A Multi-objective Framework for Effective Performance Fault Injection in Distributed Systems

Luca Traini
University of L'Aquila, Italy
luca.traini@graduate.univaq.it

ABSTRACT

Modern distributed systems should be built to anticipate performance degradation. Often requests in these systems involve ten to thousands Remote Procedure Calls, each of which can be a source of performance degradation. The PhD programme presented here intends to address this issue by providing automated instruments to effectively drive performance fault injection in distributed systems. The envisioned approach exploits multi-objective search-based techniques to automatically find small combinations of tiny performance degradations induced by specific RPCs, which have significant impacts on the user-perceived performance. Automating the search of these events will improve the ability to inject performance issues in production in order to force developers to anticipate and mitigate them.

CCS CONCEPTS

• **Software and its engineering** → *Software performance*; *Search-based software engineering*;

KEYWORDS

Fault Injection, Software Performance, Search-Based Software Engineering, Distributed Systems

ACM Reference Format:
Luca Traini. 2018. A Multi-objective Framework for Effective Performance Fault Injection in Distributed Systems. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3238147.3241535

1 INTRODUCTION

The current "fast to market" trend has shaped a new way of design, build and release software. Successful high-tech companies deliver new software in production every day [7] and perceive this capability as a key competitive advantage [18]. In order to support this fast-paced release cycle, IT organizations often employ several independent teams that are responsible "from development to deploy" [17] of loosely coupled independently deployable services [16]. Despite their benefits, these architectures introduce complexity in performance monitoring and troubleshooting. Each request involves ten to thousand Remote Procedure Calls (RPCs), each of which can introduce some form of performance degradation.

Performance is a key quality aspect of software and has a huge impact on user engagement and satisfaction [5]. In the last few years several tools have been developed in order to enhance developers awareness on the performance behavior of distributed systems [13, 19]. Although these tools can be useful for system performance debugging, there is lack of techniques to anticipate unexpected events that could significantly hamper the user perceived software. Predicting performance issues upfront can be difficult, given the continuously changing load nature [3] and frequent roll out of software changes. High-tech companies automate failure injection in production to build confidence in systems behavior and anticipate unexpected events [4]. The combinatorial space of potential failures is too large to be explored exhaustively, hence in practice failure testing solutions rely on two search policies: random search and programmer-guided search. The former randomly chooses failures, in favor of speed and simplicity, but it is unlikely to discover interesting combinations of failures. The latter leverages domain expert intuitions to build heuristics capable to find more interesting failures scenarios, but it is fundamentally unscalable. Recent work from Alvaro et al. [1] have showed promising results in guiding the search through a more structured policy.

The envisioned approach proposed in this paper provides a framework, which turns the problem of finding "interesting combinations" of performance issues into a multi-objective search problem. The rationale at the basis of this approach is that the concept of "interesting combination" includes several objective often in tradeoff. For example, if we inject a huge latency in all the services involved in all user requests, then this certainly has a negative *impact* on the system, but it is not interesting, because it is obvious and is not likely to happen in real systems. Less evident combinations that may lead to performance degradations are more interesting and difficult to detect. The key idea is to search for combinations of performance issues by minimizing the number of issues introduced (we will term it *cardinality*), and their *intensity* (e.g. the amount of latency introduced) while maximizing the *impact* on the user-perceived performance. Clearly the higher will be number of issues introduced and their *intensity* the higher will be the *impact*. By using a Pareto-optimal search-based approach we will not sacrifice solutions with high *cardinality* and/or *intensity* when they have a significant *impact*, but they will be always dominated by combinations with lowers values of *cardinality*/*intensity* when they cause the same *impact*.

This research was supported by the Electronic Component Systems for European Leadership Joint Undertaking through the MegaMart2 project (grant agreement No 737494).

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3241535

We envisage that automated search with appropriate objectives can help in identifying interesting combinations of performance issues which have significant impact on the user-perceived performance.

2 ENVISIONED APPROACH

2.1 Approach

The approach is inspired by the work of Alvaro et al. [1], which is a first attempt to guide the search of combinations of fault injection with a structured policy [2]. This work is related to functional aspects of distributed systems, whereas the approach proposed here focuses on performance aspects. The goal is to provide automation to fault injection in distributed systems by improving the search of interesting combinations of performance issues. In this paper we introduce a conceptual framework, on top of which customized *instantiations* of the approach can be built. The framework is based on several components, where the semantics and the way they interact will be described in the following. Each *instantiation* of our approach is obtained by implementing these components in a specific context. An *instantiation* defines the kind of targeted performance issues and the way on which their "interesting" combinations will be searched. In order to build an *instantiation*, five core questions have to be addressed:

(1) **Which kinds of performance issues are *targeted*?**
Distributed systems can experience different kinds of performance issues. Network latency spikes, performance bugs [12], hardware failures are only few examples. An *instantiation* of our approach has to choose one or more type of performance issues as *target*.

(2) **How do we model the *intensity* of combinations of performance issues?**
Different types of performance issues have different properties. Some of these properties define the severity of performance issues, for example in network latency spikes a property could be the latency introduced and/or the percentage of traffic affected by the problem. An *instantiation* has to provide a way to synthesize interesting properties of targeted performance issues in a single value. Hence, given any possible combination of performance issues, the approach has to derive this value, which will be named *intensity*. Intuitively, by following previous examples, higher latencies or more percentage of affected traffic will induce higher *intensity*.

(3) **What are the *impacts* we are interested to?**
Performance issues can have different *impacts* on the system. Some can be strictly related to the user-perceived performance properties, others can be correlated to inner characteristics of the system, such as the energy consumption of servers or the cost of cloud resources. Our approach is devised to simultaneously support different types of *impacts*, and an *instantiation* has to define what are the interesting ones.

(4) **How do we estimate the *impact* of combinations of performance issues?**
The *intensity* of a combination of performance issues is not always strictly correlated with their *impact* on the system performance. In contexts where requests involves multiple RPCs, performance issues with the same *intensity* in different RPCs can have different *impact* on the system behavior. Trivially, for example, more frequently invoked RPCs have more *impact* than less frequently invoked ones. An *instantiation* has to provide a way to estimate *impacts* of interest.

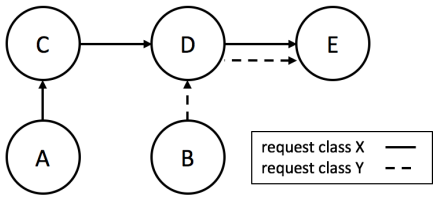

Figure 1: Calls among services in requests of classes X and Y

(5) **How do we *search* for interesting combinations of performance issues?**
The goal here is to find interesting combinations, i.e. the ones with higher *impacts* on the system. However, not all combinations are interesting, for example if we consider huge latencies, then they will clearly have huge *impacts*, but they do not sharply provide useful information on which are the performance-critical elements of a system. Combinations of performance issues with lower *intensity* are more interesting than others. In fact, if a latency introduced in a particular RPC has a negative *impact* on the system, then the most interesting value to identify is the smallest latency able to produce that *impact*. This value, in fact, represents a critical threshold to keep into account for effective fault injection. Furthermore, light combinations of performance issues are most likely to happen in production, hence they are more interesting. The number of performance issues considered in a combination will be named as *cardinality*.
The problem of searching interesting combinations can be turned into a multi-objective search problem, with the following objectives: 1) maximize all the *impacts* considered, 2) minimize *intensity* and 3) minimize *cardinality*.
Through the use of Pareto-optimality, this approach does not neglect combinations with high *cardinality* and high *intensity* when they are the only ones to show a significant impact, whereas other solutions can replace the latter ones if they show same *impact* with lower *cardinality* and lower *intensity*. *Instantiations* have to provide appropriate Search Based Software Engineering techniques [10] to solve the problem.

Different answers to these questions will provide different *instantiations* of the approach. One of the goal of this research program will be to provide several *instantiations*.

2.2 Explanatory example

In order to provide a better understanding of the proposed framework, an illustrative coarse *instantiation* in a specific scenario is introduced here. The scenario is not very representative of real applications, but it is useful to give insights on how the approach could be instantiated and how interesting combinations of performance issues can be searched.

Scenario. The context, as sketched in Figure 1, is a web application composed by 5 services named {*A*, *B*, *C*, *D*, *E*}. The application provides only two URLs accessible by users, one provided by service

A and the other provided by B. Any request to each of these URLs will produce always the same path of calls among the application services. Hence, two different classes of requests can be defined, named X and Y. When a request of class X occurs in the system, the service A is first called, thereafter A calls C which, in turn, calls D that call E. When a request of class Y occurs in the system, the B is first called which, in turn, calls D that calls E. We assume that a performance requirement is associated to each request class, which imposes that their average response times should be under certain thresholds named t_X and t_Y. We also assume to have monitored these response times, and their average values will be denoted as r_X and r_Y, respectively.

Instantiation. I will explain the illustrative *instantiation* by answering the previously introduced questions:

(1) *Which kinds of performance issues are targeted?*
Trivially here we consider the latency that can be introduced in each service (in addition to its own response time), as our targeted performance issues. For example if a request of class X occurs in the system, latency can be introduced in services A, C, D and E.

(2) *How do we model the intensity of combinations of performance issues?*
The only property considered in this *instantiation* is the amount of additional latency introduced in a service. If the service in which the latency is introduced is A then this amount will be denoted as l_A. A straightforward way to synthesize *intensity* could be to sum all latencies introduced in the combination. For example, a combination of performance which introduces a l_A latency in service A and a l_B latency in service B will have a $l_A + l_B$ *intensity*.

(3) *What are the impacts we are interested to?*
In order to keep the illustrative example as simple as possible, this *instantiation* considers only one *impact*. Here the *impact* is defined as the number of classes of requests for which the performance requirement is violated. In our case we have two requests classes, hence the maximum *impact* value will be 2 when the performance requirements are both violated.

(4) *How do we estimate the impact of combinations of performance issues?*
In order to give insights on the meaning of "estimating *impact*", we introduce here a simple implementation of this concept. The problem of estimating *impact* shifts into the problem of checking, for each class, if its performance requirement is violated. Since for any request class C its response time r_C is known, this information can be exploited in order to quantify the *impact* of a combination of performance issues. For example a combination of performance issues composed by latencies l_B, l_A and l_E, leads to violate the performance requirement associated to request of class Y if $l_B + l_E + r_Y > t_Y$. Note that l_A does not appear in this constraint because service A is not involved in the path originated by requests of class Y, thus introducing latency in A will not affect the service Y response time. Hence, we can quantify *impact* of a combination of performance issues by simply iterating for each request class, and by increasing the impact by one if the combination induces the violation of the requirement imposed on that class.

(5) *How do we search for interesting combinations of performance issue?*
Clearly the aim will be to maximize the number of performance requirements violated by the performance issues combination, namely the *impact*. Trivially large values of latencies will easily lead to violate requirements. For example, if we introduce a huge latency l_E in service E, which is involved in both classes, this will clearly have the maximum *impact*, but it will not really provide useful information about the performance criticality of specific services. A more interesting value is instead the minimum l_E that leads to violate the requirement, hence we also need to minimize the *intensity* of the performance issues introduced. Widely used multi-objective evolutionary algorithms, such as NSGA-II[6], can be adopted for this task.

2.3 Instantiations

A major challenge of this PhD program will be to provide several *instantiation* of the presented approach. The previously described *instantiation* is intentionally simple. Obviously, in order to model the complexity of real-world distributed systems, more elaborate *instantiations* shall be devised.

For example, beyond latency-related problems mentioned before, other performance issues can be targeted, e.g., utilization of resources such as CPU or memory. Also, performance issues can have several properties, for example in the context of latency issues these other properties can be considered: rate of impacted requests, average response time, standard deviation, percentiles, etc. The higher the number of properties and their complexity, the harder will be to define a synthesis of *intensity*. We do not exclude that, for some *instantiation*, *intensity* shall be split in two or more concepts. Also, the concept of *impact* shall be more realistic with respect to the one presented in the explanatory example. A refinement of *impact* can be obtained by considering the number of requests impacted by a requirement violation according to the workload of the application, rather than the number of classes of requests for which a requirement is violated,. Certainly a very critical point to develop will be the estimation of *impact*. The technique used in the illustrative example is a bit shallow, partly because the properties of performance are limited and because the notion of *impact* is overly reductive. More complex notions of properties of performance issue and impact will require more elaborate estimation techniques, such as machine learning (i.e., regressions models, neural networks, etc.) or Markov models.

From a search perspective, we plan to choose suitable search algorithms according to the *instantiation*, such as NSGA-II, which are widely used in search-based software engineering [10].

3 EXPECTED CONTRIBUTION

The expected contribution of this PhD program is to demonstrate how multi-objective search-based techniques can be applied effectively for automated fault-injection. The research questions that are intended to be addressed are:

(1) Can this framework be used to target different kinds of performance issues?
(2) What are the challenges of employing *instantiations* in real-world systems?

(3) Do *instantiations* of the approach outperform state-of-practice approaches ?

In order to evaluate RQ1 I am planning to build different *instantiations* with different kinds of targeted performance issues.
This research program relays on an industrial PhD grant, in collaboration with Microsoft and Karlsruhe Institute of Technology. Therefore, I am collaborating with Microsoft to identify real-world distributed systems for sake of experimentation, where the *instantiations* produced can be validated. One of the key challenges of pushing research in production system is the impedance mismatch between the model of reality of the research prototype and the often messy realities of production systems [1]. This experience will be very useful to gain feeling on the criticality of adapting *instantiations* for real-world scenario, therefore to answer RQ2.
At the best of my knowledge, an overly used state-practice in automated fault-injection is random search [1]. For each *instantiation* produced, I'm planning to compare, both in laboratory and in real-world systems, random fault-injection with the proposed multi-objective approach (RQ3). This can be done, for example, by iteratively comparing the actual effect on real systems of random combinations of performance issues against their dominant solutions founded by the search.

4 RELATED WORK

Fault injection is a quite mature subject in literature [8, 9, 14]. In the last few years fault injection is gaining increasing attention and it is used in production by high-tech companies [4, 11]. A recent work from Alvaro et al.[1] is the first attempt to devise a more effective strategy with respect to state-of-practice random fault injection, by smartly searching the combinatorial space of possible failures. The work presented in this paper differs from the one from Alvaro et al. by a more directed focus on performance issues of distributed systems, and by a different formulation of the problem, in that we formulate the problem of finding interesting combinations of performance issue as a multi-objective optimization problem.
The problem of anticipating unexpected performance behaviors in modern large-scale distributed systems has been tackled in different ways. Several works aim at devising tracing infrastructures to provide developers performance awareness on distributed systems. Some tools requires explicit instrumentations [13, 15, 19] while others do not require it[3]. Although these tools can be very useful for performance debugging of distributed systems, they rely uniquely on developer intuition to anticipate unexpected events. More sophisticated experimentation-oriented approaches aim at anticipating unexpected system behaviors. For example, in order to anticipate system behavior in face of unexpected workload spikes, the work of Veeraraghavan et al.[20] uses live traffic to load test the system and identify scalability limits. JustRunIt [21] proposes the use of sandboxed deployments that can execute shadow traffic from a real world deployment to answer various "what-if" questions.

5 CONCLUSION

In this paper I present a multi-objective framework for performance fault injection in distributed systems. Combinations of performance issues can have different kinds of impacts on a system, and smaller combinations with non-negligible impacts are more interesting than others, because they are harder to be identified. This leads to competing objectives often in conflict. The space of possible combinations of performance issues is too large to be exhaustively explored. The strength of this framework lays on the intuition of exploiting search-based software engineering techniques for effective performance fault injection. The framework is intended to be general, several *instantiations* can be devised. I will evaluate the applicability of the produced *instantiations* in real-world distributed systems. I also plan to compare, both in laboratory and in real-world applications, the effectiveness of the produced *instantiations* against state-of-pratice fault injections approaches.

REFERENCES

[1] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. 2016. Automating Failure Testing Research at Internet Scale. In *the ACM Symposium on Cloud Computing*. 17–28.
[2] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *SIGMOD*. 331–346.
[3] Dan Ardelean, Amer Diwan, and Chandra Erdman. 2018. Performance Analysis of Cloud Applications. In *the Symposium on Networked Systems Design and Implementation*. 405–417.
[4] Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, and Casey Rosenthal. 2016. Chaos Engineering. *IEEE Software* 33, 3 (May 2016), 35–41.
[5] Jake Brutlag. 2009. Google AI Blog: Speed matters. https://ai.googleblog.com/2009/06/speed-matters.html
[6] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transaction Evolutionary Computation* 6, 2 (April 2002), 182–197.
[7] Dror Feitelson, Eitan Frachtenberg, and Kent Beck. 2013. Development and Deployment at Facebook. *IEEE Internet Computing* 17, 4 (July 2013), 8–17.
[8] Haryadi S. Gunawi, Thanh Do, Joseph M. Hellerstein, Ion Stoica, Dhruba Borthakur, and Jesse Robbins. 2011. *Failure as a Service (FaaS): A Cloud Service for Large-Scale, Online Failure Drills*. Technical Report. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-87.html
[9] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *the Conference on Networked Systems Design and Implementation*. 238–252.
[10] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based Software Engineering: Trends, Techniques and Applications. *Comput. Surveys* 45, 1, Article 11 (Dec. 2012), 11:1–11:61 pages.
[11] Lorin Hochstein and Casey Rosenthal. 2016. Chaos Engineering Panel. In *ICSE (Companion)*. 90–91.
[12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *PLDI*. 77–88.
[13] Jonathan Kaldor, Jonathan Mace, MichałBejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *SOSP*. 34–50.
[14] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. 1995. FERRARI: A Flexible Software-Based Fault and Error Injection System. *IEEE Transanctions on Computers* 44, 2 (Feb. 1995), 248–260.
[15] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *the Symposium on Operating Systems Principles*. 378–393.
[16] Sam Newman. 2015. *Building Microservices* (1st ed.). O'Reilly Media, Inc.
[17] Charlene O'Hanlon. 2006. A Conversation with Werner Vogels. *Queue* 4, 4, Article 14 (May 2006), 14:14–14:22 pages.
[18] Julia Rubin and Martin Rinard. 2016. The Challenges of Staying Together While Moving Fast: An Exploratory Study. In *ICSE*. 982–993.
[19] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf
[20] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. 2016. Kraken: Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services. In *OSDI*. 635–651.
[21] Wei Zheng, Ricardo Bianchini, G. John Janakiraman, Jose Renato Santos, and Yoshio Turner. 2009. JustRunIt: Experiment-based Management of Virtualized Data Centers. In *the USENIX Annual Technical Conference*. 18–18.