Cost-Aware Stage-Based Experimentation: Challenges and Emerging Results

Ilias Gerostathopoulos, Christian Prehofer
Chair of Software and Systems Engineering
Technical University Munich
Munich, Germany
{gerostat, prehofer}@in.tum.de

Lubomír Bulej, Tomáš Bureš, Vojtěch Horký, Petr Tůma
Department of Distributed and Dependable Systems
Charles University in Prague
Prague, Czech Republic
{bulej, bures, horky, tuma}@d3s.mff.cuni.cz

***Abstract*—Experimentation at post-deployment phases (in production environments) can be a powerful tool for both learning how a deployed system operates and how it is being used. Though this knowledge is invaluable for optimization of the system, collecting it may require long time and experiments may even worsen the system with negative effects on users and business. This calls for methods for performing experimentation in production environments that balance the profit of experimentation with its cost. In this paper, we describe related challenges and our emerging results towards cost-aware stage-based experimentation. In particular, we aim for performing experiments that optimize towards their profit while making sure that the overall experimentation cost (e.g. total experimentation time) stays within given bounds. First, we illustrate the challenges and needs of such experimentation in two use cases from different domains. Second, we describe the main concepts behind our method in a semi-formal notation. Third, we exemplify the method by applying it in the two use cases and we report interesting first results.**

Keywords—experimentation; cost-aware; optimization

I. INTRODUCTION

Systems we build are ultimately evaluated based on the value they deliver to their users and stakeholders. Modern software-intensive systems have several characteristics that make it easier for such value to be estimated and optimized. First, they typically have a number of *tunable parameters* or *knobs* that can be used for changing their operation even in post-deployment phases. Second, they have a number of probes that can be used for obtaining telemetry data and calculating different system metrics, such as web page load time, timing performance, or even end-user satisfaction.

To increase the value delivered by a system, one needs to capture the relation between the values of its tunable parameters and the values of its measurable metrics, then apply the combination of parameter values that optimizes the metrics of interest—the optimization criterion. In certain classes of systems, such a relation is very hard to capture in pre-deployment phases, since they either involve metrics related to human users (e.g. web-facing applications) or they have complex, non-linear interactions between input parameters and metric values that are hard to model a priori. The practical solution in such cases is to deploy the system and experiment with it in production environments.

The state of the art in experimentation in production environments, also called online experimentation, mainly reports on the experience of web-facing companies such as Google [1], Uber [2], and Microsoft [3]–[5] in building scalable experimentation platforms and performing controlled experiments in order to evaluate changes that affect user experience and prioritize feature development. A/B tests stand out as the most common case of experiments in practice, albeit not without challenges in their execution [6]. Recent research has gone a step further to propose that systems (instead of humans) should control and run experiments on themselves in order to self-optimize [7]–[9]. We follow here this line of research, i.e. of automated online experimentation, and formulate what we believe is a challenging problem.

The problem with experimentation in production environments is that it is very risky due to the high cost it may incur. Consider experimenting with the parameters of a web server: the end-to-end latency for a request or a page load time might accidentally increase. While this experiment is profitable in terms of knowledge acquired about the system, it also has high cost if real users get affected, manifested in negative user experience. On the contrary, performing the same experiment in a user acceptance testing environment yields far lower cost (if not none) in terms of negative user experience. As another example, consider experimenting with the parameters of a recommender system used via a web-facing application: the time to get a good recommendation might again accidentally increase, yielding an outdated, irrelevant recommendation that reduces the value of the system as perceived by its end-users. Again, this is not an issue in a testing environment where delaying a relevant recommendation does not incur any business cost.

What is actually needed when experimenting in production environments is a method by balancing the profit of experimentation (the knowledge acquired about the system) with its cost [10]. This should allow one to follow risky strategies which e.g. gather more data for evaluating a statistical hypothesis even when the cost is high. On the other side, it should also support one to follow risk-averse strategies that e.g. cut off an experiment when the actual or the predicted cost reaches a certain threshold.

In this paper, we describe our research towards such a method. Our method prescribes the execution of an experiment in stages (i.e. steps of an experiment) to better control its profit and cost. The main idea is to decide on the stages that are most profitable to run but need to be carefully chosen since they have high cost, via first running a number of "exploratory stages", which typically yield no direct profit, but are also cheaper and provide insights into the system.

In particular, we make the following contributions. First, we illustrate the challenges and needs for such experimentation method in two use cases from different

domains (Section II). Second, we describe the main concepts behind our method in a semi-formal notation and, third, exemplify the method by applying it in our two use cases and we report interesting first results (Section III).

II. USE CASES

A. Use Case 1: Optimizing City Traffic Navigation

The first use case concerns a traffic navigation system for cars in a city. The system consists of the individual cars and of a centralized traffic router. The router is a service used by the cars to plan and guide their routes (series of streets to follow) from their current location to a destination in the city. The router is parametric: it offers several interval-scaled variables that can be tuned to affect the way new routes are calculated. For route calculation, the router uses a Dijkstra algorithm for calculating the shortest path between two nodes (intersections), based on costs assigned to each edge (street). Costs are assigned to edges based on a function that combines *static* traffic information (e.g. street length, maximum speed on street) with *dynamic* traffic information (e.g. average speed on street, calculated for a timespan t from cars that travelled on the street). The router parameters parametrize the per-edge cost assignment function by specifying the weight of static traffic information, the weight of dynamic traffic information, the timespan t, and other values (a complete list of tunable parameters is in Table 1 of [11]).

The goal of changing the parameters of the router is to maximize the value delivered to the drivers. Assuming a uniform driver profile capturing driver's preferences, drivers are satisfied when they reach their destination as soon as possible. This forms the optimization criterion of minimizing the trip overhead metric. A trip overhead is obtained by dividing the actual trip time by the theoretical trip time where the car travels at full speed in all streets.

The relation between router parameter values and the trip overhead metric values is not easy to model at design time because of the complex, non-linear interactions that exist between the router parameters and the metric values. Consider e.g. that increasing the value of a parameter increases the value of the metric, but only in combination with a decrease in the value of another parameter. Such relation is also not easy to deduce at pre-deployment phases, because it depends on runtime situations. For instance, a different configuration of parameters will be optimal in the situation of low traffic in the city (where the static traffic information will probably be more important in calculating per-edge costs than dynamic information) compared to the situation of high traffic.

Ideally, the system should run under different configurations and in different situations in production environments in order to determine the optimal configuration per situation. The risk of doing so is that, while traversing the search space, certain configurations can incur high cost in terms of driver's annoyance. Such cost can be calculated by e.g. summing up the complaints filed by drivers to the router service. A method for experimentation in production environments should strive for *keeping the number of complaints below a certain limit*.

B. Use Case 2: Identifying JIT Performance Regressions

The second use case concerns a system for automatically detecting performance regressions, i.e. differences in performance delivered by different versions of a Just-in-Time (JIT) compiler in a Java Virtual Machine (JVM). The goal is to detect changes in the code base that impact performance significantly. Performance is measured by inspecting the execution time of multiple Java applications acting as compiler benchmarks. Apart from the execution time, other metrics are also collected for each benchmark, e.g. compilation time, instruction count and number of cache references and misses. Assuming a scenario with a list of JIT versions (from a version control system) the system outputs one or more pairs of subsequent versions in which a performance regression has been detected in at least one of the benchmarks. To detect a regression in a benchmark, some of the essential metrics (e.g. execution time or instruction count) need to show a significant difference at a given significance level. To detect regressions in neighboring versions, pair-wise comparisons are performed.

The system offers a parameter, called *benchmark selection*, than can be tuned at runtime and specifies which of the benchmarks will be run, for how long, and in which order. The selection of the benchmarks is important, since even when parallel execution of benchmarks is employed (currently we run up to 36 experiments in parallel), it is typically not possible to run all (close to 80) benchmarks for all JIT versions. This is because there are typically several JIT versions released every day that need to be measured and because every benchmark has to be run multiple times with long enough warmup and measurement phase to achieve statistical significance (each run takes from tens of minutes to several hours).

The goal of tuning the benchmark selections is to execute such benchmarks on a JIT version for which there is high likelihood that a regression will show up and therefore maximize the value delivered to the developers of the JIT compiler, who need to know whether their changes introduced a regression or not within a certain time frame (after which this information is not so valuable anymore). As such, the *number of correctly identified regressions within a time frame t* is the optimization criterion in this use case, whereas the *time spent in detecting regressions* is the cost of experimentation.

III. APPROACH

Our approach aims at running experiments on a system to derive the learned knowledge, i.e. the knowledge needed to optimize the system. Each experiment is split into several stages which exercise the system in different ways. Some of the stages are costlier and more profitable (in terms of learning) at the same time; a typical case is to execute these stages at the end of an experiment, after having exercised the system in a number of relatively cheap stages. The main purpose of these stages is *not* to derive the learned knowledge but to profile the system and reduce the number of costly experiments—that are also profitable—that we ultimately need to run.

In the rest of the section, we describe the concepts of our approach in a semi-formal notation and exemplify our approach on the two use cases.

A. Concepts

System Under Experimentation. A tuple (I, O, S, C, C_0) where I is a non-empty set of identifiers of input (tunable) system parameters, O is a non-empty set of identifiers of observables, i.e. system parameters that can be observed, S is a set of identifiers of situational parameters, which may be controllable, C is a (possibly infinite) set of applicable configurations for the parameters in I, and C_0 is a function from S to C producing default configurations (used when no experiments are run). A *configuration* of the system is a valuation of each input parameter in I. A *situation* of the system is a scenario with a specific valuation of the situational parameters in S. Exemplary values for I, O, and S for the use cases are depicted in Table 1.

System Observation. A tuple obs = (id, val) where id is a unique identifier and val is a valuation of each parameter in a (non-strict) subset of O, i.e. a function that assigns a value to some of the output parameters in O. The identifier is used to relate an observation to different context such as time and experiment stages, defined next.

Experiment Stage. A process in which first a configuration is applied to the system and then observations are collected.

Experiment Stage Result. An experiment stage result is a tuple st = ($conf$, s, Obs, c) where the configuration $conf$ is applied to the system in situation s. Obs is a set of system observations collected in the stage, whereas cost c of the stage can depend on $conf$, s, Obs or execution time of the stage. For example, in Use Case 1, the cost of the stage is the number of complaints received while being in the stage. In Use Case 2, the cost of a stage is the sum of the time each benchmark was running.

Experiment. A process that generates experiment stages in order to maximize the value of learned knowledge for one situation. An experiment has a *cost budget*, i.e. a threshold on the sum of costs of the experiment stages that cannot be exceeded.

Experiment Result. The result of an experiment is expressed as a tuple ex = (s, lk) where s is a system situation, and lk is the learned knowledge, e.g. the values of input parameters that improve the system compared to default values of such parameters in the situation.

B. Application on Optimizing City Traffic Navigation

We have applied the main idea of stage-based cost-aware experimentation on CrowdNav, a self-adaptation testbed we developed in our recent research for comparing solutions to the city traffic navigation optimization problem [11], [12]. CrowdNav offers a total of seven input parameters that can be tuned for minimizing the average trip overhead. From a mathematical viewpoint, the problem of optimizing CrowdNav is one of minimizing a multivariate black-box function f that returns an average trip overhead value for a configuration of CrowdNav. Since the valuations of f are both noisy and expensive (it takes time to obtain a stable value), a fitting optimization method is Bayesian Optimization with Gaussian Processes (BOGP). BOGP runs in a number of steps; at each step it builds a probabilistic model of f based on the so-far observed valuations of f, determines the next configuration for sampling and evaluates f on this configuration. BOGP is an *anytime* algorithm for global optimization, i.e. it tends to provide the best possible solution for a given number of steps [13]. When applying BOGP on CrowdNav we observed that it was indeed very effective in traversing the configuration space and providing a configuration that minimizes trip overhead. Each step of BOGP corresponds to an experiment stage in our approach that is both costly, since it has to run for long time to gather enough samples of trip overheads and thus the probability of getting more complaints also increases, and profitable, since a model of f is trained and directly used to find a minimum.

TABLE 1. SYSTEMS UNDER EXPERIMENTATION IN THE USE CASES.

Use Case 1	
I	route randomization, static info weight, dynamic info weight, exploration percentage, …
O	trip overhead, complaint
S	total cars number, street network, …
Use Case 2	
I	benchmark selection
O	detected regressions
S	list of JIT versions, hardware platform, operating system version, …

To minimize the number of BOGP steps, we preceded them with a number of stages whose role was to profile CrowdNav and determine a subset of input parameters that have the strongest effect on the trip overhead metric. BOGP would then work with a smaller configuration space, consisting only of this subset, and need less steps to find a minimum. In particular, we used 2^7=128 stages corresponding to the configurations of a full factorial design where each input parameter becomes a factor with two values (minimum and maximum). After running each stage for only a limited time—a fraction of the time needed for a BOGP step—we applied the between-subjects factorial analysis of variance statistical test to determine the factors (input parameters) or the interactions of factors that have a significant effect on the trip overhead metric. Our initial results show that this strategy can reduce the number of profitable stages (the BOGP steps) to a minimum of five, without compromising the solution quality [11]. Investigating how this reduction translates to a reduction in experimentation cost is our ongoing work.

C. Application on Identifying JIT Performance Regressions

In this section we assess how our approach applies to detecting JIT performance regressions. Our assessment is based on data gathered from implementing and operating a system for detecting performance regressions between different open source versions of the Graal compiler, a JIT compiler project based in Oracle Labs (so far, this amounts to

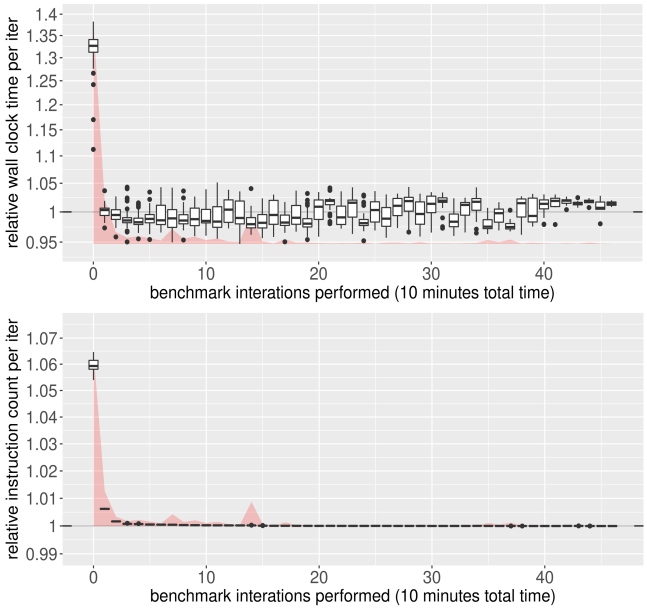

Figure 1. Warmup behavior with JIT compilation activity in backdrop of a single version of JIT on the scalaxb-huge benchmark.

some 650 thousand measurement experiments spread across approximately 1700 versions created over two years of development). The system identifies regressions in individual Graal git merge commits as described in Section II.B. Each commit is currently analyzed by running close to 80 benchmark workloads from the DaCapo, ScalaBench, SPECjvm2008 suites and additional (proprietary) benchmarks. This process is generally time-consuming and can be optimized by running only a subset of the benchmarks for each comparison.

Using the terminology introduced before, an experiment stage is the application of a benchmark selection value and the observation of detected regressions. Our approach will first determine which benchmark selection to use in later stages by first running stages with "profiling selections" which prescribe shorter benchmark runs. As an example, Figure 1 (top) shows how the benchmark time changes during warmup for the scalaxb benchmark. Technically, warmup does not stop before the JIT compiler activity (backdrop) subsides, here around iteration 20 or later. A short run can potentially save on the warmup time by using less iterations, here e.g. 5-10. Fewer iterations may not yield statistically significant differences in the wall clock time (because of the high variance of this metric in the warmup phase); however, such differences may in the instruction count metric which we observe alongside wall clock time. As seen in Figure 1 (bottom), the instruction count has lower variance, and thus provides statistically significant results with fewer samples. The main idea is that, even though changes in instruction count and wall clock time do not always occur together or in the same direction, instruction count is a useful profiling indicator. The actual regressions always need to be detected on the wall clock time metric when executing the benchmarks after proper warmup (longer execution time) and collecting enough data to provide statistical confidence.

IV. Conclusions

In this paper we focused on the particular challenge of performing experiments to optimize systems at runtime while trading off the profit of experimentation with its cost. The novelty in this respect is conceptualization of an experiment in stages which provides a framework for experimentation across domains. We have customized this framework for two different use cases with different notions of profit and cost. The key technical idea is the introduction of initial stages which do not yield direct profit but allow us to guide the experimentation process towards determining the profitable experiments to run at later stages.

Acknowledgment

This work has been partly funded by the Bavarian Ministry of Economic Affairs, Energy and Technology as part of the TUM Living Lab Connected Mobility Project and by the German Ministry of Education and Research under grant no 01Is16043A. The work has been partly funded by project no. LTE117003 (ESTABLISH) from the INTER-EUREKA LTE117 programme by the Ministry of Education, Youth and Sports of the Czech Republic.

References

[1] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer, "Overlapping experiment infrastructure: More, better, faster experimentation," in *Proc. of SigKDD 2010*, ACM, 2010, pp. 17–26.

[2] "Uber Experimentation Platform," 15-Mar-2018. [Online]. Available: https://eng.uber.com/tag/experimentation/.

[3] R. Kohavi *et al.*, "Online experimentation at Microsoft," in *Data Mining Case Studies and Practice Prize III*, 2009, vol. 11.

[4] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: survey and practical guide," *Data Min. Knowl. Discov.*, vol. 18, no. 1, pp. 140–181, Feb. 2009.

[5] A. Fabijan, P. Dmitriev, H. H. Olsson, and J. Bosch, "The evolution of continuous experimentation in software product development: from data to a data-driven organization at scale," in *Proc. of ICSE 2017*, IEEE, 2017, pp. 770–780.

[6] H. H. Olsson, J. Bosch, and A. Fabijan, "Experimentation that Matters: A Multi-case Study on the Challenges with A/B Testing," in *Software Business*, vol. 304, Springer, 2017, pp. 179–185.

[7] J. Bosch and H. H. Olsson, "Data-driven continuous evolution of smart systems," in *SEAMS 2016*, ACM, 2016, pp. 28–34.

[8] D. I. Mattos, J. Bosch, and H. H. Olsson, "More for Less: Automated Experimentation in Software-Intensive Systems," in *Product-Focused Software Process Improvement*, Springer, Cham, 2017, pp. 146–161.

[9] D. I. Mattos, J. Bosch, and H. H. Olsson, "Your System Gets Better Every Day You Use It: Towards Automated Continuous Experimentation," in *SEAA 2017*, IEEE, 2017, pp. 256–265.

[10] I. Gerostathopoulos, T. Bures, S. Schmid, V. Horky, C. Prehofer, and P. Tuma, "Towards Systematic Live Experimentation in Software-Intensive Systems of Systems," in *Proc. SiSoS@ECSA '16*, 2016, p. 7.

[11] I. Gerostathopoulos, C. Prehofer, and T. Bures, "Adapting a System with Noisy Outputs with Statistical Guarantees," in *Proc. of SEAMS 2018, to appear.*, 2018.

[12] S. Schmid, I. Gerostathopoulos, C. Prehofer, and T. Bures, "Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool," in *Proc. of SEAMS 2017*, IEEE, 2017, pp. 102–108.

[13] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas, "Taking the Human Out of the Loop: A Review of Bayesian Optimization," *Proc. IEEE*, vol. 104, no. 1, pp. 148–175, Jan. 2016.