number of "Failed" FI experiments from a *complete fault-space scan* is a valid metric for comparing benchmarks.

### B. Fault Coverage and Failure Probability in the Real World

Figure 2e on page 5 shows the application of this finding to the BIN_SEM2 and SYNC2 benchmarks by plotting their weighted, raw failure counts. Comparing the new results to the weighted coverage from Figure 2b exhibits that BIN_SEM2 indeed turns out to be protected effectively by the SUM+DMR protection scheme, the same trend predicted by the misguiding fault-coverage plot in Figure 2b. More surprisingly, though, SYNC2 seems to *worsen* by more than a factor of five compared to its baseline – a fact that was completely hidden by the fault-coverage factor, resulting in a wrong design decision. The fact that the *hardened* variant of SYNC2 has an extremely increased runtime over the *baseline*, as indicated in Figure 2g, points at the reason: In this case, a massively increased "No Effect" rate (just as in our artificial "Hi" example in Section IV) completely hid the increase in absolute "Failure" results.

> *Pitfall 3: Fault-Coverage Percentages for Benchmark Comparison*
> Subsequently, our third and most important pitfall is the *usage of fault-coverage percentages* for benchmark comparison. Unless the fault space dimensions of two program variants are identical – which is practically never the case when effective software-based fault-tolerance mechanisms are in place –, their fault coverages are measured in percent relative to different fault-space areas, and are *by definition not comparable*. Instead, **absolute failure counts from a full fault-space scan must be used** for comparison.

Unfortunately, there exists an endless amount of studies that use fault coverage as a comparison metric for program susceptibility to soft errors in memory. Examples are Fuchs's analysis of the MARS operating system [4], Rebaudengo et al. [5] with a source-to-source compiler for dependable software, Nicolescu et al. [6] analyzing a hardened space communications application, Chen et al. [7] measuring the effectiveness of object duplication in a Java runtime environment, or our own work [38] analyzing a protection scheme for virtual-function pointers in C++. Depending on the overhead (affecting the fault-space size in time or memory dimensions) the analyzed protection mechanisms introduce, the conclusions in these studies may be wrong to the point that the mechanisms actually worsen the system's fault resilience.

### C. No Effect Results, and Sampling

For our metric, only "Failure" results are relevant for comparison. As we demonstrated and discussed in Section IV, "No Effect" results can be arbitrarily skewed (either voluntarily, or by chance) by artificially modifying the benchmark's runtime or memory usage.

> *Pitfall 3 (Corollary 1): "No Effect" Result Counts*
> Thus, Corollary 1 of Pitfall 3 is that **"No Effect" experiment outcomes are irrelevant** for the comparison of program susceptibility to soft errors in memory, and **should be excluded** from the data. Their occurrence is arbitrarily influenced by the activation and subsequent masking of faults, and ultimately by the sophistication of fault-space pruning mechanisms (cf. Section IV-B).

This also means that when combining def/use pruning with sampling (Section III-E), it is not necessary to sample from equivalence classes that are known to result in "No Effect". This reduces the population size from $w$ to $w' \leq w$.

When sampling is used (cf. Section III-B and III-E), the number of samples $N_{\text{sampled}}$, and indirectly also the measured "Failure" count $F_{\text{sampled}}$, is arbitrarily chosen by the developer (but potentially also influenced by the envisaged confidence level). Hence, the raw $F_{\text{sampled}}$ cannot be used directly for comparison. To get sampling results into a form usable for our metric, the raw sample counts must be extrapolated to the fault-space size to estimate the number of "Failure" results from a full fault-space scan.

> *Pitfall 3 (Corollary 2): Raw Sample Counts*
> Thus, Corollary 2 of Pitfall 3 is *not to use raw sample counts*. If sampling is used, the raw result counts are insufficient for benchmark comparison. The **result counts must be extrapolated to the population size** $w$ (or $w'$, see above) to be usable for this purpose.
>
> $$F_{\text{extrapolated}} \quad = \quad w \cdot \frac{F_{\text{sampled}}}{N_{\text{sampled}}}$$

One example from the literature that provides raw result numbers, but omits the extrapolation step from sampling to the full fault space, is from Nicolescu et al. [6].

*Summary: Avoiding Pitfalls 1–3*
To summarize, the comparison ratio $r$ to objectively compare hardware-fault tolerant software systems must be calculated as follows:

$$r = \frac{P(\text{Failure})_{\text{hardened}}}{P(\text{Failure})_{\text{baseline}}} = \frac{w_{\text{hardened}} \cdot \frac{F_{\text{hardened,sampled}}}{N_{\text{hardened,sampled}}}}{w_{\text{baseline}} \cdot \frac{F_{\text{baseline,sampled}}}{N_{\text{baseline,sampled}}}}$$

In the case of a complete fault-space scan, $w$ and $N$ are equal, and the given formula reduces to $r = \frac{F_{\text{hardened}}}{F_{\text{baseline}}}$.

## VI. Discussion and Generalization

In the following, we briefly revisit Pitfall 1 to get an intuition how weighting affects the absolute "Failure" counts from Pitfall 3. Subsequently, we describe possible generalizations of our findings to other fault models. Finally, we discuss a specific case of cross-layer fault-coverage comparison, and its implications for the validity of high-level FI.

### A. Revisiting Pitfall 1: The Effect of Weighting on Raw Numbers

In Section III-D, we discussed the necessity to weight def/use equivalence classes by their corresponding data lifetimes (Pitfall 1), and demonstrated the impact of this decision on the fault coverages of the BIN_SEM2 and SYNC2 benchmarks. After being convinced that fault coverage is an inadequate metric for comparison, how much influence does weighting have on the *absolute* "Failure" counts advocated in Section V-B (Pitfall 3)?

Figure 2d on page 5 presents the absolute failure counts *without* weighting. In this case, both benchmarks seem to be *less* resilient to soft errors in their hardened variant when compared to the baseline. In contrast, the weighted results in Figure 2e reveal that BIN_SEM2 in fact improves dramatically – again (cf. Section V-B), a wrong design decision would have been made.

This example underlines that *all* pitfalls mentioned in this paper must be paid attention to, as each of them independently can significantly falsify the results, and lead to incorrect design decisions.

### B. Possible Generalizations

In Section II-D, we simplified the failure modes to only "Failure" and "No Effect" types. Our findings can easily be generalized to more different experiment outcomes, for example, Pitfall 3 (Corollary 1) still holds: only "No Effect" results (denoting *no* visible effect for the observer) should be excluded, while the remaining *effective* result-type counts (e.g., "Silent Data Corruption", "Timeout", ...) should be included in the analysis and separately extrapolated to the fault-space size (Pitfall 3, Corollary 2).

In Section II-C, we strongly restricted the machine and fault model to simplify the explanations throughout this paper. Nevertheless, some of our findings may be generalizable to both complex machines and a broader hardware fault model.

A modern superscalar out-of-order CPU with several cache levels would primarily change the *timing* of memory-access events. The def/use equivalence-class sizes would be derived from more detailed timing information, and therefore would gain a more accurate weight (Section III-D). However, the order of instruction execution is irrelevant for the FI methodologies and the pitfalls we identified.

When a much more detailed simulator, e.g., on the flip-flop level, is available, our findings may be extensible to other parts of the memory hierarchy. Every bit in the caches, the CPU registers, or the microarchitectural state of the CPU, could be part of the fault space – requiring to also record read and write accesses to these bits for def/use pruning. Then, faults propagating from the CPU logic should also be taken into account for weighting. Even without weighting, especially Pitfall 3 may still be applicable. We will look into these issues in future work.

### C. Cross-Layer Comparisons, and the Invalidity of High-Level Fault Injection

Recently, two studies analyzed the general validity of high-level FI, e.g., injection into main memory and CPU registers, in Cho et al. [39]. Likewise, Wei et al. [40] use the state of an artificial virtual-machine model for high-level FI validation. Both studies compare the results from FI experiments running the *same* benchmarks on fault-injected low-level simulators (e.g., simulating at the flip-flop level [39], or the ISA level [40]), providing a ground truth to match against. Similarly to our findings in Section IV, the authors use the fault-coverage metric with different fault-space sizes. In this case, the different fault-space sizes are not caused by varying benchmark runtimes or memory usages, but by the vastly different simulator models affecting both the state-space size and timing granularity.

From their analysis, Cho et al. [39] conclude that high-level FI *"can result in high degrees of inaccuracies by more than an order of magnitude"*, quoting an error of up to factor 45. Without challenging the possibility that high-level FI may indeed be inaccurate, the used fault-coverage metric (with differing fault-space size quotients) may contribute significantly to this error, and suggest reevaluating the obtained result data using our comparison metric.

## VII. Related Work

Over time, several metrics for the assessment of fault-tolerance mechanisms have been devised. The classic *fault-coverage factor* metric from Bouricius et al. [29] defines a mathematical model that is used and instantiated by many subsequent approaches, and is described more in detail in Section III-B. In Arlat et al. [12], fault injection was initially *defined* to be a practical measurement method for fault coverage. Consequently, most FI tools provide a way to map their results to this metric [13], [14], [15].

Reis et al. [41] recognize the need for a metric that adequately captures the tradeoff between performance and reliability of software-based fault tolerance techniques, and devise the *Mean Work To Failure* (MWTF) metric based on an application-specific definition of "work units" and FI measurements. Unlike our metric, MWTF is based on measuring the *Architectural Vulnerability Factor* (AVF [42], see below) implying a constant $\Delta m$. Furthermore, the authors do not derive the connection between MWTF and $P(Failure)$, or the relation to common practices in the field. More recently, Santini et al. [43] introduced a similar *Mean Workload Between Failures*

(MWBF) metric parametrized with results from radiation measurements.

Several other metrics do not explicitly take the aforementioned performance/reliability tradeoff into account. Many are also based on dynamic analysis techniques, such as FI, but abstract from the low-level details to provide more information to guide software development. Hiller et al. [44] analyze the propagation of errors in modular software with their EPIC framework, and detect the most exposed modules and signals in the system using the *permeability* and *exposure* metrics. Johansson and Suri [45] extend this approach to the analysis of the dynamic behavior of an operating system. Similarly, Gawkowski and Sosnowski [46] also use FI to trace fault propagation over several levels, from logic up to the application level.

With the *Architectural Vulnerability Factor* (AVF), Mukherjee et al. [42] devised a classic *static* fault-tolerance assessment metric. Using low-level simulations, they measured the reliability of microarchitectural structures. On the software level, Sridharan et al. [47] developed the *Program Vulnerability Factor* that is independent of expert knowledge on the microarchitecture. Both AVF and PVF weight their results by the observed data lifetimes, and, thus, avoid Pitfall 1 (Section III-D). Similarly, Benso et al. [48] created a high-level *data criticality* metric determining the probability for each variable that it propagates an error to the program's output. More recently, Rehman et al. [49] proposed the *Application Vulnerability Index* (AVI), composed of values from their *Function Vulnerability Index* (FVI), and recursively their *Instruction Vulnerability Index* (IVI), the latter being derived in a comparable way as Mukherjee's AVF [42]. Based on their metrics, they control reliability optimization passes in a compiler. On an even more abstract level, Oz et al. [50] analyze multithreaded applications with the *Thread Vulnerability Factor* (TVF).

## VIII. Conclusions

After a step-by-step analysis of current practices in software-implemented FI, we identified three common pitfalls in interpreting FI result data for the comparison of program susceptibility to soft errors in memory. Showing the effects on a real-world data set, we demonstrated that each pitfall independently can skew or even completely invalidate the analysis, and lead to wrong conclusions regarding the effectiveness of software-based fault-tolerance.

Concretely, we pointed out 1) that special care has to be taken when processing the FI results after def/use fault-space pruning has been applied, 2) that sampling combined with def/use pruning must account for different equivalence-class sizes, and, most importantly, 3) that the widely used *fault coverage* metric is inadequate for the comparison of different benchmark variants. As a remedy, we derived an objective comparison metric that can be calculated both with full fault-space scans and from sampling results: Absolute failure counts, extrapolated to the fault-space size in the case of sampling.

For each pitfall we identified, we found FI studies that are most probably affected. Especially the usage of the fault-coverage metric for benchmark comparison is widespread – the few examples cited in Section V-B by no means particularly stand out. Although we believe that many of the described software-based hardware fault-tolerance mechanisms would prevail, we suggest to reevaluate them with our comparison metric to sort out mechanisms that in fact decrease fault tolerance of programs they are deployed in. With a similar motivation, a recent study by Shrivastava et al. [51] using the AVF metric [42] surprisingly showed that five control-flow checking schemes – claimed effective by their original authors – actually *increase* the system vulnerability.

In future work, we intend to look into different fault models, to compare simulation-obtained results of our metric to radiation measurements, and to evaluate and improve existing software-based hardware fault-tolerance mechanisms.

## Acknowledgments

We thank our anonymous reviewers for their very helpful and encouraging comments. We also thank Michael Engel for detailed comments, and his suggestion for naming the "dilution delusion" (Section IV-B). This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. SP 968/5-3.

## References

[1] D. Binder, E. Smith, and A. Holman, "Satellite anomalies from galactic cosmic rays," *IEEE TNS*, vol. 22, no. 6, pp. 2675–2680, Dec. 1975.

[2] T. C. May and M. H. Woods, "Alpha-particle-induced soft errors in dynamic memories," *IEEE Transactions on Electron Devices*, vol. 26, no. 1, pp. 2–9, Jan. 1979.

[3] S. Mukherjee, *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann, 2008.

[4] E. Fuchs, "An evaluation of the error detection mechanisms in MARS using software-implemented fault injection," in *2nd Europ. Depend. Comp. Conf. (EDCC '96)*, A. Hlawiczka, J. G. Silva, and L. Simoncini, Eds. Springer, 1996, pp. 73–90.

[5] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano, "A source-to-source compiler for generating dependable software," in *1st IEEE Int. W'shop on Source Code Analysis and Manipulation*, 2001, pp. 33–42.

[6] B. Nicolescu, Y. Savaria, and R. Velazco, "Software detection mechanisms providing full coverage against single bit-flip faults," *IEEE TNS*, vol. 51, no. 6, pp. 3510–3518, Dec. 2004.

[7] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Object duplication for improving reliability," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '06. Piscataway, NJ, USA: IEEE, 2006, pp. 140–145.

[8] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *43rd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '13)*. IEEE, Jun. 2013.

[9] X. Li, M. C. Huang, K. Shen, and L. Chu, "A realistic evaluation of memory hardware errors and software system susceptibility," in *2010 USENIX TC*. Berkeley, CA, USA: USENIX, 2010.

[10] V. Sridharan and D. Liberty, "A study of DRAM failures in the field," in *Int. Conf. for High Perf. Computing, Networking, Storage and Analysis (SC '12)*. Los Alamitos, CA, USA: IEEE, 2012, pp. 76:1–76:11.

[11] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi, "Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults," in *Int. Conf. for High Perf. Computing, Networking, Storage and Analysis (SC '13)*. ACM, 2013.

[12] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: A methodology and some applications," *IEEE TOSE*, vol. 16, no. 2, pp. 166–182, Feb. 1990.

[13] J. A. Clark and D. K. Pradhan, "Fault injection: A method for validating computer-system dependability," *IEEE Comp.*, vol. 28, no. 6, pp. 47–56, Jun. 1995.

[14] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *IEEE Comp.*, vol. 30, no. 4, pp. 75–82, Apr. 1997.

[15] A. Benso and P. E. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*, ser. Frontiers in electronic testing. Boston, Dordrecht, London: Kluwer, 2003.

[16] H. Ziade, R. A. Ayoubi, and R. Velazco, "A survey on fault injection techniques," *The International Arab Journal of Information Technology*, vol. 1, no. 2, pp. 171–186, 2004.

[17] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo, "Using heavy-ion radiation to validate fault-handling mechanisms," *IEEE Micro*, vol. 14, no. 1, pp. 8–23, Feb. 1994.

[18] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger, "Application of three physical fault injection techniques to the experimental assessment of the MARS architecture," in *Conf. on Dep. Comp. for Crit. App. (DCCA '95)*. Washington, DC, USA: IEEE, 1995.

[19] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *17th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '12)*. New York, NY, USA: ACM, 2012, pp. 123–134.

[20] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates," in *44th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '14)*. IEEE, Jun. 2014, pp. 622–629.

[21] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *40th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '10)*. Los Alamitos, CA, USA: IEEE, Jun./Jul. 2010, pp. 557–562.

[22] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE TDSC*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[23] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *IEEE Int'l Reliab. Physics Symp. (IRPS '11)*, Apr. 2011, pp. 5B.4.1–5B.4.7.

[24] A. Massa, *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[25] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th Int. Conf. on Arch. of Comp. Sys. (ARCS '12), Workshop Proceedings*, ser. Lecture Notes in Informatics, G. Mühl, J. Richling, and A. Herkersdorf, Eds., vol. 200. German Society of Informatics, Mar. 2012, pp. 201–210.

[26] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, 2nd ed. Wiley, 2002.

[27] D. Powell, E. Martins, J. Arlat, and Y. Crouzet, "Estimators for fault tolerance coverage evaluation," *IEEE TC*, vol. 44, no. 2, pp. 261–274, Feb. 1995.

[28] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: quantified error and confidence," in *2009 Conf. on Design, Autom. & Test in Europe (DATE '09)*. IEEE, 2009, pp. 502–506.

[29] W. G. Bouricius, W. C. Carter, and P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in *24th National Conference*, ser. ACM '69. New York, NY, USA: ACM, 1969, pp. 295–309.

[30] D. T. Smith, B. W. Johnson, J. A. Profeta, III, and D. G. Bozzolo, "A method to determine equivalent fault classes for permanent and transient faults," in *Annual Reliability and Maintainability Symposium*. IEEE, Jan. 1995, pp. 418–424.

[31] J. Güthoff and V. Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection method," in *25th Annual Int. Symp. on Fault-Tol. Comp. (FTCS '95)*. IEEE, Jun. 1995, pp. 196–206.

[32] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo, "Fault-list collapsing for fault-injection experiments," in *Annual Reliability and Maintainability Symposium*. IEEE, Jan. 1998, pp. 383–388.

[33] L. Berrojo, I. Gonzalez, F. Corno, M. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New techniques for speeding-up fault-injection campaigns," in *2002 Conf. on Design, Autom. & Test in Europe (DATE '02)*. IEEE, 2002, pp. 847–852.

[34] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *5th Europ. Depend. Comp. Conf. (EDCC '05)*, vol. 3463. Springer, Apr. 2005, p. 246.

[35] J. Grinschgl, A. Krieg, C. Steger, R. Weiss, H. Bock, and J. Haid, "Efficient fault emulation using automatic pre-injection memory access analysis," in *25th IEEE SoC Conf. (SOCC '12)*. IEEE, 2012, pp. 277–282.

[36] M. Hoffmann, C. Dietrich, and D. Lohmann, "Failure by design: Influence of the RTOS interface on memory fault resilience," in *2nd GI W'shop on SW-Based Methods for Robust Embedded Sys. (SOBRES '13)*, ser. Lecture Notes in Informatics. German Society of Informatics, Sep. 2013.

[37] R. Alexandersson and J. Karlsson, "Fault injection-based assessment of aspect-oriented implementation of fault tolerance," in *41st IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '11)*. IEEE, Jun. 2011, pp. 303–314.

[38] C. Borchert, H. Schirmeier, and O. Spinczyk, "Protecting the dynamic dispatch in C++ by dependability aspects," in *1st GI W'shop on SW-Based Methods for Robust Embedded Sys. (SOBRES '12)*, ser. Lecture Notes in Informatics. German Society of Informatics, Sep. 2012, pp. 521–535.

[39] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *50th Design Autom. Conf. (DAC '13)*. IEEE, May 2013.

[40] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *44th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '14)*. IEEE, Jun. 2014, pp. 375–382.

[41] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and evaluation of hybrid fault-detection systems," in *32nd Int. Symp. on Comp. Arch. (ISCA '05)*. IEEE, Jun. 2005, pp. 148–159.

[42] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *IEEE/ACM MICRO 36*. Los Alamitos, CA, USA: IEEE, 2003.

[43] T. Santini, P. Rech, G. Nazar, L. Carro, and F. Rech Wagner, "Reducing embedded software radiation-induced failures through cache memories," in *19th IEEE Europ. Test Symp. (ETS '14)*, May 2014.

[44] M. Hiller, A. Jhumka, and N. Suri, "EPIC: Profiling the propagation and effect of data errors in software," *IEEE TC*, vol. 53, no. 5, pp. 512–530, May 2004.

[45] A. Johansson and N. Suri, "Error propagation profiling of operating systems," in *35th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '05)*, Jun./Jul. 2005, pp. 86–95.

[46] P. Gawkowski and J. Sosnowski, "Evaluation of transient fault susceptibility in microprocessor systems," in *Euromicro Symp. on Digital System Design (DSD '04)*, Aug. 2004, pp. 432–439.

[47] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *15th IEEE Int. Symp. on High Performance Computer Architecture (HPCA '09)*. IEEE, Feb. 2009, pp. 117–128.

[48] A. Benso, S. Di Carlo, G. Di Natale, P. E. Prinetto, and L. Tagliaferri, "Data criticality estimation in software applications," in *2003 Int'l Test Conf. (ITC '03)*. Los Alamitos, CA, USA: IEEE, 2003.

[49] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel, "Reliable software for unreliable hardware: Embedded code generation aiming at reliability," in *9th IEEE/ACM Int. Conf. on HW/SW Codesign and Sys. Synth. (CODES+ISSS '11)*, Taipei, Taiwan, Oct. 2011, pp. 237–246.

[50] I. Oz, H. R. Topcuoglu, M. Kandemir, and O. Tosun, "Examining thread vulnerability analysis using fault-injection," in *21st Int. Conf. on Very Large Scale Integration (VLSI-SoC '13)*. IEEE, Oct. 2013, pp. 240–245.

[51] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu, "Quantitative analysis of control flow checking mechanisms for soft errors," in *51st Design Autom. Conf. (DAC '14)*. ACM, 2014.