Antipattern-Based Problem Injection for Assessing Performance and Reliability Evaluation Techniques

Philipp Keck, André van Hoorn, Dušan Okanović, Teerat Pitakrat, and Thomas F. Düllmann
University of Stuttgart, Institute of Software Technology, Reliable Software Systems Group
Universitätsstraße 38, 70569 Stuttgart, Germany
{van.hoorn, okanovic, pitakrat}@informatik.uni-stuttgart.de, {keckpp, duellmtk}@studi.informatik.uni-stuttgart.de

***Abstract*—A challenging problem with today's increasingly large and distributed software systems is their performance behavior. To help developers avoid or detect mistakes that lead to performance problems, many researchers in software performance engineering have come up with classifications of such problems, called antipatterns. To test the approaches for antipattern detection, data from running systems is required. However, the usefulness of this data is doubtful as it may or may not include manifestations of performance problems. In this paper, we classify existing performance antipatterns w.r.t. their suitability for being injected and, based on this, introduce an extensible tool that allows to inject instances of these antipatterns into existing applications. The approach can be useful for researchers to test and validate their automated runtime problem evaluation and prevention techniques. Using two exemplary performance antipatterns, it is demonstrated that the injection is easily possible and produces feasible, though currently rather clinical results.**

I. Introduction

The performance of a software application directly influences user satisfaction and operational costs, and therefore the overall profit. However, some performance problems can be hard to detect or even cannot be detected in the testing phase. They can be triggered later, only when software is deployed, or require time and/or special conditions to manifest themselves, e.g., as part of effects known as software aging [6, 15]. Research shows that there are some common software design and implementation mistakes, called *performance antipatterns* that cause performance problems [7, 9, 10, 36, 40, 42]. Some examples of antipatterns are "Ramp" — an increase in task duration or resource consumption over time, "One Lane Bridge" — an execution stage that can be passed only by a single thread (or small number of threads) concurrently, and "Traffic Jam" — when a slow execution of a single task causes further tasks to pile up.

While many performance problem detection [22], diagnosis [17, 33, 47], prediction [21, 34], and prevention [19, 15] techniques have been developed, it is very hard to evaluate them. It is common to use either specific benchmarks [33, 47], generated applications [20], or case studies that involve some real applications [12, 13, 22], where the results are compared to what experienced developers/testers would have expected. Each of these evaluation approaches has its own advantages and drawbacks: benchmarks and toy applications are created only for the purpose of testing and might therefore be unrealistic. Creating random sample applications tackles the problem of the approach's validity, but it is limited to very simple applications. Real applications are by definition realistic, but due to, e.g., confidentiality restrictions, unless they are open-source, their usability is limited. Unlike sample applications created for testing, real applications might feature (often a limited number of) performance problems (if any), and there is uncertainty that the problem will be activated and manifests itself.

Therefore, the injection of faults into real application has become an interesting field of research. Different tools and approaches have been developed to inject faults in software systems, both functional [27, 38] and non-functional [3, 14, 21], and investigating their behavior. Not only can faults be turned on and off, they can also be injected without access to the source code. The recent survey by Natella et al. [29] evaluates different techniques and approaches for software fault injections in dependability assessment. Three types of problem injections are covered. *Data error injection* performs corruption of memory and/or registers. *Interface error injection* corrupts input and output values at component interfaces. *Injection of code changes* introduces the code that mimics typical bugs. The latter type is of interest in this paper.

The contribution of this paper is twofold. First, we propose an alternative classification of performance antipatterns that focuses on the suitability for injection. Second, we propose the *PPInject* framework for the injection of performance and reliability problems into (object-oriented) applications, without the need to modify its source code. The framework is extensible and allows the implementation of additional performance antipatterns and injecting them into real applications. We evaluate our tool by implementing two typical antipatterns related to software aging, namely the Ramp and the One Lane Bridge antipatterns. As stated before, the Ramp represents an increase in resource consumption or task duration over time, while the One Lane Bridge can cause congestion over time and slowdown in task execution.

The remainder of this paper is structured as follows. Section II presents the classification of antipatterns. Sections III and IV describe *PPInject* and its evaluation. In Section V we discuss related work. Section VI draws the conclusions and outlines future works. *PPInject* and the evaluation scenarios are publicly available under an open-source license.[1]

[1] https://github.com/diagnoseIT/diagnoseIT.ppinject

To appear in Proceedings of the IEEE 27th International Symposium on Software Reliability Engineering Workshops (ISSREW 2016), 8th International Workshop on Software Aging and Rejuvenation (WoSAR 2016)

II. Injection-Specific Antipattern Classification

Smith and Williams present a number of typical classes of performance antipatterns (e.g., [40, 41]). While they are similar to normal design patterns as they "document recurring solutions to common design problems" [40], their impact is negative. In addition to their work, there are also works on identifying and classifying typical problems in the area of specific technologies like Java (EE) [9, 33, 42, 43].

To allow easier implementation of detection approaches, some authors propose the classification of antipatterns. For example, both Parsons et al. [33] and Wert et al. [47] introduced hierarchal organizations of performance problems by their symptoms.

While the mentioned classifications distinguish antipatterns by their symptoms, our does this by the system scope in which they were introduced and whether or not they can be injected. Also, note that this classification is not disjunctive. Some antipatterns can appear in different classes as they can have different causes. For example, One Lane Bridge can be a local (unnecessary locks), micro-architecture (improper database structure) or deployment problem (undersized pools).

1) Local implementation problems: A performance problem is local if it affects only a single method. Many methods and even the entire application may contain the problem, but only a single method needs to be incorrect for a single instance of the problem. The problem has probably been caused by a bad decision of the programmer during the implementation of the method. Some antipatterns that fit into this group are Ramp, Empty Semi Trucks, One Lane Bridge, and Spin Wait.[2] Local problems are injected by modifying a single method.

2) Micro-architecture problems: These problems are introduced by the detailed design of the software system. They usually affect one or few classes. Because they affect multiple parts of the application code, these problems cannot be injected directly. However, the characteristic behavior of these problems can be analyzed and replicated in addition to the normal functionality of the modified methods. Example antipatterns in this group are Treasure Hunt, Ramp, One Lane Bridge, and Session as a Data Store.

3) Macro-architecture problems: These problems are introduced during the design phase and affect the overall structure of the system. For example, all the work is performed in a single ("God") class, although, e.g., a pipe-and-filter architecture would have been more suitable. These problems cannot be injected or simulated because the responsible classes or modules would not even exist in a correctly designed system. They have to be detected long before the implementation, and therefore are not detected (and injected) in the source code or the running application. Beside "God" Class, other antipatterns from this group are Traffic Jam and Unbalanced Processing.

4) Deployment problems: Even if an application has been developed correctly, its performance can be impaired by incorrect deployment. For example, the parts of a multi-component application could be improperly distributed over multiple nodes, or pools can be undersized by configuration. Note that the way an application is deployed can also increase the negative effects of one of the above problems. For example, the Round Tripping problem only causes a perceivable slowdown if the unnecessary calls need to be sent over a remote connection. Simulation of deployment performance problems using code injection obviously makes no sense, which is why they are not further considered in this paper. Other examples in this group are Falling Dominoes, One Lane Bridge, and More is Less.

This classification may be useful for application developers, as they need the antipatterns to be organized along their development process in order to consider them in the right situation. For example, local implementation problems should be considered when implementing a new, non-trivial method, micro-architecture problems when developing a group of methods or a class, macro-architecture problems should be considered in the design phase, and deployment problems when configuring an application.

In this paper, we implemented one problem from each of the first two groups prototypically. Local implementation problems have a root cause that can be injected directly. Micro-architecture problems can be more difficult to inject, but it is at least possible to inject code that emulates the characteristic effects of these problems. As stated above, macro-architecture and deployment problems are not considered in this paper, as they are introduced during architecture design or application deployment, respectively.

III. Performance Problem Injection

In Section III-A, we consider the requirements for an injection framework. Then, we propose a framework design in Section III-B.

A. Requirements

For the validation of new detection techniques, a performance problem injection library should satisfy a number of properties (cf. [29]):

- Using this library, it should be possible to implement individual performance problems with only a few lines of code. The library itself should already provide universal injection and management code. In other words, the library's architecture should support **extensibility**.
- Injection should be possible at both **method and class level**, depending on the respective performance problem.
- Integration of the library should be **unobtrusive** and easy to handle. In particular, the source code of the target application may not be available, so the library must be attached to either the compiled or the running application.
- It should be possible to inject and remove performance problems **dynamically**, or at least to activate and deactivate an injected performance problem. Because the application itself cannot be changed, this implies that some kind of external **configuration interface** is required to control the injection.

[2] Antipattern names are taken from [40] and [41]

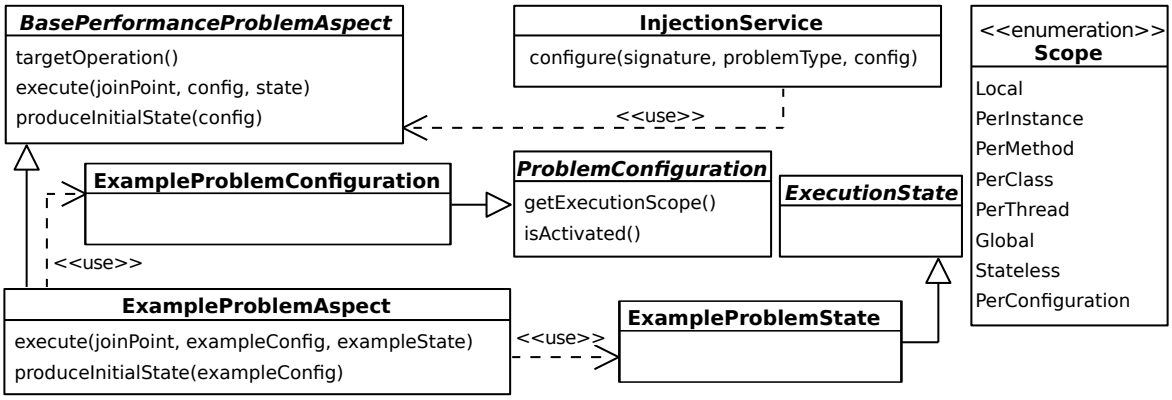


Fig. 1: Class diagram of the implemented framework interface

- Injected performance problems should be **configurable** dynamically regarding their **intensity** and behavior.
- The **performance overhead** of the library itself should be negligible, especially when the injected performance problems are deactivated.
- The library only needs to support **Java**-based applications.

B. Design of the Library

1) Framework: The part of the framework depicted in Figure 1 features base classes for specific antipattern implementations and their configurations and states. Through the `InjectionService`, injected antipatterns can be configured. Antipatterns implement the (abstract) class `BasePerformanceProblem`, which ensures that the `execute()` method is always called with the correct configuration and state objects, so that the actual problem implementation is free from boilerplate code.

To meet the requirements regarding unobtrusiveness, dynamic activation and performance overhead, we chose to make use of the Aspect-Oriented Programming (AOP) paradigm [24]. AOP is also commonly used, for instance, for program instrumentation in dynamic software analysis and performance monitoring [46]. Antipatterns are implemented using aspects, while the library provides infrastructure for their injection. Since the AOP *join point*, at which an injection can be performed, is usually a method call or a field access, method and class level injection is technically possible. While it may be possible to derive heuristics from production experience, this is not the focus of the current work and we assume that the user of the library will define the correct pointcuts.

An aspect that implements an antipattern (in this case `ExampleProblemAspect`) can derive its own versions of `ProblemConfiguration` and/or `ExecutionState` to retrieve/store additional information. The configuration stores static information for a particular performance problem instance like the number of available "lanes" (One Lane Bridge) or the slope of the "ramp" (Ramp). The configuration also determines the execution scope of an injected problem instance, as detailed in Section III-B2. The `ExecutionState` is then used to store runtime data, such as *"How many lanes are still available?"* or "*How far up the ramp the application has already gone?*".

By providing multiple `ProblemConfiguration` instances for different pointcuts and/or by specifying fine-grained scopes, multiple simultaneous instances of a perceived performance problem can be injected. For example, the "One Lane Bridge" antipattern, might affect all methods of a class, e.g., if they were all `synchronized`, or only those with database accesses (because the connection pool is limited).

2) Execution Scopes: A configuration item that is always required is the execution scope of the injected performance problem. While local problems (see Section II-1) are mostly stateless, the micro-architecture performance problems (see Section II-2) have to behave differently in subsequent executions. To define which parts of the application are affected by a *single instance* of the performance problem (as multiple problems can be injected independently), we use the following scopes:

- **Local**: The problem is local to a single method of an object.
- **Per-Instance**: The problem affects all instrumented methods of an object instance.
- **Per-Method**: The problem affects a single method of all instances of its class.
- **Per-Class**: The problem affects all instances of a class.
- **Per-Thread**: The problem affects all instrumented methods in all classes, but only when executed within a single thread.
- **Global**: The problem affects all instrumented methods in all classes.
- **Stateless**: The problem affects all instrumented methods in all classes, but there is no connection between subsequent executions of the problem.
- **Per-Configuration**: The problem affects all instrumented methods which are activated by the configuration for the problem instance.
- **Other scopes** are also imaginable, but not considered in the current design.

Consider the "Ramp" antipattern, which is one of the major

causes of software aging, as an example. It can occur, for example, because a single method uses an inefficient cache. The class in which the method is implemented might be instantiated multiple times. Therefore, the execution scope is *per-class* if the cache is `static`, or *per-instance* if not. On the other hand, the "One Lane Bridge" antipattern can be caused by an undersized database connection pool or connection leaks. This affects all the data access methods and all the connection instances, so multiple methods have to be targeted. However, as there is only a single global connection pool, this is considered a *global* problem.

3) Activation: Antipatterns are activated by specifying signatures of classes to which an antipattern should be injected, with the possibility of using wild-cards. Activation based on instances or threads would be imaginable, too, but this would be almost impossible to configure from outside because instances and threads are not available or cannot be referenced there.

Instead of mapping to a boolean value indicating the activation, the signature is mapped to a configuration object. A missing configuration indicates that the aspect is deactivated.

IV. Evaluation

As an example, we implemented a prototype of the proposed *PPInject* injection framework, as well as two injectable performance antipatterns. First we show the implementation of the Ramp (Section IV-A), which is implemented by injection of symptoms. Next, we present the implementation of the One Lane Bridge, which is implemented using injection of problematic code (Section IV-B).

A. Ramp

The Ramp antipattern implementation emulates a method's increasing response time.

During each execution, the time t that the execution took is measured and a delay of $i \cdot \alpha \cdot t$ is added to the i-th execution, where α is the ramp factor/slope (usually $<< 1$). By letting the freshly measured runtime t affect the delay time, the natural behavior of the underlying method is carried through. For clarity, the implementation is only presented as pseudo code:

```
int i = increaseAtomicCounter();
executeRealMethod(); -> measuredDuration,
    result
sleep(measuredDuration * alpha * i);
reproduceResult(result);
```

B. One Lane Bridge

The implementation of the One Lane Bridge antipattern uses a semaphore to let only the configured number of threads pass at a time. The number of threads is provided by configuration.

The following listing provides an excerpt of the aspect that implements this antipattern:

```
static class Config
extends ProblemConfiguration {
  int numberOfLanes;
}

static class State
extends ExecutionState {
  Semaphore semaphore = new
      Semaphore(config.numberOfLanes, true);
}

public Object execute(.. joinPoint, Config
    config, State state) {
  state.semaphore.acquire();
  try {
    return joinPoint.proceed();
  } finally {
    state.semaphore.release();
  }
}
```

C. Evaluation Results

For the evaluation, the MyBatis[3] JPetStore on a Jetty server was used, with JMeter[4] as a load generator. As an example, we instrumented one method (`CatalogAction-Bean#viewProduct()`) and increased its runtime by adding `Thread.sleep(1000)` to simulate a long-running method.

To evaluate the implementation of One-/n-Lane Bridge, we measured the response time of the method while varying the number of active threads. The tests lasted for 2 minutes, with a ramp-up from 1 to 12 threads. We discarded the very first measurement (with one thread) as well as the measurements during ramp-down.

Without any additional performance problems, the method behaves like a normal long-running method (see Figure 2a). When only one thread can enter the method at a time (a classic One Lane Bridge), the response times increase linearly as more threads are added (Figure 2b). When five lanes are available (*5-Lane Bridge*), the response times stay at $1000\,\mathrm{ms}$ until more than five threads are active, and then it increases linearly (Figure 2c).

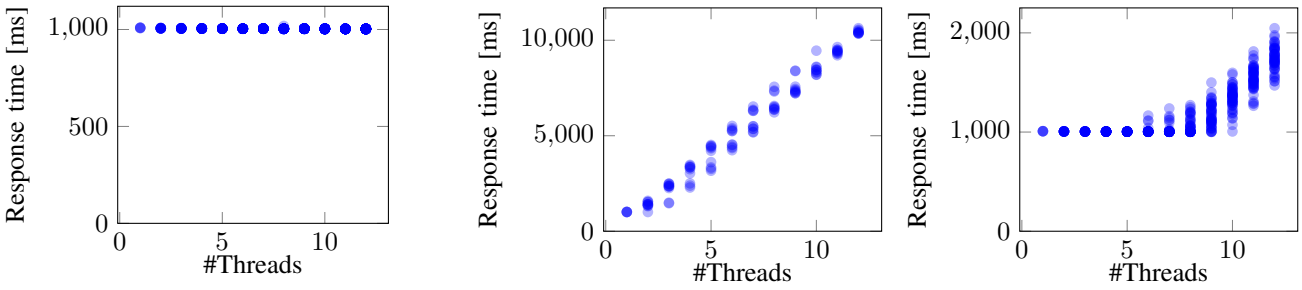
For the Ramp antipattern, the number of threads was constant (10) and the experiment was again run for 2 minutes. The results show the increase in the method's response time, which is typical for the Ramp antipattern (Figure 3).

D. Known Issues and Future Work

As discussed by Natella et al. in their survey [29], there are three important aspects of each fault injection tools: *usability*–the ability to use it in a new system, *representativeness*–the ability to represent real faults, *efficiency*–the ability to achieve results with reasonable effort.

Regarding the *usability*, using our library, we have shown that antipatterns can be implemented using fairly simple aspects. However, with the current implementation, all aspects

[3]http://mybatis.org/
[4]http://jmeter.apache.org/



(a) Method runtime without injected problems

(b) 1 Lane

(c) 5 Lanes

Fig. 2: Method runtime with One Lane Bridge

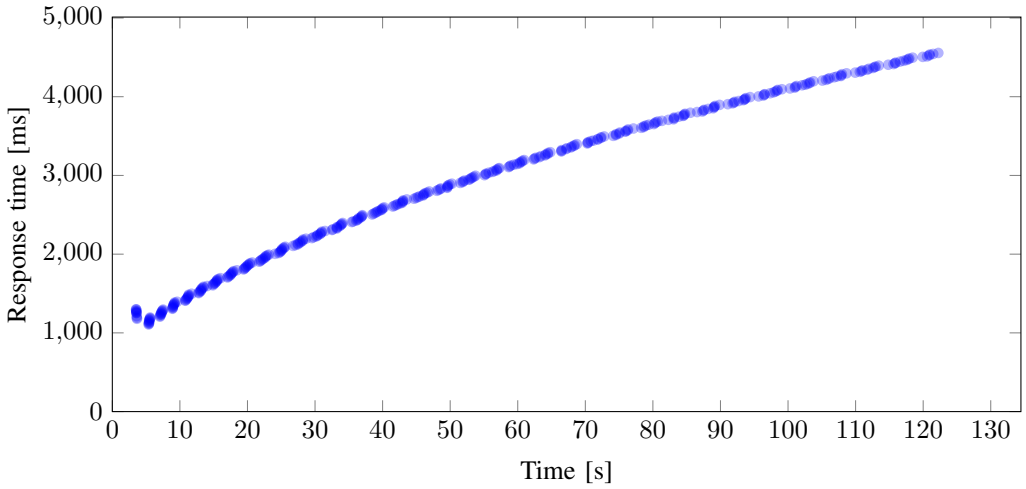


Fig. 3: Method runtime with the Ramp and $\alpha = 0.01$

are woven into the bytecode and are thus permanently injected into the running application. They need to be deactivated when they are not needed. Each time an advice is executed, it needs to check if it is activated and retrieve the corresponding configuration, which leads to an overhead, and it can only be avoided by dynamically injecting the problems. Wert et al. [47] use dynamic instrumentation with Javassist and HotSwap for measuring and monitoring performance. A similar approach could be used for dynamic injection.

Representativeness and *efficiency* highly depend on the antipattern implementation. We only implemented two simple antipatterns so far. Providing realistic implementations for the other antipatterns is more difficult, especially for the micro-architecture problems. A general drawback of problem injection is its artificiality: while the problems are injected into a realistic application and may also produce realistic side-effects and interferences, the injected problem itself is still artificial, and poses a threat to validity of an experiment. With functional faults, this is less of a problem because a failure is a failure, regardless of its origin. Micro-architecture performance problems can only be injected by emulating the symptoms, which are usually increased response time or increased memory consumption. Both of these symptoms are not binary, rather a function of time and other variables. When these are injected, the results (see Figure 3) look rather clinical. This could be compared to real instances of these performance problems to make the injected instances more realistic.

V. Related Work

Fault injection has been subject of research for a long time [29]. The main motivation is to test the dependability of a system in the presence of faults [18]. Both on the hardware level (using radioactive substances such as Californium [16]) and the software level [4, 25], faults were injected to simulate faulty behavior of processors and other hardware components [5]. Techniques for fault injection at software level include source code modifications [37], changing the compiled code, manipulations at the operating system layer [44], using special processor debugging features [8], altering the execution state from a second process [23], as well as computational reflection [26] or bytecode manipulation [27, 38].

Most of the previously mentioned approaches intend to simulate hardware faults and then test the system's robustness. Although the technical means to achieve the injection are similar, this needs to be distinguished from the (realistic) simulation of mistakes in the code that a programmer could

have made. The latter is the goal of this work and has also been covered in the literature [11, 28, 31, 45].

The idea to build an extensible framework for fault injection is not new. For example, the tools GOOFI [2], Jaca [27] and J-SWFIT [38] are clearly designed to be reusable and extensible. However, these tools provide functional, not performance-problem-based code injection.

Modern enterprise applications and cloud systems usually use container deployment. Fault injection systems for these platforms are either limited in functionality, e.g., ChaosMonkey is only for shutting down VMs in AWS, or have high resource overhead, e.g., Cocoma [35]. Sheridan et al. [39] propose a tool that solves the resource overhead issue and provides different failure scenarios, but still only on the VM level. It can be observed that there is a trend in designing the tools that allow fault injection at runtime, however they focus on system level problems.

Irrera and Vieira [21] and Pitakrat et al. [34] used fault injection to generate failure data in the context of online failure prediction. Alonso et al. [3] and Gross et al. [14] injected memory leaks to validate crash prediction algorithms or software-aging detection, respectively. To test the capabilities of APM tools in detecting performance regressions, Ahmed et al. [1] inject ineffective resource usage, high CPU utilization and inefficient database access. These approaches used proprietary fault injection, developed for their specific evaluation use cases, while the goal of *PPInject* is to be a reusable and extensible injection framework.

As to our knowledge, no research has been conducted on realistically injecting typical performance antipatterns.

VI. Conclusions

In this paper, we presented the design of the extensible *PPInject* framework for performance problem injection based on known antipatterns, and tested its capabilities on a proof-of-concept implementation. The framework is designed to help researchers validate their newly developed approaches for performance problem detection, diagnosis, prediction, and prevention. Problems can be injected by either injecting a root cause, or by emulating the runtime behavior that a real problem would have with respect to, e.g., response times and memory consumption. We have implemented and shown how two typical performance antipatterns can be injected into the sample application. Additionally, we proposed a classification of performance problems by the scope in which they are introduced. In the future, our goal is to implement injectors for other antipatterns, and to extend *PPInject* to serve as a part of a community benchmark suite for assessing performance and reliability evaluation techniques. Another part of our future research is to investigate fault injection to microservice architectures [30], e.g., based on the respective (anti)patterns for these types of systems [32].

It is important to note that presented concepts do not limit the implementation of the library to Java. There are AOP implementations for other platforms, that would allow porting the library to them.

Acknowledgement

This work is being supported by the German Federal Ministry of Education and Research (grant no. 01IS15004, diagnoseIT), and by the Research Group of the Standard Performance Evaluation Corporation (SPEC, http://research.spec.org).

References

[1] T. M. Ahmed, C.-P. Bezemer, T.-H. Chen, A. E. Hassan, and W. Shang. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: An experience report. In *Proc. 13th Int. Con. on Mining Software Repositories (MSR '16)*, pages 1–12, 2016.

[2] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN '01)*, pages 83–88, 2001.

[3] J. Alonso, J. Torres, and R. Gavaldà. Predicting web server crashes: A case study in comparing prediction algorithms. In *Proc. 5th Int. Conf. on Autonomic and Autonomous Systems (ICAS '09)*, pages 264–269, 2009.

[4] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. on Software Engineering*, 16(2):166–182, 1990.

[5] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Trans. on Computers*, 52(9):1115–1133, 2003.

[6] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[7] S. Boroday, A. Petrenko, J. Singh, and H. Hallal. Dynamic analysis of Java applications for multithreaded antipatterns. In *Proc. 3rd Int. Workshop on Dynamic Analysis (WODA '05)*, pages 1–7.

[8] J. Carreira, H. Madeira, and J. G. Silva. Xception: Software fault injection and monitoring in processor functional units. *Dependable Computing and Fault Tolerant Systems*, 10:245–266, 1998.

[9] B. Dudney. *J2EE antipatterns*. Wiley, 2003.

[10] R. F. Dugan, Jr., E. P. Glinert, and A. Shokoufandeh. The sisyphus database retrieval software performance antipattern. In *Proc. 3rd Int. Workshop on Software and Performance (WOSP '02)*, pages 10–16, 2002.

[11] J. A. Duraes and H. S. Madeira. Emulation of software faults: A field data study and a practical approach. *IEEE Trans. on Software Engineering*, 32(11):849–867, 2006.

[12] K. Foo, Z. M. Jiang, B. Adams, A. Hassan, Y. Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Proc. 10th Int. Conf. on Quality Software (QSIC '10)*, pages 32–41, 2010.

[13] M. Grechanik, C. Fu, and Q. Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proc. 34th Int. Conf. on Software Engineering (ICSE '12)*, pages 156–166, 2012.

[14] K. Gross, V. Bhardwaj, and R. Bickford. Proactive detection of software aging mechanisms in performance critical computers. In *Proc. 27th Annual NASA Goddard/IEEE Software Engineering Workshop (SEW-27'02)*, pages 17–23, 2002.

[15] M. Grottke, R. Matias, and K. Trivedi. The fundamentals of software aging. In *IEEE Int. Conf. on Software Reliability Engineering Workshops (ISSRE '08)*, pages 1–6, 2008.

[16] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation.

In *Proc. 19th Int. Symposium on Fault-Tolerant Computing (FTCS '89)*, pages 340–347, 1989.

[17] C. Heger, A. van Hoorn, D. Okanović, S. Siegl, and A. Wert. Expert-guided automatic diagnosis of performance problems in enterprise applications. In *Proc. 12th European Dependable Computing Conf. (EDCC '16)*. IEEE, 2016. To appear.

[18] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[19] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton. Software rejuvenation: Analysis, module and applications. In *Proc. 25th Int. Symposium on Fault-Tolerant Computing (FTCS '95)*, pages 381–390, 1995.

[20] I. Hussain, C. Csallner, M. Grechanik, Q. Xie, S. Park, K. Taneja, and B. M. Mainul Hossain. Rugrat: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Software: Practice and Experience*, 46(3):405–431, 2016.

[21] I. Irrera and M. Vieira. A practical approach for generating failure data for assessing and comparing failure prediction algorithms. In *Proc. IEEE 20th Pacific Rim Int. Symposium on Dependable Computing (PRDC '14)*, pages 86–95, 2014.

[22] Z. M. Jiang. Automated analysis of load testing results. In *Proc. 19th Int. Symposium on Software Testing and Analysis (ISSTA '10)*, pages 143–146, 2010.

[23] G. Kanawati, N. Kanawati, and J. Abraham. FERRARI: a flexible software-based fault and error injection system. *IEEE Trans. on Computers*, 44(2):248–260, 1995.

[24] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. *Aspect-oriented programming*. Springer Berlin Heidelberg, 1997.

[25] H. Madeira, M. Z. Rela, F. Moreira, and J. G. Silva. RIFLE: A general purpose pin-level fault injector. In *Proc. 1st European Dependable Computing Conf. (EDCC '94)*, pages 199–216, 1994.

[26] E. Martins and A. Rosa. A fault injection approach based on reflective programming. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN '00)*, pages 407–416, 2000.

[27] E. Martins, C. Rubira, and N. Leme. Jaca: A reflective fault injection tool based on patterns. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN '02)*, pages 483–487, 2002.

[28] R. Natella, D. Cotroneo, J. Duraes, and H. Madeira. On fault representativeness of software fault injection. *IEEE Trans. on Software Engineering*, 39(1):80–96, 2013.

[29] R. Natella, D. Cotroneo, and H. S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3):44:1–44:55, 2016.

[30] S. Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[31] W. T. Ng and P. M. Chen. The design and verification of the Rio file cache. *IEEE Trans. on Computers*, 50(4):322–337, 2001.

[32] M. Nygard. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007.

[33] T. Parsons and J. Murphy. A framework for automatically detecting and assessing performance antipatterns in component based systems using run-time analysis. In *Proc. 9th Int. Workshop on Component Oriented Programming (part of ECOOP)*, 2004.

[34] T. Pitakrat, D. Okanović, A. van Hoorn, and L. Grunske. An architecture-aware approach to hierarchical online failure prediction. In *Proc. 12th Int. ACM SIGSOFT Conf. on the Quality of Software Architectures (QoSA '16)*. IEEE, 2016.

[35] C. Ragusa, P. Robinson, and S. Svorobej. A framework for modeling and execution of infrastructure contention experiments. In *Proc. 2nd Int. Workshop on Measurement-based Experimental Research, Methodology and Tools (MERMAT 2013)*, 2013.

[36] D. Rayside and L. Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proc. 22nd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE '07)*, pages 194–203, 2007.

[37] H. Rosenberg and K. Shin. Software fault injection and its application in distributed systems. In *Proc. 23rd Int. Symposium on Fault-Tolerant Computing (FTCS '93)*, pages 208–217, 1993.

[38] B. Sanches, T. Basso, and R. Moraes. J-SWFIT: A java software fault injection tool. In *Proc. 5th Latin-American Symposium on Dependable Computing (LADC '11)*, pages 106–115, 2011.

[39] C. Sheridan, D. Whigham, and M. Artač. Dice fault injection tool. In *Proc. 2nd Int. Workshop on Quality-Aware DevOps (QUDOS '16)*, pages 36–37, 2016.

[40] C. U. Smith and L. G. Williams. Software performance antipatterns. In *Proc. 2nd Int. Workshop on Software and Performance (WOSP '00)*, pages 127–136, 2000.

[41] C. U. Smith and L. G. Williams. Software performance antipatterns: Common performance problems and their solutions. In *Proc. 27th Int. Computer Measurement Group Conf. (CMG '01)*, pages 797–806, 2001.

[42] B. Tate, M. Clark, and P. Linskey. *Bitter EJB*. Manning Publications Co., 2003.

[43] B. A. Tate. *Bitter Java*. Manning Publications Co., 2002.

[44] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the FTAPE fault injection tool. In *Proc. 8th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems (MMB '95)*, pages 26–40, 1995.

[45] E. van der Kouwe, C. Giuffrida, and A. S. Tanenbaum. Finding fault with fault injection: An empirical exploration of distortion in fault injection experiments. *Software Quality Journal*, pages 1–30, 2014.

[46] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proc. 3rd ACM/SPEC Int. Conf. on Performance Eng. (ICPE '12)*, pages 247–248, 2012.

[47] A. Wert, J. Happe, and L. Happe. Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In *Proc. 2013 Int. Conf. on Software Engineering (ICSE '13)*, pages 552–561, 2013.