Performance Annotations for Cloud Computing

Daniele Rogora[*] *Steffen Smolka*[†] *Antonio Carzaniga*[*] *Amer Diwan*[‡] *Robert Soulé*[*¶]
[*]*Università della Svizzera italiana* [†]*Cornell University* [‡]*Google* [¶]*Barefoot Networks*

Abstract

Web services and applications are complex systems. Layers of abstraction and virtualization allow flexible and scalable deployment. But they also introduce complications if one wants predictable performance and easy trouble-shooting. We propose to support the designers, testers, and maintainers of such systems by annotating system components with performance models. Our goal is to formulate annotations that can be used as oracles in performance testing, that can provide valuable guidance for debugging, and that can also inform designers by predicting the performance profile of an assembly of annotated components. We present an initial formulation of such annotations together with their concrete derivation from the execution of a complex web service.

1 Introduction

Understanding and predicting system behavior is crucial for cloud-service providers. An operations manager might want to know how to respond to a growth of requests (e.g., *"should I invest in more compute or storage nodes?"*). A developer might want to know which updated component has slowed down parts of the Web application, and how that affects operations (e.g., *"is a particular method allocating too much memory?"*). A designer might want to predict the performance of a service that relies on other services, both internal and external (e.g., *"assuming the authentication and storage lookup service run in under 50 milliseconds, would a delete operation terminate in less than 100 milliseconds?"*).

Unfortunately, operations managers, developers, and designers have very few tools at their disposal that allow them to answer such questions. Currently, they mostly rely on profilers and performance monitors [5, 13]. However, profilers are inherently limited, in that they report aggregate resource usage of one or more functions under a specific input workload. Profilers do not relate the performance of a function to its input. Furthermore, despite recent research advances [19], profilers provide little information about the system behavior under various workloads, and offer no predictive capabilities.

In this paper, we propose an alternative approach, in which APIs are explicitly annotated with information about their expected performance. We envision three main uses of *performance annotations*. First, they can serve as assertions and therefore as failure detectors in both testing and production systems. Second, annotations can provide diagnostic information beyond a binary pass/fail test. For example, the annotation can be checked by monitoring the run-time behavior (as an assertion) and the result could be a distance or fitness value (or a sort of p-value) that expresses how far the behavior deviates from the prescribed distribution. This information computed for a series of components can in turn be useful to inform both on-line operational management and off-line performance engineering for the whole system. Third, perhaps the most ambitious goal, is to use annotations in a compositional analysis, together with code, to predict aggregate behaviors of a system from the behavior of individual components.

There are at least two major challenges in developing performance annotations: (i) providing meaningful information that is useful in practical deployments, and (ii) determining how to derive annotations in a complex system stack. To address these challenges, we argue that performance annotations should assert *expected performance* and should be *automatically derived* using statistical techniques.

By expected performance, we mean concrete performance in the average case, as opposed to worst-case and/or asymptotic performance. Although worst-case analysis is useful in many scenarios, it is often impractical to reason about only the worst case in real-world deployments. For example, not every lookup will result in a cache miss; not every allocation causes a TLB miss, etc. Moreover, if we consider only the worst case, and

then try to reason compositionally about services in a distributed system, we will quickly conclude that everything can be slow and expensive. Finally, it is often easy to detect and deal with worst-case or extreme behaviors. For example, a rogue process that continually consumes resources can be sandboxed and separated from the rest of the system. In contrast, a process whose performance varies slightly from its nominal behavior is harder to single out and to understand.

By automatically derived, we mean that developers should not have to annotate code themselves. Cloud-based, web applications are complex systems, involving a number of interconnected components that interact in intricate ways. There is typically (at least) a storage component, a database component for meta-data, and a web front-end with user-facing functionality. The storage component is usually replicated for fault-tolerance, and accessed through a distributed file system; the database, which is also replicated, must handle all sorts of internal calls, from user authentication and user sessions services, as well as store all the necessary state for the web application. And the whole system is typically deployed on a set of virtual machines interconnected with a number of virtual networks, perhaps hosted on multiple data centers. Given this complexity, one can not expect programmers to accurately predict system performance. Instead, we argue that system behavior must be learned.

In this paper, we first present a general notion of performance annotation (§2), and how they are automatically derived (§3). We then apply them in the study of a system that implements a complex cloud storage service, and categorize a wide variety of behaviors we observe in such a system (§4). Finally, we show that the annotations can be used to reveal anomalous behaviors (§5).

2 Performance Annotations

A performance annotation formulates a probabilistic model of the behavior of a sub-system or component in a distributed system. Performance annotations differ from profilers [19] in that they correlate measured performance metrics with semantically meaningful input or system state, allowing operators greater understanding into the behavior of the system. Importantly, performance annotations model the average-case, rather than worst-case behavior [9, 10]. The average-case gives the expected behavior given a typical usage scenario.

A natural way to describe the expected performance of a service is as a probability distribution for a set of *measured* values (i.e., the dependent variable) that change in response to some system input or *feature* (i.e., the independent variable) of the input or the system state. The annotations we propose take this form.

As an example, consider the following annotated code:

```
1 @Time~Norm(2ms * path_length(path), 1ms)
2 void delete(String path) {
3   ...
4 }
```

In this example, the *delete* method, declared on Line 2, deletes the file specified in the parameter *path*. The annotation, on Line 1, asserts that the running time of the method follows a normal distribution with a mean of $2ms$ times the number of path components, and variance $1ms$. In this example, *time* is the measured value, and *path_length(path)* is the related feature.

3 Deriving Annotations

Deriving an annotation involves three steps: (i) choosing and measuring a specific target metric y, (ii) determining a feature, x, and measuring its values, and (iii) learning a model or hypothesis, h, such that $y \approx h(x)$ (or $y \sim h(x)$, if $h(x)$ is a distribution). Below, we briefly discuss each of these three steps.

Target metrics. We assume that code is (or can be) instrumented to measure the desired values. This is common practice in many web-scale companies, such as Google for Facebook. Instrumentation necessarily incurs an overhead that may result in a performance penalty. However, this subject has been researched extensively and many techniques exist to minimize the impact of instrumentation. In principle, annotations could be used to characterize the behavior in terms of any measured value. As a first step, we focus on running time, as well as other widely used performance indicators, including heap allocation, lock holding time, and number and duration of remote procedure calls. Other useful measurements that we have not yet collected might include latency or congestion for network communication.

Feature discovery. Our features are derived from the input parameters by using a set of basic heuristics: for scalar numeric parameters, we record their value; for collections, we record the size; for strings, we check if the string points to a file. If it does, we record the file size and the depth of the path. Otherwise, we record the length of the string. In our initial experiments, we have found that although these heuristics often work, our system will ultimately depend on the knowledge of a developer to guide the choice. Although our prototype uses a single feature, it should be possible to combine features.

Formulating annotations. Given a set of target measurements (e.g., time), each with a corresponding set of features, we formulate an annotation as a relation between one feature and the target metric. In essence, we try to find a simple-enough model that predicts the target

data from a feature with reasonable accuracy.

We proceed by first selecting a feature, and then by modeling the way the feature affects the target metric. We select a feature by computing the Pearson correlation coefficient (PCC), which measures the linear correlation between the feature and the target metric, and we choose the feature with the highest PCC value greater than some threshold. We then model the relation between the chosen feature and the target metric with a regression, using polynomials of increasing degree, from degree 1 (linear) to degree 3 (cubic). For each degree, we compute the mean squared error for the measured data. If the relative error is greater than a threshold value, then we try a larger degree. We refer to annotations derived using this method as *correlated annotations*.

If we are unable to find a feature with a PCC value greater than the minimum threshold, or if no regression has a relative error less than the threshold, then we conclude that the chosen features set is not informative, and therefore formulate an annotation based exclusively on the measured metric. In particular, we sort the data, and run a k-means clustering algorithm. To determine an appropriate value for k, we use the standard iterative approach: we start with one cluster, and compute the mean square error. We increase the number of clusters until the ratio between the errors in subsequent runs is below a threshold (i.e., we "reach the elbow"). We refer to annotations derived using this method as *clustered annotations*. A clustered annotation consists of a set of clusters, each described by a centroid and a standard deviation. With this notation we imply that clusters contain normally distributed values, which is not necessarily true. Ideally, our model makes no assumptions on the actual distribution of the target metric, so in the future we may extend this form of annotations to account for different distributions.

Note that our current implementation focuses on annotations based on a single feature, which are easy to interpret. The approach can be extended to more complex models that use several features.

4 SWITCHdrive Case Study

In order to explore the space of behaviors of complex cloud systems, to gain an understanding of their specific behaviors, and ultimately to validate our assertion model, we performed a case study on a scaled-down replica of a real-world cloud application called SWITCHdrive. Below, we report on some results from our experience.

Experimental Configurations. SWITCHdrive is a file hosting service offered by SWITCH, the national ISP for academic institutions in Switzerland. SWITCHdrive provides the same functionality as DropBox, allowing users access to cloud storage and file synchronization.

SWITCHdrive requires a complex software stack that includes the virtualization framework OpenStack [14], the storage platform Ceph [2], and a server-side web application written in PHP called ownCloud [15] running within an NGINX web server. The production SWITCHdrive cluster consists of 41 powerful machines.

For our experimentation, we faithfully recreated the same deployment on a smaller scale. In particular, we used the Puppet and Ansible scripts written by SWITCH to install and configure the required software on a small cluster of 12 servers. The servers were placed in a local network provided by two top-of-the-rack switches. Of the 12 servers, 5 are dedicated storage nodes, 5 are compute nodes, and 1 machine runs the OpenStack controllers. The last machine is used as a client to generate workload. All servers have the same hardware configuration: a Xeon x3360 processor with 4 cores, 8GB of RAM, two 1GB/s Ethernet interfaces, and a single 750GB hard disk storage unit. On the bare hardware, we installed Ubuntu 14.04 server images.

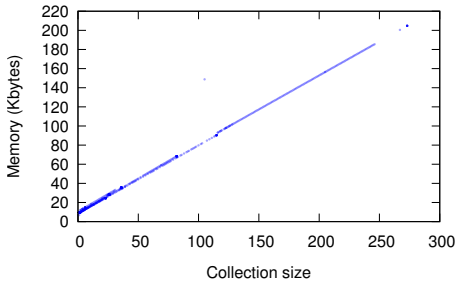
Instrumentation. To gather target metrics, we instrumented ownCloud, the PHP web server application. To instrument the code, we used the runkit library [18], which dynamically augments existing code using the decorator design pattern. Every method was dynamically replaced with a wrapper that records time-stamps and heap-allocated memory before and after the method execution. All of the information is then logged to the *tmp* filesystem of the machine running ownCloud.

Workload generation. To interact with the SWITCHdrive software, we relied on the WebDAV interface provided by ownCloud. This allowed us to mount the folder of a specific user on a remote machine, using the *davfs* filesystem. To generate workload for our experiments, we initiated an rsync operation for a complete Linux kernel source tree to the WebDAV directory. The entire workload consists of more than 200k WebDAV requests. To improve the clarity of the visualization, we present a subset consisting of about 1.5k requests.

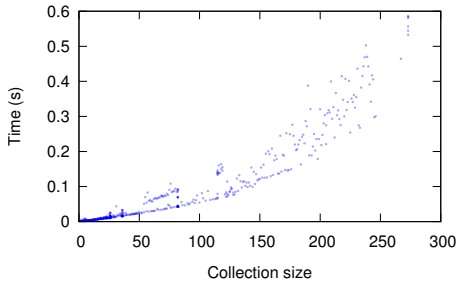
Observed behavior. Figure 1 shows some interesting behaviors that we observed. The x-axis indicates the feature used in the derived annotation, and the y-axis indicates the target measured metric.

The first two graphs show two different correlations for the *generateMultiStatus* method, which takes a single parameter, *fileProperties*, which is an array of properties about a file. The method iterates over the objects in the *fileProperties* array and computes some data about each one of them.

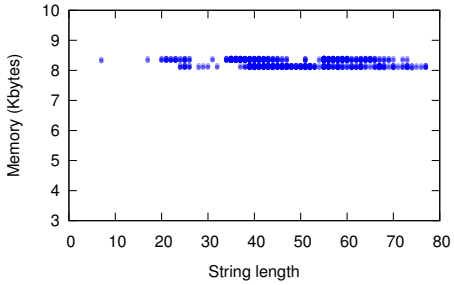
Figure 1a clearly shows that the the memory usage of the method is linearly correlated to the size of array that the method takes as parameter. This is an example of a data set that is characterized by a good Pearson correlation coefficient.



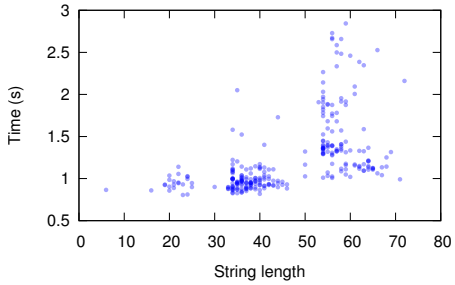
(a) Example of linearly correlated data, generated from the *generateMultiStatus* method.



(b) Example of superlinearly correlated data, generated from the *generateMultiStatus* method.



(c) Example of constant data, generated from the *getOwner* method.



(d) Example data showing no strong correlation, generated from the *createCollection* method.

Figure 1: Examples of data collected for different methods.

Figure 1b, shows the runtime of the same method for the same input set. Again, a regression produces a fitting performance model for this method, which in this case corresponds to a quadratic function of the input feature.

Figure 1c shows results for a different method, *getOwner*, which takes a single string parameter, *path*. This method returns a string representing the owner of a file specified by path. This case is interesting, because the memory usage remains constant as the length of the input string varies. One way to interpret this data is that it is a polynomial where the coefficients are all 0 (or very small). However, it is not clear if such an interpretation is semantically meaningful for program analysis. Instead, we believe that it is better to explicitly annotate the function so as to indicate a *constant* behavior.

Finally, Figure 1d shows a method that does not fall into either the *correlated* or *constant* categories. This plot shows results for the *createCollection* method, which creates a new directory with a given path and set of properties. The figure shows the running time plotted against the length of the string specified by the *path* parameter. In this case, we are unable to formulate a regression with a good Pearson correlation coefficient. Consequently, we resort to clustering to formulate a model of performance that does not depend on the input feature.

The four cases we present here are characteristic of many other behaviors observed for many methods of the ownCloud component of the SWITCHdrive system. Overall, the case study reveals a number of interesting behaviors that can be meaningfully described by relatively simple, high-level relations between input features and measured metrics. Moreover, these annotations are expressed in a human readable form that can also serve as documentation for the system.

5 Using Annotations

Having derived several performance annotations, we ask whether these annotations would indeed provide good information for developers and system operators. We start by using annotations as assertions and therefore as failure or anomaly detectors. We then want to verify that such detectors are sensitive to real anomalies at the same time as they are robust with respect to different workloads.

We proceed as follows: we first derive annotations using the workload described in Section 4. We then run the same workload in a special setting in which we artificially introduce an anomaly in the system. In this setting, we use the annotations as assertions. We implement the assertions as standard one-sample statistical tests, comparing measured metrics to the idealized model given by the annotation. We record an assertion violation whenever the test indicates that the measurements *do not* conform to the annotation. For each run, we then count the

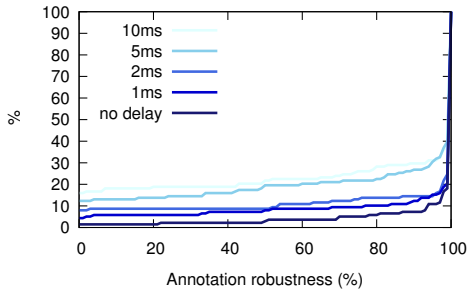


Figure 2: Cumulative Distribution Function of annotations robustness for the *extra-delay* experiments.

number of assertion passed (or failed) for all the instrumented methods.

As an anomaly, we introduce an artificial network latency between the virtual machine hosting the database server and the rest of the cluster, varying the delay from 0 to 10 milliseconds. The case of 0ms serves as a robustness check of the assertions generated during training.

Figure 2 shows the cumulative distribution function of the number of passed assertions (as a proportion of the overall number of assertions). The results indicate that assertion failures clearly expose a change in the performance behavior of the system, suggesting that there must be a performance problem somewhere.

One thing to notice is that the overall difference may appear small. Indeed, we only tweak the behavior of the database component, leaving all the other components in their original configurations. This means that only a fraction of the 138 methods analyzed were affected by the slow database. In fact, we also observed that the methods whose annotations are violated more often in the different experiments are those that directly or indirectly involve some database operations.

These experiments demonstrate the feasibility of using performance annotations to diagnose performance bugs, and how annotations allow us to reason about the impact of environment changes on a large code base.

6 Related Work

Understanding the performance of distributed systems is a topic of interest to several research communities. To a first approximation, existing work can be classified into two categories: (i) approaches based on profiling, with a focus on average-case behavior; and (ii) approaches based on static analysis, with a focus on worst-case behavior. Our work falls into the former category.

Profiling Based. PSpec [16] is a language that allows users to write performance assertions, and a set of tools for testing the assertions at runtime. Our assertions differ from PSpec in that they are probabilistic, and they are discovered, as opposed to provided by programmers explicitly. Pip [17] is similar to PSpec. While it supports automatic generation of assertions, this generation works only for qualitative correctness assertions, rather than for quantitative performance assertions as in our work. Both Pip and PSpec are limited to bounding aggregate metrics (such as average, maximum, or standard deviation) of performance measures by constants; in contrast, our system can express and automatically generate performance expectations as functions of the input size.

Our work builds on ideas from algorithmic profiling [19], which repeatedly runs a program on different inputs to relate input size to cost. Our work differs in that it applies statistical techniques to learn the cost function, and relates concrete cost measures, such as running time, as opposed to abstract "algorithmic steps". Jin et al. [11] automatically detect performance bugs using rules that are synthesized from bugfixes. Coppa and Finocchi [3] apply curve fitting to analyze the progress of MapReduce applications. Their analysis is simpler than ours, in that it is restricted to a single feature (i.e., input size).

Static Analysis Based. Hofmann et al. [10] use type-systems backed by LP solvers to infer resource bounds for functional programs. The approach has been extended to handle higher-order programs, amortized analysis [12], and multinomial polynomials in the sizes of the inputs to bound resources [8]. Hoffman et al. [9] extend this approach to OCaml. The work by Gulwani et al. [7, 6] is similar in spirit, but they use symbolic techniques and focus on imperative programs. Çiçek et al. [1] propose a relational cost analysis to reason about the *difference* in execution cost of two programs.

7 Outlook

This paper presents performance annotations, which formulate a probabilistic model of the expected, average-case performance of sub-systems and services. These models correlate measured metrics with semantically meaningful input or system state, allowing operators greater understanding into the behavior of the system. As an initial evaluation, we conducted a case study on a scaled-down replica of a real-world cloud application, SWITCHdrive, and identified several interesting observed behaviors. Overall, performance annotations are a first step towards a more general goal of providing operators and developers useful tools for analyzing and understanding the behavior of cloud applications.

Acknowledgments: We thank Simon Leinen and SWITCH for their support. This work is supported by the Swiss National Science Foundation (grants 200021-166132 and 200021-157164) and a Google Faculty Research Award.

8 Discussion

This paper describes initial results from our research on performance annotations. In on-going work, we are extending the system in a number of ways. Below, we discuss some of these efforts.

First, we are exploring an increased feature space. Our current prototype derives performance annotations based a single (scalar) feature. However, in practice, system performance is often dependent on a number of issues. Therefore, a natural extension is to add annotations that relate target metrics with two or more features (or a multi-dimensional feature).

Second, we are investigating alternative, potentially more effective, machine-learning techniques to deduce relations between features and target performance metrics. In particular, machine-learning techniques are typically tunable with several parameters, which can be adjusted to better fit the chosen application domain. To evaluate these techniques, we will need to extend our experimental analysis both for our SWITCHDrive deployment, and other distributed systems.

Third, we plan to adopt more formalized statistical methods to deduce and validate annotations. This will allow us to provide guarantees about the "quality" of our annotations, i.e., how well they predict performance.

Beyond these efforts towards the automatic derivation of performance annotations, we are also developing techniques to use the annotations to predict systems performance, and to support the design of complex systems. At a minimum, we plan to support "what-if" analysis and simulation. Pushing even further, we plan to use annotations together with code in building bottom-up abstractions. More concretely, one of our plans is to reason about the combination of different performance annotations using related concepts from probabilistic programming languages [4]. One approach used in this area that might also prove effective in our case is to use sampling techniques such as Monte Carlo simulation.

This research plan is ambitious, and and as with any project, caries with it a certain amount of uncertainty. We have identified a number of threats for the validity and success of this work. The main problem we face is the lack of information about a subject system. Up to now, we have assumed that, with a rich-enough set of heuristics and perhaps with the help of the designers and developers, we can identify significant features. The validity and usefulness of our annotations depend crucially on such features. To compensate for a possible lack of knowledge on meaningful features, we have developed the notion of clustered annotations. However, we plan to further explore ways to discover meaningful features and other other forms of auto-correlation of target metrics.

References

[1] Çiçek, E., Barthe, G., Gaboardi, M., Garg, D., and Hoffmann, J. Relational Cost Analysis. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 2017), pp. 316–329.

[2] Ceph. http://ceph.com.

[3] Coppa, E., and Finocchi, I. On Data Skewness, Stragglers, and MapReduce Progress Indicators. In *ACM Symposium on Cloud Computing (SOCC)* (Aug. 2015), pp. 139–152.

[4] Gordon, A. D., Henzinger, T. A., Nori, A. V., and Rajamani, S. K. Probabilistic programming. In *Proceedings of the on Future of Software Engineering* (New York, NY, USA, 2014), FOSE 2014, ACM, pp. 167–181.

[5] GNU gprof. https://sourceware.org/binutils/docs/gprof/.

[6] Gulwani, S., Jain, S., and Koskinen, E. Control-flow Refinement and Progress Invariants for Bound Analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2009), pp. 375–385.

[7] Gulwani, S., Mehra, K. K., and Chilimbi, T. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 2009), pp. 127–139.

[8] Hoffmann, J., Aehlig, K., and Hofmann, M. Multivariate Amortized Resource Analysis. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 2011).

[9] Hoffmann, J., Das, A., and Weng, S.-C. Towards Automatic Resource Bound Analysis for OCaml. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 2017), pp. 359–373.

[10] Hofmann, M., and Jost, S. Static Prediction of Heap Space Usage for First-order Functional Programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 2003), pp. 185–197.

[11] Jin, G., Song, L., Shi, X., Scherpelz, J., and Lu, S. Understanding and Detecting Real-World Performance Bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2012), pp. 77–88.

[12] Jost, S., Hammond, K., Loidl, H.-W., and Hofmann, M. Static Determination of Quantitative Resource Usage for Higher-Order Programs. In *ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)* (Jan. 2010), pp. 237–236.

[13] JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html.

[14] OpenStack. https://www.openstack.org/.

[15] ownCloud. https://owncloud.org.

[16] Perl, S. E., and Weihl, W. E. Performance Assertion Checking. In *ACM Symposium on Operating Systems Principles (SOSP)* (Dec. 1993), pp. 134–145.

[17] Reynolds, P., Killian, C. E., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the Unexpected in Distributed Systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (May 2006), pp. 115–128.

[18] runkit. http://php.net/manual/en/book.runkit.php.

[19] Zaparanuks, D., and Hauswirth, M. Algorithmic Profiling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (June 2012), pp. 67–76.