

# quiho: Automated Performance Regression Using Fine Granularity Resource Utilization Profiles

Ivo Jimenez

UC Santa Cruz

ivo.jimenez@ucsc.edu

Noah Watkins

UC Santa Cruz

nmwatkin@ucsc.edu

Michael Sevilla

UC Santa Cruz

msevilla@ucsc.edu

Jay Lofstead

Sandia National Laboratories

gflofst@sandia.gov

Carlos Maltzahn

UC Santa Cruz

carlosm@ucsc.edu

## ABSTRACT

We introduce *quiho*, a framework for profiling application performance that can be used in automated performance regression tests. *quiho* profiles an application by applying sensitivity analysis, in particular statistical regression analysis (SRA), using application-independent performance feature vectors that characterize the performance of machines. The result of the SRA, in particular feature importance, is used as a proxy to identify hardware and low-level system software behavior. The relative importance of these features serve as a performance profile of an application (termed fine granularity resource utilization profile or FGRUP), which is used to automatically validate performance behavior across multiple revisions of an application’s code base. We demonstrate that *quiho* can successfully discover performance regressions by showing its effectiveness in profiling application performance for synthetically induced regressions as well as those found in real-world applications.

## CCS CONCEPTS

- Software and its engineering → Software performance; Software testing and debugging; Acceptance testing; Empirical software validation;
- Social and professional topics → Automation;

### ACM Reference Format:

Ivo Jimenez, Noah Watkins, Michael Sevilla, Jay Lofstead, and Carlos Maltzahn. 2017. *quiho: Automated Performance Regression Using Fine Granularity Resource Utilization Profiles*. In *Proceedings of ACM Conference (Conference ’17)*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Quality assurance (QA) is an essential activity in the software engineering process [1–3]. Part of the QA pipeline involves the execution of performance regression tests, where the performance of the application is measured and contrasted against past versions [4–6]. Examples of metrics used in regression testing are throughput, latency, or resource utilization over time. These metrics are captured

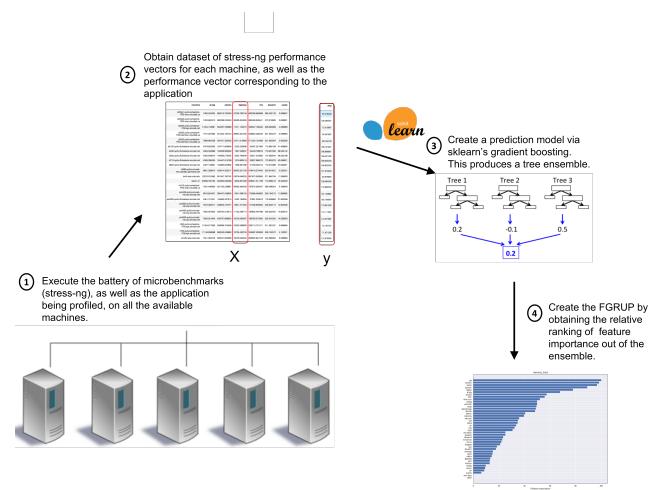
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

*Conference’17, July 2017, Washington, DC, USA*

© 2017 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnnn.nnnnnnn>



**Figure 1: *quiho*’s workflow for obtaining a fine granularity resource utilization profile (FGRUP).** A FGRUP can be used as an alternative for profiling application performance and can be used in automated regression testing. FGRUPs can also aid in root cause analysis.

and compared for multiple versions of an application (usually current and past versions) and, if significant differences are found, this constitutes a regression.

One of the main challenges in automating performance regression tests is defining the criteria to decide whether a change in an application’s performance behavior is significant [7]. Understanding the effects in performance that distinct hardware and low-level system software<sup>1</sup> components have on applications demands highly-skilled performance engineering [8–10]. Traditionally, this investigation is done by an analyst in charge of looking at changes to the performance metrics captured at runtime, possibly investigating deeply into by looking at performance counters, performance profiles, static code analysis, and static/dynamic tracing. One common approach is to find bottlenecks by generating a profile (e.g., using the perf Linux kernel tool) in order to understand which parts of the system an application is hammering on [5].

Automated solutions have been proposed in recent years [11–13]. The general approach of these is to analyze runtime logs and/or

<sup>1</sup>Throughout this paper, we use “system” to refer to the low-level compute stack composed by hardware, firmware and the operating system (OS).

metrics application in order to build a performance prediction model that can be used to automatically determine whether a regression has occurred. This relies on having accurate predictions and, as with any prediction model, there is the risk of finding false/positive/negatives. Rather than striving for highly accurate predictions, one can use performance modeling as a profiling tool.

In this work, we present *quiho* an approach aimed at complementing automated performance regression testing by using fine granularity resource utilization profiles (FGRUP) associated to an application (Fig. 1). FGRUPs are an alternative way of profiling the performance of an application that relies on applying Statistical Regression Analysis<sup>2</sup> (SRA) on a dataset of application-independent performance vectors. The main assumption behind *quiho* is the availability of multiple machines when exercising performance regression testing, a reasonable requirement that is well-aligned with current software engineering practices (performance regression is carried out on multiple architectures and OS).

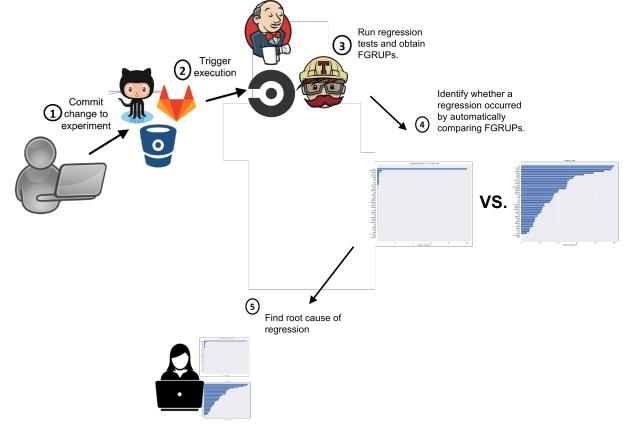
When an application is profiled using *quiho*, the underlying system is baselined by executing a battery of microbenchmarks. This matrix (or dataset) of performance vectors characterizes the available machines independently from any application and can be used (and re-used) as the foundation for applying statistical learning techniques such as SRA. In order to obtain a FGRUP, the application under study is executed on the same machines from where the performance vectors were obtained, and SRA is applied. The results of the SRA for an application, in particular feature importance, is used as a proxy to characterize hardware and low-level system utilization behavior. The relative importance of these features serve as the FGRUP of an application, which is used to automatically validate its performance behavior across multiple revisions of its code base.

In this article, we demonstrate that *quiho* can successfully identify performance regressions. We show (Section 4) that *quiho* (1) obtains resource utilization profiles for application that reflect what their codes do and (2) effectively uses these profiles to identify induced regressions as well as other regressions found in real-world applications. The contributions of our work are:

- Insight: feature importance in SRA models (trained using these performance vectors) gives us a resource utilization profile of an application without having to look at the code.
- An automated end-to-end framework (based on the above finding), that aids analysts in identifying significant changes in resource utilization behavior of applications which can also aid in identifying root cause of regressions.
- Methodology for evaluating automated performance regression. We introduce a set of synthetic benchmarks aimed at evaluating automated regression testing without the need of real bug repositories. These benchmarks take as input parameters that determine their performance behavior, thus simulating different “versions” of an application.
- A negative result: ineffectiveness of resource utilization profiles for predicting performance using ensemble learning.

Next section (Section 2) shows the intuition behind *quiho* and how can be used to automate regression tests. We then do a more

<sup>2</sup>We use the term *Statistical Regression Analysis* (SRA) to differentiate between regression testing in software engineering and regression analysis in statistics.



**Figure 2: Automated regression testing pipeline integrating fine granularity resource utilization profiles (FGRUP).** FGRUPs are obtained by *quiho* and can be used both, for identifying regressions, and to aid in the quest for finding the root cause of a regression.

in-depth description of *quiho* (Section 3), followed by our evaluation of this approach (Section 4). We briefly show how *quiho*'s resource utilization profiles can not be used to predict performance using some common machine learning techniques (Section 4.4). We then discuss different aspects of our work (Section 5), review (Section 6) related work and we lastly close with a brief discussion on challenges and opportunities enabled by *quiho* (Section 7).

## 2 MOTIVATION AND INTUITION BEHIND QUIHO

Fig. 2 shows the workflow of an automated regression testing pipeline and shows how *quiho* fits in this picture. A regression is usually the result of observing a significant change in a performance metric of interest (e.g., runtime). At this point, an analyst will investigate further in order to find the root cause of the problem. One of these activities involves profiling an application to see the resource utilization pattern. Traditionally, coarse-grained profiling (i.e. CPU-, memory- or IO-bound) can be obtained by monitoring an application's resource utilization over time. Fine granularity behavior helps application developers and performance engineers quickly understand what they need to focus on while refactoring an application.

Fine granularity performance utilization behavior can better inform the regression testing pipeline. Examples of which resources are included in this type of profiling are the OS memory mapping subsystem, the CPU's cryptographic unit, or the CPU cache. This type of profiling is time-consuming and requires use of more computing resources. This is usually done offline by analysts and involves eyeballing source code, static code analysis, or analyzing hardware/OS performance counters/profiles.



(“unseen”) machines [24], a natural next step to take with a dataset like this. As we show in Section 4.4, this is not as good as we would expect.

However, building a prediction model has a utility. If we use these performance vectors to apply SRA and we focus on feature importance [25] of the created models, we can see that they give us fine granularity resource utilization patterns. In Fig. 3, we show the intuition behind why this is so. The performance of an application is determined by the performance of the subcomponents that get stressed the most by the application’s code. Thus, intuitively, if the performance of an application across multiple machines resembles the performance of a microbenchmark, then we can say that the application is heavily influenced by that subcomponent. In other words, if the variability pattern of a feature across multiple machines resembles the variability pattern of application performance across those same machines, it is likely due to the application stressing the same subcomponent that the corresponding microbenchmark stresses. While this can be inferred by obtaining correlation coefficients, proper SRA is needed in order to create prediction models, as well as to obtain a relative rank of feature importances.

If we rank features by their relative importance, we obtain what we call a fine granularity resource utilization profile (FGRUP), as shown in Fig. 4. In the next section we show how these FGRUPs can be used in automated performance regression tests. Section 4 empirically validates this approach.

### 3 OUR APPROACH

In this section we describe *quihō*’s approach and the resulting prototype. We first describe how we obtain the performance vectors that characterize system performance. We then show that we can feed these vectors to SRA in order to build a performance model for an application. Lastly, we describe how we obtain feature importance, how this represents a fine granularity resource utilization profile (FGRUP) and the algorithm (and alternative heuristics) to comparing FGRUPs.

#### 3.1 Performance Feature Vectors As System Performance Characterization

While the hardware and software specification can serve to describe the performance characteristics of a machine, the real performance characteristics can only feasibly be obtained by executing programs and capturing metrics. One can generate arbitrary performance characteristics by interposing a hardware emulation layer and deterministically associate performance characteristics to each instruction based on specific hardware specs. While possible, this is impractical (we are interested in characterizing “real” performance). The question then boils down to which programs should we use to characterize performance? Ideally, we would like to have many programs that execute every possible opcode mix so that we measure their performance. Since this is an impractical solution, an alternative is to create synthetic microbenchmarks that get as close as possible to exercising all the available features of a system.

*stress-ng*[26] is a tool that is used to “stress test a computer system in various selectable ways. It was designed to exercise various physical subsystems of a computer as well as the various operating system kernel interfaces”. There are multiple stressors

**Table 1: List of stressors used in this paper, along with the categories assigned to them by *stress-ng*. Note that some stressors are part of multiple categories.**

stressor	CPU	Cache	Mem	VM
af-alg	X			
atomic	X		X	
bigheap				X
brk	X			
bsearch	X	X	X	
cache		X		
cpu	X			
crypt	X			
full				X
heapsort	X	X	X	
hsearch	X	X	X	
icache		X		
lockbus			X	X
longjmp	X			
lsearch	X	X	X	
malloc		X	X	X
matrix	X	X	X	
memcpy			X	
mincore			X	
mmap				X
mremap				X
msync				X
nop	X			
numa	X		X	
oom-pipe			X	
qsort	X	X	X	
remap			X	X
resources			X	
rmap			X	
shm				X
shm-sysv				X
stack			X	X
stackmmap			X	X
str	X	X	X	
stream	X		X	
tsearch	X	X	X	
vecmath	X	X		
vm			X	X
vm-rw			X	X
vm-rw				X
vm-splice				X
zero				X

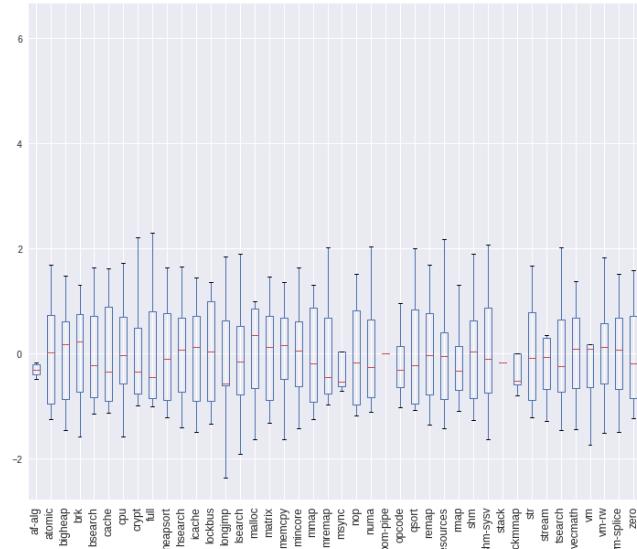
for CPU, CPU cache, memory, OS, network and filesystem. Since we focus on system performance bandwidth, we execute the (as of version 0.07.29) 42 stressors for CPU, CPU cache, memory and virtual memory stressors (Tbl. 1 shows the list of stressors used in this paper). A *stressor* (or microbenchmark) is a function that loops for a fixed amount of time, exercising a particular subcomponent of the system. At the end of its execution, *stress-ng* reports the rate of iterations executed for the specified period of time (referred to as *bogo-ops-per-second*).

Using this battery of stressors, we can obtain a performance profile of a machine (a performance vector). When this vector is compared against the one corresponding to another machine, we can quantify the difference in performance between the two at a per-stressor level. Fig. 5 shows the variability in these performance vectors.

Every stressor (element in the vector) can be mapped to basic features of the underlying platform. For example, *bigheap* is directly associated to memory bandwidth, *zero* to memory mapping, *qsort* to CPU performance (in particular to sorting data), and so on

**Table 2: Table of machines from CloudLab. The last three entries correspond to computers in our lab.**

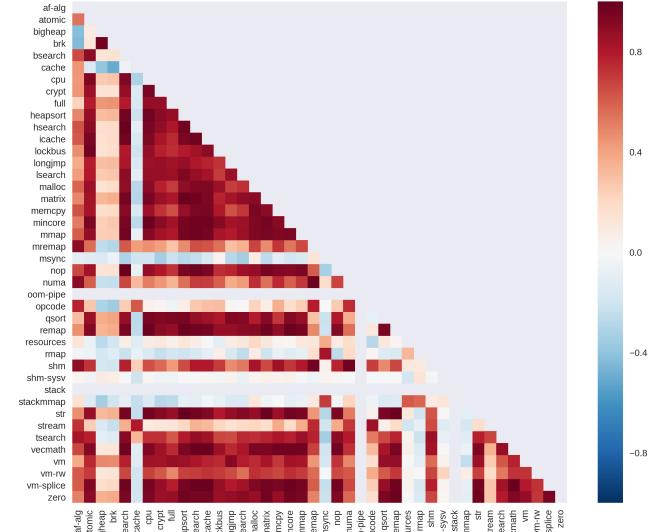
machine	cpu	num_cpus	cores
c220g2	Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz	2	8
c8220	Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz	2	10
dl360	Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz	2	8
m510	Intel(R) Xeon(R) CPU D-1548 @ 2.00GHz	1	8
pc2400	Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz	1	4
pc3000	Intel(R) Xeon(TM) CPU 3.00GHz	1	1
pc3500	Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz	1	4
r720	Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz	1	8
scruffy	Intel(R) Xeon(R) CPU E5620 @ 2.40GHz	1	4
dwill	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz	1	4
issdm-41	Dual-Core AMD Opteron(tm) Processor 2212	2	2



**Figure 5: [source] Boxplots illustrating the variability of the performance vector dataset. Each stressor was executed five times on each of the machines listed in Tbl. 2.**

and so forth. However, the performance of a stressor in this set is *not* completely orthogonal to the rest, as implied by the overlapping categories in Tbl. 1. Fig. 6 shows a heat-map of Pearson correlation coefficients for performance vectors obtained by executing `stress-ng` on all the distinct machine configurations available in CloudLab [19] (Tbl. 2 shows a summary of their hardware specs). As the figure shows, some stressors are slightly correlated (those near 0) while others show high correlation between them (in Section 4.4 we apply principal component analysis to this dataset).

In order to analyze this last point further, that is, to try to discern whether there are a few orthogonal features that we could focus on, rather than looking at the totality of the 42 stressors, we applied principal component decomposition (PCA) [27]. Fig. 7 shows the result. The figure shows the relative (blue) and cumulative (green) explained variance ratio. As we can see, having 6-8 components would be enough to explain most of the variability in the dataset. This confirms what we observe in Fig. 6, in terms of having many stressors that can be explained in function of others. However,



**Figure 6: [source] heat-map of Pearson correlation coefficients for performance vectors obtained by executing `stress-ng` on all the distinct machine configurations available in CloudLab.**

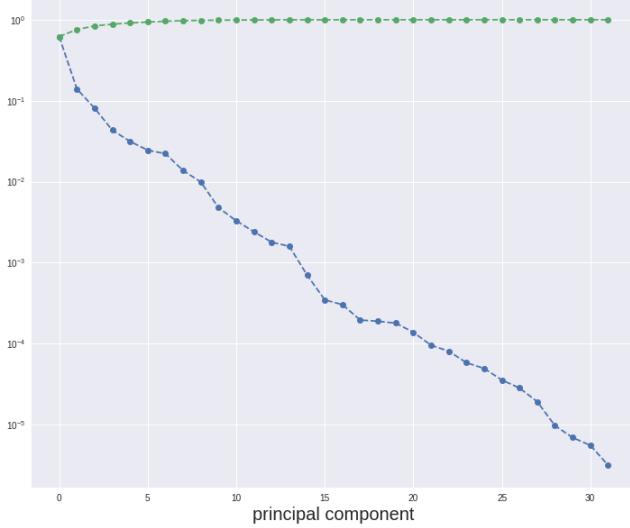
doing so would cause the loss of information with respect to what stressors are explaining the prediction. Instead of trying to reduce the number, we decide to leave all the stressors in order to not lose this information. Part of our future work is to address whether we can reduce the number of features with the goal of improving the models, without having to lose information about which stressors are involved in the prediction.

### 3.2 System Resource Utilization Via Feature Importance in SRA

SRA is an approach for modeling the relationship between variables, usually corresponding to observed data points [28]. One or more independent variables are used to obtain a *regression function* that explains the values taken by a dependent variable. A common approach is to assume a *linear predictor function* and estimate the unknown parameters of the modeled relationships.

A large number of procedures have been developed for parameter estimation and inference in linear regression. These methods differ in computational simplicity of algorithms, presence of a closed-form solution, robustness with respect to heavy-tailed distributions, and theoretical assumptions needed to validate desirable statistical properties such as consistency and asymptotic efficiency. Some of the more common estimation techniques for linear regression are least-squares, maximum-likelihood estimation, among others.

`scikit-learn` [29] provides with many of the previously mentioned techniques for building regression models. Another technique available in `scikit-learn` is gradient boosting [30]. Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees



**Figure 7: [source]** Principal Component Analysis for the performance vector dataset. The y-axis (log-scale) corresponds to the explained variance ratio, while the x-axis denotes the number of components. The blue line denotes the amount of variance reduced by having a particular number of components. The green line corresponds to the cumulative sum of the explained variance.

[31]. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function. This function is then optimized over a function space by iteratively choosing a function (weak hypothesis) that points in the negative gradient direction.

Once an ensemble of trees for an application is generated, feature importances are obtained in order to use them as the FGRUP for an application. Fig. 1 shows the process applied to obtaining FGRUPs for an application. scikit-learn implements the feature importance calculation algorithm introduced in [32]. This is sometimes called *gini importance* or *mean decrease impurity* and is defined as the total decrease in node impurity, weighted by the probability of reaching that node (which is approximated by the proportion of samples reaching that node), averaged over all trees of the ensemble.

We note that before generating a regression model, we normalize the data using the StandardScaler method from scikit-learn, which removes the mean (centers it on zero) and scales the data to unit variance. Given that the bogo-ops-per-second metric does not quantify work consistently across stressors, we normalize the data in order to prevent some features from dominating in the process of creating the prediction models (the data for Fig. 8 has been normalized prior to being plotted). In section Section 4 we evaluate the effectiveness of FGRUPs.

### 3.3 Using FGRUPs in Automated Regression Tests

As shown in Fig. 2 (step 4), when trying to determine whether a performance degradation occurred, FGRUPs can be used to compare

differences between current and past versions of an application. In order to do so, we apply a simple algorithm. Given two profiles  $A$  and  $B$ , and an arbitrary  $\epsilon$  value, look at first feature in the ranking (highest in the chart). Then, compare the relative importance value for the feature and importance values for  $A$  and  $B$ . If relative importance is not within  $+/- \epsilon$ , the importance is considered not equivalent and the algorithm stops. If values are similar (within  $+/- \epsilon$ ), we move to the next, less important factor and the compare again. This is repeated for as many features are present in the dataset.

FGRUPs can also be used as a pointer to where to start with an investigation that looks for the root cause of the regression (Fig. 2, step 5). For example, if *memorymap* ends up being the most important feature, then we can start by looking at any code/libraries that make use of this subcomponent of the system. An analyst could also trace an application using performance counters and look at corresponding performance counters to see which code paths make heavy use of the subcomponent in question.

## 4 EVALUATION

In this section we answer the following questions:

1. Can FGRUPs accurately capture application performance behavior? (Section 4.1)
2. Can FGRUPs work for identifying simulated regressions? (Section 4.2)
3. Can FGRUPs work for identifying regressions in real world software projects? (Section 4.3)
4. Can performance vectors be used to create performance prediction models? (Section 4.4)

**Note on Replicability of Results:** This paper adheres to The Popper Experimentation Protocol and convention<sup>4</sup> [33], so experiments presented here are available in the repository for this article<sup>5</sup>. We note that rather than including all the results in the paper, we instead include representative ones for each section and leave the rest on the paper repository. Experiments can be examined in more detail, or even re-executed, by visiting the [source] link next to each figure. That link points to a Jupyter notebook that shows the analysis and source code for that graph. The parent folder of the notebook (following the Popper's file organization convention) contains all the artifacts and automation scripts for the experiments. All results presented here can be replicated<sup>6</sup>, as long as the reader has an account at Cloudlab (see repo for more details).

### 4.1 Effectiveness of FGRUPs to Capture Resource Utilization Behavior

In this subsection we show how FGRUPs can effectively describe the fine granularity resource utilization of an application with respect to a set of machines. Our methodology is:

1. Given an application  $A$ , discover relevant performance features using the *quiho* framework.

<sup>4</sup><http://falsifiable.us>

<sup>5</sup><http://github.com/ivotron/quiho-popper>

<sup>6</sup>**Note to reviewers:** based on the terminology described in the ACM Badging Policy [34] this complies with the *Results Replicable* category. In the event that our paper gets accepted, we plan to submit this work to the artifact review track too.

- Do manual performance analysis of  $A$  to corroborate that discovered features are indeed the cause of performance differences.

Fig. 4 shows the profile of an execution of the hpccg miniapp [14]. This proxy (or miniapp) application [??] is a “conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors [that] generates a 27-point finite difference matrix with a user-prescribed sub-block size on each processor.” [14].

Based on the profile, stackmmap and cache are the most important features. In order to corroborate if this matches with what the application does, we profiled this execution with perf. The stacked profile view shows that ~85% of the time the application is running the function HPC\_sparsenv(). The code for this function is shown in Lst. 1. As the name implies, this snippet implements a sparse vector multiplication function of the form  $y = Ax$  where  $A$  is a sparse matrix and the  $x$  and  $y$  vectors are dense. By looking at this code, we see that the innermost loop iterates an array, accumulating the sum of a multiplication. This type of code is a potential candidate for manifesting bottlenecks associated with CPU cache locality [35].

**Listing 1** Source code for bottleneck function in HPCCG.

```
int HPC_sparsenv(HPC_Sparse_Matrix *A,
                  const double * const x,
                  double * const y)
{
    const int nrow = (const int) A->local_nrow;

    for (int i=0; i< nrow; i++) {
        double sum = 0.0;
        const double * const cur_vals =
            (const double * const) A->ptr_to_vals_in_row[i];

        const int * const cur_inds =
            (const int * const) A->ptr_to_inds_in_row[i];

        const int cur_nnz = (const int) A->n nz_in_row[i];

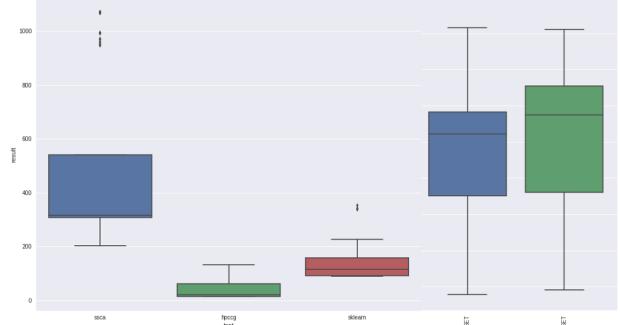
        for (int j=0; j< cur_nnz; j++)
            sum += cur_vals[j]*x[cur_inds[j]];
        y[i] = sum;
    }

    return(0);
}
```

We analyze the performance of this benchmark further by obtaining performance counters for the application and comparing the counters with those from the top three features (Tbl. 3 shows the summary of hardware-level performance counters). Given that hardware performance counters are architecture-dependent, we can not make generalizations about given that we run an application on a multitude of machines. However, we can see that in the case of the

**Table 3: Table of performance counters for the HPCCG performance test.**

counter	HPCCG	stackmmap	cache	bigheap
ins. per cycle	0.78	0.18	0.25	0.39
stalled cycles p/ins.	0.53	2.29	3.52	1.44
stalled cycles (frontend)	13.51%	41.09%	89.70%	18.95%
stalled cycles (backend)	41.19%	9.23%	0.80%	56.53%
branch misses	2.87%	9.24%	0.01%	0.69%
L1-dcache misses	5.62%	5.47%	52.91%	2.75%
LLC misses	1.03%	16.41%	51.88%	5.60%



**Figure 8: [source]** Variability of the five applications presented in this subsection. Y-axis has been normalized.

stackmmap stressor, there are some similarities between the counters, specially the ones denoting stalled cycles (which significantly determine application performance [36,37]).

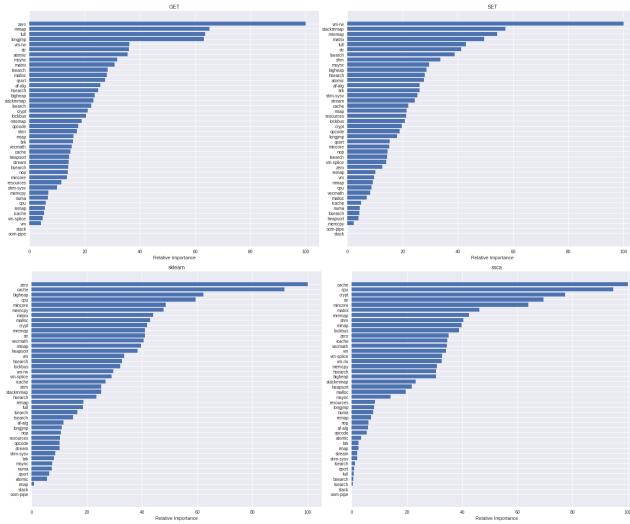
Next, we analyze FGRUPs of other four applications<sup>7</sup>. These applications are Redis [38], Scikit-learn [29], and SSCA [39]. Due to space constraints we omit a similar detailed analysis as the one presented above for hpccg. However, resource utilization characteristics of these code bases is well known and we verify FGRUPs using this high-level intuition. As a way to illustrate the variability originating from executing these applications on a heterogeneous set of machines, Fig. 8 shows boxplots of the these.

In Fig. 9 we show FGRUPs for these four applications. The first two on the top correspond to two tests of Redis, a popular open-source in-memory key-value database. These two tests are SET, GET from the redis-benchmark command that test operations that store and retrieve key-value pairs into/from the DB, respectively. The resource utilization profiles suggest that SET and GET are memory intensive operations (first 3 stressors from each test, as shown in Tbl. 1), which is an obvious conclusion.

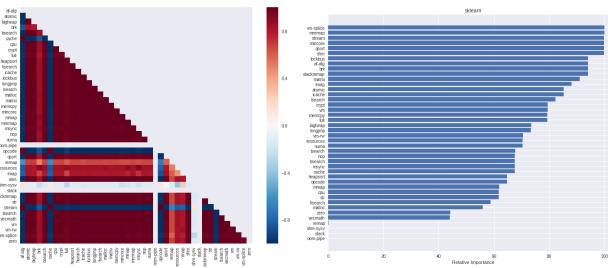
The next two FGRUPs (below) correspond to performance tests for Scikit-learn and SSCA. In the case of Scikit-learn, this test runs a comparison of a several classifiers in on a synthetic dataset. Scikit-learn uses NumPy [40] internally, which is known to be memory-bound. The profile is aligned to this known behavior since the zero microbenchmark stresses access.

The last application is SSCA, a graph analysis benchmark comprising of a data generator and 4 kernels which operate on the graph.

<sup>7</sup>For brevity, we omit other results that corroborate FGRUPs can correctly identify resource utilization patterns. All these are available in the github repository associated to this article.



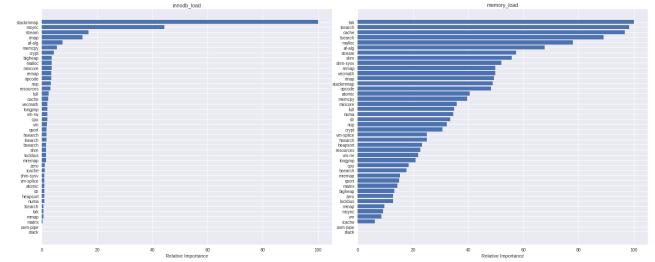
**Figure 9:** [source] FGRUPs of the four tests used in this section.



**Figure 10:** [source] Correlation matrix and FGRUP obtained from only two randomly selected machines from Tbl. 2 (issdm-41 and r320 in this case).

The benchmark is designed to have very little locality, which causes the application to generate many cache misses. As shown in the profile, the first feature corresponds to the cache stressor, which as it was explained earlier, stresses the CPU cache by generating a non-locality workload.

The reader might have noticed that, regardless of how the performance of an application looks like, SRA will always produce a model with associated feature importances. Thus, one can pose the following question: is there any scenario where a FGRUP is *not* correctly associated with what the application is doing? In other words, are FGRUPs falsifiable? The answer is yes. A FGRUP can be incorrectly representing an application's performance behavior if there is under- or over-fitting when generating the model. Fig. 10 shows the correlation matrix (left), as well as a FGRUP obtained from selecting two random machines from the set of available ones. The application in question is Scikit-learn and, as the figures show, this FGRUP is of little use since many features have 100% relevance. The correlation matrix shows why this is so: almost all the features are highly correlated. As mentioned before, an open problem is that one of systematically reducing the number of required machines.



**Figure 11:** [source] MariaDB with innodb and in-memory backends.

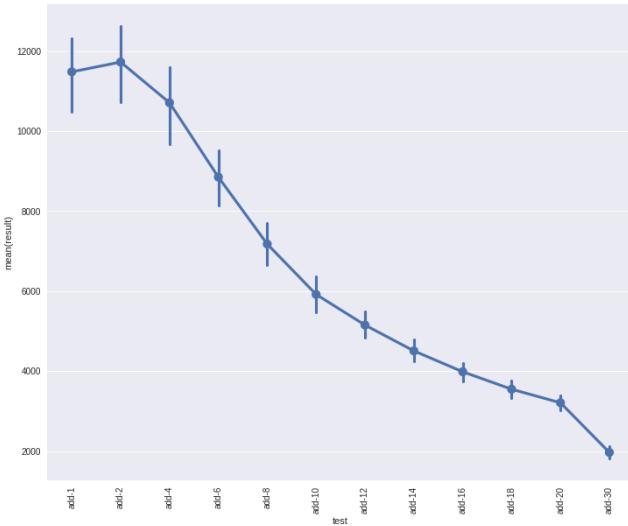
## 4.2 Simulating Regressions

In this section we test the effectiveness of *quiho* to detect performance simulations that are artificially induced. We induce regression by having a set of performance tests that take, as input, parameters that determine their performance behavior, thus simulating different “versions” of the same application. In total, we have 10 benchmarks for which we can induce several performance regressions, for a total of 20 performance regressions. For brevity, in this section we present results for two applications, MariaDB [41] and a modified version of the STREAM benchmark.

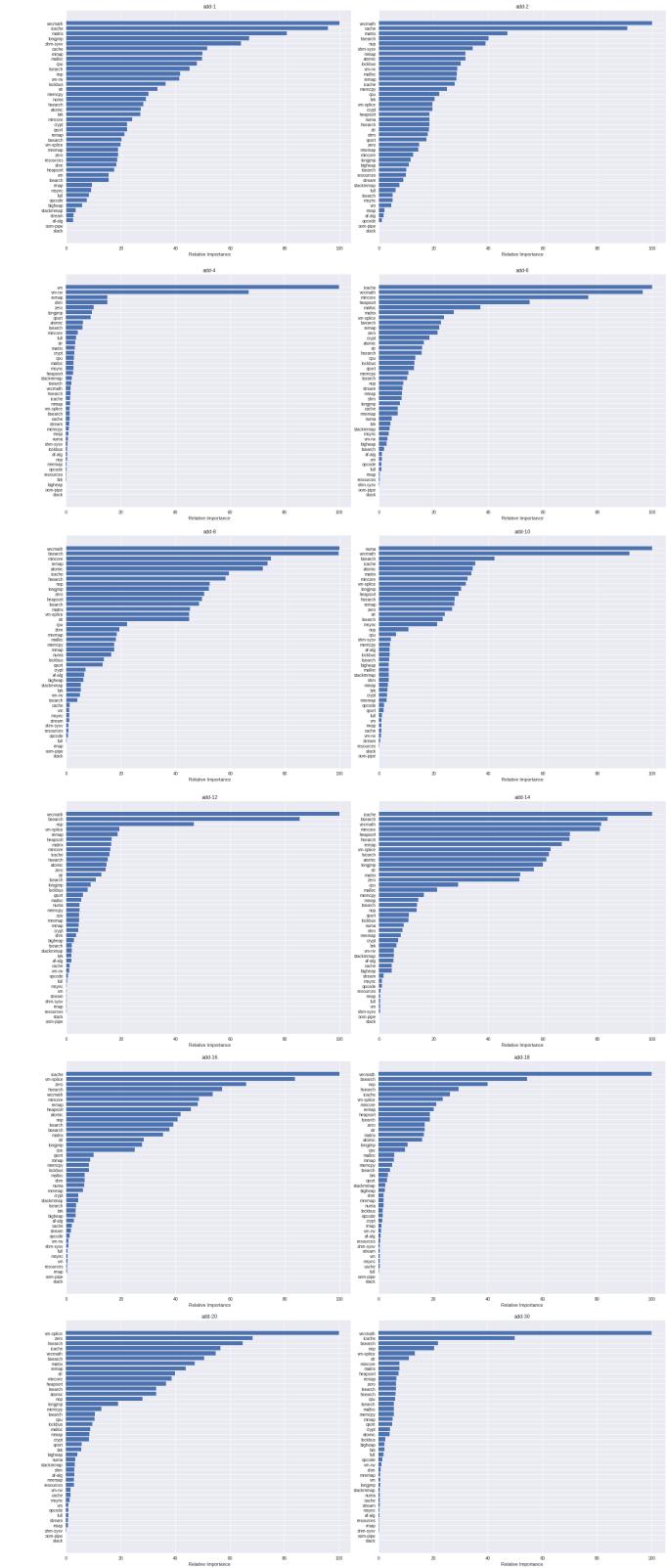
The MariaDB test is based on the `mysqlslap` utility for stressing the database engine. In our case we run the data loading test, which populates a database whose schema is specified by the user. We have a fixed set of parameters that load a 10GB database. One of the exposed parameters is the one that selects the backend (storage engine in MySQL terminology). While the workload and test parameters are the same, the code paths are distinct and thus present different performance characteristics. The two engines we use in this case are `innodb` and `memory`. Fig. 11 shows the profiles of MariaDB performance for these two engines.

The next test is a modified version of the STREAM benchmark [42], which we refer to as STREAM-NADDS (introduced in [43]). This version of STREAM introduces a NADDS pre-processor parameter that controls the number additions for the Add test of the STREAM benchmark. In terms of the code, when NADDS equals to 1 is equivalent to the “vanilla” STREAM benchmark. For any value greater than 1, the code adds a new term to the sum being executed. Intuitively, since the vanilla version of STREAM is memory bound, so adding more terms to the sum causes the CPU to do more work, eventually moving the bottleneck from memory to being cpu-bound; the higher the value of the NADDS parameter, the more cpu-bound the test gets. Fig. 12 shows this behavior.

Fig. 13 shows the FGRUPs for the four tests. On the left, we see the resource utilization behavior of the “vanilla” version of STREAM (which corresponds to a value of 1 for the NADDS parameter). As expected, the associated features (stressors) to these are from the memory/VM category, in particular `vecmath`. As the number of terms for the sum increases, the test moves all the way to being CPU-bound (at NADDS=30), which can be seen by observing the `bsearch` and `hsearch` features going up in importance as the number of additions increases.



**Figure 12:** [source] General behavior of the STREAM-NADDS performance test. The y-axis is the throughput of the test in MB/s. The x-axis corresponds to the number of terms in the sum expression of the Add STREAM subtest. The regular (“vanilla”) STREAM add test is memory bound, so adding more terms to the Add subtest moves the performance from memory- to cpu-bound; the higher the value of the NADDS parameter, the more CPU-bound the test gets. This test was executed across all available machines (5 times). The bars denote standard deviation.



**Figure 13:** [source] The FGRUPs for modified version of STREAM. The parameter of NADDS increases by taking values of 1, 2, 4, ..., 20 and 30. We see that they capture the simulated regression which causes this application to be moving from being memory-bound to being cpu-bound.

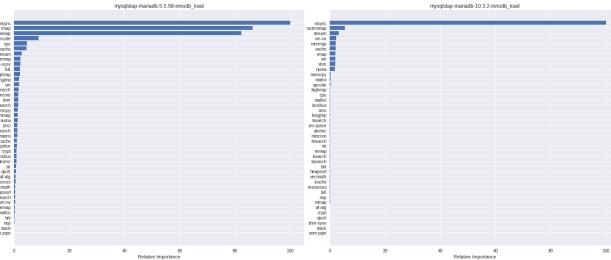
### 4.3 Real world Scenario

In this section we show that *quiho* works with regressions that can be found in real software projects. It is documented that the changes made to the innodb storage engine in version 10.3.2 improves the performance in MariaDB, with respect to previous version 5.5.58. If we take the development timeline and invert it, we can treat 5.5.58 as if it was a “new” revision that introduces a performance regression. To show that this can be captured with FGRUPs, we use *mysqlslap* again and run the load test. Fig. 14 shows the corresponding FGRUPs. We can observe that the FGRUP generated by *quiho* can identify the difference in performance.

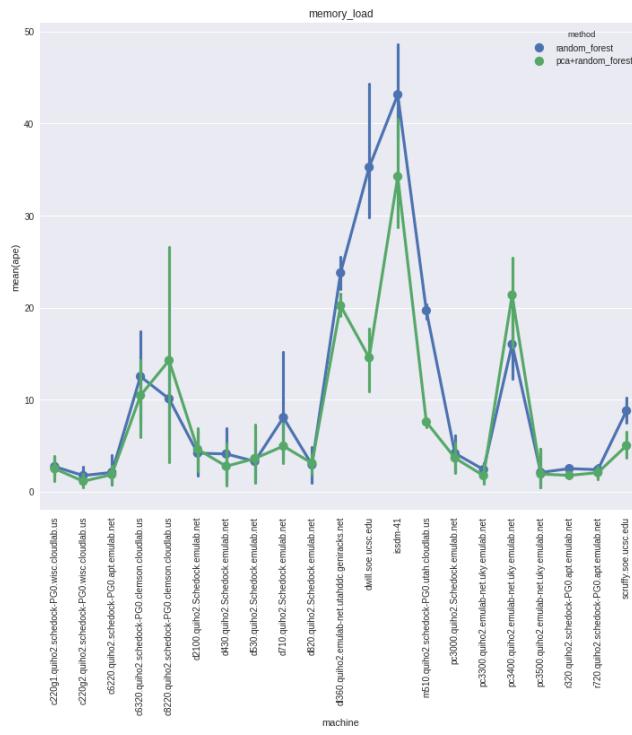
For brevity, we omit regressions found in other 4 applications (*zlog*, *postgres*, *redis*, and *apache web server*).

### 4.4 Using Performance Vectors to Predict Performance

As mentioned earlier, the set of performance vectors obtained as part of the generation of FGRUPs could be used to create prediction models that try to estimate the performance of an application using. Fig. 15 shows a plot with mean absolute percentage errors (MAPE) corresponding to the outcome of doing 1-cross-validation [44] across the distinct type of hardware architectures found in CloudLab. The 1-cross-validation is done by creating a training dataset composed of performance vectors from all but one machine.



**Figure 14:** [source] A regression that appears from going in the reversed timeline (from mariadb-10.0.3 to 5.5.38).



**Figure 15:** [source] Mean Absolute Percentage Error of cross-validation.

We then generate the model using this training subset as the independent variables and the performance metric associated to an application performance as the dependant variable. We then test obtain the accuracy of the model on the data corresponding to the machine that we left out. Before

We create two prediction models. The first one is using random forest [45] to create a linear regression performance model (blue line). We select random forest (as opposed to selecting other alternatives), since it is the one with the highest estimation accuracy from all the ones we tried. As mentioned previously, data is first normalized to prevent dimensionality issues. The second model is obtained by creating a pipeline, where principal component analysis (PCA) is applied first and then random forest is applied next (green line). As we can see, the prediction errors range from 3% up to almost 50%

in the worst-case scenario. Compared to the status quo [46], where good performance prediction models are those with less than 2-3% MAPE.

## 5 DISCUSSION

We briefly discuss some aspects related to *quiho*.

### Application-Independent Performance Characterization.

We used a subset of stress-ng microbenchmarks to quantify machine performance but the approach is not limited to this benchmarking toolkit. Ideally, we would like to extend the amount and type of stressors so that we have more coverage over the distinct subcomponents of a system. An open question is to systematically test whether the current set of stressors is sufficient to cover all sub-components of a system, and at the same time reduce the number of microbenchmarks.

**Quiho vs. other tools.** The main advantage of *quiho* over other performance profiling tools is that it is automatic and 100% hands-off. As mentioned before, the main assumption being that there exist performance vectors (or they are obtained when the as part of the test) for a sufficiently varied set of machines. We see *quiho* as a complement, not a replacement of perf, to existing performance engineering practices: once a test has failed *quiho*'s checks, then proceed to make use of existing tools.

**FGRUP Comparison.** The algorithm specified in Section 3.3 is a straight-forward one. One could think of more sophisticated ways of doing FGRUP comparison and finding equivalences. For example, using the categories from Tbl. 1, one could try to group stressors and determine coarse-grained bottlenecks, instead of fine grained ones. Another alternative is to do reduce the number of features by applying PCA, exploratory factor analysis (EFA), or singular value decomposition (SVD), and compare profiles in terms of the mapped factors.

**FGRUP as a visualization tool.** The reader might have noticed that FGRUPs can be visually compared by the human eye (and are somewhat similar in this regard to FlameGraphs [47]). Adding a coloring scheme to FGRUPs might make it easier to interpret the differences. For example, the categories in Tbl. 1 could be used to define a color palette (by assigning a color to each subset of the powerset of categories).

**Reproducibility.** Providing performance vectors alongside experimental results allows to preserve information about the performance characteristics of the underlying system that an experiment observed at the time it ran. This is a quantifiable snapshot that provides context and facilitates the interpretation of results. Ideally, this information could be used as input for emulators and virtual machines, in order to recreate original performance characteristics.

**Enforcement Learning.** Over the course of its life, an application will be tested on many platforms. If we can have an ever-growing list of machines where an application is tested, the more we run an application in a scenario like this, the more rich the performance vector dataset (and associated application performance history). This can serve as the foundation to apply becomes we learn about its properties. For example, if we had performance vectors captured as part of executions of the Phoronix benchmark suite (which has public data on openbenchmarking.org), we could leverage such a dataset to create rich performance models.

## 6 RELATED WORK

### 6.1 Automated Regression Testing

Automated regression testing can be broken down in the following three steps:

1. In the case of large software projects, decide which tests to execute [48]. This line of work is complementary to *quiho*.
2. Once a test executes, decide whether a regression has occurred [49]. This can be broken down in mainly two categories, as explained in [12]: pair-wise comparisons and model assisted. *quiho* fits in the latter category, the main difference being that, as opposed to existing solutions, *quiho* does not rely on having accurate prediction models since its goal is to describe resource utilization (obtain FGRUPs).
3. If a regression is observed, automatically find the root cause or aid an analyst to find it [13,50,51]. While *quiho* does not find the root cause of regressions, it complements the information that an analyst has available to investigate further.

### 6.2 Decision Trees In Performance Engineering

In [52] the authors use decision trees to detect anomalies and predict performance SLO violations. They validate their approach using a TPC-W workload in a multi-tiered setting. In [12], the authors use performance counters to build a regression model aimed at filtering out irrelevant performance counters. In [53], the approach is similar but statistical process control techniques are employed instead.

In the case of *quiho*, the goal is to use decision trees as a way of obtaining feature performance, thus, as opposed to what it's proposed in [12], the leaves of the generated decision trees contain actual performance predictions instead of the name of performance counters

### 6.3 Correlation-based Analysis and Supervised Learning

Correlation-based and supervised learning approaches have been proposed in the context of software testing, mainly for detecting anomalies in application performance [50]. In the former, runtime performance metrics are correlated to application performance using a variety of distinct metrics. In supervised learning, the goal is the same (build prediction models) but using labeled datasets.

Decision trees are a form of supervised learning, however, given that *quiho* applies regression rather than classification techniques, it does not rely on labeled datasets. Lastly, *quiho* is not intended to be used as a way of detecting anomalies, although we have not analyzed its potential use in this scenario.

## 7 LIMITATIONS AND FUTURE WORK

The main limitation in *quiho* is the requirement of having to execute a test on more than one machine in order to obtain FGRUPs. As mentioned, an open problem is to precisely quantify the minimum amount of required machines. On the other hand, we can avoid having to run `stress-ng` every time the application gets tested by integrating this into the infrastructure (e.g., system administrators can run `stress-ng` once a day or once a week and make this information for every machine available to users).

We are currently working in adapting this approach to profile distributed and multi-tiered applications. We also plan to analyze the viability of using *quiho* in multi-tenant configurations. Lastly, long-running (multi-stage) applications, e.g., a web-service or big-data application with multiple stages. In this case, we would define windows of time and we would apply *quiho* to each. The challenge: how do we automatically get the windows rightly placed.

In the era of cloud computing, even the most basic computer systems are complex multi-layered pieces of software, whose performance properties are difficult to comprehend. Having complete understanding of the performance behavior of an application, considering the parameter space (workloads, multi-tenancy, etc.) is challenging. One application of *quiho* we have in mind is to couple it with automated black-box (or even gray-box) testing frameworks to improve our understanding of complex systems.

**Acknowledgments:** This work was partially funded by the Center for Research in Open Source Software<sup>8</sup>, Sandia National Laboratories and NSF Award #1450488.

## REFERENCES

- [1] G.J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, 2011.
- [2] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," *2007 Future of Software Engineering*, 2007.
- [3] B. Beizer, *Software Testing Techniques*, 1990.
- [4] J. Dean and L.A. Barroso, "The tail at scale," *Commun ACM*, vol. 56, Feb. 2013.
- [5] B. Gregg, *Systems Performance: Enterprise and the Cloud*, 2013.
- [6] F.I. Vokolos and E.J. Weyuker, "Performance Testing of Software Systems," *Proceedings of the 1st International Workshop on Software and Performance*, 1998.
- [7] L. Cherkasova, K. Ozonat, N. Mi, J. Symons, and E. Smirni, "Anomaly? Application change? Or workload change? Towards automated detection of application performance anomaly and change," *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008.
- [8] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and Detecting Real-world Performance Bugs," *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.
- [9] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie, "Performance Debugging in the Large via Mining Millions of Stack Traces," *Proceedings of the 34th International Conference on Software Engineering*, 2012. Available at: <http://dl.acm.org/citation.cfm?id=2337223.2337241>.
- [10] M. Jovic, A. Adamoli, and M. Hauswirth, "Catch Me if You Can: Performance Bug Detection in the Wild," *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2011.
- [11] Z.M. Jiang, "Automated Analysis of Load Testing Results," *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.
- [12] W. Shang, A.E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters," *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, 2015.
- [13] C. Heger, J. Happe, and R. Farahbod, "Automated Root Cause Isolation of Performance Regressions During Software Development," *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, 2013.
- [14] M.A. Heroux, *Hpcg Solver Package*, Sandia National Laboratories, 2007. Available at: <https://www.osti.gov/scitech/biblio/1230960>.
- [15] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Ligouri, "Kvm: The Linux virtual machine monitor," *Proceedings of the Linux symposium*, 2007.
- [16] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux J.*, vol. 2014, Mar. 2014. Available at: <http://dl.acm.org/citation.cfm?id=2600239.2600241>.
- [17] J. Mambretti, J. Chen, and F. Yeh, "Next Generation Clouds, the Chameleon Cloud Testbed, and Software Defined Networking (SDN)," *2015 International Conference on Cloud Computing Research and Innovation (ICCCR)*, 2015.
- [18] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale Virtualization in the Emulab Network Testbed," *USENIX*

<sup>8</sup><http://cross.ucsc.edu>

- 2008 Annual Technical Conference*, 2008. Available at: <http://dl.acm.org/citation.cfm?id=1404014.1404023>.
- [19] R. Ricci and E. Eide, "Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications," *login*: vol. 39, 2014/December. Available at: <http://www.usenix.org/publications/login/dec14/ricci>.
  - [20] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *Int J High Perform Comput Appl*, vol. 20, Nov. 2006.
  - [21] A. Wiggins, "The Twelve-Factor App" Available at: <http://12factor.net/>. Available at: <http://12factor.net/>.
  - [22] M. Httermann, *DevOps for Developers*, 2012.
  - [23] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpacı-Dusseau, and A. Arpacı-Dusseau, "Characterizing and Reducing Cross-Platform Performance Variability Using OS-Level Virtualization," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.
  - [24] J.W. Boyce and D.R. Warn, "A Straightforward Model for Computer Performance Prediction," *ACM Comput Surv*, vol. 7, Jun. 1975.
  - [25] K. Kira and L.A. Rendell, "A Practical Approach to Feature Selection," *Proceedings of the Ninth International Workshop on Machine Learning*, 1992. Available at: <http://dl.acm.org/citation.cfm?id=645525.656966>.
  - [26] C.I. King, *Stress-ng*, 2017. Available at: <https://github.com/ColinIanKing/stress-ng>.
  - [27] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and Intelligent Laboratory Systems*, vol. 2, Aug. 1987.
  - [28] D.A. Freedman, *Statistical Models: Theory and Practice*, 2009.
  - [29] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, 2011. Available at: <http://www.jmlr.org/papers/v12/pedregosa11a.html>.
  - [30] P. Prettenhofer and G. Louppe, "Gradient Boosted Regression Trees in Scikit-Learn," Feb. 2014. Available at: <http://orbi.ulg.ac.be/handle/2268/163521>.
  - [31] J.H. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine," *Ann. Stat.*, vol. 29, 2001.
  - [32] L. Breiman, J. Friedman, C.J. Stone, and R.A. Olshen, *Classification and Regression Trees*, 1984.
  - [33] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpacı-Dusseau, and R. Arpacı-Dusseau, "The Popper Convention: Making Reproducible Systems Evaluation Practical," *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.
  - [34] ACM, "Result and Artifact Review and Badging" Available at: <http://www.acm.org/publications/policies/artifact-review-badging>. Available at: <http://www.acm.org/publications/policies/artifact-review-badging>.
  - [35] K. Akbudak, E. Kayaaslan, and C. Aykanat, "Hypergraph Partitioning Based Models and Methods for Exploiting Cache Locality in Sparse Matrix-Vector Multiplication," *SIAM J. Sci. Comput.*, vol. 35, Jan. 2013.
  - [36] J. Cepeda, "Pipeline Speak, Part 2: The Second Part of the Sandy Bridge Pipeline."
  - [37] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, Mar. 2003.
  - [38] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Mag.*, vol. 79, 2009.
  - [39] D.A. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," *High Performance Computing – HiPC 2005*, 2005.
  - [40] S. van der Walt, S.C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Comput. Sci. Eng.*, vol. 13, 2011.
  - [41] M. Widenius, "MariaDB SQL server project," *Ask Monty* Available at: <http://askmonty.org/wiki/index.php/MariaDB>. Available at: <http://askmonty.org/wiki/index.php/MariaDB>.
  - [42] J.D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Comput. Soc. Tech. Comm. Comput. Archit. TCCA Newslet.*, Dec. 1995.
  - [43] A. Hutcheson and V. Natoli, "Memory Bound vs. Compute Bound: A Quantitative Study of Cache and Memory Bandwidth in High Performance Applications," 2011.
  - [44] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," *Ijcai*, 1995.
  - [45] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R News*, vol. 2, 2002.
  - [46] A. Crume, C. Maltzahn, L. Ward, T. Kroeger, and M. Curry, "Automatic generation of behavioral hard disk drive access time models," *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, 2014.
  - [47] B. Gregg, "The Flame Graph," *Commun ACM*, vol. 59, May, 2016.
  - [48] R. Kazmi, D.N.A. Jawawi, R. Mohamad, and I. Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *ACM Comput Surv*, vol. 50, May, 2017.
  - [49] M.D. Syer, Z.M. Jiang, M. Nagappan, A.E. Hassan, M. Nasser, and P. Flora, "Continuous Validation of Load Test Suites," *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, 2014.
  - [50] O. Ibadunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Comput Surv*, vol. 48, Jul. 2015.
  - [51] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating Root-cause Diagnosis of Performance Anomalies in Production Software," *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012. Available at: <http://dl.acm.org/citation.cfm?id=2387880.2387910>.
  - [52] G. Jung, G. Swint, J. Parekh, C. Pu, and A. Sahai, "Detecting Bottleneck in n-Tier IT Applications Through Analysis," *Large Scale Management of Distributed Systems*, 2006.
  - [53] T.H. Nguyen, B. Adams, Z.M. Jiang, A.E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Statistical Process Control Techniques," *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, 2012.