MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment

Manolis Kaliorakis Dimitris Gizopoulos

Department of Informatics & Telecommunications
University of Athens, Greece
{manoliskal, dgizop}@di.uoa.gr

Ramon Canal Antonio Gonzalez

Computer Architecture Department
Universitat Politecnica de Catalunya, Spain
{rcanal, antonio}@ac.upc.edu

ABSTRACT

Early reliability assessment of hardware structures using microarchitecture level simulators can effectively guide major error protection decisions in microprocessor design. Statistical fault injection on microarchitectural structures modeled in performance simulators is an accurate method to measure their Architectural Vulnerability Factor (AVF) but requires excessively long campaigns to obtain high statistical significance.

We propose MeRLiN[1], a methodology to boost microarchitecture level injection-based reliability assessment by several orders of magnitude and keep the accuracy of the assessment unaffected even for large injection campaigns with very high statistical significance. The core of MeRLiN is the grouping of faults of an initial list in equivalent classes. All faults in the same group target equivalent vulnerable intervals of program execution ending up to the same static instruction that reads the faulty entries. Faults in the same group occur in different times and entries of a structure and it is extremely likely that they all have the same effect in program execution; thus, fault injection is performed only on a few representatives from each group.

We evaluate MeRLiN for different sizes of the physical register file, the store queue and the first level data cache of a contemporary microarchitecture running MiBench and SPEC CPU2006 benchmarks. For all our experiments, MeRLiN is from 2 to 3 orders of magnitude faster than an extremely high statistical significant injection campaign, reporting the same reliability measurements with negligible loss of accuracy. Finally, we theoretically analyze MeRLiN's statistical behavior to further justify its accuracy.

CCS CONCEPTS

• Computer systems organization → Reliability

ISCA '17, June 24-28, 2017, Toronto, ON, Canada
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4892-8/17/06...$15.00
http://dx.doi.org/10.1145/3079856.3080225

KEYWORDS

Microarchitecture level reliability estimation, architectural vulnerability factor, fault injection, transient faults

ACM Reference format:

M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez. 2017. MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages. https://dx.doi.org/10.1145/3079856.3080225

1. INTRODUCTION

Continuous miniaturization of transistors allows computer architects to build more complex and efficient circuits in terms of functionality and performance. However, these chips become more and more susceptible to *transient*, *intermittent* and *permanent* faults due to external factors (such as particle strikes), manufacturing defects or wear-out phenomena [1, 2, 3, 4, 5].

Unavoidably, designers devote significant resources (effort, budget, circuit area) to ensure sufficient reliability levels of the computing system before it is released to market. Design decisions for detection, diagnosis, recovery and repair of faults are always translated to performance, area and power overheads. If such design decisions are guided by inaccurate reliability assessments, they can lead to unnecessary and excessive costs for error protection [6]. Early but also accurate reliability assessment is vital for optimal selection among the available protection mechanisms.

The four more popular approaches to estimate the reliability of hardware components are: *RTL injection* [7, 8, 9], *microarchitecture level injection* [10, 11, 12, 13, 14], *ACE (Architecturally Correct Execution) analysis* [15, 16, 17, 18] and *probabilistic models* [19, 20, 21, 22].

RTL injection allows very accurate studies of the fault effects in all hardware structures but these studies are performed too late in the design cycle to facilitate effective decision-making for error protection. Moreover, RTL injection requires excessively long simulation time which prevents detailed reliability evaluation of components with

[1] **MeRLiN** = **M**icroarchitectural **e**valuation of **R**eliability using statistica**L** fault **iN**jection.

statistically significant number of injections and large workloads. *Microarchitecture level injection*, on the other hand, is less detailed than RTL injection and is used for accurate full-system studies of fault effects in early design stages; it is orders of magnitude faster than RTL injection.

ACE analysis and *probabilistic models* are significantly faster than the two injection methods because they require a single or few fault-free runs to report reliability estimations. They provide a very useful but conservative lower bound of the reliability (upper bound of the vulnerability) of hardware components [9, 23, 24]. In particular, [23] reports 7X and [9] reports 3X AVF over-estimation of ACE analysis compared to fault injection. For example, [25] reports about 30% AVF for the physical integer register file of the out-of-order Alpha 21264 microprocessor with 80 registers using ACE analysis[2]; however, our comprehensive injection campaign of 60,000 transient faults[3] targeting the same structure for the same benchmarks on the out-of-order x86-64 microarchitecture in Gem5 measures only 2.56%, 4.81%, and 8.92% AVF for 256, 128 and 64 registers respectively.[4] Moreover, ACE analysis is not suitable to evaluate fault tolerant mechanisms that are based on soft error symptoms, in contrast to microarchitecture level injection [9, 27]. Despite of its disadvantages, ACE analysis merit in early reliability assessments is indisputable because it gives the opportunity to estimate the upper bound of vulnerability for different design options (component sizes, policies, etc.) in very short time.

Figure 1 reflects the motivation of our MeRLiN methodology compared to the four aforementioned state-of-the-art methods in terms of speed and measurement accuracy. An ideal method at the top-right corner of the figure would provide the highest speed (equal to that of the ACE analysis and probabilistic models) and the highest accuracy (equal to that of the injection methods with high statistical significance). MeRLiN approaches the ideal method boosting microarchitecture level injection-based reliability assessment while keeping its measurement accuracy unaffected. The backbone of MeRLiN is built on two major observations:

- A large number of faults in a statistical fault injection campaign are over-written before being read or are injected in dead or invalid entries of the hardware structure [14]. These faults can be easily identified and pruned from the initial fault list in a single run. We call this first part of our method *ACE-like*.
- The faults that are injected in the same or different entries of a structure during the same or different vulnerable intervals are *very likely* to have the same effect on program execution if these intervals end up to the same static instruction and the same micro-operation (uop) that reads the faulty entry. MeRLiN groups these faults together and performs fault injection on a small number of representatives. While it preserves the accuracy of the reliability measurements, this grouping drastically reduces the number of required injections because instruction repetition is an extensively inherent property of all programs [28, 29, 30, 31].

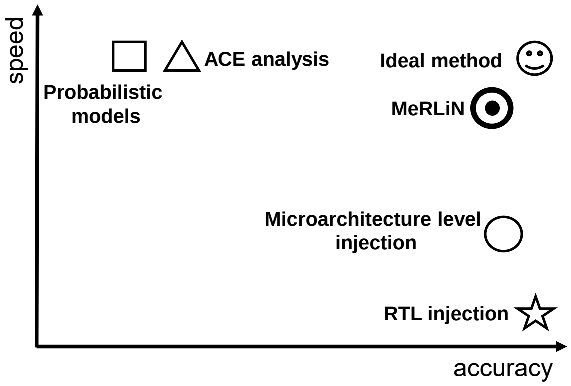

Figure 1: Reliability estimation methods: speed and accuracy.

Microarchitecture level, full-system simulators have been used for early assessment of the soft error vulnerability of hardware structures (register files, buffers, queues, caches etc.) that occupy the majority of the chip's area [10, 13, 32, 33, 34]. We implement and evaluate MeRLiN on a state-of-the-art microarchitecture level fault injector [12] [13] built on Gem5 [35]. MeRLiN's contributions are:

- It accelerates statistical microarchitecture level fault injection from 2 to 3 orders of magnitude. Our experiments with full runs of 10 MiBench benchmarks show 93X, 225X and 68X speedup on average for different sizes of the register file, the store queue and the first level data cache, respectively. When applied to 10 SPEC CPU2006 benchmarks, MeRLiN reveals larger average speedups of 1644X, 2018X and 171X for the register file, the store queue and the first level data cache, respectively.
- It reports virtually the same reliability estimations as conventional microarchitectural fault injection with extremely high statistical significance.
- It delivers fine-grained insights of the fault effects (Silent Data Corruptions-SDC, Detected Unrecoverable Errors-DUE, crashes, locks) unlike ACE analysis which only reports a gross AVF estimate. This can be used to evaluate different protection schemes or to identify benchmarks more prone to SDCs [27, 36].

2. RELATED WORK

Lifetime analysis has been previously used in several reliability-related studies. The method of [37] uses execution intervals sampling for reliability evaluation of caches. In

[2] Our ACE-like analysis corroborates this and reports about 25% AVF for a register file of 80 registers for the same benchmarks.

[3] This population of faults corresponds to an extremely low error margin (0.63%) and an extremely high confidence level (99.8%); see [26].

[4] For more details see Section 4. For 80 registers the injection-based AVF measurement is about 6%.

[38, 39], the authors separate the Hardware Vulnerability Factor (HVF) from the Program Vulnerability Factor (PVF), while [40] focuses on on-line vulnerability estimation and [25] aims to develop stressmarks to measure the maximum vulnerability of hardware structures to soft errors. The methods in [41, 42, 43, 44] use lifetime analysis to support decision-making for error protection.

Relyzer [45] aims to evaluate the effectiveness of software symptom-based error detection techniques and to identify all the SDCs [46, 47]. Relyzer injects faults only at the software level (architectural registers and output of load/store address generation units), without considering microarchitecture level masking and its features (flushes, store forwarding, dead instructions, cache write backs etc.) which our method fully supports. Relyzer comprehensively measures the application resiliency or, equivalently, reports the PVF portion [38] of the AVF. On the other hand, MeRLiN considers *both* the microarchitecture *and* the software masking and injects faults in the actual bits of *any* hardware structure at *any* cycle of the program execution; thus it reports the complete AVF including both the HVF and the PVF dimensions. Unlike Relyzer, MeRLiN:

- Reports the vulnerability of all microarchitectural structures modeled in performance simulators (physical register file, ROB, LSQ, predictors, caches, TLBs, etc.) and the vulnerability of the entire CPU. Relyzer focuses only on software resilience to faults.
- Reports the vulnerability of instruction related structures (L1 Instruction cache, fetch queue, trace cache, etc.). Relyzer only studies faults that reach data fields of the software.
- Can be used in early design stages to guide reliability design decisions concerning several microarchitectural features (components sizes, policies, etc.) or the use of several hardware and software protection mechanisms; Relyzer is limited to software symptom-based detectors.

GangES [48] is a follow-up study of [45] that accelerates injections at the software layer monitoring the intermediate execution state of each run. Finally, [12] is orthogonal to MeRLiN and can be combined with it, as it accelerates the individual microarchitectural injection runs at runtime without pruning the initial fault list.

3. MeRLiN METHODOLOGY

Our methodology consists of three phases: *Preprocessing*, *Fault List Reduction* and *Fault Injection Campaign* as shown in Figure 2. We describe the three phases in the following subsections.

3.1 Preprocessing

This first phase includes two tasks. First, MeRLiN records all vulnerable intervals of all entries of a hardware structure during the entire benchmark execution; this is the *ACE-like* analysis step. Then, MeRLiN creates the *initial fault list* repository that consists of a large number of faults for a statistically significant sampling: very low error margin and very high confidence level [26].

3.1.1 ACE-like analysis

During this first task, the benchmark runs once to completion to profile the vulnerable intervals (in which a bit flip may lead to corruption) of each entry of the target hardware structure (e.g. the registers in a physical register file). For our analysis, a *vulnerable interval* of an entry:

- Starts with a write operation and ends with a committed read of the same entry;
- Starts with a committed read and ends with another committed read of the same entry.

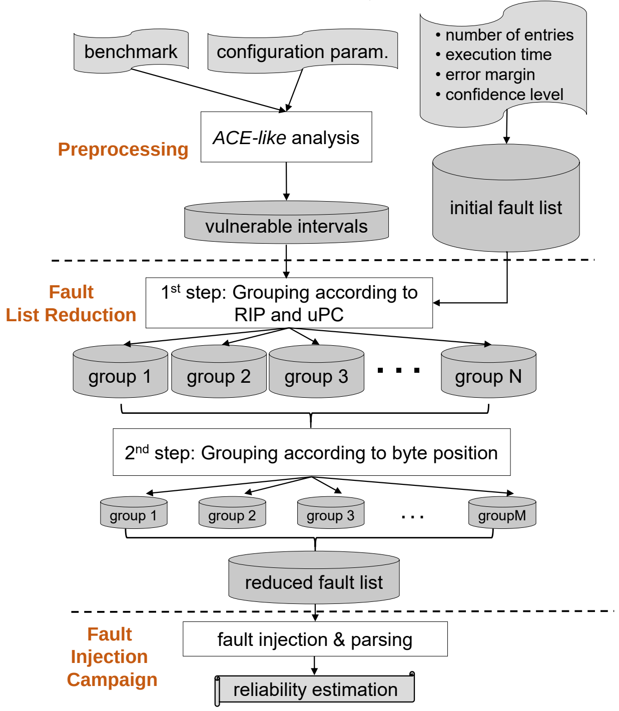

Figure 2: Flowchart of MeRLiN.

This definition differs from the typical definition of ACE intervals [15, 17] (where intermediate reads do not define the end of an interval) but the overall vulnerable time (sum of vulnerable intervals) is the same. Note that, similar to the original ACE analysis wrong-path execution instructions are not considered as part of the vulnerable intervals of MeRLiN. We highlight this difference between the two methods by an example in Figure 3, which represents the lifetime of an entry during the execution of a benchmark. The arrows directed upwards and downwards represent read and write operations, respectively. The read operations at t2, t5 and t6 are finally squashed. MeRLiN divides the interval between t7 and t9 in two individual vulnerable intervals, while ACE analysis considers them as a single interval.

This difference between MeRLiN's first step and classic ACE analysis is very important for the second phase of

MeRLiN, where the faults are grouped with respect to the instruction pointer (RIP) and the micro program counter (uPC) of the committed read that accesses the entry at the end of the vulnerable interval. Our analysis requires both the RIP and the uPC to cover cases where an x86-64 instruction consists of different micro-instructions that access the same or different entries of the hardware structure in the same or different cycles. These accesses can lead to different fault effects and are classified separately.

Our *ACE-like* analysis is significantly lighter in terms of storage overhead (10-100MB in our experiments) and more easily implemented than the complete ACE, because it does not trace the transitively dynamically dead (TDD) instructions [15]. The execution time of the *ACE-like* single-run step was less than 5 hours for all our experiments.

At the end of this step, the following information is stored in the *vulnerable intervals* repository for every *ACE-like* vulnerable interval of each entry: (i) start and end of the interval (cycle numbers), (ii) the instruction pointer (RIP) of the static x86-64 instruction that reads an entry at the end of the interval, and (iii) the micro program counter (uPC) of the micro-operation which is part of the x86 instruction and reads an entry at the end of the interval.

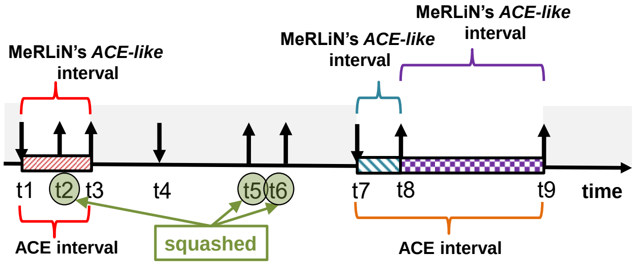
MeRLiN's ACE-like interval
MeRLiN's ACE-like interval
MeRLiN's ACE-like interval
t1
t2
t3
t4
t5
t6
t7
t8
t9
time
ACE interval
squashed
ACE interval

Figure 3: ACE and *ACE-like* intervals definition example.

3.1.2 Initial Fault List Creation

In the second task of the first phase, MeRLiN creates the *initial fault list* repository according to the statistical sampling described in [26]. The initial faults population is defined by: (1) the size (in bits) of the hardware structure, (2) the total execution time (in cycles) of the benchmark, (3) the statistical confidence level and (4) the statistical error margin. To achieve high statistical significance, the initial fault list should consist of tens or hundreds of thousands of faults. For instance, an injection campaign targeting a 256-entry integer register file of 64-bit registers with error margin 2.88%, confidence level 99% and 100M cycles of program execution time, requires 2000 fault injection runs [26]. If a higher statistical significance is needed (i.e. 0.63% error margin and 99.8% confidence level), the total number of injection runs explodes to 60,000 (an unacceptably large number of injections even for relatively short benchmarks). We use this number of 60K faults to define the baseline injection campaign for each single component, size and benchmark configuration, ensuring the same or even slightly higher statistical significance for all our structures. According to [26], for estimations of high statistical significance the confidence level and the error margin dominate in the calculation of the initial fault list population.

The outputs of the first phase of MeRLiN are the *vulnerable intervals* repository and the *initial fault list* that feed MeRLiN's second phase.

3.2 Fault List Reduction

This phase of MeRLiN classifies the faults in groups running a two-step grouping algorithm, and creates the *reduced fault list* that is used for the actual injections.

3.2.1 1st step of group creation algorithm

During the execution of the first step of the algorithm, all faults of the *initial fault list* are examined. All faults that target a non-vulnerable interval are directly classified as *Masked* as no injection is needed for them. The remaining faults that hit *ACE-like* vulnerable intervals are stored in different subdirectories (see Figure 2) according to the RIP and the uPC of the instruction that reads the entry at the end of the interval. Each of the created groups consists of transient faults on the same or different entries of the hardware structure being analyzed, during the same or different *ACE-like* vulnerable intervals that are read by an instruction with the *same* RIP and the *same* uPC.

Figure 4 shows an informative example of this first step for three entries of a hardware component during the execution of the same benchmark. When this step finishes, four groups are created containing faults that hit different hardware entries at different time intervals. The faults with the same color belong to the same group. The faults belonging to non-vulnerable intervals (gray color) are characterized as *Masked*. For instance, the faults in intervals t4-t6, t10-t13 and t7-t11 are grouped together (red color), because these intervals end up to micro-instructions with the same *ripC* and *uPC3*.

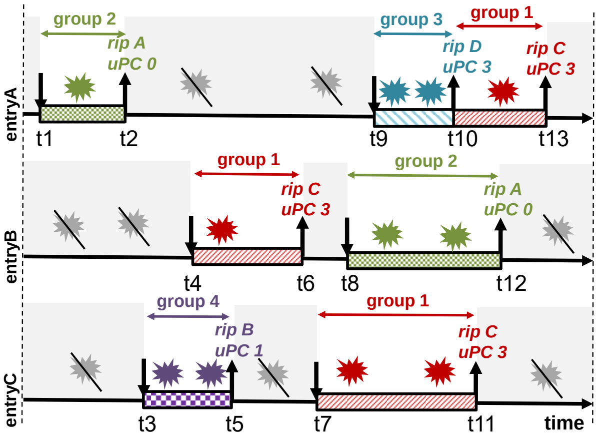

Figure 4: 1st step example of the grouping algorithm.

3.2.2 2nd step of group creation algorithm

Due to logical masking, all bits in a given faulty entry may not have the same effect when read by an instruction. To maximize MeRLiN's accuracy especially for groups with hundreds of faults, we select more than one fault for the actual fault injection runs in cases that faults hit a different byte of the entry. Moreover, faults in different bytes are selected from *different* dynamic instances of the same static instruction to increase time diversity. This can be further extended to separate faults hitting different *nibbles* or *bits*, but our experiments verify that this is not necessary.

MeRLiN ensures that for static instructions that are correlated with large population of faults, several representatives are selected from different dynamic instances of the same instruction, covering all possible byte positions of different entries. This per byte selection leads to smaller final groups ensuring the statistical significance of MeRLiN (see the theoretical analysis in Section 4.4.5), while it leads to groups of faults that are *extremely likely* to have the same effect. Figure 5 shows an example of the second step of the algorithm for three different hardware entries (*K*, *L*, *M*) during the execution of a benchmark. Note that all these faults were classified in the same group (same rip=*F* and uPC=4) from the first step of the grouping algorithm. The number next to each fault corresponds to the group in which the fault is finally classified at the end of the second step; the faults in circles are stored in the *reduced fault list* repository and are the *only* ones that will be injected. The execution time of the entire MeRLiN's single-run *group creation algorithm* was less than 50 minutes for all our experiments.

At the end of this phase, the *reduced fault list* repository contains all the selected faults. Only these faults are injected using the microarchitecture level fault injector.

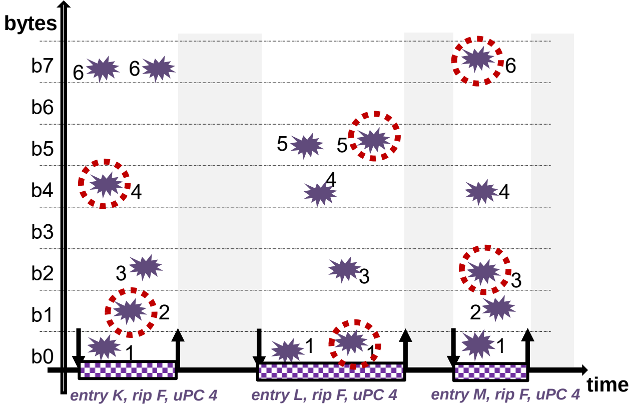

Figure 5: 2nd step example of the grouping algorithm.

3.3 Fault Injection Campaign

In the last phase of MeRLiN, the *fault injection* campaign is launched using all faults of the *reduced fault list* repository. During the *parsing* step, the outputs of all the injection runs per reduced group are compared to that of the golden run to identify the fault effect and calculate the final *reliability estimation* of the structure.

4. MeRLiN EVALUATION

4.1 Microarchitecture level fault injector – GeFIN

We employ GeFIN [13] a Gem5-based [35] microarchitectural injector and extend it to implement and evaluate MeRLiN on three structures of an x86-64 out-of-order processor:

- The physical integer Register File (RF) for three sizes: 256, 128, 64 registers.
- The data field of the Store Queue (SQ) of the Load/Store Queue for three sizes: 64 load and 64 store, 32 load and 32 store, and 16 load and 16 store entries. Gem5 doesn't implement data fields in the Load Queue.
- The data field of L1 data cache (L1D) for three sizes: 64KB, 32KB and 16KB.

MeRLiN can be used for: (i) all hardware structures of the CPU (caches, buffers, queues, registers, etc.), (ii) different input sets and benchmarks, (iii) different architectures and ISAs.

4.1.1 Configuration

Table 1 shows the baseline microprocessor configuration of our experiments. For all the experiments, we used machines with Intel Core i7-4771 at 3.5GHz, 16GBytes of RAM at 1600MHz and 1TByte hard disk.

Table 1: Baseline microprocessor configuration.

Parameter	x86 microprocessor model configuration
Pipeline	OoO
Physical register file	**256/128/64 int;** 192 FP
Issue Queue entries	32
Load/Store Queue	**64/32/16 load & 64/32/16 store entries**
ROB entries	100
Functional units	6 int ALUs; 2 complex int ALUs; 4 FP ALUs, 2 FP mul/div, 4 SIMD
L1 Instruction Cache	32KB,64B line,128 sets,4-way,write back
L1 Data Cache	**16KB/32KB/64KB, 64B line,64/128/256 sets,4-ways,write back**
L2 Cache	1MB,64B line,1024 sets,16-way,write back
Branch Predictor	Tournament predictor
Branch Target Buffer	conditional and unconditional branches BTB (direct-mapped, 4K entries)

4.1.2 Fault effect classification

For each injection run, we classify the fault effect in one of the six categories shown in Table 2.

Table 2: Fault effect classification.

Category	Effect
Masked	Output and x86 exceptions were identical to the golden run
SDC	The output is corrupted, but there was no abnormal behavior of the simulation process or the x86 exceptions
DUE	Simulation process and output are not corrupted, but there were indications of extra x86 exceptions
Timeout	Includes program flow *Deadlocks* (not committing further instructions) and *Livelocks* (redirected but continuing to commit instructions) that exceed execution time of benchmarks by three times
Crash	Includes *process* (abnormal termination of simulated program), *system* (full-system is unable to recover) and *simulator* (simulator process terminated abnormally) crashes
Assert	Simulator stopped due to *assert* instruction

4.2 Fault Sampling

An *exhaustive* fault list at the microarchitecture level consists of all flips for every bit of a hardware structure and for every program execution cycle. At the software the same list consists of bit flips in the operands of the assembly instructions; these faults are not correlated to the execution time of the program and the actual bits of the hardware.

Table 3 presents a high-level quantitative comparison of Relyzer [45] and MeRLiN using as starting point the exhaustive fault list of the corresponding level of abstraction (first column). The second column shows the faults of the exhaustive list that remain for injection after the application of each method, and the third column presents the *gains* (speedup) in terms of fault list reduction achieved by each method over the corresponding exhaustive list. The last two columns show the time needed to inject the *exhaustive* list and the remaining faults in both methods, respectively. Assume that we run one benchmark of 1 billion cycles and we inject faults in the L1D (32KB), the SQ (16 entries) and the RF (64 registers). The throughput of Gem5 for full-system cycle-accurate simulation is 10^5 cycles/sec while for software emulation it is 10^6 cycles/sec [35]. MeRLiN delivers 5 orders of magnitude higher gains than Relyzer having as starting point the exhaustive list, while it reports the reliability of the exhaustive list 10 orders of magnitude faster.

Table 3: MeRLiN vs. Relyzer using exhaustive fault list.

	Exhaustive fault list	Remaining faults	Gain	Evaluation time using exhaustive fault list	Evaluation time using remaining faults
MeRLiN	10^{13}	10^3	10^{10}	~3×10^9 years	4 months
Relyzer [45]	10^{11}	10^6	10^5	~3×10^6 years	32 years

Statistical fault sampling is unavoidable due to the huge number of faults in the exhaustive fault list. Thus, the *initial fault list* for each campaign of this paper was generated using statistical fault sampling [26] (Section 3.1.2) and consists of 60,000 faults (99.8% confidence level and 0.63% error margin). To study the scalability of MeRLiN (Section 4.4.2.4), we used an initial fault list of 600,000 faults (99.8% confidence level and 0.19% error margin).

4.3 Benchmarks

We employ 10 benchmarks from the MiBench suite [49] and 10 from the SPEC CPU2006 suite. We ran the MiBench benchmarks to the end to evaluate both MeRLiN's accuracy and speedup. Their execution time ranges from 1 to 55 million cycles, while they are very similar in instruction mixes and throughput with SPECs. Thus, they have extensively been used in many reliability studies [13, 14, 23, 25, 50]. In the case of SPEC benchmarks, we evaluate MeRLiN running Simpoint samples of 100M committed instructions with the largest weight [51]. MeRLiN's purpose is not to propose new benchmark intervals sampling approach for reliability evaluation, but any existing approach can be used (e.g. [37] for large caches or Simpoints that were used in many reliability studies [15, 17, 19]).

We selected to evaluate MeRLiN's accuracy executing MiBench benchmarks till the end instead of running entire SPEC benchmarks, because the execution time of each baseline comprehensive injection campaign (60,000 faults for each entire SPEC program, component and configuration) would make the evaluation infeasible. Also, we evaluated the accuracy of MeRLiN at the end of the Simpoint intervals of two selected SPEC CPU2006 benchmarks (bzip2 and gcc) (Section 4.4.3.4).

4.4 Results and Analysis

We evaluate MeRLiN in terms of reliability estimation accuracy and speedup against the comprehensive campaign and the *ACE-like* analysis. Then, we discuss Relyzer's heuristics if employed in MeRLiN's concept. Finally, we analyze the statistical properties of MeRLiN.

4.4.1 Homogeneity of fault effects

First, to measure the effectiveness of our grouping algorithm we define the *homogeneity* metric. In equation (1), *N* is the number of the groups that MeRLiN generates and *#faults* is the number of faults of a group. The *dominant* class of a group is defined as the category among those of Table 2 that contains the largest number of faults in the group. Thus, *dominant_class%* is the percentage of faults of the group that are classified in the dominant class. When *dominant_class%* equals 100%, it means that all the faults in that group have the same fault effect. Finally, *#total_faults* is the total population of faults that hit vulnerable intervals. Large values of *homogeneity* close to 1.0, denote that the vast majority of faults across all groups lead to the same effect, and the accuracy of the algorithm is high.

$$homogeneity = \frac{\sum_{\text{group1}}^{\text{groupN}} \#\text{faults} \times \text{dominant\_class}\%}{\#\text{total\_faults} \times 100\%} \qquad (1)$$

Figure 6 shows the *homogeneity* for all our experiments running the 10 MiBench. On the average, the highest *homogeneity* for the RF is 0.940, for the SQ is 0.982 and for L1D is 0.920. In general, the homogeneity values are very high for this fine-grained classification (the 6 classes). If homogeneity is calculated in coarser granularity (masked vs. not-masked faults) and all classes that lead to non-masking are combined together, then homogeneity is even larger; see the values at the top of each bar in Figure 7. In Figure 7 the value at the bottom of each bar represents the percentage of groups (average for all our experiments with MiBench) that consist of faults with *exactly* the same effect (masked, non-masked) meaning that they have a perfect homogeneity value of 1.0. Finally, homogeneity climbs to 0.99 if we count the faults excluded by the ACE-like, but here we focus only on MeRLiN's grouping part. All these results indicate the extremely high accuracy of MeRLiN.

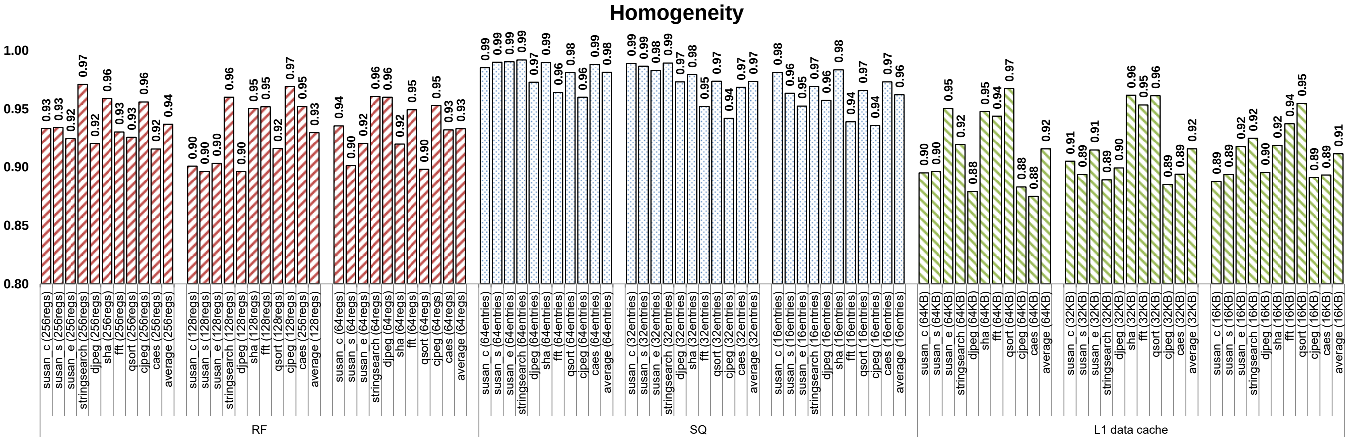

Figure 6: Fine-grained homogeneity of fault effects in the RF, SQ and L1D for 10 MiBench benchmarks; using 6 classes of Table 2.

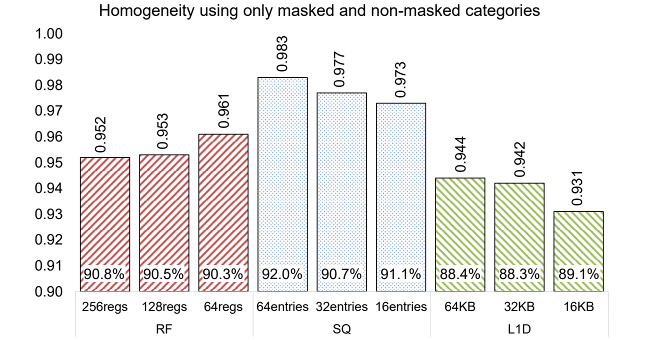

Figure 7: Coarse-grained homogeneity (top of bars) and percentage of groups with perfect homogeneity (equal to 1.0) (bottom of bars); average for 10 MiBench.

4.4.2 *Speedup*

We evaluate the speedup of MeRLiN against the comprehensive baseline fault injection campaigns (60,000 faults).

4.4.2.1 *MiBench benchmarks*

Figure 8 presents the speedup of the method for 256, 128 and 64 physical registers for the 10 MiBench benchmarks. The lower (blue) segment and the value on top of it indicate the speedup compared to the comprehensive baseline injection method (60,000 faults) after the first *ACE-like* pass. The higher (red) segment of each bar indicates the speedup achieved by the grouping algorithm on top of the first *ACE-like* step. The value on top of the red bar represents the final speedup achieved by MeRLiN. For example, for 64 registers and the *qsort* benchmark the *ACE-like* step reduces the initial fault list by 4.1X (60,000/14,757). The remaining 14,757 faults are further reduced by the grouping algorithm to 1126 faults that should be actually injected; this totally corresponds to 53.3X (60,000/1126) reduction of the initial fault list. The average speedups are 93.1X, 62.1X and 43.7X for 256, 128 and 64 registers, respectively. Similarly, Figure 9 and Figure 10 present the speedup for the store queue and the data cache, respectively. The average speedups for the store queue are 224.9X, 186.7X and 146.9X for 64, 32 and 16 entries respectively, while for the data cache they are 67.9X, 61.6X and 59.0X for 64KB, 32KB and 16KB respectively.

4.4.2.2 *Actual Estimation Time running MiBench*

Figure 11 depicts the actual time required for the fault injection campaigns in the three structures with the comprehensive fault injection method (60,000 faults per campaign; blue bars) and MeRLiN method (red bars) for all MiBench benchmarks and all component configurations. We assume that all injections run sequentially in the same machine.

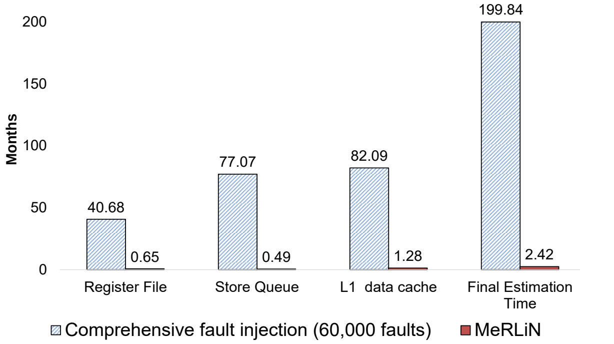

Figure 11: Actual reliability estimation times of the comprehensive baseline injection vs. MeRLiN for all structures configurations of this study running 10 MiBench benchmarks.

4.4.2.3 *SPEC CPU2006 benchmarks*

To evaluate the efficiency of MeRLiN in terms of speedup in larger benchmarks, we ran Simpoint samples of 100M committed instructions with the highest weight from 10 selected integer benchmarks of the SPEC CPU2006 suite assuming an initial fault list of 60,000 faults. We used the configuration of Table 1, with 128 physical integer registers, 16 store and 16 load queue entries and a 32KB L1 data cache. The results of the speedup that MeRLiN delivers are reported in Figure 12. MeRLiN leads to very high final speedups of 1644X, 2018X and 171X on average for the RF, the SQ and the L1D cache, respectively, which are higher than the speedups obtained for MiBench programs since the Simpoint samples we used for SPECs correspond to the most representative part of their execution.

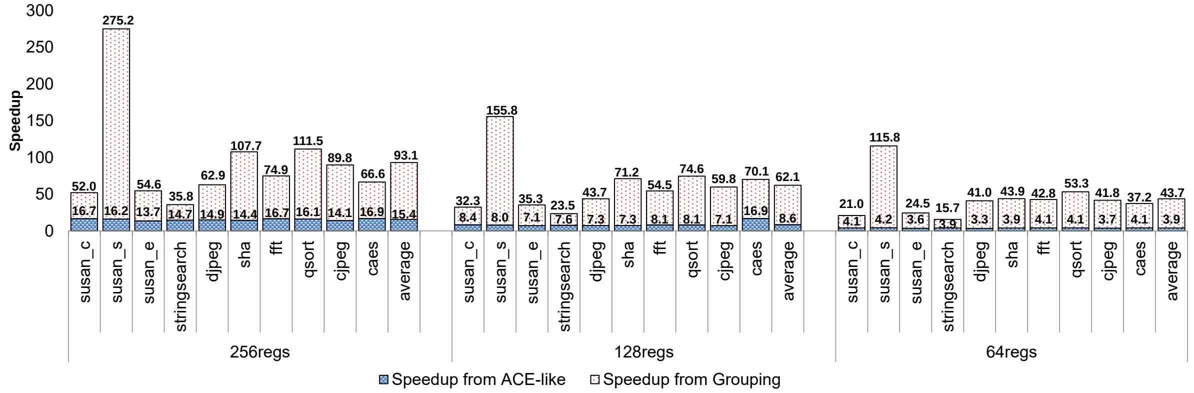

Figure 8: MeRLiN speedup for the three sizes of the Physical Integer Register File running 10 MiBench benchmarks.

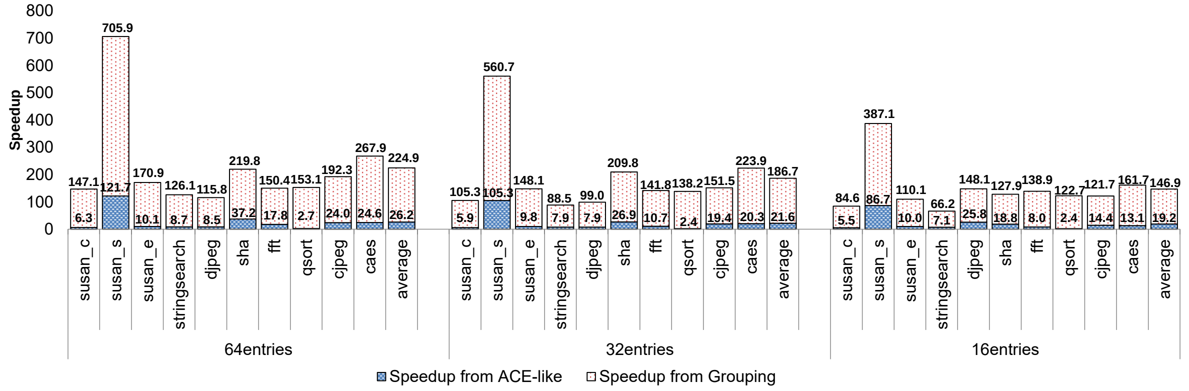

Figure 9: MeRLiN speedup for the three sizes of the Store Queue running 10 MiBench benchmarks.

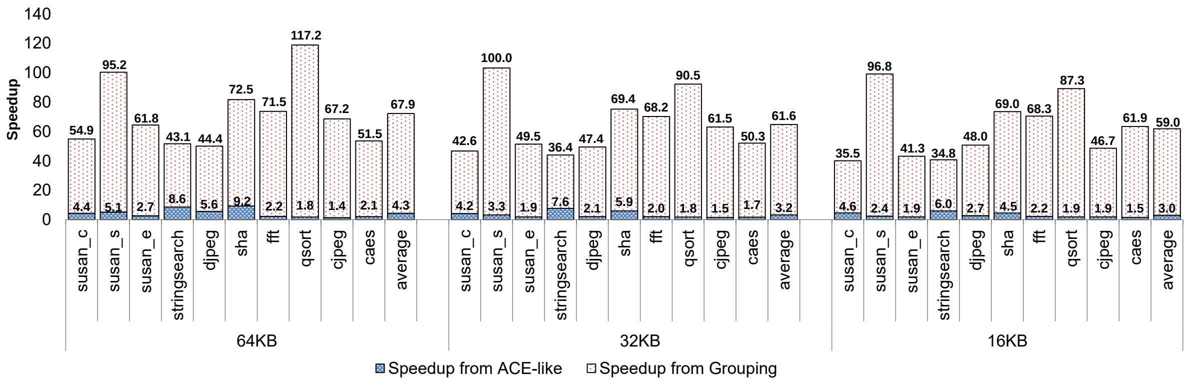

Figure 10: MeRLiN speedup for the three sizes of the L1 data cache running 10 MiBench benchmarks.

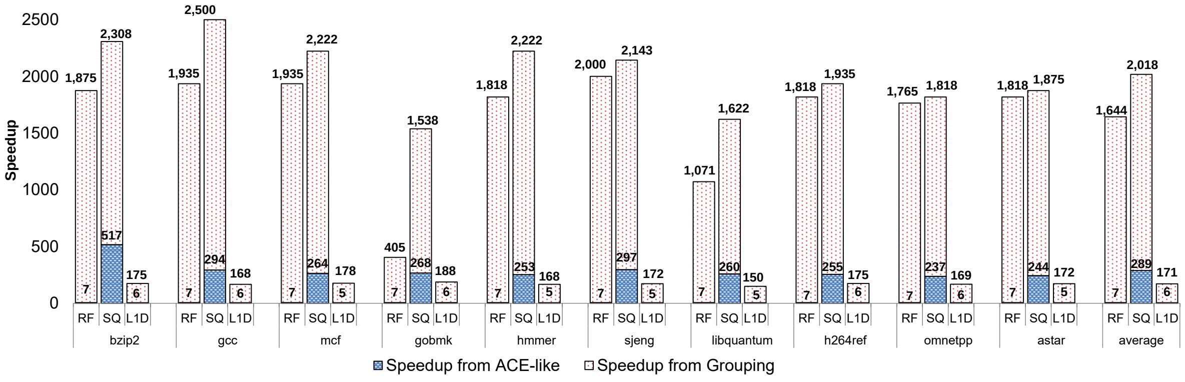

Figure 12: MeRLiN speedup for the Register File (RF), Store Queue (SQ), and L1 data cache (L1D) running 10 SPEC CPU2006.

4.4.2.4 Scaling of the MeRLiN method

The higher the statistical significance of the initial fault list the larger the speedup that MeRLiN offers. In our initial set of campaigns, we ran all MiBench benchmarks using 60,000 faults per campaign (99.8% confidence level and 0.63% error margin). To stress MeRLiN even further, we repeated all these campaigns using a huge 10 times larger initial list of 600,000 faults (99.8% confidence level and 0.19% error margin)[5]. Figure 13 presents the average speedup achieved for these two sets of campaigns by the ACE-like (lower purple segment of each bar) and the grouping step (upper white segment) of MeRLiN, as well as the final speedup achieved (value on top of each bar) for each configuration. The final speedup was scaled up 3.46 times on average; practically meaning that for a 10 times increase of the initial fault list, MeRLiN finally applies only 2.89 times more faults.

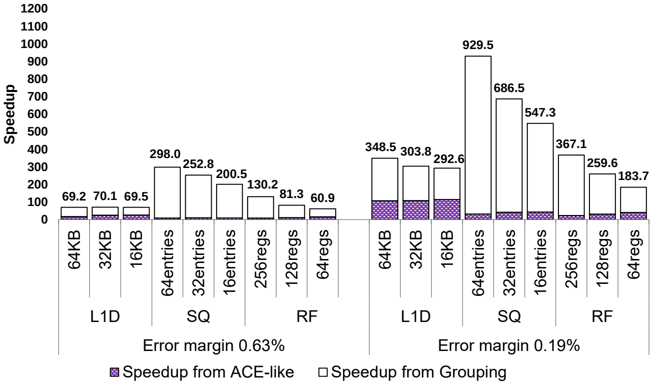

Figure 13: MeRLiN speedup scaling for 0.63% (60K faults) and 0.19% error margin (600K faults); 10 MiBench average.

4.4.3 Reliability Estimation Accuracy

We measure the accuracy of the reliability estimations of MeRLiN for the three components running 10 MiBench benchmarks till the end. We compare MeRLiN's accuracy against the injection in: (i) the remaining fault list after the exclusion of the faults that target non-vulnerable intervals (identified by the *ACE-like* step of the method), (ii) the comprehensive baseline fault list (60,000 faults). Finally, we evaluate MeRLiN's accuracy for the RF with 60K faults using Simpoints from the bzip2 and the gcc.

4.4.3.1 Accuracy in the remaining fault list after ACE-like

The estimation accuracy of MeRLiN for the three structures of this study against the injection using the remaining fault list *after* the *ACE-like* step is shown in Figure 14. Each graph shows the average fault effect classification across the 10 MiBench benchmarks used in our study for the three configurations of each structure. The first bar (blue) in each class corresponds to the results of the fault injection in the remaining fault list *after* the *ACE-like* analysis, while the second bar (red) illustrates the results on the same fault list after applying MeRLiN's grouping algorithm and injecting only the selected faults. The values on top of each bar represent the measurement per fault effect category. Similar behavior is observed across all benchmarks. For all component configurations, MeRLiN reports negligible differences compared to the injection using all the faults that hit only vulnerable intervals.

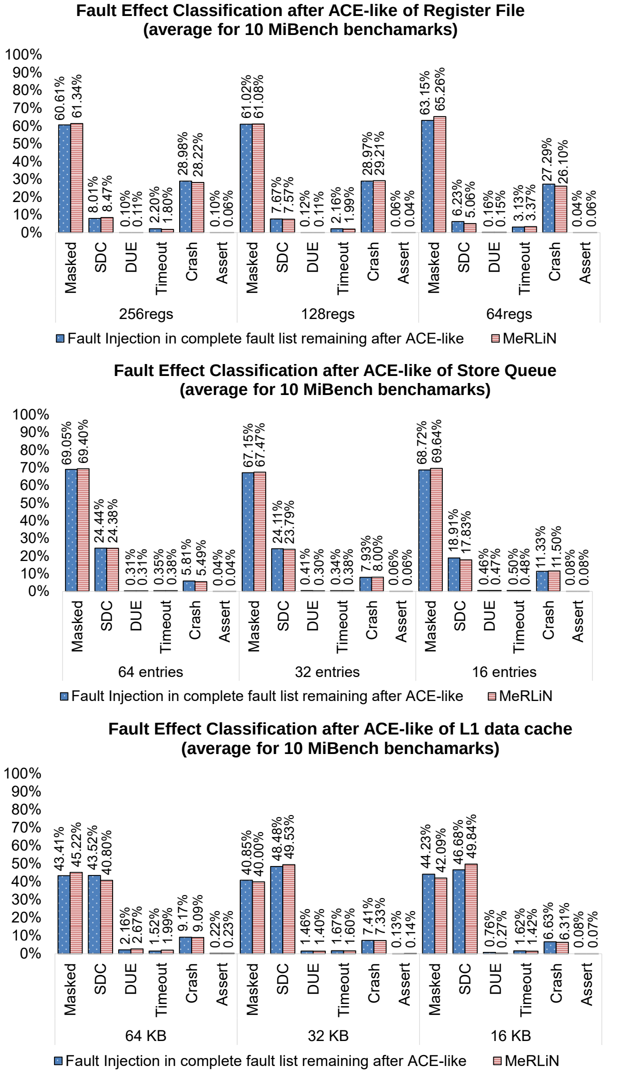

Figure 14: Classification of MeRLiN against injection with the remaining faults after *ACE-like* step for the RF, SQ, L1D.

4.4.3.2 Accuracy in the comprehensive list of 60K faults

Figure 15 shows the bigger picture for MeRLiN's accuracy, in which the final fault effect classification of the comprehensive baseline fault injection of 60,000 faults (blue bar) is compared to the final classification of MeRLiN (red bar). Each bar represents the average values across the 10 MiBench benchmarks. Similar behavior is observed across all benchmarks. MeRLiN for all cases is extremely accurate and delivers virtually the same reports with the comprehensive injection, but orders of magnitude faster.

[5] In all our experiments we round up the number of injections (60,000 and 600,000) instead of rounding the error margins.

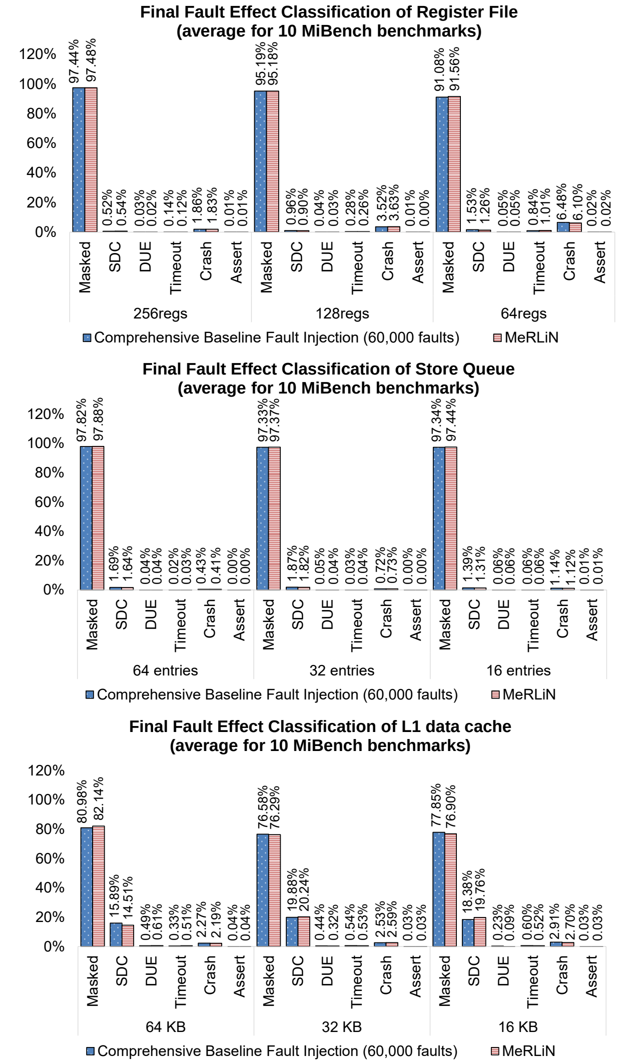

Figure 15: Final classification of MeRLiN against comprehensive baseline injection (60,000 faults) for the RF, SQ, L1D.

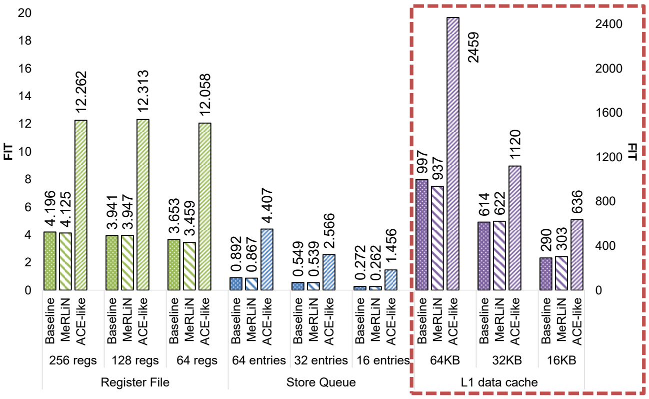

Figure 16: Final reliability assessment (FIT) for RF, SQ, and L1D (average for 10 MiBench benchmarks).

4.4.3.3 Final Reliability Assessment (FIT)

Figure 16 demonstrates the final reliability estimation in Failures-in-Time (FIT) rates for the comprehensive baseline campaign (60,000 faults), the MeRLiN method and the *ACE-like* method running the 10 MiBench benchmarks to the end. The reported FIT rates are the products of AVF, raw FIT rate and number of structure's bits. The AVF of the injection-based methods is the ratio of the non-masked injections over the total injections, while the AVF of the *ACE-like* is measured as in [15]. Any raw FIT rate can be used; we use 0.01 FIT per bit.

MeRLiN reports negligible differences compared to the comprehensive baseline injection, while the *ACE-like* delivers a pessimistic lower bound of structures' reliability.

4.4.3.4 Accuracy using SPEC CPU2006 benchmarks

The evaluation of MeRLiN's accuracy for SPEC CPU2006 benchmarks executed until the end in detailed microarchitectural simulation mode is infeasible as was discussed in Section 4.3. To overcome this difficulty and in order to evaluate the accuracy that MeRLiN provides for SPEC CPU2006 benchmarks, we applied MeRLiN injecting faults in the physical register file for the gcc and bzip2 benchmarks and terminating the fault injection runs at the end of the Simpoint interval. The configuration for these experiments is the one of Table 1, with 128 physical registers, 16 store and 16 load queue entries and a 32KB L1 data cache.

As we do not execute the fault injection runs to the end, we are not able to identify SDCs, timeouts or any other abnormal behavior after the end of the Simpoint interval. Thus, only for these experiments we used a different fault effect classification than the classification presented in Table 2. The classification consists of the following categories: (i) Masked; indicates a fault that was not over-written or hit a non-vulnerable interval without affecting program execution, (ii) DUE (as in Table 2), (iii) Crash (as in Table 2), (iv) Assert (as in Table 2), and (v) Unknown; indicates a fault that still exists but at the end of the Simpoint interval it is not known if it will eventually be classified in one of the previous classes or if it will lead to an abnormal behavior.

Table 4 summarizes our measurements per fault effect category using MeRLiN and the comprehensive baseline fault list of 60K faults for the two benchmarks. In both cases, MeRLiN delivers very accurate results per fault effect category compared to the comprehensive baseline method, while the maximum inaccuracy that was observed is only 1.11 percentile points for the Unknown category of the bzip2 benchmark.

Table 4: MeRLiN's accuracy for gcc and bzip2 benchmarks.

Category	gcc (MeRLiN)	gcc (baseline 60K faults)	bzip2 (MeRLiN)	bzip2 (baseline 60K faults)
Masked	85.08%	85.08%	84.98%	84.98%
DUE	0.06%	0.07%	0.29%	0.81%
Crash	3.67%	3.13%	3.50%	4.10%
Assert	0.01%	0.01%	0.03%	0.02%
Unknown	11.18%	11.71%	11.20%	10.09%

4.4.4 Analysis of Relyzer's heuristics

Both MeRLiN and Relyzer prune faults of the initial fault list being injected at different levels of the system stack. Thus, in this section we analyze the applicability of Relyzer heuristics at the microarchitecture level injection.

Bounding addresses: It prunes faults in the address field of store and load instructions if the valid address space is violated. This heuristic requires an unaffordable amount of memory to track the addresses in data related structures (e.g. caches). Also, MeRLiN provides finer grained effect classification for non-masking categories (Table 2) and is not limited to symptom-based techniques.

Def-use: It prunes faults in the destination architectural register of an instruction followed by another instruction that consumes this value, as these faults will have the same effect. **Store-equivalence** is similar to the def-use for store and load instructions. These two heuristics cannot be applied at the microarchitecture level of our work. The destination register of an instruction and the source register of a subsequent correspond to the same physical entity [47].

Control-equivalence: Software analysis using basic blocks tracks the control flow paths of all the dynamic instances of all the static instructions to separate Masked from SDC faults [46]. For each path Relyzer randomly chooses only one pilot. To evaluate this heuristic, we ran the 10 MiBench to the end with 128 registers, 16 SQ entries and 32KB L1D. Exhaustive fault injection is infeasible; thus, we used the remaining faults (from 60,000 initial faults) after the pruning by our ACE-like step. We used a control flow path depth of 5, exactly as Relyzer does [45].

In terms of speedup, MeRLiN slightly prevails on average in the RF (62.1X compared to 60.5X) and the L1D (60.1X compared to 59.1X), while for the SQ, MeRLiN provides 146.9X speedup compared to 150.6X of Relyzer's heuristic. Figure 17, illustrates the results of the comparison in terms of inaccuracy in percentile units compared to the injection using the same fault list.

A source of Relyzer's inaccuracy is the static instructions with large population of faults that are represented by only one randomly selected pilot. In [45], 52% on average of all static instructions have only 1 pilot. We measured that Relyzer leaves 9% of the groups correlated to a static instruction with large population of faults (more than 100 faults) with only 1 pilot, while MeRLiN leaves less than 2%. The heuristic of Relyzer if applied to our statistical concept selects only one pilot for code loops with large number of iterations. Assume a for-loop with 1000 iterations that consists of only one static instruction with only two control flow paths with 995 and 5 instances, respectively. Due to statistical sampling all faults may come only from the first path. In this case, Relyzer chooses only one pilot for this loop. On the contrary, MeRLiN, due to the homogenous distribution of faults, chooses more than one from different bytes and dynamic instances. These large loops exist in most program execution phases, including initialization and output phase that are not examined by [45]. Despite of Relyzer's indisputable merit in software resilience, this heuristic of Relyzer is not so efficient to be employed in our concept.

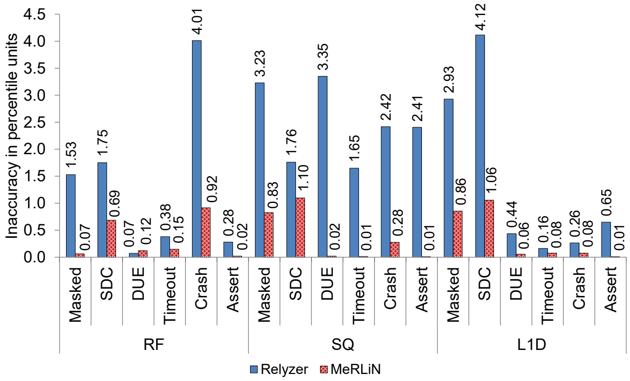

Figure 17: Inaccuracy of MeRLiN and Relyzer vs. injection with the remaining faults after *ACE-like*;avg. for 10 MiBench.

4.4.5 Theoretical analysis of MeRLiN

In this section we analyze the statistical behavior of MeRLiN comparing the mean and the variance of the AVF measurements it reports to the corresponding mean and variance of the comprehensive fault injection campaign. We assume that soft errors affecting the microprocessor bits follow a normal distribution [26]. A fault injection campaign can be described as a binomial experiment of F individual injections, each of which has a probability of *success* (program is affected) or *failure* (program is not affected; fault is masked). Thus, the AVF measurement k ($0 \leq k \leq 1$) in our case means that $k \cdot F$ faults are Not-Masked.

MeRLiN's first phase prunes a fraction m ($0 \leq m \leq 1$) of the F faults that are guaranteed masked: $m \cdot F$. The remaining $(1-m) \cdot F$ faults (which now contain all $k \cdot F$ Not-Masked faults of the initial list of F faults) are forwarded to the second phase of MeRLiN (grouping). This second phase produces n groups of faults with sizes s_i (i=1, 2, … , n). The sum of the group sizes is equal to the number of faults passed to the second phase: $s_1 + s_2 + \ldots + s_n = (1-m) \cdot F$.

When the comprehensive injection campaign (without MeRLiN) is applied, all F faults are injected and the outcome r of each run is observed (Not-Masked=1 or Masked=0). In this case, the AVF (k) is[6]:

$$k = \frac{\sum_{i=1}^{n} \sum_{j=1}^{s_i} r_i^j}{F}$$

We assume that the probability of Non-Masking within a group i is p_i. Within a group i, all faults have the same

[6] We could consider as group 0 with size $s_0 = m \cdot F$ the group of faults from MeRLiN's pre-processing step but since all faults of this group are masked, i.e. r=0, this group is not needed in the calculations.

probability p_i because of MeRLiN's grouping criterion: faults in a group hit the *same* byte of the entries during a vulnerable interval that ends with the *same* instruction that reads the entry. The results of Figure 7 show the validity of this assumption; they indicate that the vast majority of groups have homogeneity close to 1.0 (considering only the masked and non-masked categories) and that the percentage of groups with perfect homogeneity is very large in all cases. Across groups, probabilities p_i are different since the groups correspond to faults eventually read by different instructions. The mean (expected value; E) of the AVF measurement k in the comprehensive campaign is[7]:

$$E(k) = E\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} r_i^j}{F} \right) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} E(r_i^j)}{F} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} p_i}{F} = \frac{\sum_{i=1}^{n} s_i \cdot p_i}{F}$$

When MeRLiN is employed it delivers a new AVF measurement k_{MeRLiN}. For each run r of the selected fault from a group i all faults are assumed to have the same result (1=Not-Masked, 0=Masked). So, the true measurement in this case is $s_i r_i$ for each group i and the new AVF k_{MeRLiN} is:

$$k_{MeRLiN} = \frac{\sum_{i=1}^{n} s_i \cdot r_i}{F} \text{, which has a mean}$$

$$E(k_{MeRLiN}) = E\left(\frac{\sum_{i=1}^{n} s_i \cdot r_i}{F} \right) = \frac{\sum_{i=1}^{n} E(s_i \cdot r_i)}{F} = \frac{\sum_{i=1}^{n} s_i E(r_i)}{F} = \frac{\sum_{i=1}^{n} s_i \cdot p_i}{F} = E(k)$$

therefore, MeRLiN reports AVF with the same mean value as the original comprehensive set of F fault injections. The variance of the AVF measurements k and k_{MeRLiN} is shown in the following equations[8]:

$$\sigma^2(k) = \sigma^2\left(\frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} r_i^j}{F} \right) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} \sigma^2(r_i^j)}{F^2} = \frac{\sum_{i=1}^{n}\sum_{j=1}^{s_i} p_i \cdot (1-p_i)}{F^2} \Rightarrow$$

$$\Rightarrow \sigma^2(k) = \frac{\sum_{i=1}^{n} s_i \cdot p_i \cdot (1-p_i)}{F^2}$$

$$\sigma^2(k_{MeRLiN}) = \sigma^2\left(\frac{\sum_{i=1}^{n} s_i \cdot r_i}{F} \right) = \frac{\sum_{i=1}^{n} s_i^2 \cdot \sigma^2(r_i^j)}{F^2} = \frac{\sum_{i=1}^{n} s_i^2 p_i \cdot (1-p_i)}{F^2}$$

The values of both $\sigma^2(k)$ and $\sigma^2(k_{MeRLiN})$ are very small (several orders of magnitude smaller than the means of k and k_{MeRLiN}, respectively) for two reasons: (a) the groups generated by MeRLiN are very homogeneous (thus either p_i or $(1-p_i)$ is zero or is very small) as shown in Section 4.4.1 and (b) the sizes of the groups (s_i values) are very small compared to F. In our experiments, the average size of a MeRLiN group is always less than100 and typically ranges between 5 and 40. Thus, with simple calculations on the above equations the variance of the initial AVF value when F consists of 60K faults is about 8 to 10 orders of magnitude smaller than the mean. Therefore, the multiplication with the s_i values in the variance of MeRLiN's AVF measurements $\sigma^2(k_{MeRLiN})$ keeps this variance from 6 to 8 orders of magnitude smaller than the mean (assuming s_i values up to 100): still *a very small variance*, only slightly increased compared to the initial one.

Overall our analysis shows that the AVF measurement of MeRLiN has the same mean as the comprehensive experiment of F injections, while both have a very small variance. These two statistical properties make them almost statistically equivalent although MeRLiN reports AVF in 2 to 3 orders of magnitude shorter time.

5. CONCLUSIONS

We presented MeRLiN, a methodology to accelerate comprehensive, statistically significant microarchitecture level fault injection campaigns on hardware structures modeled in performance simulators. MeRLiN's effectiveness is based on the combination of the principle of dynamic instruction repetition and the identification of the non-vulnerable intervals for the entries of the hardware structures. We demonstrated its efficiency using microarchitecture level fault injection on a Gem5 model of a contemporary microprocessor. We reported results for the method's speedup, accuracy, and scaling for different sizes of the physical register file, store queue and first level data cache.

MeRLiN achieves several orders of magnitude speedup (reduction of the number of injections) while it virtually delivers the same reliability measurements compared to exhaustive (but computationally infeasible) fault injection campaigns. Our experimental results and theoretical analysis validate MeRLiN's accuracy.

ACKNOWLEDGMENT

This work has been funded by the European Union through the CLERECO FP7 Project (Grant Agreement 611404) and the UniServer H2020 Project (Grant Agreement 688540).

6. REFERENCES

[1] Robert Baumann. 2005. Soft errors in advanced computer systems. In IEEE Design & Test of Computers, vol. 22, no. 3, pp. 258-266, May-June. DOI:http://doi.org/10.1109/MDT.2005.69

[2] Zeshan Chishti, Alaa R.Alameldeen, Chris Wilkerson, Wei Wu, and Shih-Lien Lu. 2009. Improving cache lifetime reliability at ultra-low voltages. In *Proceedings of the IEEE/ACM*

[7] We use the linearity property of the means of independent variables which holds for binomial distribution. The mean of a binomially distributed variable is $E(X) = n\,p$ with n experiments and p success probability.

[8] We use the relation $\sigma^2(a \cdot X + b \cdot Y) = a^2 \cdot \sigma^2(X) + b^2 \cdot \sigma^2(Y)$ for the variances of independent variables. Group 0 has zero variance.

International Symposium on Microarchitecture (MICRO). DOI:http://dx.doi.org/10.1145/1669112.1669126

[3] Cristian Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. In *IEEE Micro*, vol. 23, pp. 14-19, July. DOI:http://dx.doi.org/10.1109/MM.2003.1225959

[4] Lin Huang, and Qiang Xu. 2010. AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs. In *Proceedings of Design, Automation and Test in Europe (DATE)*. ISBN:978-3-9810801-6-2

[5] Sani R.Nassif, Nikil Mehta, and Yu Cao. 2010. A resilience roadmap. In *Proceedings of Design, Automation and Test in Europe (DATE)*. ISBN:978-3-9810801-6-2

[6] Yixin Luo, Sriram Govindan, Bikash Sharma, Mark Santaniello, Justin Meza, Aman Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. 2014. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous reliability memory. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2014.50

[7] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A.Abraham, and Subbanish Mitra. 2013. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of ACM/EDAC/IEEE Design and Automation Conference (DAC)*. ISBN:978-1-4503-2071-9

[8] Michail Maniatakos, Naghmeh Karimi, Chandra Tirumurti, Abhijit Jas, and Yiorgos Makris. 2011. Instruction- level impact analysis of low-level faults in a modern microprocessor controller. In *IEEE Transactions on Computers*, vol. 60, no. 9, pp.1260-1273. DOI:http://dx.doi.org/ 10.1109/TC.2010.60

[9] Nicholas J.Wang, Aqeel Mahersi, and Sanjay J.Patel. 2007. Examining ACE analysis reliability estimates using fault-injection. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/10.1145/1250662.1250719

[10] Gulay Yalcin, Osman S.Unsal, Adrian Cristal, and Mateo Valero. 2011. FIMSIM: A fault injection infrastructure for microarchitectural simulators. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*. DOI:http://dx.doi.org/10.1109/ICCD.2011.6081435

[11] Nikos Foutris, Dimitris Gizopoulos, John Kalamatianos, and Vilas Sridharan. 2013. Assessing the impact of hard faults in performance components of modern microprocessors. In *Proceedings of IEEE International Conference on Computer Design (ICCD)*. DOI:http://dx.doi.org /10.1109/ICCD.2013.6657044

[12] Athanasios Chatzidimitriou, and Dimitris Gizopoulos. 2016. Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. DOI:http://dx.doi.org/10.1109/ISPASS.2016.7482075

[13] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. 2015. Differential fault injection on microarchitectural simulators. In *Proceedings of IEEE International Symposium on Workload Characterization (IISWC)*. DOI:http://dx.doi.org/10.1109/IISWC.2015.28

[14] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. 2015. Accelerated microarchitectural fault injection-based reliability assessment. In *Proceedings of IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. DOI:http://dx.doi.org/10.1109/DFT.2015.7315134

[15] Shubhendu S.Mukherjee, Christopher Weaver, Joel Emer, Steven K.Reinhardt, and Todd Austin. 2004. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessors. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ISBN:0-7695-2043-X

[16] Arun Nair, Stijn Eyerman, Lieven Eeckhout, and Lizy K.John. 2012. A first-order mechanistic model for architectural vulnerability factor. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/10.1145/2366231.2337191

[17] Arijit Biswas, Paul Racunas, Romulus Cheveresan, Joel Emer, Shubhendu S.Mukherjee, and Ram Rangan. 2005. Computing architectural vulnerability factors for address-based structures. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/10.1109/ISCA.2005.18

[18] Hossein Asadi, Vilas Sridharan, Mehdi Tahoori, and David Kaeli. 2005. Balancing performance and reliability in the memory hierarchy. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. DOI:http://dx.doi.org/ 10.1109/ISPASS.2005.1430581

[19] Xiaodong Li, Sarita V.Adve, Pradip Bose, and Jude A.Rivers. 2005. SoftArch: An architecture-level tool for modeling and analyzing soft errors. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2005.88

[20] Jinho Suh, Murali Annavaram, and Michel Dubois. 2012. MACAU: A markov model for reliability evaluations of caches under single-bit and multi-bit upsets. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. DOI:http://dx.doi.org/ 10.1109/HPCA.2012.6168940

[21] Jinho Suh, Mehrtash Manoochehri, Murali Annavaram, and Michel Dubois. 2011. Soft error benchmarking of L2 caches with PARMA. In *Proceedings of ACM SIGMETRICS*. DOI:http://dx.doi.org/10.1145/2007116.2007127

[22] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: probabilistic soft error reliability on the cheap. In *Proceedings of IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI:http://dx.doi.org/10.1145/1736020.1736063

[23] Nishant J.George, Carl R.Elks, Barry W.Johnson, and John Lach. 2010. Transient fault models and AVF estimation revisited. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2010.5544276

[24] Xiaodong Li, Sarita V.Adve, Pradip Bose, and Jude A.Rivers. 2007. Architecture-level soft error analysis: Examining the limits of common assumptions. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI: http://dx.doi.org/ 10.1109/DSN.2007.15

[25] Arun A.Nair, Lizy K.John, and Lieven Eeckhout. 2010. AVF Stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. DOI:http://dx.doi.org/ 10.1109/MICRO.2010.34

[26] Regis Leveugle, A.Calvez, Paolo Maistri, and Pierre Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *Proceedings of Design, Automation and Test in Europe (DATE)*. DOI:http://dx.doi.org/ 10.1109/DATE.2009.5090716

[27] Arijit Biswas, Paul Racunas, Joel Emer, and Shubhendu S.Mukherjee. 2008. Computing accurate AVFs using ACE analysis on performance models: a rebuttal. In *IEEE Computer Architecture Letters*, vol.7, no. 1, January-June. DOI:http://dx.doi.org/ 10.1109/L-CA.2007.19

[28] Aashish Phansalkar, Ajay Joshi, and Lizy K.John. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/ 10.1145/1273440.1250713

[29] Avinash Sodani, and Gurinhar S.Sohi. 1997. Dynamic instruction reuse. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/10.1145/384286.264200

[30] Avinash Sodani, and Gurinhar S.Sohi. 1998. An empirical analysis of instruction repetition. In *Proceedings of IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI:http://dx.doi.org/10.1145/384265.291016

[31] Saisanthosh Balakrishnan, and Gurinhar S.Sohi. 2003. Exploiting value locality in physical register files. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ISBN:0-7695-2043-X

[32] Xin Fu, Tao Li, and Jose Fortes. 2006. Sim-SODA: A unified framework for architectural level software reliability analysis. In *Workshop on Modeling, Benchmarking and Simulation*.

[33] Lide Duan, Bin Li, and Lu Peng. 2009. Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. DOI:http://dx.doi.org/ 10.1109/HPCA.2009.4798244

[34] Konstantinos Parasyris, Georgios Tziantzoulis, Christos D. Antonopoulos, and Nikolaos Bellas. 2014. GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates. *In Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2014.96

[35] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K.Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R.Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D.Hill, and David A.Wood. 2011. The Gem5 simulator. In *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, May. DOI:http://dx.doi.org/10.1145/2024716.2024718

[36] Man-Lap Li, Pradeep Ramachandran, Swarup K.Sahoo, Sarita V.Adve, Vikram S.Adve, and Yuanyuan Zhou. 2008. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI:http://dx.doi.org/ 10.1145/1353534.1346315

[37] Jinho Suh, Murali Annavaram, and Michel Dubois. 2013. PHYS: Profiled-HYbrid Sampling for soft error reliability benchmarking. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2013.6575352

[38] Vilas Sridharan, and David R.Kaeli. 2009. Eliminating microarchitectural dependency from architectural vulnerability. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*. DOI:http://dx.doi.org/10.1109/HPCA.2009.4798243

[39] Vilas Sridharan, and David R.Kaeli. 2010. Using hardware vulnerability factors to enhance AVF analysis. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/10.1145/1816038.1816023

[40] Arijit Biswas, Niranjan Soundararajan, Shubhendu S.Mukherjee, Sudhanva Gurumurthi. 2009. Quantized AVF: a means of capturing vulnerability variations over small windows of time. In *International Workshop on Silicon Errors in Logic-System Effects (SELSE)*.

[41] Pablo Montesinos, Wei Liu, and Josep Torrellas. 2007. Using register lifetime predictions to protect register files against soft errors. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2007.99

[42] Xin Xu, and Man-Lap Li. 2012. Understanding soft error propagation using efficient vulnerability-driven fault injection. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. ISBN: 978-1-4673-1624-8

[43] Vimal Reddy, and Eric Rotenberg. 2007. Inherent Time Redundancy (ITR): Using program repetition for low-overhead fault tolerance. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2007.59

[44] Mohamed A.Gomaa, and T.N.Vijaykumar. 2005. Opportunistic transient-fault detection. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/ 10.1109/ISCA.2005.38

[45] Siva K.S.Hari, Sarita V.Adve, Helia Naemi, and Pradeep Ramachandran. 2012. Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In *Proceedings of IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI:http://dx.doi.org/ 10.1145/2150976.2150990

[46] Guanpeng Li, Qining Lu, and Karthik Pattabiraman. 2015. Fine-grained characterization of faults causing long latency crashes in programs. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/10.1109/DSN.2015.36

[47] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. 2015. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. DOI:http://dx.doi.org/ 10.1109/DSN.2015.44

[48] Siva K.S.Hari, Radha Venkatagiri, Sarita V.Adve, and Helia Naemi. 2014. GangES: Gang error simulation for hardware resilience evaluation. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture (ISCA)*. DOI:http://dx.doi.org/10.1145/2678373.2665685

[49] Matthew R.Guthaus, Jeff S.Ringenberg, Damien Ernst, Todd M.Austin, Trevor Mudge, and Richard B.Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of IEEE International Workshop on Workload Characterization (WWC)*. DOI:http://dx.doi.org/ 10.1109/WWC.2001.990739

[50] Daya S.Khudia, and Scott Mahlke. 2014. Harnessing soft computations for low budget fault tolerance. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*. DOI:http://dx.doi.org/ 10.1109/MICRO.2014.33

[51] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *Proceedings of IEEE/ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. DOI:http://dx.doi.org/10.1145/635506.605403