quiho: Automated Performance Regression Testing Using Inferred 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, feature importance specifically, 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 inferred resource utilization profile or IRUP), which is used to automatically validate performance behavior across multiple revisions of an application's code base without having to instrument code or obtain performance counters. We demonstrate that *quiho* can successfully discover performance regressions by showing its effectiveness in profiling application performance for synthetically introduced 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. 2018. *quiho*: Automated Performance Regression Testing Using Inferred Resource Utilization Profiles. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.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 and compared for multiple versions of an application (usually current and past versions) and, if significant differences are found, this constitutes a regression.

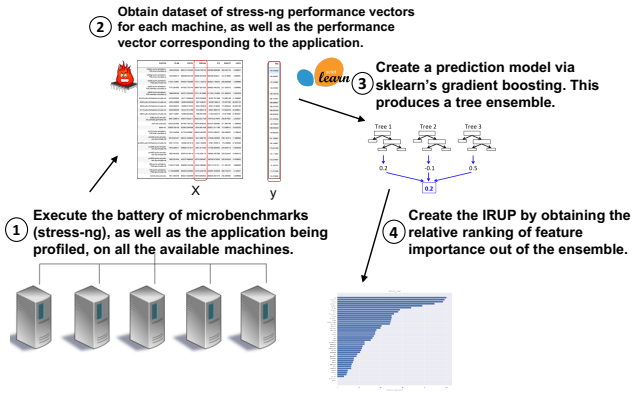


Figure 1: *quiho*'s workflow for generating inferred resource utilization profiles (IRUPs) for an application. An IRUP is used as an alternative for profiling application performance and can complement automated regression testing. For example, after a change in the runtime of an application has been detected across two revisions of the code base, an IRUP can be obtained in order to determine whether this change is significant. IRUPs can also aid in root cause analysis.

One of the main challenges in automating performance regression tests is defining the criteria to decide whether a change in application performance behavior is significant [7]. Understanding the impact that distinct hardware and low-level system software[1] components have on the performance of 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 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]. Profiling involves recording resource utilization for an application over time.

[1]Throughout this paper, we use "system" to refer to the low-level compute stack composed by hardware, firmware and the operating system (OS).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In general, this can be done in two ways: timed- and event-based profiles. Timed-based profiling samples the instruction pointer at regular intervals and generates a function call tree with each node having a percentage of time associated with it, which represents the amount of time that the CPU spends within that piece of code. Event-based profiling samples at regular intervals different events at the hardware- and OS-level in order to obtain a distribution of events over time. In either case, the system needs to execute the application in a "profiling" mode in order to enable the instrumentation mechanisms that the OS has available for carrying out this task.

Automated solutions have been proposed in recent years [11–13]. The general approach of these is to analyze runtime logs and/or 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 negatives/positives. In addition to striving for highly accurate predictions, one can also use performance modeling as a profiling tool.

In this work we present *quiho*, an approach aimed at complementing automated performance regression testing by using inferred resource utilization profiles (IRUP) associated to an application. *quiho* is an alternative framework for profiling an application where the utilization of one or more subsystems (e.g. virtual memory) is inferred by applying Statistical Regression Analysis[2] (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 OSs).

When an application is profiled using *quiho* (Fig. 1), the machines available to the performance tests are baselined by executing a battery of microbenchmarks on each. This matrix 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 infer resource utilization, the application under study is executed on the same machines from where the performance vectors where obtained, and SRA is applied. The result 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 constitutes what we refer to as an *inferred resource utilization profile* (IRUP).

In this article, we demonstrate that our approach successfully identifies performance regressions by showing (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 application-independent performance vectors) gives us a resource utilization profile (an IRUP) of an application without having to look at the code.

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

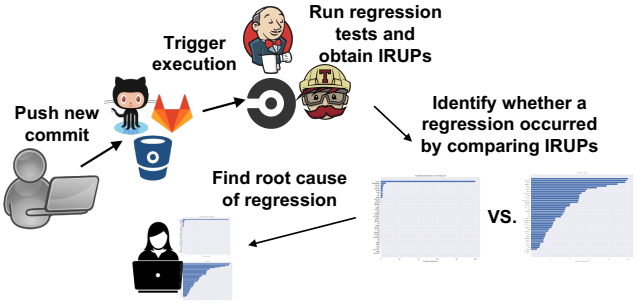


Figure 2: Automated regression testing pipeline integrating inferred resource utilization profiles (IRUP). IRUPs are obtained by *quiho* and can be used both, for identifying regressions, and to aid in the quest for finding the root cause of a regression.

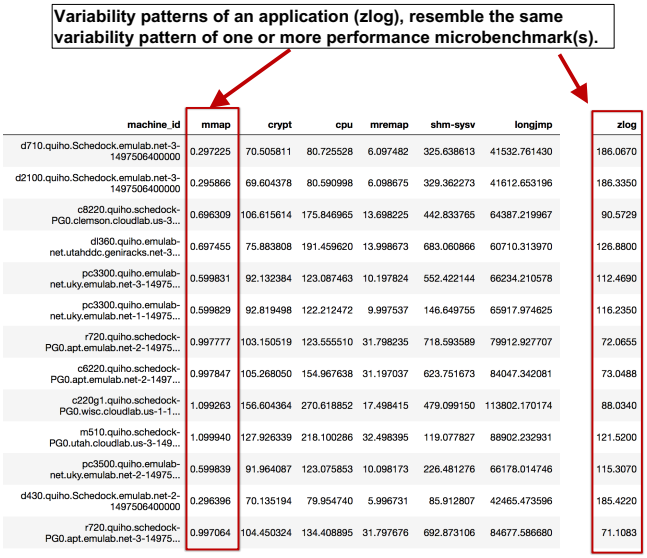
- An automated end-to-end framework (based on the above finding), that aids analysts in identifying significant changes in resource utilization behavior of applications which can also aid in identifying root cause of regressions, and that is resilient to code refactoring.
- Methodology for evaluating 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.

Next section (Section 2) shows the intuition behind *quiho* and how can be used to automate regression tests. We then do a more in-depth description of *quiho* (Section 3), followed by our evaluation of this approach (Section 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

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



machine_id	mmap	crypt	cpu	mremap	shm-sysv	longjmp	zlog
d710.quiho.Schedock.emulab.net-3-1497506400000	0.297225	70.505811	80.725528	6.097482	325.638613	41532.761430	186.0670
d2100.quiho.Schedock.emulab.net-3-1497506400000	0.295866	69.604378	80.590998	6.098675	329.362273	41612.653196	186.3350
c8220.quiho.schedock-PG0.clemson.cloudlab.us-3...	0.696309	106.615614	175.846965	13.698225	442.833765	64387.219967	90.5729
dl360.quiho.emulab-net.utahddc.geniracks.net-3...	0.697455	75.883808	191.459620	13.998673	683.060866	60710.313970	126.8800
pc3300.quiho.emulab-net.uky.emulab.net-3-14975...	0.599831	92.132384	123.087463	10.197824	552.422144	66234.210578	112.4690
pc3300.quiho.emulab-net.uky.emulab.net-1-14975...	0.599829	92.819498	122.212472	9.997537	146.649755	65917.974625	116.2350
r720.quiho.schedock-PG0.apt.emulab.net-2-14975...	0.997777	103.150519	123.555510	31.798235	718.593589	79912.927707	72.0655
c6220.quiho.schedock-PG0.apt.emulab.net-2-1497...	0.997847	105.268050	154.967638	31.197037	623.751673	84047.342081	73.0488
c220g1.quiho.schedock-PG0.wisc.cloudlab.us-1-1...	1.099263	156.604364	270.618852	17.498415	479.099150	113802.170174	88.0340
m510.quiho.schedock-PG0.utah.cloudlab.us-3-149...	1.099940	127.926339	218.100286	32.498395	119.077827	88902.232931	121.5200
pc3500.quiho.emulab-net.uky.emulab.net-2-14975...	0.599839	91.964087	123.075853	10.098173	226.481276	66178.014746	115.3070
d430.quiho.Schedock.emulab.net-2-1497506400000	0.296396	70.135194	79.954740	5.996731	85.912807	42465.473596	185.4220
r720.quiho.schedock-PG0.apt.emulab.net-3-14975...	0.997064	104.450324	134.408895	31.797676	692.873106	84677.586680	71.1083

Figure 3: A matrix of performance feature vectors over a colection of CloudLab servers (left), and an array of a performance metric for an application on those same machines (right). Every column in the matrix comes from executing a microbenchmark on that machine. This dataset of microbenchmarks allows us to create a performance prediction model for application. Variability patterns of an application (zlog in the example), resemble the same variability pattern of one or more performance microbenchmark(s). Thus, the system subcomponent exercised by the microbenchmark is likely to be also the cause of why the application exhibits such performance behavior.

involves eyeballing source code, static code analysis, or analyzing hardware/OS performance counters/profiles.

An alternative is to infer resource utilization behavior by comparing the performance of an application on platforms with different performance characteristics. For example, if we know that machine A has higher memory bandwidth than machine B, and an application is memory-bound, then this application will perform better on machine A. There are several challenges with this approach:

1. Consistent Software. We need to ensure that the software stack is the same on all machines where the application runs.
2. Application Testing Overhead. The amount of effort required to run applications on a multitude of platforms is not negligible.
3. Hardware Performance Characterization. It is difficult to obtain the performance characteristics of a machine by just looking at the hardware spec, so other more practical alternative is required.
4. Correlating Performance. Even if we could solve the above issue (Hardware Performance Characterization) and infer performance characteristics by just looking at the hardware specification of a machine, there is still the problem of not being able to correlate baseline performance with application behavior, since between two platforms is rarely the case where the change of performance is observed in only one subcomponent of the system (e.g., a newer machine doesn't have just faster memory sticks, but also better CPU, chipset, etc.).

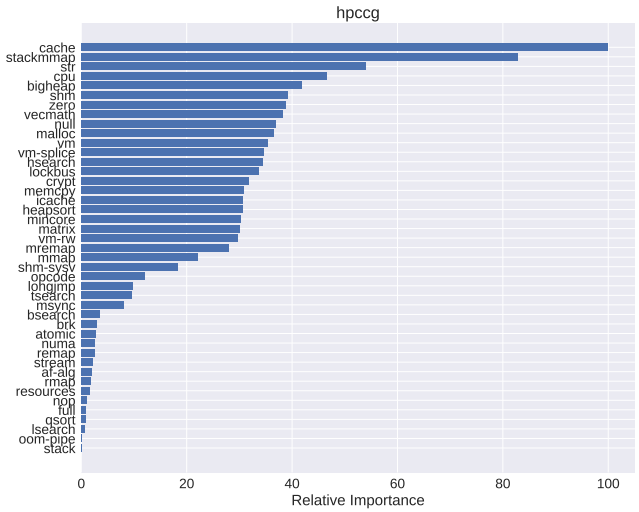


Figure 4: An example profile showing the relative importance of features for an execution of the hpccg miniapp [14]. The x-axis corresponds to the relative performance value, normalized with respect to the most important feature, which corresponds to the first one on the y-axis (from top to bottom). Section 3.2 describes in detail how feature importances are calculated.

The advent of cloud computing allows us to solve (1) using solutions like KVM [15] or software containers [16]. ChameleonCloud [17], CloudLab [18,19] and Grid5000 [20] are examples of bare-metal-as-a-service infrastructure available to researchers that can be used to automate regression testing pipelines for the purposes of investigating new approaches. These solutions to infrastructure automation coupled with DevOps practices [21,22] allows us to address (2), i.e. to reduce the amount of work required to run tests.

Thus, the main challenge to inferring resource utilization patterns lies in quantifying the performance of the platform in a consistent way (3,4). One alternative is to look at the hardware specification and infer performance characteristics from this, a highly inaccurate task due to the lack of correspondence between advertised (or theoretical peak throughput) and actual performance observed in reality. For example, the spec of a platform might specify that the machine has DDR4 memory sticks, with a theoretical peak throughput of 10 GB/s, but the actual memory bandwidth could be less (usually is, by a non-deterministic fraction of the advertised performance).

quiho solves this problem by characterizing machine performance using microbenchmarks. These performance vectors are the "fingerprint" that characterizes the behavior of a machine [23]. These vectors, obtained over a sufficiently large set of machines[3],

[3]In Section 5 we briefly sketch how we would apply PAC to find the minimal set of machines needed to obtaining meaningful results from SRA.

can serve as the foundation for building a prediction model of the performance of an application when executed on new ("unseen") machines [24]. Thus, a natural next step to take with a dataset like this is to try to build a prediction model.

While building a prediction model is obviously something that can be used to estimate the performance of an application, building one can also serve as a way of identifying resource utilization. If we use these performance vectors to apply SRA and focus on feature importance [25] of the generated models, they can allow us to infer 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 over the same set of machines, then we can say that the application is heavily influenced by that subcomponent. In other words, if the variability of a feature across multiple machines resembles the variability 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.

Relying on SRA as a way of inferring resource utilization behavior has the practical consequence of *quiho* benefiting heavily from an heterogeneous setup. The more the "performance diversity" of machines that are available for testing, the easier that *quiho* can discover an application's resource utilization behavior. Intuitively, this can be explained as follows. If we run a IO-bound application on distinct machines with very different CPU and memory subsystem performance but similar IO throughput, we won't be able to discover that the application's bottleneck is on the IO subsystem. If we create a more heterogeneous mix of machines, with larger IO performance variability, we can discover that this application is IO-intensive since the performance of the application will vary, depending on the capabilities of the underlying IO subsystem of each distinct machine.

Thus, having high performance variability allows *quiho* to infer resource utilization patterns by discovering the underlying correlations between the performance of microbenchmarks and an application's performance. Since SRA results in creating a performance prediction model for an application, we can rank features by sorting them with respect to their relative performance prediction importance. We call this ranking an *Inferred Resource Utilization Profile* (IRUP), as shown in Fig. 4. In the next section we explain how these IRUPs are obtained and how they can be used in automated performance regression tests. Section 4 empirically validates this approach.

3 OUR APPROACH

In this section we describe *quiho*'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,

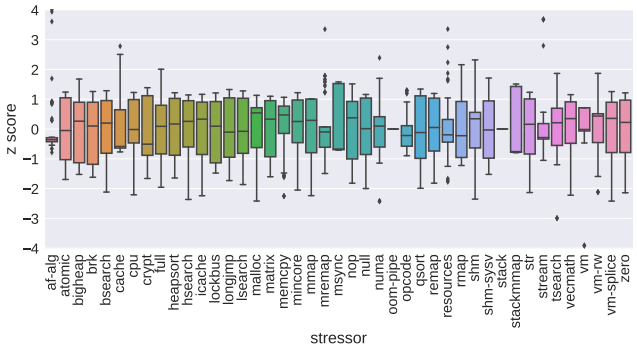


Figure 5: Boxplots illustrating the variability of the performance vector dataset. The data is normalized in order to guard against dimensionality issues. Thus, the y-axis shows variability in terms of the z-score (signed value representing the number of standard deviations by which the value of an observation is below or above the mean). Each stressor was executed five times on each of the machines listed in Tbl. 2.

how this represents an inferred resource utilization profile (IRUP) and the algorithm (and alternative heuristics) to comparing IRUPs.

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 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

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				
vm-splice				X
zero			X	

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

can quantify the difference in performance between the two at a per-stressor level. Fig. 5 shows the variability in these performance vectors.

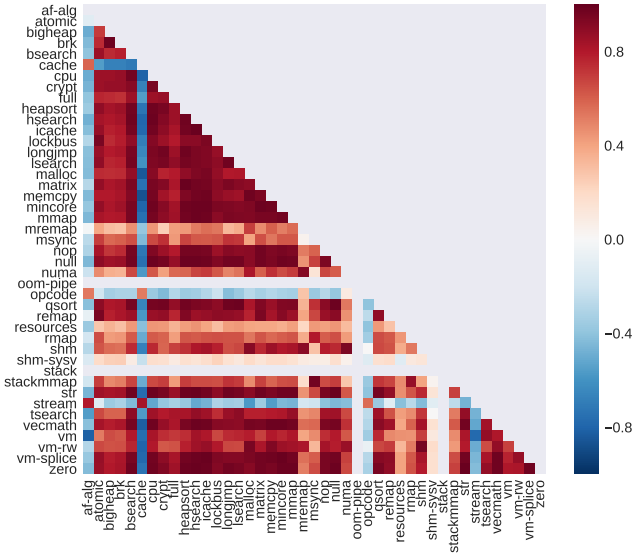


Figure 6: Heat-map of Pearson correlation coefficients for performance vectors obtained by executing `stress-ng` on all the distinct machine configurations available in CloudLab.

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 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 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 relative (blue) and cumulative (green) explained variance ratio. The explained variance ratio is the amount of variability that a component removes from the dataset. The higher the variance associated to a component, the more the data can be explained by that component. 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. So the reader might wonder, why not remove stressors in order to simplify the analysis? If we use the correlation matrix, we would need to define an arbitrary correlation index threshold. If we use PCA, we lose information with respect to what stressors are explaining a prediction. Instead of trying to reduce the number of features, we decide to leave all the stressors in order to not lose any information or having to define arbitrary thresholds. Part of our future work is to address whether we can reduce the number of features with the goal of

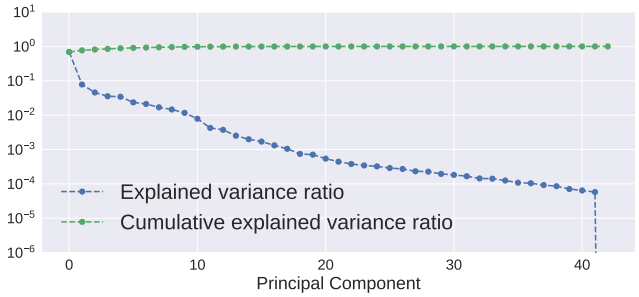


Figure 7: Principal Component Analysis for the performance vector dataset. The y-axis (log-scale) corresponds to the explained variance ratio, while the x-axis denotes the number of components. The blue line denotes the amount of variance reduced by having a particular number of components. The green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

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 [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 IRUP for an application. Fig. 1 shows the process applied to obtaining IRUPs for an application. `scikit-learn` implements the feature importance calculation algorithm introduced in [32] and is sketched in the following pseudo-code algorithm. Given an ensemble of trees:

1. Initialize an `f_importance` array to hold a score for each feature in the dataset.
2. Take an unseen tree of the ensemble and traverse it using the following steps:
   a. For each node that splits on feature i, compute the error reduction of that node, multiplied by the number of samples that were routed to the node.
   b. Add this quantity to the `f_importance` array (value corresponding to feature i).
   c. Once all nodes are traversed, pick another unseen tree from the ensemble and go to 2.
3. Assign a score of 100 to the most important feature and normalize the rest of elements in the `f_importance` array with respect to this one.

For step 2.a, the error reduction is recursively defined by obtaining the difference between the parent node impurity and the weighted sum of the two child node impurities. The impurity criterion depends on whether the problem is a classification or regression one. Gini or MSE (among many others) can be used for classification. For regression, variance impurity is employed and corresponds to the variance of all data points that are routed through that node.

We note that before generating a regression model, we normalize the data by obtaining the z-score of the dataset. 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. In Section 4 we evaluate the effectiveness of IRUPs.

3.3 Using IRUPs in Automated Regression Tests

As shown in Fig. 2 (step 4), when trying to determine whether a performance degradation occurred, IRUPs 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, 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 does not have the same value, the importance is considered not equivalent and the algorithm stops. If values are similar, we move to the next, less important factor and the compare again. This is repeated for as many features are present in the dataset.

IRUPs 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 the *stream* stressor (mimics the STREAM benchmark [33]) 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 by capturing performance counters over time and look at corresponding 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. How well can IRUPs accurately capture application performance behavior? (Section 4.1)

2. How well can IRUPs work for identifying simulated regressions? (Section 4.2)
3. How well can IRUPs work for identifying regressions in real world software projects? (Section 4.3)

Note on Replicability of Results: This paper adheres to The Popper Experimentation Protocol and convention[4] [34], so experiments presented here are available in the repository for this article[5]. 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, as long as the reader has an account at Cloudlab (see repo for more details).

4.1 Effectiveness of IRUPs to Capture Resource Utilization Behavior

In this subsection we show how IRUPs 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.
2. 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 [35] 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_sparsemv()`. 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 [36].

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. Having said this, we can try to analyze the counter results for the particular machine where we ran this test.

[4]http://falsifiable.us
[5]http://github.com/ivotron/quiho-popper

Listing 1 Source code for bottleneck function in HPCCG.

```
int HPC_sparsemv(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->nnz_in_row[i];

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

  return(0);
}
```

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%

We can see that the performance counters values for the hpccg application correspond to a combination of values for the three most relevant features (stressors). In the case of the `stackmmap` stressor, similarities between stalled cycle counters are noticeable denoting similarities in stalled cycles, which are associated to application performance [37,38].

Next, we analyze the IRUPs of other three applications[6]. These applications are Redis [39], Scikit-learn [29], and SSCA [40]. 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 IRUPs using this knowledge. As a way of illustrating the performance variability of these applications on an heterogeneous set of machines, Fig. 8 shows boxplots of their runtime.

[6]For brevity, we omit other results that corroborate IRUPs can correctly identify resource utilization patterns. All these are available in the github repository associated to this article.

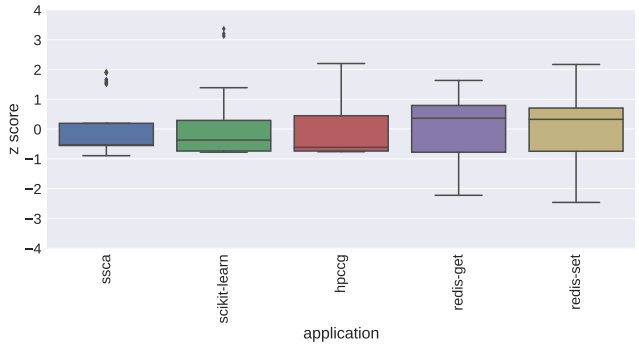


Figure 8: Variability of the four applications presented in this subsection. Y-axis has been normalized.

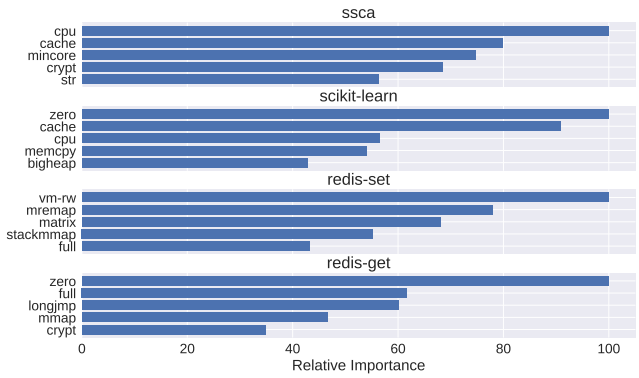


Figure 9: IRUPs for the four tests benchmarked in this section. This and subsequent figures show only the top 5 most important features in order to improve visualization of the plots.

In Fig. 9 we show IRUPs for these four applications[7]. 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 IRUPs (below) correspond to performance tests for Scikit-learn and SSCA. In the case of Scikit-learn, this test runs a comparison of several classifiers in on a synthetic dataset. Scikit-learn uses NumPy [41] 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. The benchmark is designed to have very little locality, which causes the application to generate a 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.

[7]In order to enhance the visualization of the IRUPs we only show the top 5 most important features. Complete profiles can be visualized on the Jupyter notebook contained in the github repository.

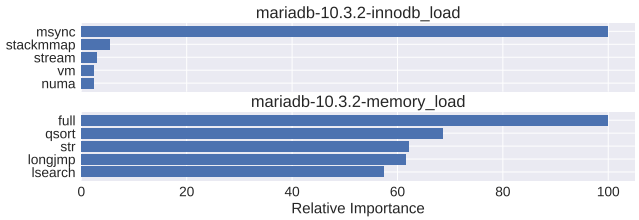


Figure 10: MariaDB with innodb and in-memory backends.

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 [42] 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. 10 shows the profiles of MariaDB performance for these two engines.

The next test is a modified version of the STREAM benchmark [33], 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. 11 shows this behavior.

Fig. 12 shows the IRUPs 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.

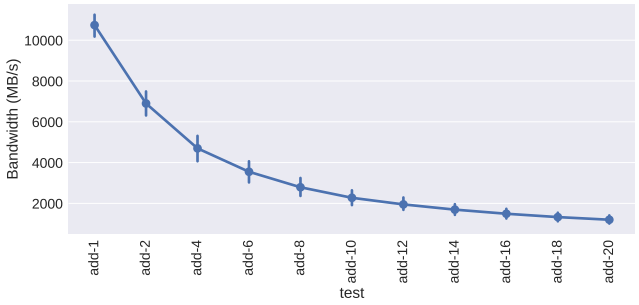


Figure 11: General behavior of the STREAM-NADDS performance test. The y-axis is the throughput of the test in MB/s. The x-axis corresponds to the number of terms in the sum expression of the `Add` STREAM subtest. The regular ("vanilla") STREAM add test is memory bound, so adding more terms to the `Add` 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.

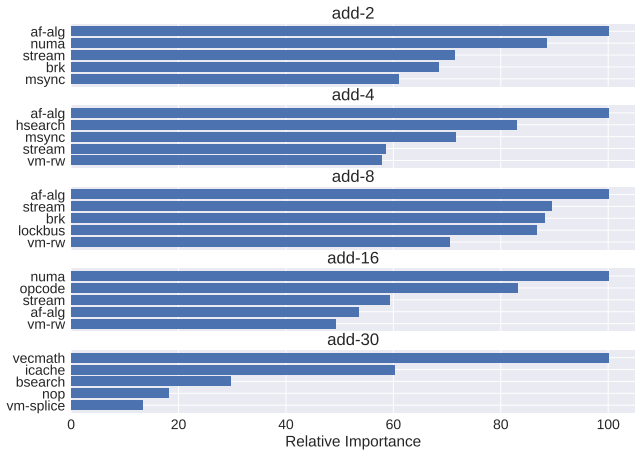


Figure 12: The IRUPs for modified version of STREAM. The parameter of `NADDS` increases by taking values of 1, 2, 4, ..., 20 and 30. We see that they capture the simulated regression which causes this application to be moving from being memory-bound to being cpu-bound.

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 IRUPs, we use `mysqlslap` again and run the `load` test. Fig. 13 shows the corresponding IRUPs. We can observe that the IRUP generated by *quiho* can identify the

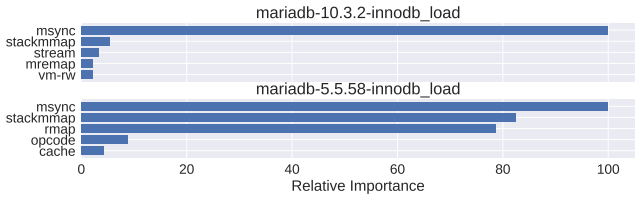


Figure 13: A regression that appears from going in the reversed timeline (from mariadb-10.0.3 to 5.5.38).

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

5 DISCUSSION

Application-Independent Performance Characterization. The main advantage of the *quiho* approach is its resiliency. By inferring resource utilization instead of directly instrumenting code to generate profiles, the *quiho* approach is resilient to code refactoring and requires no manual intervention. 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 subcomponents of a system, and at the same time reduce the number of microbenchmarks.

Falsifiability of IRUPs 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 an IRUP is *not* correctly associated with what the application is doing? In other words, are IRUPs falsifiable? The answer is yes. An IRUP can be incorrectly representing an application's performance behavior if there is under- or over-fitting when generating the model. Fig. 14 shows the correlation matrix obtained from a dataset containing only 3 data points (generated by selecting two random machines from the set of available ones). Almost all stressors are highly correlated among each other, which suggests (as explained in Section 3) there is little that a prediction model can learn about the underlying resource utilization behavior of an application in this dataset (which contains only a couple of points, coming from two machines with very similar characteristics). This is confirmed by obtaining an IRUP multiple times for an application contained in this small dataset (Fig. 15). The application in this case is `redis-set`. If we obtain the IRUP 3 times and compare them, we observe that they give completely random and contradictory results (for example, the bottom IRUP ranks CPU stressors as the top important features). This is in contrast to what we observe with well-fitted models, such as the ones in figure Fig. **??** for which multiple IRUPs show consistent results in their results. The correlation matrix shows why this is so: almost all the features are highly correlated. One way of determining the right amount of machines needed in order to generate good models is to apply probable aproximate correct learning (PAC) [44] to this dataset in order to quantify the probability of obtaining highly accurate estimations.

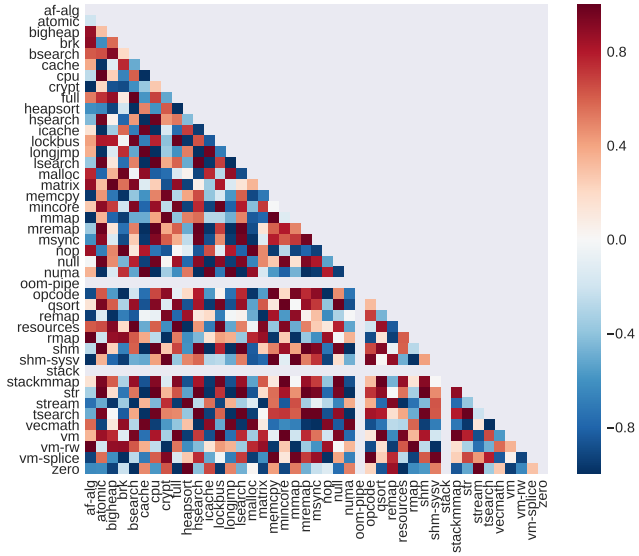


Figure 14: Correlation matrix, obtained from only two randomly selected machines from Tbl. 2.

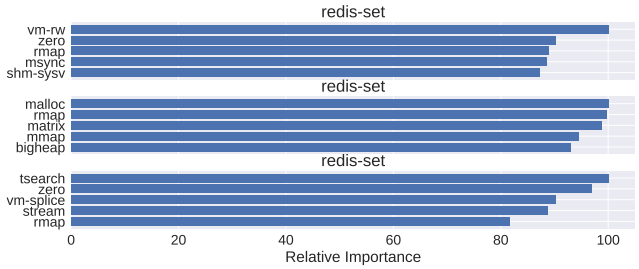


Figure 15: Three IRUPs for the `redis-set` benchmark, obtained sequentialy from the same dataset, which consistes of only two randomly selected machines from Tbl. 2.

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 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.

IRUP Comparison. The algorithm specified in Section 3.3 is a straight-forward one. One could think of more sophisticated ways of doing IRUP 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.

IRUP as a visualization tool. The reader might have noticed that IRUPs can be visually compared by the human eye (and are somewhat similar in this regard to FlameGraphs [45]). Adding a coloring scheme to IRUPs 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.

Reinforcement 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 applycbecomes 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

Automated Regression Testing. Automated regression testing [46] can be broken down in the following three steps. 1) In the case of large software projects, decide which tests to execute [47]. This line of work is complementary to *quiho*. 2) Once a test executes, decide whether a regression has occurred [48]. 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 IRUPs). 3) If a regression is observed, automatically find the root cause or aid an analyst to find it [13,49]. While *quiho* does not find the root cause of regressions, it complements the information that an analyst has available to investigate further.

Profiling-based Performance Modeling. Modeling performance based on application profiles has been studied before [50–52]. In [50], the MAPS benchmark is used to characterize the performance of machines. These profiles are then convoluted with application traces obtained by the MetaSim tool in order to obtain a prediction on the performance of an application. In [52] the authors use randomized optimization (genetic algorithms) to systematically explore the parameter space of an application in order to create a record of `<input, runtime>` pairs. *quiho* can be used in this case to augment the available information and have an IRUP associated to the inputs of the application under study.

Performance Profile Visualization. An IRUP can be used to visualize performance and thus have a resemblance with a flame graph [45]. In [53] the authors introduce the concept of differential flame grahps, which can be used to visually compare the changes between two or more flame graphs. A similar approach could be

applied to IRUPs in order to visualize the differences between two flame graphs.

Inducing Performance Regressions. In [54], the authors analyzed the code repositories of two open source projects in order to device a way of systematically inducing performance regressions. Our methodology instruments an application in order to parametrize performance and control when changes in performance are triggered, as a way of testing methods that are aimed at detecting these changes.

Decision Trees In Performance Engineering. In [55] 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 [56], 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

Correlation-based Analysis and Supervised Learning. Correlation and supervised learning approaches have been proposed in the context of software testing, mainly for detecting anomalies in application performance [49]. 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 IRUPs. 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 applying *quiho* in multi-tenant configurations and to profile long-running (multi-stage) applications such as a web-service or big-data applications. In these cases, we would define windows of time and apply *quiho* to each. The main challenge in this scenario is to automatically define the windows in such a way that we can get accurate profiles.

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[8], 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, *Hpccg 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. Liguori, "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 (ICCCRI)*, 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 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. Arpaci-Dusseau, and A. Arpaci-Dusseau, "Characterizing and Reducing Cross-Platform Performance

[8] http://cross.ucsc.edu

Variability Using OS-Level Virtualization," *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

[24] J.W. Boyse 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] J.D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Comput. Soc. Tech. Comm. Comput. Archit. TCCA Newsl.*, Dec. 1995.

[34] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "The Popper Convention: Making Reproducible Systems Evaluation Practical," *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.

[35] M.A. Heroux, D.W. Doerfler, P.S. Crozier, J.M. Willenbring, H.C. Edwards, A. Williams, M. Rajan, E.R. Keiter, H.K. Thornquist, and R.W. Numrich, "Improving performance via mini-applications," *Sandia Natl. Lab. Tech Rep SAND2009-5574*, vol. 3, 2009.

[36] 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.

[37] S. Cepeda, "Pipeline Speak, Part 2: The Second Part of the Sandy Bridge Pipeline."

[38] C. McNairy and D. Soltis, "Itanium 2 processor microarchitecture," *IEEE Micro*, vol. 23, Mar. 2003.

[39] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Mag.*, vol. 79, 2009.

[40] D.A. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," *High Performance Computing – HiPC 2005*, 2005.

[41] 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.

[42] 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.

[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] L.G. Valiant, "A Theory of the Learnable," *Commun ACM*, vol. 27, Nov. 1984.

[45] B. Gregg, "The Flame Graph," *Commun ACM*, vol. 59, May. 2016.

[46] S.E. Perl and W.E. Weihl, "Performance Assertion Checking," *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[47] 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.

[48] 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.

[49] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Comput Surv*, vol. 48, Jul. 2015.

[50] A. Snavely, N. Wolter, and L. Carrington, "Modeling application performance by convolving machine signatures with application profiles," *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001.

[51] S. Ghaith, M. Wang, P. Perry, and J. Murphy, "Profile-Based, Load-Independent Anomaly Detection and Analysis in Performance Regression Testing of Software Systems," *2013 17th European Conference on Software Maintenance and Reengineering*, 2013.

[52] D. Shen, Q. Luo, D. Poshyvanyk, and M. Grechanik, "Automating Performance Bottleneck Detection Using Search-based Application Profiling," *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015.

[53] C.P. Bezemer, J. Pouwelse, and B. Gregg, "Understanding software performance regressions using differential flame graphs," *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[54] J. Chen and W. Shang, "An Exploratory Study of Performance Regression Introducing Code Changes," *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[55] 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.

[56] 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.