Rapid Fault-Space Exploration by Evolutionary Pruning

Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk

Technische Universität Dortmund, Computer Science 12
Otto-Hahn-Str. 16, 44221 Dortmund, Germany
{horst.schirmeier,christoph.borchert,olaf.spinczyk}@tu-dortmund.de

Abstract. Recent studies suggest that future microprocessors need low-cost fault-tolerance solutions for reliable operation. Several competing software-implemented error-detection methods have been shown to increase the overall resiliency when applied to critical spots in the system. Fault injection (FI) is a common approach to assess a system's vulnerability to hardware faults. In an FI campaign comprising multiple runs of an application benchmark, each run simulates the impact of a fault in a specific hardware location at a specific point in time. Unfortunately, exhaustive FI campaigns covering all possible fault locations are infeasible even for small target applications. Commonly used sampling techniques, while sufficient to measure overall resilience improvements, lack the level of detail and accuracy needed for the identification of critical spots, such as important variables or program phases. Many faults are sampled out, leaving the developer without any information on the application parts they would have targeted.
We present a methodology and tool implementation that application-specifically reduces experimentation efforts, allows to freely trade the number of FI runs for result accuracy, and provides information on *all* possible fault locations. After training a set of Pareto-optimal heuristics, the experimenting user is enabled to specify a maximum number of FI experiments. A detailed evaluation with a set of benchmarks running on the eCos embedded OS, including MiBench's *automotive* benchmark category, emphasizes the applicability and effectiveness of our approach: For example, when the user chooses to run only 1.5 % of all FI experiments, the average result accuracy is still 99.84 %.

1 Introduction

Recent technology roadmaps [1,2,3] suggest that future hardware designs for embedded systems will exhibit an increasing rate of soft errors, trading reliability for smaller structure sizes, lower supply voltage, and reduced production costs. This trend creates new challenges for embedded software development, which must application-specifically place error detection [4] and recovery mechanisms [5,6,7] (EDM/ERMs) that do not diminish all gains from these new hardware designs. In future embedded software, critical spots in the software stack must be hardened against hardware faults, while the remaining unprotected components economize resource consumption by occasionally tolerating incorrect results.

Architecture-level fault injection (FI) has been the standard analysis technique in the software fault-tolerance community for at least two decades [8,9]. In an FI *campaign*

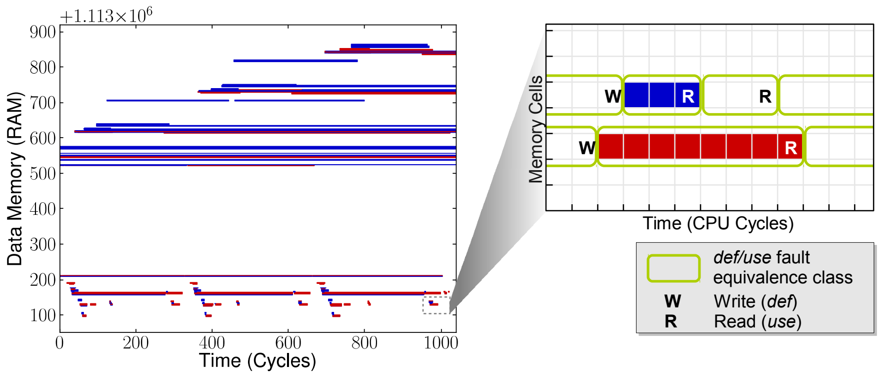


Fig. 1: Fault-space plot (left): Each coordinate in the fault space (RAM $\times$ Time) shows the outcome of one *independent* experiment that injects one transient fault at the particular coordinate. Faults that have no effect are shown as *white* points. Timeouts are denoted in *blue*, CPU exceptions in *red*, and SDCs in *black*. The magnified subplot (right) reveals def/use fault equivalence classes (see Sect. 2.1)

comprising many similar runs of an application benchmark, each run simulates the impact of a fault in a specific hardware location (e.g., a single bit in main memory or a CPU register) at a specific point in time during the benchmark's execution (e.g., one µs after the start). Unfortunately, FI campaigns exhaustively covering *all* possible fault locations are infeasible even for small target applications.

Consequently, most studies leveraging FI for dependability analysis purposes resort to *statistically sampling* fault locations [10,11]: A randomized selection of fault locations (usually in the thousands) is used to drive the FI campaign, until statistics predict a "good enough" probability for the overall result distribution to lie within the desired confidence interval. The result is an estimate on the aggregated campaign outcome – i.e., the usual probability breakup that an experiment finishes with the expected output (*no effect*), with a different output (a *silent data corruption*, SDC), that it terminates prematurely with a CPU exception, or that it runs into an endless loop and never terminates (*timeout*).[1] While an estimate on the aggregated results suffices for measuring the resiliency improvement between a benchmark's baseline and an ERM-protected variant, it gives no authoritative insights on critical spots and local phenomena, such as the vulnerability of specific data structures or program phases.

In contrast, the left half of Fig. 1 shows a visual representation of FI campaign results that were collected injecting faults into *all* possible fault locations of a particular benchmark application, requiring enormous computing power using the Fail* [12] FI framework with an x86 simulator backend. The fault model used for Fig. 1 constitutes uniformly distributed transient bit flips in the main memory. The *fault space* spans all CPU cycles during a benchmark run, and all bits in the address space. Thus, each coordinate in Fig. 1 shows the outcome of one independent FI experiment after injecting a burst bit-flip at a specific point in time (*CPU cycles* axis) and a specific byte in main

[1] This result categorization is just a (common) example; depending on the analyzed benchmarks and EDMs/ERMs, the FI campaign designer may choose a more fitting categorization.

memory (*memory address* axis). A large fraction of the injected bit flips is never read by the benchmark. This fact is acknowledged by most of the white areas in Fig. 1. Only injections into memory locations that are read in subsequent CPU cycles can have an effect, indicated by a colored coordinate. With information on the memory locations of program variables, and the phases in time certain system modules are active, the user can draw detailed conclusions on the vulnerability of specific variables and program phases.

In this paper, we describe a *fault-space pruning* methodology that massively reduces the FI campaign runtime *and* keeps information on all possible fault locations in the program. The basic idea stems from a simple insight: If in two FI experiments *the machine state is similar* (or *identical*) at the point in time where the fault is injected, *the experiment result will be similar* (or *identical*), too. The primary contributions of this paper are (1) a detailed description of an effective **application-specific fault-space pruning technique** that **preserves local features** of the application's reaction to faults, (2) a means to **freely trade accuracy for experimentation runtime** without suffering the drawbacks of randomized sampling, and (3) **detailed evaluation results with benchmarks running on the eCos embedded OS [13]**, including the *automotive* category of the MiBench [14] benchmark suite.

In the following, we focus on transient burst bit flips (flipping all eight bits at one address) in main memory to approximate multi-bit flips induced by single-event upsets [15], which are commonly caused by particle strikes. We assume a uniform distribution of faults, meaning that a fault can be triggered independently at any CPU cycle and memory address. We believe, though, that our approach can be easily applied to other fault models, such as single-bit flips or transient faults in CPU registers or caches, and intend to analyze this in future work. The following section revisits related FI and fault-space pruning principles, and points to related work in these areas. Sections 3 and 4 describe the design and implementation of our generic fault-equivalence heuristic, Section 5 presents and discusses evaluation results, and Section 6 summarizes and concludes the paper.

2 Background and Related Work

Recent FI techniques that proceed with more sophistication than randomly sampling locations in the fault space are based on deterministic experiment runs[2] and recorded instruction and memory-access traces. These traces are created during a *"golden run"*, which exercises the target software without injecting faults, and, thus, serves as a reference for the expected program behavior. In the following we describe the most important FI techniques based on trace information, which opens up a wide variety of possibilities to systematically reduce the number of FI experiments.

2.1 Conservative Def/use Analysis

Smith et al. are among the first concisely describing the classical *def/use analysis* technique [16] that was subsequently reinvented several times, e.g., by Benso et al. [17],

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

Berrojo et al. [18], Barbosa et al. [19], and recently by Grinschgl [20]. This method *conservatively* prunes the fault space, i.e., without compromising the result quality in any way. The basic insight is that all fault locations between a *def* (a **W**rite) or *use* (a **R**ead) of data in memory[3], and a subsequent *use*, are equivalent (see the right half of Fig. 1): regardless of when exactly in this time frame a fault is injected there, the earliest point where it will become architecturally visible is when the corrupted data is read. Instead of conducting one experiment for every point within this time frame, it suffices to conduct a *single* experiment (for example at the latest possible time directly before the **R**ead), and assume the same outcome for all remaining coordinates within that *def/use equivalence class* (green frame in Fig. 1). Similarly, all points in time between a **R** or **W** and a subsequent **W** are known to result in *no effect*, as the corrupted data will be overwritten in all cases. The result is a partitioning of the fault space into def/use equivalence classes, some of which a single experiment needs to be conducted for (those ending with a **R**ead), and some with a priori known experiment outcome. Note that even if we conduct only one experiment for a particular def/use equivalence class, the experiment's result contributes to many fault locations in the fault space. For example, in the right half of Fig. 1, the lower equivalence class, which ends with a **R**ead, represents *eight* fault-injection outcomes, and has a greater weight in the outcome distribution than the upper def/use class representing *three* FI outcomes.

2.2 Fault Equivalence Heuristics

Although def/use pruning significantly reduces the number of experiments, the computational efforts for complete fault-space coverage are still far too heavy for most benchmarks. For example, the experiment count for the MiBench benchmark BASICMATH (*small* input data set, x86 build) is reduced from 5.3×10^{13} to 1.8×10^{8} – still prohibitively many, even if a single experiment only takes a few seconds.

Only recently, more advanced fault-equivalence pruning techniques appear in literature, leaving the realm of conservative, accuracy-neutral methods. A significant contribution constitutes the RELYZER tool by Hari et al. [21], who describe several heuristics that combine multiple def/use equivalence classes into larger groups. From each group, only *one* representing def/use equivalence class (the *pilot*) gets picked, and the experiment result is assumed to be identical for the remaining group members. Although rather effective, a major deficiency of this approach is the inflexibility regarding the result accuracy (the authors report an average accuracy of 96 % for pilots representing their group) and experiment count tradeoff: If the result accuracy turns out too low, no alternative is offered, and if the experiment count is still too high, the authors suggest sampling from the set of pilots (resulting in the fault-space coverage problems mentioned in Sect. 1). Additionally, the grouping heuristics are based on complex control and data-flow analyses, SPARC platform specifics, and assumptions on the experiment result interpretation: RELYZER only differentiates between *no effect* and *SDC* outcomes, while in many use cases more outcome types, or even quality thresholds on the output [22], become relevant.

[3] This technique also works for any other level in the memory hierarchy, e.g., CPU registers or cache memory.

Li and Tan [23] describe similar pruning heuristics in their SmartInjector tool. They provide a slightly better experiment count reduction than Relyzer, but otherwise share the aforementioned drawbacks, including the single focus on SDCs, and a fixed tradeoff between accuracy (reportedly 94 % on average) and pruning effectivity.

3 A Generic Fault-Equivalence Heuristic

In the following, we outline a generalization – and simplification – of the *fault equivalence* notion coined in the works of Hari et al. [21] and Li & Tan [23], and subsequently derive a generic heuristic that ameliorates the inflexibilities mentioned in Sect. 2.2.

3.1 Fault Similarity, and a Generalization of Fault Equivalence

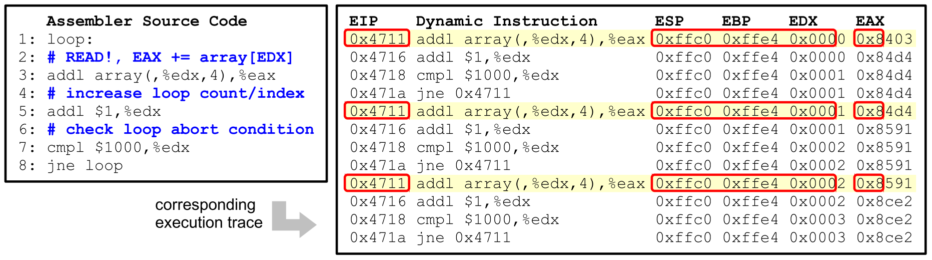
Instead of using the term "fault-equivalence class" from Hari and Li to denote groupings of multiple def/use equivalence classes, we will use the term *fault-similarity class* in the following to avoid confusion with *def/use equivalence classes* (cf. Sect. 2.1), and to capture the following facts:

Mispredictions may occur: One def/use class (the *pilot*) represents all the other fault-similarity class members. Due to the approximative nature of heuristics, one or more non-pilot members can have a different experiment outcome than the pilot, making the word "equivalence" unwarranted.

Equivalence is in the eye of the beholder: The equivalence of two FI experiment outcomes purely depends on the experimenter's definition. While for one type of FI campaign, any deviation from the golden run is a *failure*, for others a detailed differentiation into several outcome types (cf. Sect. 1) is important. Hence, multiple def/use classes are *similar* only under the chosen evaluation metric.

Although both studies [21,23] invent various complicated analysis techniques to combine multiple def/use equivalence classes into larger groups, the general notion of *fault similarity* can be reduced to a simple insight: If at one point in time during the benchmark run the machine state is *completely identical* to the state at another point in time, conducting two FI experiments that inject a fault (e.g., a single-bit flip) at a specific memory location will yield the same result, no matter whether the fault was injected at the first or the second point in time. For example, if one FI experiment results in an *SDC* that affects the program's output, the very same thing will happen in the other experiment.

Of course, in reality the machine state is never completely identical at two points in time during a program run; even if the benchmark would enter an infinite loop, some parts of the machine (e.g., a wallclock timer) would be in a different state. Nevertheless, over the program's runtime, a *relevant part* of the machine state may be identical at several points in time. To gain a better intuition on what we mean by a relevant part of the machine state, observe the x86 assembler code snippet in Fig. 2 (left-hand side): In a loop, the (integer) elements of an array are added up in the EAX register, keeping the array index (also used for the loop abort condition) in the EDX register. When considering faults in memory only, and applying the def/use pruning method described in Sect. 2.1, the only memory-reading *(use)* instruction is the one in Line 2 (marked with the comment



```
   Assembler Source Code
1: loop:
2: # READ!, EAX += array[EDX]
3: addl array(,%edx,4),%eax
4: # increase loop count/index
5: addl $1,%edx
6: # check loop abort condition
7: cmpl $1000,%edx
8: jne loop
```

```
EIP    Dynamic Instruction       ESP    EBP    EDX    EAX
0x4711 addl array(,%edx,4),%eax  0xffc0 0xffe4 0x0000 0x8403
0x4716 addl $1,%edx              0xffc0 0xffe4 0x0000 0x84d4
0x4718 cmpl $1000,%edx           0xffc0 0xffe4 0x0001 0x84d4
0x471a jne 0x4711                0xffc0 0xffe4 0x0001 0x84d4
0x4711 addl array(,%edx,4),%eax  0xffc0 0xffe4 0x0001 0x84d4
0x4716 addl $1,%edx              0xffc0 0xffe4 0x0001 0x8591
0x4718 cmpl $1000,%edx           0xffc0 0xffe4 0x0002 0x8591
0x471a jne 0x4711                0xffc0 0xffe4 0x0002 0x8591
0x4711 addl array(,%edx,4),%eax  0xffc0 0xffe4 0x0002 0x8591
0x4716 addl $1,%edx              0xffc0 0xffe4 0x0002 0x8ce2
0x4718 cmpl $1000,%edx           0xffc0 0xffe4 0x0003 0x8ce2
0x471a jne 0x4711                0xffc0 0xffe4 0x0003 0x8ce2
```

Fig. 2: Short x86 assembler snippet (left) adding up the contents of an array: All memory reads in the dynamic execution (right) share a similar machine state, and FI will lead to similar results in all cases (a wrong sum). For simplicity registers carry dummy 16-bit values.

"READ!"). Injecting a fault into the memory location being read from directly before the read will lead to a similar result in all loop iterations: The resulting sum will be calculated faultily. Thus, it would suffice to do a single experiment instead of 1000, and predict the same result outcome for the others.

Now consider the machine state right before each memory read in the dynamic execution (right-hand side, highlighted lines): Among others, the EIP (instruction pointer), ESP and EBP registers are the same in all cases, EDX only differs by its lower-order bits in most consecutive loop iterations, and EAX may (depending on the magnitude of the values in the array) not change too much from one iteration to the next either. A (geometric) *projection function* of the machine-state vector – preserving only the components EIP, ESP, EBP, and the higher-order bits of EDX and EAX – therefore serves very well as a criterion to combine all these def/use equivalence classes into a single group, and to conduct a single experiment instead of one per loop iteration.

3.2 A Flexible Fault-Similarity Heuristic

Our working hypothesis is that a *projection of the machine-state vector* can successfully be used to combine multiple def/use equivalence classes with high result accuracy, depending on the user's requirements. We assume that this projection highly depends on the analyzed program(s) (including the underlying operating system) and their chosen input, the CPU architecture, the compiler, the chosen compiler optimizations, the chosen time discretization, the fault model, and what experiment outcome differentiation the user chooses. A generic fault-similarity heuristic therefore has to adapt to these factors; a detailed analysis of their impact is beyond the scope of this paper, though.

The basic idea behind our heuristic is to find a suitable machine-state projection that can be used to accurately combine def/use classes with "mostly" equivalent FI results. More in detail, we first record a machine-state vector for each *def* and *use* when recording the golden run trace; for efficiency reasons, we do not record the complete machine state, but only the values listed in Tab. 1. Then we determine FI results by running actual experiments for a feasible, randomly chosen subset of all def/use classes, which serve as the *training set* for searching a state-vector projection that accurately groups def/use classes with mostly equivalent experiment outcomes. An optimization algorithm then searches for a projection function that is optimal regarding the specified result accuracy, or the limit on the number of FI experiments, which refers to the number of fault-similarity classes.

These two criteria – *maximum accuracy* and *minimal number of fault-similarity classes* – are contradictory and can be traded for each other. One extremal point (in favor of *accuracy*) uses the identity function as the state-vector projection – hence defines all available machine state as *relevant* for grouping – and combines *no* def/use partition with another: it produces fault-similarity classes with *one* def/use member each, hence conducts an FI experiment for *every* def/use class, and achieves maximum accuracy. The other extremal point combines *all* def/use classes to a single similarity class and only conducts a single experiment, resulting in minimal experimentation effort and maximal result error. Between these extremal points exists a large search space with all possible machine-state vector projections, some of them representing Pareto-optimal solutions that optimally trade experiment effort for accuracy.

3.3 Applying the Similarity Heuristic

After finding a projection function that satisfies the user's requirements regarding accuracy and FI campaign efforts, the user can apply it to the remaining def/use classes with yet unknown outcome: multiple def/use classes with identical values in the projected machine-state vector are grouped into one common fault-similarity class. Speaking in the example from Sect. 3.1, all def/use classes with the same values in EIP, ESP, EBP, and the higher-order bits of EDX and EAX, are assigned the same fault-similarity class. For all fault-similarity classes that do not yet contain at least one member with a known outcome (because an FI experiment was run in the training phase), one pilot def/use class gets picked, and one experiment is run. After this step is completed, all def/use classes (and, thus, every coordinate in the fault space, as depicted in Fig. 1) either directly – by running an FI experiment for them – or indirectly – by looking at the pilot in their fault-similarity class – can be assigned an experiment outcome.

4 Implementation

We implemented a tool set for the outlined fault-space pruning approach in the FAIL* [12] FI experimentation framework, configured to run with the Bochs x86 simulator [24]. Ideally, we would simulate faults in a detailed register transfer and gate-level processor model; however, since simulation of realistic benchmarks on low-level models is extremely slow, this work chooses a fast architecture simulator. We extended the *tracing* plugin of FAIL* with the capability to record the additional machine state listed in Tab. 1 alongside the usual instruction and memory-access trace.

We encode the recorded machine state (Tab. 1) into a single, long bit vector with all state variables concatenated. This *state vector* exists once for every dynamic instruction in the golden run that reads memory. The projection function we want to search for is also encoded as a bit vector (with the same length as the state vector), which we call the *projection vector*. Its bits indicate whether the corresponding bit position in the machine state is **used** (1) or **not used** (0) for comparing the machine state of def/use

data_address	Memory address the def/use writes/reads
data_value	Actual value that is written/read
EIP	Instruction pointer of the def/use instruction
dyn_instr	Dynamic instruction count since benchmark start
opcode	Instruction's opcode
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, EFLAGS	Contents of general-purpose registers, stack pointer, CPU flags
*EAX, *EBX, *ECX, *EDX, *ESI, *EDI, *ESP, *EBP	Contents of the machine word the respective register points to (if interpretable as a mapped memory address)
jumphistory	RELYZER [21] style (control-equivalence) bit list indicating whether each of the last 16 and next 16 conditional branches was taken
duration	Temporal duration of the def/use equivalence class (e.g., in CPU cycles)
benchmark_id	An ID uniquely identifying each benchmark

Table 1: Information recorded for every dynamic *def* or *use* instruction executed during the golden run of each benchmark.

classes, deciding whether or not to group them into a common fault-similarity class. The bitwise AND of an – initially randomly chosen – projection vector and each state vector yields a vector uniquely identifying the similarity class each def/use class belongs to. The example projection function from Sect. 3.1 – preserving only the components EIP, ESP, EBP, and the higher-order bits of EDX and EAX – can directly be encoded by setting the corresponding bits in this vector.

In order to find an *optimal* projection vector, we model the search problem on the training data using the SPEA2 multi-objective evolutionary algorithm [25] (implemented in the PISA library [26]) with the projection vector as the genome, and simple multi-bit mutation and single-point crossover operators [27]. Initially we run a fixed number of 100,000 FI experiments[4] per benchmark to gain training data. Then the genetic algorithm is initialized with a population of randomized projection vectors. In every *generation* of the search algorithm, each individual's (i.e., projection bit vector's) fitness is evaluated by **1.** performing the aforementioned grouping of def/use classes from the training set into similarity classes, **2.** picking the largest[5] def/use class in each similarity class as the pilot and pretending it properly represents the remaining class members, and **3.** measuring the two fitness criteria *accuracy* and the emerging *number of fault-similarity classes* within the training set.

The *accuracy* measures how accurately the pilots *actually* represent the remaining members within their similarity classes. As a misprediction of a large (i.e., many clock cycles wide) def/use class has a greater impact on the outcome quality than a small one (cf. Sect. 2.1), the correctly predicted *area* in the fault space is used for this metric, taking the weights of the def/use classes into account:

$$\text{Accuracy} = \frac{\text{Correctly predicted fault-space area}}{\text{Total fault-space area}}$$

[4] This number was arbitrarily chosen for the purpose of this article, but may be selected application-specifically in the future.

[5] Currently the def/use class spanning the most CPU cycles (cf. Sect. 2.1) is chosen as each similarity class's pilot to minimize error when mispredictions occur.

The second fitness criterion is the *number of different similarity classes* emerging from the def-use class grouping step. These two criteria – the number of correctly represented faults, and the number of similarity classes that directly translates into the number of pilots (and, hence, the total number of necessary FI experiments) – are used as the optimization objectives for the SPEA2 algorithm.

5 Evaluation

In the following, we will elaborate on the evaluation setup we used, and subsequently analyze the effectiveness and efficiency of our fault-space pruning heuristic.

5.1 Evaluation Setup and Ground Truth

First, we chose a subset of the benchmark programs that accompany the eCos operating system (namely those we already used in a previous work [5]). The 19 eCos/baseline programs are relatively small, and test eCos kernel capabilities, such as synchronization, scheduling, and inter-process communication (refer to [5] for a more detailed description). They are also reasonably similar to each other regarding their fault propagation, which allows us to consider them as a single, combined benchmark for the remainder of this section. A second variant of the 19 programs – eCos/CRC – is hardened against memory faults by error-detection measures (CRC32 codes for all kernel objects [5] and for the stacks of preempted threads [28]), and subsequently executes about three times more dynamic executions (cf. Tab. 2). Additionally, we picked MiBench's [14] *automotive* benchmark category as a set of real-world application benchmarks. The four benchmarks (qsort, basicmath, bitcount, susan, using the *small* input data set) each execute more dynamic instructions than all eCos benchmarks combined and represent a more heavyweight workload for our tooling.

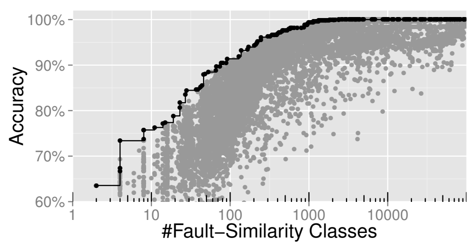
To determine the "ground truth" for our pruning experiments, we ran FI experiments for *all* def/use equivalence classes of our benchmarks, resulting in the total (single-CPU) simulation time shown in the last column of Tab. 2. We limited FI to the first 10^7 dynamic instructions for the MiBench benchmarks to keep our computing time budgets reasonable.

For each benchmark in Tab. 2 we randomly[6] picked 100,000 def/use classes as the training set, and parametrized the genetic algorithm with a population size of 100 individuals, 400 optimization generations, and a mutation probability of 10 %. The values for these parameters were picked from experiences in early evaluation rounds; we will analyze their impact on the approach more in detail in future work.

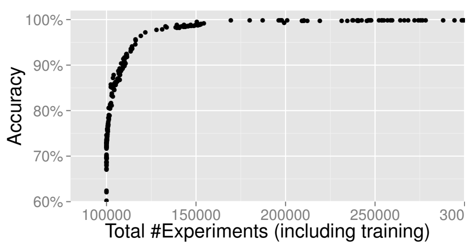
5.2 Heuristic Training and Test

Fig. 3a shows the training results and fitness values for eCos/baseline after $3'40''$ of optimization (on a 32-core Intel Xeon E5-4650): Each point represents an individual (the *machine-state projection vector*; cf. Sect. 3.2), which partitions the training set into a number of fault-similarity classes (X-axis, logarithmic scale, with possible values from 1

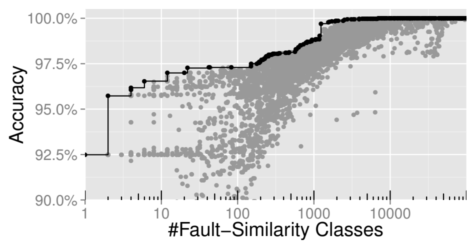
[6] Uniform sampling without taking the def/use class size into account.



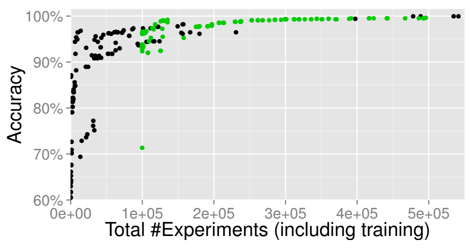
(a) eCos/baseline (training)



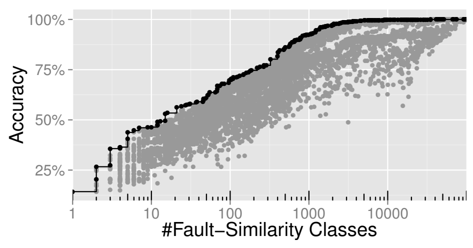
(b) eCos/baseline (total)



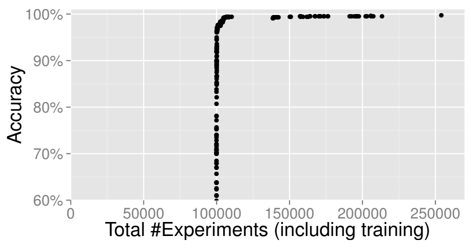
(c) eCos/CRC (training)



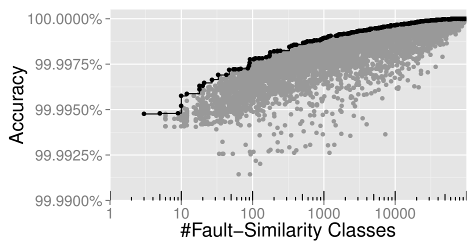
(d) eCos/CRC (total, w/ and w/o training)



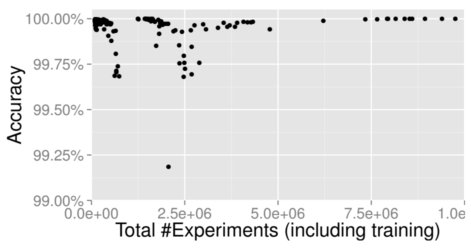
(e) MiBench/basicmath (training)



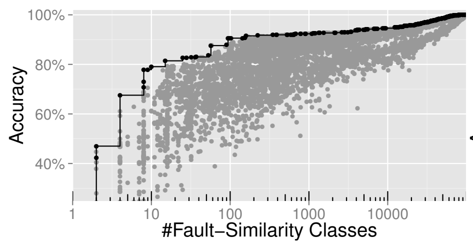
(f) MiBench/basicmath (total)



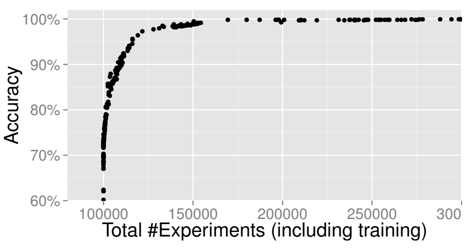
(g) MiBench/qsort (training)



(h) MiBench/qsort (total)



(i) MiBench/susan (training)



(j) MiBench/susan (total)

Fig. 3: Accuracy in the training set (left) and in the complete fault space (right)

Benchmark	Dyn. instr.	CPU cycles	# FI exp. after def/use pruning	FI simulation runtime
eCos/baseline	1.08×10^7	9.7×10^9	1.48×10^7	4,946 hrs (33.4 hrs)
eCos/CRC	2.73×10^7	9.7×10^9	4.15×10^7	15,035 hrs (36.1 hrs)
MiBench/qsort	4.20×10^7	4.20×10^7	1.48×10^7 (of 5.49×10^7)	44,139 hrs (297.6 hrs)
MiB/basicmath	1.47×10^8	1.47×10^8	1.24×10^7 (of 1.84×10^8)	95,068 hrs (767.8 hrs)
MiB/bitcount	4.08×10^7	4.08×10^7	2.89×10^6 (of 9.15×10^6)	3,621 hrs (124.9 hrs)
MiBench/susan	2.95×10^7	2.95×10^7	1.25×10^7 (of 3.68×10^7)	23,526 hrs (186.9 hrs)

Table 2: Dynamic instruction counts, simulated CPU cycles, num. of 8-bit burst FI experiments necessary after basic def/use pruning (cf. Sect. 2.1), and FI simulation runtime for all experiments (and the 100,000 experiment training set). For the eCos benchmarks, the total CPU cycles differ from the dynamic instructions due to idle phases; for MiBench, we limited FI to the first 10^7 dynamic instructions.

to 100,000). As described in Sect. 4, the accuracy of such a partioning, plotted on the Y-axis, is defined as the percentage of correctly represented fault-space area.

The optimization yields a set of Pareto-optimal solutions (black) the user can choose from to partition the benchmark's *complete* fault space in the next step (cf. Sect. 3.3). Fig. 3b shows the test results when applying all previously determined, Pareto-optimal projection vectors to the complete fault space while reusing all results from the 100,000 training FI experiments: Depending on the partitioning our heuristic creates in the complete fault space, more experiments than the initial training experiments need to be conducted to get a representing pilot for *all* new fault-similarity classes. For a chosen projection, the accuracy drops by a nonlinear factor (in the order of 0.1 % for highly-accurate, up to 20 % for the low-quality solutions) from training to test. For example, a solution with 99.9986 % accuracy in the training set (9,012 fault-similarity classes) requires the user to conduct 99,308 additional FI experiments (totaling in 199,308 including the training set, from a total of 14.8 million, cf. Tab. 2) and reconstructs the complete fault space with 99.2512 % accuracy; a solution with 99.9903 % training set accuracy (2,100 fault-similarity classes) yields 90.4036 % accuracy with 8,243 additional experiments (108,243 total).

In Fig. 4, we apply the latter example solution to the complete eCos/baseline fault space, illustrating the advertised local fault-space feature preservation of our heuristic: A close-up of a small area of the fault-space plot of the eCos/baseline mutex1 benchmark (actually the same excerpt as in Fig. 1) even remains largely intact when after training only 8,243 additional FI experiments (totaling 108,243 including the training set, a mere 0.73 % of all 14.8 million eCos/baseline experiments) are conducted.

Subsequently we investigated how well previously trained solutions apply to a new, unknown (yet not completely different) benchmark. As described in the previous section,

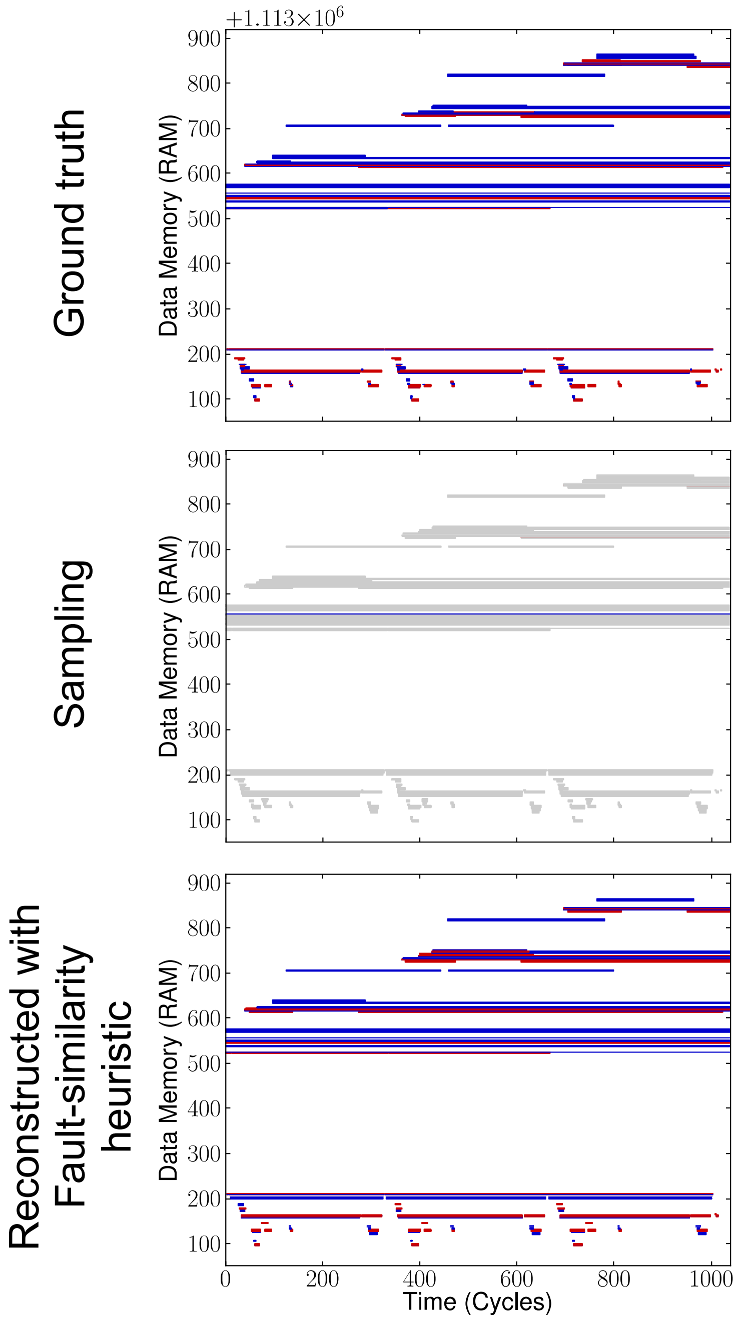


Fig. 4: A tiny fault-space plot excerpt from a stack memory area of the MUTEX1 benchmark (from top to bottom; color coding as in Fig. 1): Ground-truth results (100 % FI experiments), results from sampling 0.73 % of all experiments (gray areas are unknown results, i.e., def/use classes that were not sampled or known a priori), and reconstructed results with also a total of 0.73 % (including training set) of all experiments for the eCos/BASELINE benchmarks.

the eCos/CRC benchmark comprises the same 19 programs as eCos/baseline, yet they are hardened against memory faults, and execute substantially more dynamic instructions. Probably most notably they introduce a new FI experiment outcome type "detected" that signals a successful error detection of an EDM: This outcome type does not exist in eCos/baseline, and, thus, cannot have been observed by the training process from Fig. 3a. The black points in Fig. 3d show how well these projection vectors perform for eCos/CRC without any eCos/CRC-specific training. (Hence, there is no initial 100,000 FI experiment penalty for the training set). One interesting observation is that the solutions requiring up to 300,000 FI experiments are partially in the 90–97 % accuracy range, but by far not as close to 100 % as in the eCos/baseline plot (Fig. 3b). Nevertheless, previously trained heuristics seem to be reusable even for unknown benchmarks: As our training process only learns how to group def/use classes into fault-similarity classes, but does not try to completely *predict* experiment outcomes without carrying out new FI experiments, it can even deal with previously unseen experiment outcomes, such as "detected" in this case. Fig. 3c and the green points in Fig. 3d show the accuracy results after training specifically for eCos/CRC:[7] The accuracy (and especially the accuracy mapping from training to test) is significantly better than without training (the low-quality left margin vanishes), but at the cost of an initial training phase and more FI experiments depending on the desired accuracy.

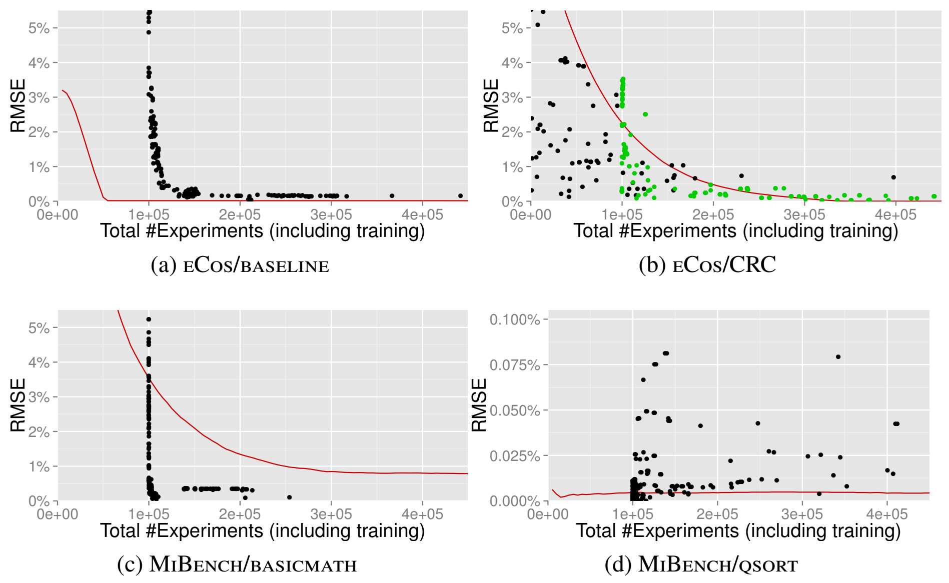
Among the remaining Fig. 3e–3j (MiBench/bitcount is omitted due to space constraints, but closely resembles the plots for susan), MiBench/qsort displays an extremely high accuracy even for minimal numbers of additional experiments: With 100,012 FI experiments (results from training, plus 12 new fault-similarity classes in the complete fault space) it achieves an accuracy of 99.9902 %. The primary reason for this is that the vast majority (99.9892 %) of experiment outcomes for this benchmark are SDCs (not very astonishing for a benchmark sorting a long list of text strings), which allows to create extremely large fault-similarity classes that yield the same outcome.

5.3 Experiment Outcome Breakup & Comparison with Sampling

As the user is, apart from fault-space details, also interested in the usual experiment outcome breakup (for example, in the basicmath benchmark, 34.18 % of all faults result in *no effect*, 19.34 % *SDC*, 31.80 % *CPU exception*, 14.68 % *timeout*), this aggregate should not turn out to be inaccurate either. Fig. 5 shows the root mean squared error (RMSE) of experiment outcome breakups for a selection of benchmarks with the Pareto-optimal heuristics from Fig. 3 applied to their complete fault space (black points and green points with the same meaning as in the previous section): As expected, the fault-space reconstruction accuracy and the outcome breakup RMSE correlate quite well – good local accuracy also yields a good global accuracy.

The plots in Fig. 5 also include the outcome breakup RMSE for FI sampling (with fault expansion [29]; red lines in the figure), a technique commonly used to *only* estimate the breakup without gaining any information on local fault-space details. Interestingly, in some cases sampling yields inferior results – especially for the un-trained heuristic

[7] ... and reusing some of the eCos/baseline projection vectors as the initial population for the optimization algorithm.



(a) eCos/baseline (b) eCos/CRC

(c) MiBench/basicmath (d) MiBench/qsort

Fig. 5: Comparison to sampling (red line): RMSE of the outcome probability breakup is comparable to, and many cases better than the common sampling approach – which completely lacks information on local fault-space details.

configurations in the eCos/CRC case (Fig. 5b, black points) as it does not have the experiment-count penalty of a training set. This means our heuristic can compete with sampling, although it yields much more detailed information on the fault space, e.g., for EDM/ERM placement.

6 Conclusions & Future Work

We presented an adaptive, application-specific fault-space pruning technique that preserves local features of the fault space for detailed susceptibility analyses. The approach allows the user to freely trade accuracy for experimentation runtime, choosing a Pareto-optimal heuristic that was trained with a feasible FI experiment subset from the program(s) under analysis. Our results confirm the assumption that a machine-state subset can be successfully used to partition the fault space into fault-similarity classes, allowing to gain insights on local phenomena for EDM/ERM placement with massively reduced experimentation efforts: For example, when the user chooses to run 1.5 % of all FI experiments, the average (weighted by the total number of faults in each benchmark) result accuracy is 99.84 %.[8] In many cases our fault-space pruning technique even outperforms classic sampling techniques, although they do not preserve any fault-space details.

[8] Except for bitcount, where training yields no solution that only needs 1.5 % experiments. Here, the user can achieve, e.g., 99.84 % accuracy for 4 % of all experiments.

Future work includes a detailed analysis of the various free parameters of our approach, including the impact of different training-set sizes, the genetic algorithm's configuration (population size, generation number, and mutation probability), and the chosen genome representation itself. To gain more confidence in the genericity of our approach, we intend to evaluate it on other instruction-set architectures and with other types of benchmark applications. We also plan to analyze different fault models, and consider to completely replace the evolutionary algorithm with a more sophisticated machine-learning algorithm. Beyond this, merging the training phase and the FI campaign into a continuously adapting online training might speed up the dependability assessment process even more.

Acknowledgments This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. SP 968/5-2. The authors would like to thank Jochen Streicher for lenghty discussions on quality metrics and machine learning, Adrian Böckenkamp, Richard Hellwig and Lars Rademacher for feedback on the core idea of the pruning heuristic, and Daniel Cordes for initial help with the PISA library.

References

1. Borkar, S.Y.: Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. IEEE Micro **25**(6) (2005) 10–16
2. Duranton, M., Yehia, S., de Sutter, B., de Bosschere, K., Cohen, A., Falsafi, B., Gaydadjiev, G., Katevenis, M., Maebe, J., Munk, H., Navarro, N., Ramirez, A., Temam, O., Valero, M.: The HiPEAC vision. Technical report, HiPEAC (2010)
3. Narayanan, V., Xie, Y.: Reliability concerns in embedded system designs. IEEE Comp. **39**(1) (2006) 118–120
4. Hari, S.K.S., Adve, S.V., Naeimi, H.: Low-cost program-level detectors for reducing silent data corruptions. In: 42nd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '12), IEEE (2012)
5. Borchert, C., Schirmeier, H., Spinczyk, O.: Generative software-based memory error detection and correction for operating system data structures. In: 43nd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '13), IEEE (June 2013)
6. Borchert, C., Schirmeier, H., Spinczyk, O.: Protecting the dynamic dispatch in C++ by dependability aspects. In: 1st GI W'shop on SW-Based Methods for Robust Embedded Sys. (SOBRES '12). LNI, German Society of Informatics (September 2012) 521–535
7. Borchert, C., Schirmeier, H., Spinczyk, O.: Return-address protection in C/C++ code by dependability aspects. In: 2nd GI W'shop on SW-Based Methods for Robust Embedded Sys. (SOBRES '13). LNI, German Society of Informatics (September 2013)
8. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., Powell, D.: Fault injection for dependability validation: A methodology and some applications. IEEE TOSE **16**(2) (February 1990) 166–182
9. Benso, A., Prinetto, P.: Fault injection techniques and tools for embedded systems reliability evaluation. Frontiers in electronic testing. Kluwer, Boston, Dordrecht, London (2003)
10. Leveugle, R., Calvez, A., Maistri, P., Vanhauwaert, P.: Statistical fault injection: quantified error and confidence. In: 2009 Conf. on Design, Autom. & Test in Europe (DATE '09), IEEE (2009) 502–506
11. Ramachandran, P., Kudva, P., Kellington, J., Schumann, J., Sanda, P.: Statistical fault injection. In: 38th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '08), IEEE (2008) 122–127
12. Schirmeier, H., Hoffmann, M., Kapitza, R., Lohmann, D., Spinczyk, O.: FAIL*: Towards a versatile fault-injection experiment framework. In Mühl, G., Richling, J., Herkersdorf, A., eds.: 25th Int. Conf. on Arch. of Comp. Sys. (ARCS '12), Workshop Proceedings. Volume 200 of LNI., German Society of Informatics (March 2012) 201–210

13. Massa, A.: Embedded Software Development with eCos. Prentice Hall (2002)
14. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: MiBench: A free, commercially representative embedded benchmark suite. In: IEEE Int. W'shop. on Workload Characterization (WWC '01), Washington, DC, USA, IEEE (2001) 3–14
15. Mukherjee, S.: Architecture Design for Soft Errors. Morgan Kaufmann (2008)
16. Smith, D.T., Johnson, B.W., Profeta, III, J.A., Bozzolo, D.G.: A method to determine equivalent fault classes for permanent and transient faults. In: Annual Reliability and Maintainability Symposium. (January 1995) 418–424
17. Benso, A., Rebaudengo, M., Impagliazzo, L., Marmo, P.: Fault-list collapsing for fault-injection experiments. In: Annual Reliability and Maintainability Symposium. (January 1998)
18. Berrojo, L., Gonzalez, I., Corno, F., Reorda, M., Squillero, G., Entrena, L., Lopez, C.: New techniques for speeding-up fault-injection campaigns. In: 2002 Conf. on Design, Autom. & Test in Europe (DATE '02). (2002) 847–852
19. Barbosa, R., Vinter, J., Folkesson, P., Karlsson, J.: Assembly-level pre-injection analysis for improving fault injection efficiency. In: 5th Europ. Depend. Comp. Conf. (EDCC 2005). Volume 3463., Springer (April 2005) 246
20. Grinschgl, J., Krieg, A., Steger, C., Weiss, R., Bock, H., Haid, J.: Efficient fault emulation using automatic pre-injection memory access analysis. In: SOC Conference. (2012) 277–282
21. Hari, S.K.S., Adve, S.V., Naeimi, H., Ramachandran, P.: Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults. In: 17th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '12), New York, NY, USA, ACM (2012) 123–134
22. Döbel, B., Schirmeier, H., Engel, M.: Investigating the limitations of PVF for realistic program vulnerability assessment. In: 5rd HiPEAC W'shop on Design for Reliability (DFR '13), Berlin, Germany (January 2013)
23. Li, J., Tan, Q.: SmartInjector: Exploiting intelligent fault injection for SDC rate analysis. In: IEEE Int. Symp. on Defect & Fault Tol. in VLSI & Nanotech. Sys. (DFT 2013). (2013)
24. Lawton, K.P.: Bochs: A portable PC emulator for Unix/X. Linux Journal **1996**(29es) (1996)
25. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In Giannakoglou, K.C., Tsahalis, D.T., Périaux, J., Papailiou, K.D., Fogarty, T., eds.: Evolutionary Methods for Design Optimization and Control with Applications to Industrial Problems, Athens, Greece, International Center for Numerical Methods in Engineering (September 2001) 95–100
26. Bleuler, S., Laumanns, M., Thiele, L., Zitzler, E.: PISA — a platform and programming language independent interface for search algorithms. In Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L., eds.: Evolutionary Multi-Criterion Optimization (EMO 2003). LNCS, Berlin, Springer (2003) 494 – 508
27. Mitchell, M.: An Introduction to Genetic Algorithms. MIT Press (1998)
28. Hoffmann, M., Borchert, C., Dietrich, C., Schirmeier, H., Kapitza, R., Spinczyk, O., Lohmann, D.: Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In: 17th IEEE Int. Symp. on OO Real-Time Distrib. Computing (ISORC '14), IEEE (2014)
29. Smith, D.T., Johnson, B.W., Andrianos, N., Profeta, III, J.A.: A variance-reduction technique via fault-expansion for fault-coverage estimation. IEEE TR **46**(3) (September 1997) 366–374