

# Towards a Fault-Injection Benchmarking Suite

Tianhao Wang, Robin Thunig and Horst Schirmeier  
Chair of Operating Systems, TU Dresden, Germany  
tianhao.wang2@mailbox.tu-dresden.de  
{robin.thunig, horst.schirmeier}@tu-dresden.de

**Abstract**—Soft errors in memories and logic circuits are known to disturb program execution. In this context, the research community has been proposing a plethora of fault-tolerance (FT) solutions over the last decades, as well as fault-injection (FI) approaches to test, measure and compare them. However, there is no agreed-upon benchmarking suite for demonstrating FT or FI approaches. As a replacement, authors pick benchmarks from other domains, e.g. embedded systems. This leads to little comparability across publications, and causes behavioral overlap within benchmarks that were not selected for orthogonality in the FT/FI domain.

In this paper, we want to initiate a discussion on what a benchmarking suite for the FT/FI domain should look like, and propose criteria for benchmark selection.

## I. INTRODUCTION

Soft errors have been posing threats to modern computer systems, particularly in mission-critical applications such as satellites. Software-implemented hardware fault tolerance is a cost-effective way of mitigating soft errors. Researchers typically demonstrate their fault-tolerance (FT) mechanisms or fault-injection (FI) frameworks with a set of benchmarking programs. While there exists abundant research on FT and FI, to the best of our knowledge, there is no dedicated benchmarking suite for the FT/FI domain.

To test their FT and FI mechanisms, researchers usually resort to benchmarking suites from other domains such as *TACLeBench* [1] for *Worst-Case Execution Time (WCET)* research or *MiBench* [2] for embedded systems. These benchmarking suites are designed for different purposes and metrics than those relevant in the FT/FI domain. This leads to three major shortcomings:

(1) *Little Comparability across FT/FI Papers.* It is hard to compare the effectiveness of different FT/FI mechanisms when they are tested with different benchmarking programs. We also show in Sec. III that even the same benchmarking program could have a dramatically different fault space when built with different runtime.

(2) *Overlapping Benchmarks.* It is difficult to select a minimal set of benchmarks from other domains that achieves a full coverage of FT/FI relevant properties. Researchers usually pick as many benchmarks as possible to deliver a convincing demonstration, regardless of the potential overlap within the benchmarks. This leads to inefficiency as each additional benchmark could cost a huge amount of computing resources and time, particularly in the FI experiments.

(3) *Limited Configurability.* Technically, the out-of-the-box benchmarks from other domains are usually not optimal for

FI campaigns. For example, we often need programs to meet certain requirements such as a minimal amount of memory-access events or a maximal execution time. However, in most cases – if at all – we can only adjust the scale of the input.

Therefore we would like to open a discussion whether the FT/FI domain needs its own benchmarking suite, and which requirements it should meet.

## II. PREFERABLE BENCHMARK PROPERTIES

As an example, Guthaus et al. [2] select benchmarks for *MiBench* on a basis of their relevance in industrial embedded systems, and primarily focus on the system performance as a benchmarking metric. Falk et al.’s *TACLeBench* [1] is apt for WCET research. The focus is on the metrics of WCET analysis tools. The authors further differentiate usage-type categories like *kernel*, *sequential*, *application*, *test* and *parallel*.

While these domain-specific criteria and metrics could also be important for FT/FI research, they provide us with very limited insight when analysing the fault tolerance of programs. Some benchmarks could also be totally undesired by FT/FI experiments, such as the *test* group in *TACLeBench* or *MiBench*’s *basicmath* benchmark, which are designed to stress test the system under test with a repeated and simple workload.

From the experience and observations in the literature, we would like to propose several preferable properties for a FT/FI benchmarking suite.

*A. Different Granularities.* We observe that the FT/FI literature uses both simple, bare-metal algorithm implementations and complex system compositions for demonstration purposes. Therefore we suggest to include benchmarks in different granularities, e.g. 1) isolated algorithm implementations and program parts for a more targeted analysis and 2) integrated systems that demonstrate more realistic use cases.

*B. Selection of Relevant Benchmarks.* We suggest that it should be possible to classify benchmarks into different groups that can be utilized by FT/FI researchers to achieve a representative and preferably minimal experiment setup. Currently we have two categorizations in mind which are 1) categorizing benchmarks with respect to their program characteristics, which might include memory usage, runtime, fault space characteristics, etc., with which redundant benchmarks could be avoided and 2) categorizing benchmarks into specific domains, like possibly *MiBench* [2] in the embedded systems domain.

*C. Resource-Efficient Fault Injection.* An FI benchmarking suite should be designed with heavy FI evaluations in mind.

In this regard, we aim for a benchmark suite that 1) relies on a lightweight infrastructure and 2) is *configurable* to meet requirements such as execution time and memory usage. These properties would be helpful to adjust the fault-space size to specific needs, e.g. to reduce the number of necessary injections, enabling large-scale FI experiment campaigns.

*D. Self-contained Runtime.* Besides being lightweight, we propose that the benchmarking suite should also be self-contained and portable. On one hand, FI frameworks such as FAIL\* [3] run the benchmarks in an instrumented virtual machine. This requires target programs to bring their own runtime, including a barebone operating system and in most cases, a standard library. On the other hand, the dependency on uncertain external runtime support also makes experiment results less comparable across different studies. *TACLeBench* [1] shows a good example of portability in this regard.

### III. EVALUATION OF SELECTED PROPERTIES

In this section, we exemplarily explore some of the aforementioned benchmarking suite properties in more detail. For this fast abstract, we try to find criteria to classify benchmarks based on their characteristics and demonstrate what benefit a lightweight infrastructure could bring to speed up FI.

The plot in Fig. 1 shows some of the aforementioned properties derived from an FI campaign with *MiBench* [2] and *TACLeBench* [1], [4]. As a preliminary study, it includes the number of dynamic instructions, the number of memory-access locations<sup>1</sup>, and the *Silent Data Corruption* (SDC) count. From a bird’s eye view, the benchmarks fill in three quadrants, and exhibit some clustering patterns. This may suggest that 1) we are missing benchmarks that have very high data throughput (e.g., using SIMD instructions), 2) some benchmarks may be overlapping in terms of their fault-space characteristics, and 3) the outliers may be interesting targets.

Interesting program characteristics beyond those shown are, e.g., stack and heap usage, branching behavior, or memory-access granularity.

To highlight the relevance of program granularities, we measured the runtimes of a subset of *MiBench* benchmarks compiled (1) in a full-system setting with the *eCos* [6] OS, and (2) in a bare-metal setting based on *picolibc* [5]. The results show that for the *MiBench* benchmarks *dijkstra*, *susan*, *crc* and *cutcp*, the *eCos* variants have 37%–128% more dynamic instructions. The number of dynamic memory accesses is increased by the same order of magnitude. In the same experiments, the FI results are also vastly different across the two variants: Notably the full-system setting is more prone to failure modes such as timeout. For example, the *eCos* variant of *crc* has 97% more SDCs, 801% more timeouts and 1491% more CPU exceptions compared to the *picolibc* variant.

### IV. CONCLUSION

To start a discussion on a potential dedicated FT/FI-specific benchmarking suite, we demonstrated how this domain could

<sup>1</sup>The number of unique memory locations where at least one memory read or write event is recorded during the FI campaign.

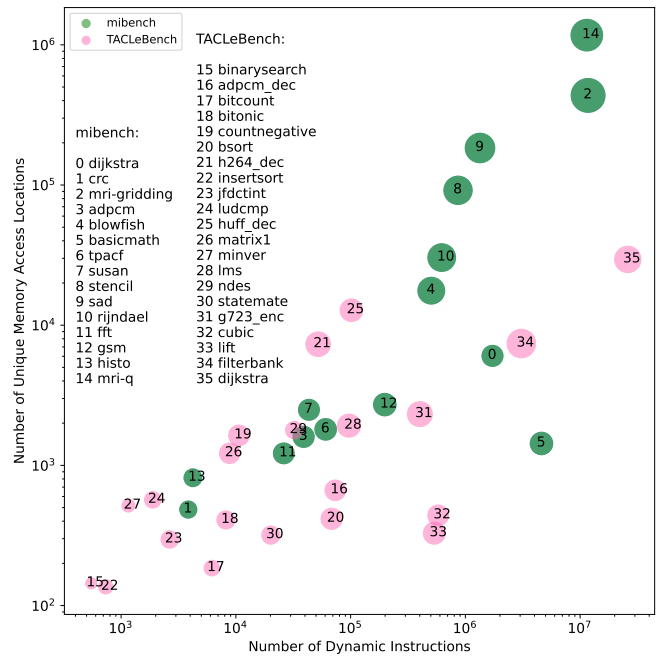


Fig. 1. Number of dynamic instructions, unique memory-access locations and SDCs (circle sizes) of *MiBench* [2] benchmarks compiled with *picolibc* [5] and *TACLeBench* [1] as used by Borchert et al. [4], measured with the *FAIL\** fault-injection framework.

benefit. Such a benchmarking suite should have a full coverage in terms of both practical applicability and program characteristics, include benchmarks in different granularities, ship a self-contained and lean runtime, and expose rich configurability. Nevertheless, it still remains an open question what the most relevant program properties for FT/FI would be.

### REFERENCES

- [1] H. Falk, S. Altmeyer, P. Hellinckx, et al., “TACLeBench: A benchmark collection to support worst-case execution time research,” in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, M. Schoeberl, Ed., ser. OpenAccess Series in Informatics (OASiCs), vol. 55, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016, 2:1–2:10. DOI: 10.4230/OASiCs.WCET.2016.2.
- [2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, “MiBench: A free, commercially representative embedded benchmark suite,” in *IEEE Int. W’shop on Workload Characterization (WWC ’01)*, Washington, DC, USA: IEEE, 2001, pp. 3–14, ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.15.
- [3] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk, “FAIL\*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance,” in *11th Europ. Depend. Comp. Conf. (EDCC ’15)*, (Paris, France), Piscataway, NJ, USA: IEEE, Sep. 2015, pp. 245–255. DOI: 10.1109/EDCC.2015.28.
- [4] C. Borchert, H. Schirmeier, and O. Spinczyk, “Compiler-implemented differential checksums: Effective detection and correction of transient and permanent memory errors,” in *53rd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN ’23)*, (Porto, Portugal), Piscataway, NJ, USA: IEEE, Jun. 2023. DOI: 10.1109/DSN58367.2023.00021.
- [5] Keith Packard, *Picolibc: C Libraries for Smaller Embedded Systems*, https://keithp.com/picolibc/, 2023.
- [6] A. Massa, *Embedded Software Development with eCos*. Upper Saddle River, NJ, USA: Prentice Hall, 2002, ISBN: 0130354732.