ACTOR: Accelerating Fault Injection Campaigns using Timeout Detection based on Autocorrelation

Tim-Marek Thomas[1,✉], Christian Dietrich[2,✉], Oskar Pusz[1], and Daniel Lohmann[1]

[1] Leibniz Universität Hannover, Germany
{thomas,pusz,lohmann}@sra.uni-hannover.de
[2] Technische Universität Hamburg, Germany
christian.dietrich@tuhh.de

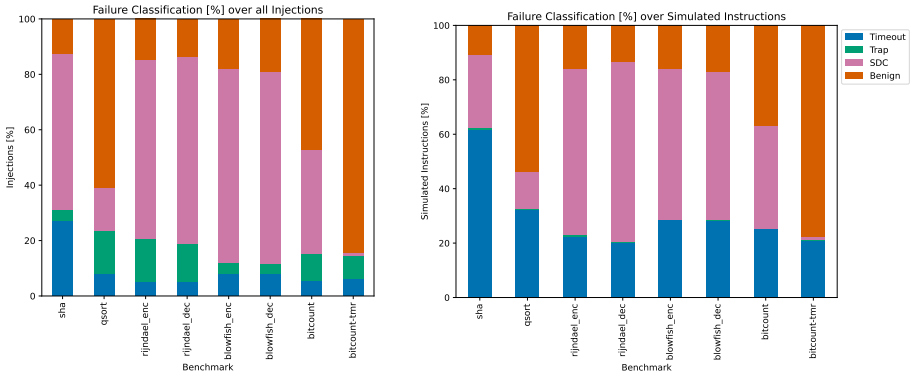
Accepted Version. Final Version: https://doi.org/10.1007/978-3-031-14835-4_17

Abstract. Fault-injection (FI) campaigns provide an in-depth resilience analysis of safety-critical systems in the presence of transient hardware faults. However, FI campaigns require many independent injection experiments and, combined, long run times, especially if we aim for a high coverage of the fault space. Besides reducing the number of pilot injections (e.g., with def-use pruning) in the first place, we can also speed up the overall campaign by speeding up individual experiments. From our experiments, we see that the timeout failure class is especially important here: Although timeouts account only for 8 percent (QSort) of the injections, they require 32 percent of the campaign run time.

In this paper, we analyze and discuss the nature of timeouts as a failure class, and reason about the general design of dynamic timeout detectors. Based on those insights, we propose ACTOR, a method to identify and abort stuck experiments early by performing autocorrelation on the branch-target history. Applied to seven MiBench benchmarks, we can reduce the number of executed post-injection instructions by up to 30 percent, which translates into an end-to-end saving of 27 percent. Thereby, the absolute classification error of experiments as timeouts was always less than 0.5 percent.

1 Introduction

Functional safety standards, such as ISO 26262 or IEC 61508 [15, 14], demand that we assess (and, if necessary, mitigate) the effects of transient hardware faults (soft errors) on our systems. As soft errors are rare in reality [28, 21], we often use *fault injection (FI)* [1, 29] to quantify the resilience of a program. Unlike radiation or heat experiments [9], which are probabilistic by nature, FI also gives us the chance to gain systematic insights as we can inject different *faults* into repeated re-executions of the same program. By observing the resulting erroneous misbehavior(s), we can classify the *failure* of the *system-under-test*



(a) Executed Injections (b) Simulated Instructions after Inj.

Fig. 1: Injection Count vs. Simulated Executions

(SUT) and provide a summarized overview. Fig. 1a shows the (unweighted) failure classification for seven MiBench [10] benchmarks if injected on the ISA level.

If the campaign designer wants to cover the entire *fault space (FS)*, which gives the most comprehensive picture of the potential misbehavior, we have to execute millions of injections. Even after applying standard fault-pruning methods [11, 27], our benchmarks require $2.8 \cdot 10^7$ injections. For these, our simulation-assisted FI platform [25] executed $1.3 \cdot 10^{12}$ instructions after the injection, which took us around 13 CPU days (at 1.16 MHz simulation rate). And although FI sampling [8] can reduce the number of injections, long-running programs with a large state will still require many independent injections.

Whenever the injected program execution deviates from the *golden run* (i.e., the fault is not *benign*), this typically also impacts its execution time, that is, the time it takes until the error is detected and the simulation terminates with a failure classification. However, the different failure classes can differ significantly in their share of the simulation time (see Fig. 1b): For example, although 15.6 percent of all QSort injections yield a trap (e.g., division by zero), they only account for 0.4 percent of the simulated instructions, so apparently, trap errors are detected early. On the other hand, *timeout* faults are detected late: 8.1 percent of faults in QSort account for 32.3 percent of the simulated instructions.

Timeout is meant to catch fault-induced endless loops and is a special failure class: It does not convey a ground truth, as deriving the ground truth would imply a solution for the halting problem. Instead, we need to heuristically classify an experiment as a timeout (and abort the simulation) by invoking the timeout-handler after some time $t_{\text{inv}} \in [t_1, \infty)$, with t_1 being the fault-free execution time. The selection of t_{inv} is a tradeoff between accidentally misclassifying longer-running experiments as timeout (false positives) and prolonging simulation time (as each true positive runs until t_{inv}).[1] The common approach is to select t_{inv} by

[1] In hard real-time settings, the situation is somewhat different: Here, the respective task's deadline would actually define a ground truth for timeout errors and, thus,

stretching the execution time t_1 by a *timeout factor*. This factor is arbitrary by definition; in the literature, commonly a factor between two and five is chosen without any further justification [24, 17, 23], some even suggest a factor of ten [6]! Following this, we assume the apparently most common factor of three (i.e., $t_{\text{inv}} = 3t_1$) throughout this paper, which, for our above campaign, let to 3.6 CPU days for alleged endless loops to complete. To sum up: *Timeout detection* is notoriously imprecise, while accounting for a considerable share of simulation time in FI campaigns.

About this Paper We propose and analyze *Autocorrelation-based Timeout Restriction* (ACTOR), an approach for dynamic timeout prediction that mitigates the costs of timeouts. ACTOR employs a low-overhead autocorrelation-based predictor that classifies faults early on as timeouts by observing the jump patterns of the continued execution, thereby reducing the overall campaign time. In particular, we claim the following contributions:

- We analyze the nature of the timeout failure class, reason about the maximal achievable savings of any timeout detector, and give guidelines for their design.
- We propose and implement autocorrelation to heuristically detect faulty executions that will lead to a timeout.
- We evaluate our ACTOR prototype on seven MiBench benchmarks and quantify the achieved end-to-end savings (up to 27.6 %) and the classification error.

The rest of the paper is structured as follows: In Sec. 2, we describe our fault-injection model and discuss the problem of timeout detection. Sourced by those insights, we design ACTOR in Sec. 3 and evaluate it in Sec. 4. After the discussion of our results (Sec. 5) and the related work (Sec. 6), we conclude this paper in Sec. 7.

2 Problem Analysis

In a nutshell, we aim to reduce FI-campaign run times by detecting experiments that are most likely to result in a timeout early and abort their continuation. For this, we will first describe our targeted models of FI campaigns and reason afterwards in general about *timeout detectors*.

2.1 Fault-Injection Model

ACTOR targets systematic FI campaigns, were a single deterministic program run on a specific system is examined for its resiliency. For this, we record a fault-free *golden run* of the SUT and plan a number of faults that cover the (partial or complete) FS. The start of the golden run is t_0, its end is t_1. Each

also the upper bound for t_{inv}. However, depending on the tightness of the deadline, this might still prolong the simulation time too much.

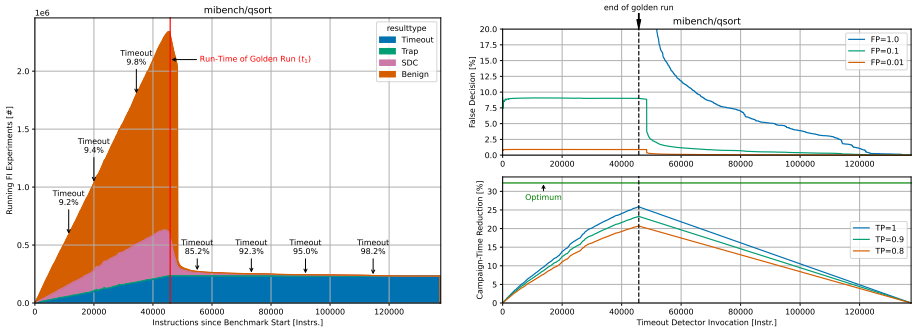
fault is identified by its fault location (e.g., register `r0` bit 3) and its relative fault time t_f (i.e., $t_f = t_0 + a$, $t_0 \leq t_f \leq t_1$).

For injecting a specific fault, the *FI platform* (e.g., a modified x86 emulator) *forwards* the program to the fault time and injects the fault (e.g., toggling one or multiple fault-location bits). Depending on the FI platform, forwarding is made more efficient using checkpointing [18, 3] or (hardware-assisted) break points [26]. In contrast, we cannot speedup the *post-injection execution* as the faulty control flow can deviate from the golden run. Therefore, this paper looks only on the time-budget spent *after* injecting the fault.

After injection, the platform continues the SUT, observes its behavior, and comes to a failure classification. While such classification is always application-specific, the classes *benign*, *silent-data corruption (SDC)*, *trap*, and *timeout* are commonly used.

For our approach, we furthermore assume that the FI platform can report the last m jumps. This can either be done via actively recording jumps (in a simulator) or by a hardware-implemented branch-history buffer. Without loss of generality, we explore the ACTOR approach on an ISA-level fault injection.

2.2 Timeout Detectors



(a) Histogram of Running Experiments

(b) Detectors of Different Quality

Fig. 2: Running Experiments for QSort and the Influence of the Timeout-Detector Quality

As already mentioned, timeout is a special failure class as the FI platform cannot surely classify stuck programs into one of the other classes. Therefore, the campaign designer must define a *timeout detector* that classifies the currently injection as a timeout.

These detectors can either be *static* and ignore the current system state or they are *dynamic* and make a heuristic decision. Furthermore, the detector-invocation time t_{inv} can either be *relative* to the FI time t_f (e.g., 50 cycles after injection, $t_{inv} = t_f + 50$), *absolute* with regard to the golden run (e.g., 300% of the normal run time, $t_{inv} = 3t_1$), or *continuously* applied after injection. Also, any real-world

detector induces an overhead and will produce incorrect results (as they can only be a heuristic).

Usually, campaign designers define that executions that take N-times longer (usually $N = 2 \dots 10$ [24, 17, 23, 6]) than the golden-run length are considered as a timeout. In our taxonomy, this is a static detector with an absolute invocation point at $N \cdot t_1$, whose *true positive (TP)* and *false positive (FP)* rate is 1.

To give you a better intuition, Fig. 2a shows a stacked histogram of the FI-experiment "population" (for QSort). Thereby, the population is the number of parallel running experiments that execute at a given point if we would start them all in parallel. For example, at $t_0 + 20000$, we execute one million experiments and from those 9.4 percent will still execute at $3t_1$. Please note that this graph ramps up until t_1 as we only consider the post-injection time. Furthermore, the integral over Fig. 2a is the total number of executed post-injection instructions (i.e., the minimal campaign time) that, if broken down by resulting failure class, has been shown in Fig. 1b.

For QSort and the static $3t_1$ detector, we spend 32 percent of the campaign time for executing stuck programs. With the (hypothetical) timeout detector called OPT (relative, TP=1, FP=0), which surely stops all timeouts at fault time $t_{\text{inv}} = t_{\text{f}}$, we reach the theoretical optimum. In Fig. 2a, OPT removes the complete blue area. Between these extremes (OPT and $3t_1$), we will now explore the possible design space of dynamic detectors. Hence, the dynamic detectors are an *addition* to the static $3t_1$ detector, which keeps experiments surely bounded.

First, we ask when to invoke a detector and if its invocation should be relative to fault-time. For this, we look at the population size at a given t and its composition. Shortly after t_1, executions that masked the fault or that incorporated it into their outputs without running longer terminate. For QSort, the population shrinks by around 80 percent from its maximum, while the share of timeout experiments rises from below 10 percent to over 80 percent. Please note that these experiments check their result within the simulator, whereby the described drop does not happen immediately at t_1. As every detector has overhead, which multiplies with the population size, the timeout-detection cost drops significantly after t_1, which results in larger end-to-end savings.

Furthermore, real-world detectors will have a FP rate > 0 that, if applied to a population with many non-stuck experiments, will lead to a large number of *false positives (FPs)*. As FPs skew the failure classification, we consider them to be more important than *false negatives (FNs)*, which only prolong the campaign. To illustrate this, Fig. 2b (upper half) shows the influence of the FP rate for an absolute detector on the percentage of false decisions. Before t_1, a detector that is 90 percent correct makes wrong decisions in about 9 percent of the cases, while after t_1, even a detector that labels all experiments as timeout (FP=1) quickly becomes usable. Even better for detectors that have a lower FP rate. Therefore, we argue that detectors should be invoked after t_1, which also rules out relative detectors as they would often become active before t_1.

On the other hand, we should invoke the detector as early as possible to maximize its effect and avoid executing stuck programs. For this, Fig. 2b (lower

half) shows the campaign-time reduction that absolute detectors with different TP rates can achieve over the $3t_1$ detector. Before t_1, which we already ruled out, an absolute detector cannot help much as many timeout experiments have not started yet. However, with progressing time, we lose saving potential (Lost Cycles) as the $3t_1$ limit comes closer and closer. Nevertheless, we also see that right after t_1 even detectors that achieve only a TP rate of 80 percent save 20 percent of our overall campaign run time. Please also note that absolute detectors invoked at or after t_1 have a benchmark-specific maximum that they can reach. For QSort, this upper limit is at 80 percent of OPT's savings.

To conclude our considerations: We should use absolute timeout detectors that we invoke shortly after t_1, where they cannot do much harm, even if they have a high FP rate. At this point, even if they are bad at detecting stuck programs (low TP rate), their saving potential is still high.

3 Timeout Detection using Autocorrelation

The core idea of ACTOR is that stuck programs will probably execute in a (rather tight) loop, whereby their instruction stream becomes periodic. If the observed periodicity exceeds a certain threshold, which we have to choose above the periodicities of the fault-free execution, we abort the FI experiment and classify it as a timeout although we have not waited until $3t_1$. We base our detector on Ibing et. al. [16], who use autocorrelation on the branch-target history to detect stuck executions on the fly. We adapt this technique for the FI context to achieve actual end-to-end savings and chose parameters for the specific benchmark.

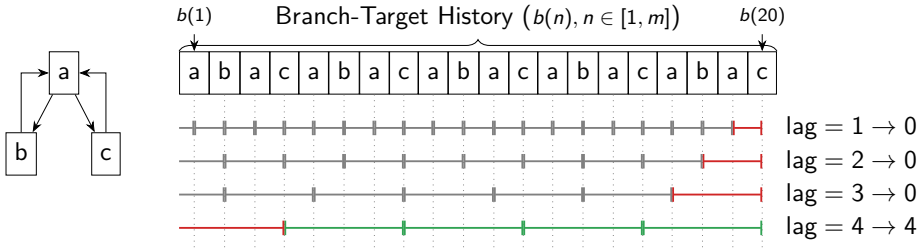


Fig. 3: Autocorrelation for Branch-Target History. $b(m)$ is the latest branch, while $b(1)$ is the oldest recorded branch. For lag 1-3, we cannot fit a periodic pattern, while with lag 4, the pattern continues throughout the branch-target history.

First, we want to give you a brief overview of the autocorrelation, which is often used in signal processing and statistical analysis, in the context of detecting periodic infinite loops [16]. The authors of this article apply discrete autocorrelation on the branch-target history instead of the full program trace, since the sequence of jump targets is sufficient to reconstruct the full path through a program. At a certain point in time, we look at the last m branches and compare the recorded branch-target sequence with a time-lagged version of itself. The

discrete autocorrelation can be simplified to a recursion $R_{\text{bb}}(l, m)$, where l is the currently examined *lag*:

$$R_{\text{bb}}(l, m) = \begin{cases} R_{\text{bb}}(l, m-l) + 1, \text{if } b(m) = b(m-l) \\ 0, \text{else} \end{cases} \tag{1}$$

In a nutshell, we count, beginning from the last taken branch ($b(m)$), how often we can jump l branches backwards in time before we hit a branch target unequal to $b(m)$. If we repeat this with different lags (e.g., $l \in [1, 64]$) on a fixed branch history, we end up with a vector of autocorrelation values $\vec{R}_{\text{bb}}(m)$. For example, in Fig. 3, the program is stuck within an endless loop that takes alternating conditional branches with each iteration. As the last branch target c is taken every 4 jumps, we end up with $\vec{R}_{\text{bb}} = \langle 0, 0, 0, 4 \rangle$. If the autocorrelation value exceeds a given threshold T, it can be classified as a timeout.

3.1 Adaption as Timeout Detector

To use autocorrelation as an absolute, dynamic timeout detector, we have to make adaptions and choose parameters. A static $3t_1$ detector is used as a fallback for potential false negatives. The main problem with the integration is the overhead of the detector and detection latency, as both are crucial to achieve actual end-to-end savings.

First, we have to decide on the history length and when to execute the *autocorrelation (AC)*. Ibing et.al. [16], which looked at natively run programs, used the binary-instrumentation package Pin [22] to hook all branches. They ran the autocorrelation continuously on every branch and, for a history length of 100, they report slowdown *factors* of 100x to 225x, which would diminish all savings that we could achieve with a timeout detector.

Therefore, guided by our discussion of timeout detectors (see Sec. 2.2), we diverge in several points from Ibing et.al: (1) Since recording branches will slowdown most FI platforms, we only start recording branches at t_1 where the execution's population starts to dwindle quickly (see Fig. 2a). (2) From thereon, we record branches until the branch-history buffer is filled up to a certain level and then execute the autocorrelation *exactly once*. This bounds the overhead per experiment but comes at the cost of detecting less timeouts. If ACTOR does not detect a timeout, no further overheads are induced afterwards. Now, we only have to choose three parameters: the history length, the maximal lag $l_{\max}$, and the threshold T at which we classify an experiment as a timeout.

For the history length m, we look at the development of the population size (Fig. 2a). As we have argued that time detector should run shortly after t_1, we choose to run the detector-invocation point at around $1.2t_1$. To achieve this, we derive the size of the history buffer from the average-branch density and the length of the golden run. For example, for a benchmark where every tenth instruction is a branch and $t_1 = 1000$ instrs., we set the history length to $0.2 \cdot \frac{1000}{10} = 20$ branches. Please note that, since the faulty programs can deviate from the original program, the branch buffer can be filled before or after $1.2t_1$.

The second parameter that we have to choose is the maximal lag $l_{\max}$. With a large $l_{\max}$, our detector becomes sensitive to patterns with a larger periodicity, which we expect to result in more brittle decisions (higher FP rate). For example, with a lag of 128, ACTOR could detect periodic sequences that repeat only every 128 branches. Therefore, we choose our maximum lag to be 16, which also is in concordance with our goal of detecting tight loops.

At the threshold T, we classify an experiment as a timeout, which challenges us to choose T such that ACTOR does not trigger on regular program behavior but is still able to detect timeouts. Since Ibing et al. [16] did not restrict the history length, they could use a rather large threshold (i.e., $T = 500$) that was applied regardless of the lag. However, with our fixed-sized history length, $R_{\text{bb}}(l, m)$ is always less than $\lfloor m/l \rfloor$, whereby the need for a lag-specific threshold vector $\vec{T} = (T_1, \ldots, T_{l_{\max}})$, which we will compare against $\vec{R}_{\text{bb}}(m)$, arises. If any observed value surpasses its threshold, we report a timeout.

$$T_l = 1 + \max_{s \in [0,(|H|-m)]} R_{\text{bb}}(l, H[s, s+m]) \tag{2}$$

To calculate T_l for lag l, we find the maximum R_{bb} value that we observe if we perform autocorrelation on the golden run and increase it by 1. For this, we shift an m-sized window over the branch-target history H of the golden run and calculate the autocorrelation. With the resulting $\vec{T}$, ACTOR cannot trigger if confronted with a regular program run even if the injected fault shifts the execution beyond t_1.

3.2 FAIL* Integration

We integrated the ACTOR approach in the simulation-assisted open-source FI framework FAIL* [25], which provides infrastructure for golden-run tracing, fault planning, distributed and parallelized campaign execution, and result analysis. FAIL* utilizes the independence of injections using a client-server-architecture to highly parallelize FI campaigns. We integrated ACTOR into the IA-32 injector client, which is based on the Bochs simulator [19]. With a deterministic timer breakpoint, we start recording the branch-target history at t_1, whereby we use the FAIL* infrastructure to record branches directly from Bochs' simulator loop which keeps the overheads as low as possible. The source code is publicly available.[2]

4 Evaluation

With our evaluation, we demonstrate that ACTOR is able to reduce the end-to-end campaign run-times without skewing the result statistic towards the timeout class. We use the classification results and the campaign run time of the static $3t_1$ detector as the ground truth and the baseline. We also show the theoretical optimum that OPT would achieve (see Sec. 2) if invoked at injection time t_{f} ($\text{OPT}_{t_{\text{f}}}$) and compare ACTOR to a static detector invoked at $1.2t_1$.

[2] https://doi.org/10.5281/zenodo.6534708

Benchmark	Classification Error [Δ %]				$1.2t_1$ Detector	ACTOR Detector		
	Ben.	SDC	Trap	TO	TO [Δ %]	Inv.	TPR	FPR
BitCount	+0.00	−0.02	+0.00	+0.02	+0.04	4.69 %	98.51 %	93.40 %
BitCount-TMR	−0.21	−0.01	+0.00	+0.22	+38.14	43.86 %	86.12 %	0.59 %
QSort	−0.05	−0.35	−0.03	+0.43	+1.41	9.01 %	88.55 %	37.72 %
SHA	+0.00	−0.16	−0.11	+0.27	+13.75	27.50 %	99.75 %	42.29 %
Blowfish (enc)	−0.06	−0.12	−0.01	+0.19	+0.31	8.37 %	98.27 %	82.40 %
Blowfish (dec)	−0.07	−0.13	+0.00	+0.20	+0.28	8.21 %	96.76 %	84.12 %
AES (enc)	−0.03	−0.26	−0.11	+0.40	+0.46	5.63 %	99.92 %	56.39 %
AES (dec)	−0.07	−0.08	−0.03	+0.19	+1.49	8.51 %	85.65 %	5.27 %

Table 1: Quality of the Failure Classification. For each failure class, we report the relative classification error compared to a static $3t_1$ timeout detector. For ACTOR, we report the TP and FP rates, the percentage of experiments involving a detector invocation (Inv.). For comparison, we also show the classification error in percentage points that a $1.2t_1$ detector would exhibit.

We ran seven benchmarks from the automotive and security branch of the MiBench [10] benchmark suite on FAIL*'s IA-32 backend (Bochs). Additionally, ACTOR was also applied to a modified BitCount benchmark using *triple modular redundancy (TMR)*. As a fault model for this evaluation, we use uniformly-distributed single-bit flips in registers and memory, and classify the failure into benign, SDC, trap, and timeout (TO). For the evaluation, we also record whether a timeout was detected by ACTOR or by the static fall-back 3_{t_1} detector. We performed the FIs on a 17-node Intel X5650 @ 2.67 GHz (12 cores) cluster, leading to 204 simultaneously run simulations. Timestamps were both taken in simulated instructions and in wall-clock time.

First, we look at the influence of ACTOR on the failure-classification statistic (see Tab. 1, Δ %). In total, we see that ACTOR has only a small impact on the failure classification over all benchmarks and that it shifts less than 0.5 percent of all FIs from another failure class into the timeout class. We also see that our invocation strategy (at around $1.2t_1$) successfully restricts the usage of the ACTOR detector to less than 10 percent of all experiments. Only for SHA, which exhibits a high number of long-running timeouts (see Fig. 1a), and BitCount-TMR, which naturally has a longer runtime when one of the results is corrupted, our detector is invoked more often. Furthermore, the ACTOR detector is very good (TPR $> 85\%$) at aborting experiments that would still execute at $3t_1$. In all cases, the static $1.2t_1$ detector, which is invoked at around the same time as ACTOR, shifts more experiments into the TO class.

However, the FP rate of our ACTOR detector varies widely between 1 to 94 percent, which means that ACTOR marks experiments as timeouts although they would eventually result in a different classification before $3t_1$. We still achieve good results for the classification error for two reasons: (1) we invoke the detector only on a small share of experiments (see Tab. 1, Inv.), and (2) from these experiments, only a small share will yield a non-timeout (e.g. Fig. 2a). Therefore, even a large FP rate yields small changes in the result. We will discuss the FP-rate issue in more detail in Sec. 5.

Benchmark	Autocorrelation			Sim. Post-Inj. Instr. [%]			E2E Saving
	H-Ln.	l_{avg}	Cost	ACTOR	$1.2t_1$	OPT_{t_f}	
BitCount	2 705	4.0	64 µs	−13.1	−17.3	−25.2	−12.66 %
BitCount-TMR	5 296	4.6	62 µs	−9.9	−14.6	−21.1	−7.39 %
QSort	1 385	1.6	53 µs	−19.5	−23.3	−32.3	−16.07 %
SHA	1 766	1.0	68 µs	−30.6	−44.4	−61.7	−27.64 %
Blowfish (enc)	1 019	9.6	67 µs	−16.1	−20.7	−28.7	−15.91 %
Blowfish (dec)	981	9.5	65 µs	−15.8	−20.4	−28.4	−15.72 %
AES (enc)	851	1.0	47 µs	−17.2	−16.0	−22.2	−17.59 %
AES (dec)	794	1.0	38 µs	−13.1	−14.5	−20.1	−12.24 %

Table 2: Campaign Run-Time Reductions. Besides the achieved end-to-end savings (w/ overheads), we show the reduction of simulator time for ACTOR, the OPT_{t_f}-detector, and a $1.2t_1$ detector. We also quantify the autocorrelation with the history length (H-Ln.), the average abort lag l_{avg}, and the run-time cost.

In Tab. 2, we show the run-time savings that ACTOR achieves by aborting experiments early. For the number of simulated post-injection instructions, we reduce the campaign run time by at least 9.9 percent and by up to 30 percent. In comparison to the theoretical optimum (OPT), ACTOR achieves a respectable reduction, although it is invoked on average $0.7t_1$ time units later. In direct competition with a $1.2t_1$ detector, which acts around the same time as ACTOR, we stay within 5 percent points (except for SHA). Please note, that ACTOR sometimes reaches bigger savings than the static detector since executions that are stuck in a tight loop often fill the history buffer before $1.2t_1$.

We are able to translate these simulation-time reduction into actual end-to-end savings for the campaign run time by at least 7.4 percent and up to 27 percent. This success is rooted in two design decisions: (1) The AC itself is fast (< 70µs) since we bound the lag and the branch-target history and we invoke the AC exactly once. (2) Recording the branch-target history within the simulator loop reduces the simulation frequency by 28 percent. However, as we only activate this in the interval $[1.0, 1.2] \cdot t_1$, the simulation-time reductions translate well into end-to-end savings.

5 Discussion

Clearly, the decision between aborting an experiment as a timeout and continuing its execution (also beyond $3t_1$) is a trade-off between campaign run-time and result quality. In essence, for programs without a hard deadline, no timeout detector can distinguish between stuck programs that will never halt and faulty-programs that execute for a (very) long but bounded time. So, in general, there is no ground truth for timeout detection but only similarity between different timeout detectors. Therefore, the absolute share of timeouts is a source of uncertainty in the resilience assessment of a program and the campaign designer has to decide whether the observed uncertainty is acceptable in the current design stage.

Without ACTOR, the only way to reduce the number of timeouts and to classify more experiments as non-timeouts is to prolong the observation time. With ACTOR, we are able to achieve campaign run-times similar to an aggressive $1.2t_1$ detector (see Tab. 2), but at timeout rates close to the $3t_1$ detector (see Tab. 1). For example, in the best case (SHA), we produce 13 percent points less timeouts than $1.2t_1$ while the end-to-end campaign run time reduces by 27 percent. Therefore, a viable route for a campaign designer is to use ACTOR in combination with a $3t_1$ fall-back detector. If the timeout rate exceeds his safety margins, e.g. those required by a certain standard, he can re-run aborted experiments with an ACTOR-variant that gets activated later (e.g., $1.4t_1$).

The other important aspect to discuss is the widely varying FP rate of our AC-based detector, which sometimes results in high ($> 80\,\%$) FP rates. These FPs stem from the detection principle of ACTOR to abort highly periodic executions. For our *non-TMR* benchmarks, which perform no error mitigation, timeouts occur if the injected fault hits a loop counter that prolongs the execution by a certain time depending on the flipped bit: *least-significant bit (LSB)* flips prolong the execution only slightly, while a *most-significant bit (MSB)* hit results in a large number of additional iterations. While both injections result in a highly-periodic branch pattern, which triggers ACTOR, some experiments still terminate before $3t_1$. In our eyes, the categorization of those injections as timeouts at $3t_1$ is quite arbitrary as they could still terminate with an SDC or a benign after this static time mark. We basically chose $3t_1$ as our "ground truth", because it best reflects the numbers commonly reported in the literature [24, 17, 23, 6].

To put these results in more context, we build the TMR-protected variant of BitCount, which actively schedules a third execution on a detected error, whereby 86 percent of the executions at $1.2t_1$ will terminate before $3t_1$. While this is a similar execution-prolongation pattern as a loop-counter injection, ACTOR is able to differentiate with a very low FPR of 0.59 percent between the third execution and a behavior that leads to a timeout. Therefore, we conclude that ACTOR is well suited to work on benchmarks with enabled mitigation techniques – which are of special interest for the campaign designers.

ACTOR is furthermore limited to FI scenarios where a fault-free execution trace is available, which we use to derive the detector parameters (H and $\vec{T}$). If those parameters can be chosen otherwise, ACTOR can also be used without a trace. Furthermore, we found the third parameter $l_{\max} = 16$ worked quite well for our benchmarks. However, for SUTs with many convoluted loops and conditional branches a higher limit could be chosen to detect more timeouts.

A threat to the external validity is our limited selection of benchmarks, which we chose from the automotive and security branch of the MiBench suite [10] and which is generally considered to be representative for applications in safety-critical environments. We also evaluated ACTOR only on the ISA level. However, ACTOR can be generalized to other levels (e.g., RTL, gates) as long as a mechanism to collect branch targets (i.e., branch-target buffer) is available.

6 Related Work

ACTOR is a fault-outcome prediction, which makes heuristic decisions about the outcome of a running experiment. To our knowledge, ACTOR is the first attempt at *dynamic* timeout detection for the FI of transient hardware faults into running programs; usually timeout detectors with a static execution budget [24] are used. Nevertheless, others have proposed outcome-prediction strategies for other failure classes. For example, SmartInjector [20] chooses predictor instructions and trigger values for which the predictor instruction will produce a benign or SDC result. During the FI experiment, when the faulty control-flow reaches a predictor instruction, they compare the actual value to trigger value and abort the experiment in case. On the gate level, we [7] have proposed fault-masking terms to detect benign faults within the first cycle after injection. GangES [13] runs several FI experiments in parallel and looks for equal execution states in different experiments. If two experiments have the same state, only one experiment is completed and its outcome is transferred to the other. However, they only perform matching for a limited time after injection, whereby their measures become ineffective for experiments that become stuck late.

In a broader sense, infinite loops can, as discussed throughout the paper, be detected by autocorrelation [16], although the original 100x to 225x overhead makes the unmodified approach unsuitable for FI. Another approach is Looper [4], which use an SMT theorem solver to generate non-termination formulas, which are checked at run-time. However, they also report prohibitive run-time overheads of up to 10 000x. The third route is to detect recurring program states, which was done by Carbin et al. [5]. They record the *whole* program state and report an infinite loop if that state did not change in between two loop iterations. However, for our FI benchmarks, we have observed that timeouts often continue to change their program state (e.g., decrementing a faulty loop counter).

Fault pruning, which reduces the number of planned faults by choosing pilot injections that represent a group of faults, are a different way to speed up FI campaigns. These techniques are complementary to ACTOR. Bartsch et al. [2] have proposed a static program analysis based on unrolled data-flow graphs ("program netlists") that finds faults that will surely become benign. Relyzer by Hari et al. [12] applies heuristic known-outcome pruning to reduce the amount of experiments but is only able to find benign, SDC and trap experiments.

7 Conclusion

With this paper, we investigate on the nature of the failure classification *timeout* in the context of FI campaigns for transient hardware faults. We observe that timeouts require an over-proportional large amount of execution time, which makes them a prime target for experiment-speedup techniques. From our analysis, we derive that *timeout detectors* should execute shortly after the fault-free program run-time to achieve the highest end-to-end savings while limiting their negative effect of the failure classification.

Based on this, we present ACTOR, an autocorrelation-based dynamic timeout detector that detects highly-periodic branch patterns and aborts the FI experiment early if it exceeds thresholds we derive from the golden run. Applied to seven benchmarks from the MiBench benchmark suite, ACTOR achieves end-to-end savings that range from 7.4 percent up to 27.6 percent in comparison to a static timeout detector. Thereby, ACTOR maintains a low classification error of less than 0.5 percentage points and a high ($> 85\,\%$) true-positive rate for experiments that can be stopped early.

References

1. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E., and Powell, D.: Fault Injection for Dependability Validation: A Methodology and Some Applications. IEEE Trans. on Soft. Engin. 16(2) (1990). DOI: 10.1109/32.44380
2. Bartsch, C., Villarraga, C., Stoffel, D., and Kunz, W.: A HW/SW Cross-Layer Approach for Determining Application-Redundant Hardware Faults in Embedded Systems. J Electron Test 33(1) (2017). DOI: 10.1007/s10836-017-5643-3
3. Berrojo, L., González, I., Corno, F., Reorda, M.S., Squillero, G., Entrena, L., and Lopez, C.: New techniques for speeding-up fault-injection campaigns. In: Design, Automation and Test in Europe Conference and Exhibition (2002)
4. Burnim, J., Jalbert, N., Stergiou, C., and Sen, K.: Looper: Lightweight Detection of Infinite Loops at Runtime. In: Automated Software Engineering (ASE'09) (2009). DOI: 10.1109/ASE.2009.87
5. Carbin, M., Misailovic, S., Kling, M., and Rinard, M.C.: Detecting and Escaping Infinite Loops with Jolt. In: Mezini, M. (ed.) 25th European Conference on Object-Oriented Programming (ECOOP'11) (2011). DOI: 10.1007/978-3-642-22655-7_28
6. Di Leo, D., Ayatolahi, F., Sangchoolie, B., Karlsson, J., and Johansson, R.: On the Impact of Hardware Faults – An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions. In: Ortmeier, F., and Daniel, P. (eds.) Computer Safety, Reliability, and Security (SAFECOMP'12) (2012)
7. Dietrich, C., Schmider, A., Pusz, O., Payá-Vayá, G., and Lohmann, D.: Cross-Layer Fault-Space Pruning for Hardware-Assisted Fault Injection. In: 55th Annual Design Automation Conference (DAC '18) (2018). DOI: 10.1145/3195970.3196019
8. Ebrahimi, M., Sayed, N., Rashvand, M., and Tahoori, M.B.: Fault injection acceleration by architectural importance sampling. In: Hardware/Software Codesign and System Synthesis (CODES+ISSS) (2015). DOI: 10.1109/CODESISSS.2015.7331384
9. Gunneflo, U., Karlsson, J., and Torin, J.: Evaluation of Error Detection Schemes Using Fault Injection by Heavy-ion Radiation. In: 19th Intl. Symp. on Fault-Tolerant Computing (FTCS-19) (1989). DOI: 10.1109/FTCS.1989.105590
10. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., and Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: Fourth Annual IEEE Intl. Workshop on Workload Characterization. WWC-4 (2001). DOI: 10.1109/WWC.2001.990739
11. Guthoff, J., and Sieh, V.: Combining software-implemented and simulation-based fault injection into a single fault injection method. In: 25rd Intl. Symp. on Fault-Tolerant Computing (FTCS-25) (1995). DOI: 10.1109/FTCS.1995.466978
12. Hari, S.K.S., Adve, S.V., Naeimi, H., and Ramachandran, P.: Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In: ACM SIGPLAN Notices (2012). DOI: 10.1145/2189750.2150990

13. Hari, S.K.S., Venkatagiri, R., Adve, S.V., and Naeimi, H.: GangES: Gang error simulation for hardware resiliency evaluation. In: ACM/IEEE 41st Intl. Symp. on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014 (2014). DOI: 10.1109/ISCA.2014.6853212
14. IEC: IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems (1998)
15. ISO 26262-9: ISO 26262-9:2011: Road vehicles – Functional safety – Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses, Geneva, Switzerland (2011)
16. Ibing, A., Kirsch, J., and Panny, L.: Autocorrelation-Based Detection of Infinite Loops at Runtime. In: IEEE Int. Conf. Dependable, Autonomic and Secure Computing (2016). DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.78
17. Kaliorakis, M., Tselonis, S., Chatzidimitriou, A., Foutris, N., and Gizopoulos, D.: Differential Fault Injection on Microarchitectural Simulators. In: 2015 IEEE Intl. Symp. on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015 (2015). DOI: 10.1109/IISWC.2015.28
18. King, S.T., Dunlap, G.W., and Chen, P.M.: Debugging Operating Systems with Time-Traveling Virtual Machines (Awarded General Track Best Paper Award!) In: 2005 USENIX Annual Technical Conference (2005)
19. Lawton, K.P.: Bochs: A Portable PC Emulator for Unix/X. Linux Journal (1996)
20. Li, J., and Tan, Q.: SmartInjector: Exploiting Intelligent Fault Injection for SDC Rate Analysis. In: Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT '13) (2013). DOI: 10.1109/DFT.2013.6653612
21. Li, X., Huang, M.C., Shen, K., and Chu, L.: A Realistic Evaluation of Memory Hardware Errors and Software System Susceptibility. In: 2010 USENIX Annual Technical Conference (2010)
22. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., and Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. Acm sigplan notices 40(6) (2005)
23. Mansour, W., and Velazco, R.: SEU fault-injection in VHDL-based processors: A case study. In: 13th Latin American Test Workshop, (LATW' 12) (2012). DOI: 10.1109/LATW.2012.6261258
24. Schirmeier, H., and Breddemann, M.: Quantitative Cross-Layer Evaluation of Transient-Fault Injection Techniques for Algorithm Comparison. In: 15th European Dependable Computing Conference, EDCC (2019). DOI: 10.1109/EDCC.2019.00016
25. Schirmeier, H., Hoffmann, M., Dietrich, C., Lenz, M., Lohmann, D., and Spinczyk, O.: FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance. In: Sens, P. (ed.) 11th European Dependable Computing Conference (EDCC '15) (2015). DOI: 10.1109/EDCC.2015.28
26. Schirmeier, H., Rademacher, L., and Spinczyk, O.: Smart-Hopping: Highly Efficient ISA-Level Fault Injection on Real Hardware. In: 19th IEEE European Test Symp. (ETS '14), Paderborn, Germany (2014)
27. Smith, D.T., Johnson, B.W., Profeta, J.A., and Bozzolo, D.G.: A method to determine equivalent fault classes for permanent and transient faults. In: Reliability and Maintainability Symp. (1995). DOI: 10.1109/RAMS.1995.513278
28. Sridharan, V., Stearley, J., DeBardeleben, N., Blanchard, S., and Gurumurthi, S.: Feng Shui of Supercomputer Memory: Positional Effects in DRAM and SRAM Faults. In: High Performance Computing, Networking, Storage and Analysis, Denver, Colorado (2013). DOI: 10.1145/2503210.2503257

29. Ziade, H., Ayoubi, R.A., and Velazco, R.: A Survey on Fault Injection Techniques. The Intl. Arab Journal of Information Technology 1(2) (2004)