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 used in automated performance regression tests. *quiho* discovers hardware and system software resource utilization patterns that influence the performance of an application. It achieves this by applying sensitivity analyisis, in particular statistical regression analysis (SRA), using application-independent performance feature vectors to 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, which is used to automatically validate its performance behavior across revisions. We demonstrate that *quiho* can successfully identify performance regressions by showing its effectiveness in profiling application performance for synthetically induced regressions as well as several found in real-world applications.

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 compared and when significant differences are found, this constitutes a regression.

One of the main challenges in performance regression testing is defining the criteria to decide whether a change in an application's performance behavior is significant, that is, whether a regression has occurred [7]. Simply comparing values (e.g. runtime) is not enough, even if this is done in statistical terms (e.g. mean runtime within a pre-defined variability range). Traditionally, this investigation is done by an analyst in charge of looking at changes, possibly investigating deeply into the issue and finally determining whether a regression exists.

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

When investigating a candidate of a regression, one important task is to find bottlenecks [8]. Understanding the effects in performance that distinct hardware and low-level system software[1] components have on applications is an essential part of performance engineering [9–11]. One common approach is to monitor an application's performance in order to understand which parts of the system an application is hammering on [5]. Automated solutions have been proposed [7,12–14]. The general approach of these is to analyze logs and/or metrics obtained as part of the execution of an application in order to automatically determine whether a regression has occurred. Most of them do this by creating prediction models that are checked against runtime metrics. As with any prediction model, there is the risk of false/positive negatives.

In this work, we present *quiho* an approach aimed at complementing automated performance regression testing by using system resource utilization profiles associated to an application. A resource utilization profile is obtained using Statistical Regression Analysis[2] (SRA) where application-independent performance feature vectors are used to characterize the performance of machines. The performance of an application is then analyzed applying SRA to build a model for predicting its performance, using the performance vectors as the independent variables and the application performance metric as the dependant variable. 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 a performance profile 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.
- We identify a set of synthetic benchmarks that accurately represent different resources. This mapping is used in our root cause analysis.

[1] Throughout this paper, we use "system" to refer to hardware, firmware and the operating system (OS).
[2] We use the term *Statistical Regression Analysis* (SRA) to differentiate between regression testing in software engineering and regression analysis in statistics.

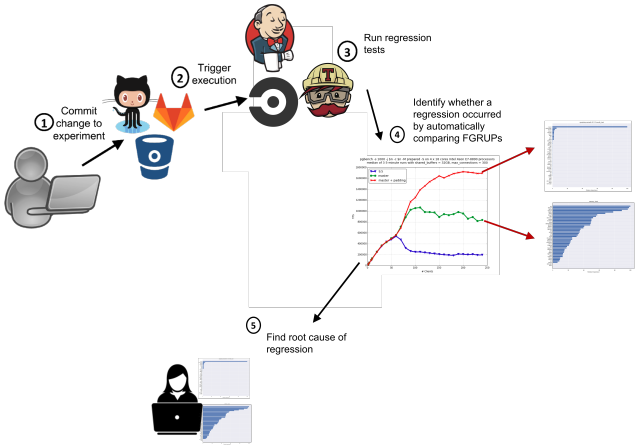

Figure 1: Automated regression testing pipeline integrating fine granularity resource utilization profiles (FGRUP). FGRUPs are obtained by *quiho* and can be used both, for identifying regressions, and to aid in the quest for finding the root cause of a regression.

- An automated end-to-end framework that brings administrators from the point that a regression is identified to the bottleneck that caused the regression.

This last point is especially relevant in today's world, where software grows rapidly and hardware changes constantly. Instead of understanding the stack from end-to-end, it is often more efficient to run workloads, measure their effects on the stack, and optimize resource usage. For example, Google' Vizier [15] is a general auto-tuning service, implemented because "any sufficiently complex system acts as a black box when it becomes easier to experiment with than to understand". Our automated framework acknowledges this trend and can identify the root cause of performance regressions when software versions change or hardware gets upgraded.

Next section (Section 2) shows the intuition behind *quiho* and how can be used to automate regression tests (Section 2). We then do a more 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). Section 5 reviews related work and we subsequently close with a brief discussion on challenges and opportunities enabled by *quiho* (Section 6).

2 MOTIVATION AND INTUITION BEHIND *QUIHO*

Fig. 1 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 what's the patter in terms of resource utilization. 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 allows application developers and performance engineers to quickly understand what they need to focus on while refactoring an application.

Obtaining fine granularity performance utilization behavior, for example, system subcomponents such as the OS memory mapping submodule or the CPU's cryptographic unit is usually time-consuming or requires implicates the use of more computing resources. This usually involves eyeballing source code, static code analysis, or analyzing hardware/OS performance counters.

An alternative is to infer fine granularity resource utilization behavior by comparing the performance of an application on platforms with different system 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. We need to ensure that the software stack is the same on all machines where the application runs.
2. The amount of effort required to run applications on a multitude of platforms is not negligible.
3. 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. Even if we could solve 3 and infer performance characteristics by just looking at the hardware specification of a machine, there is still the issue 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.).

The advent of cloud computing allows us to solve 1 using solutions like KVM [16] or software containers [17]. ChameleonCloud [18], CloudLab [19,20] and Grid5000 [21] 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 [[22] ; htter-mann_devops_2012] allows us to address 2, i.e. to reduce the amount of work required to run tests.

Thus, the main challenge to inferring fine granularity resource utilization patterns (3 and 4) lies in quantifying the performance of the platform in a consistent way. One alternative is to look at the hardware specification and infer performance characteristics from this. As has been shown **[needs-citation]**, this is not consistent. For example, the spec 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 (Fig. 2). These performance vectors are the "fingerprint" that characterizes the behavior of a machine [23].

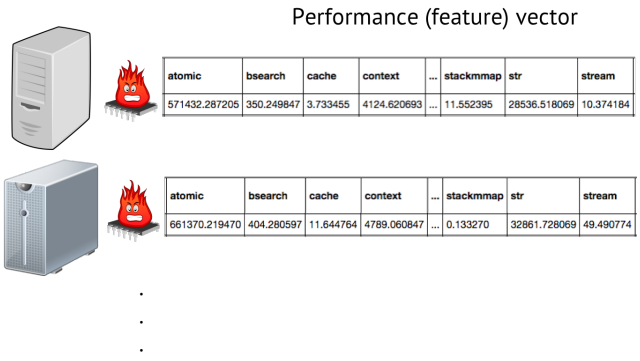

Figure 2: Performance vectors are obtained by executing a battery of microbenchmarks that quantify the performance of multiple subcomponents of a machine.

These performance vectors, obtained over a sufficiently large set of machines[3], can serve as the foundation for building a prediction model of the performance of an application when executed on new ("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.

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 do an in-depth description of *quiho*'s approach. We first describe how we obtain the performance vectors that characterize system performance. We then show how using these vectors we can feed SRA to build performance models for an application. Lastly, we describe how we obtain feature importance and how this represent a fine granularity resource utilization profile (FGRUP).

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

[3] As mentioned in Section 6, an open problem is to identify the minimal set of machines needed to obtaining meaningful results from SRA.

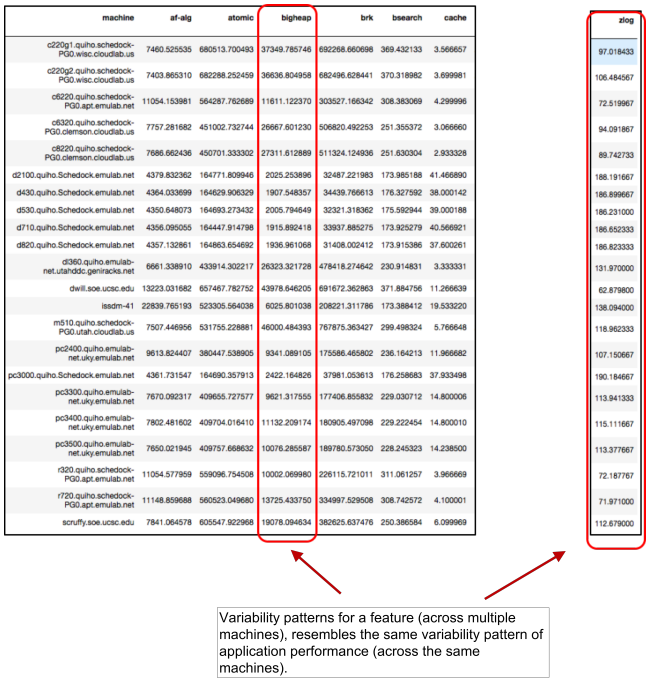

Figure 3: Intuition behind why feature importance implies resource utilization behavior. The variability patterns for a feature (across multiple machines), resembles the same variability pattern of application performance across the same machines. While this can be inferred by obtaining correlation coefficients, proper SRA is needed in order to create prediction models, as well as to obtain a relative rank of feature importances.

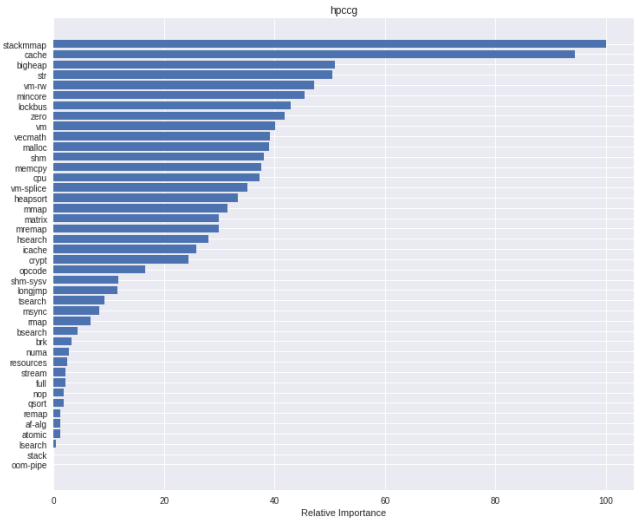

Figure 4: An example profile showing the relative importance of features for a particular application.

machine	cpu	cores	num_cpus	threads	vcpus	board_name
\c220g1.quiho.schedock-pg0.wisc.cloudlab.us\"	\"Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz\"	8	2	2	32	\"UCSC-C220-M4S\"
\c220g2.quiho.schedock-pg0.wisc.cloudlab.us\"	\"Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz\"	8	2	2	32	\"UCSC-C220-M4S\"
\c6220.quiho.schedock-pg0.apt.emulab.net\"	\"Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz\"	8	1	2	16	\"PowerEdge R320\"
\c6320.quiho.schedock-pg0.clemson.cloudlab.us\"	\"Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz\"	10	2	1	20	\"PowerEdge C8220\"
\c8220.quiho.schedock-pg0.clemson.cloudlab.us\"	\"Intel(R) Xeon(R) CPU E5-2660 v2 @ 2.20GHz\"	10	2	1	20	\"PowerEdge C8220\"
\d2100.quiho.schedock.emulab.net\"	\"Intel(R) Xeon(TM) CPU 3.00GHz\"	1	1	2	2	\"PowerEdge 2850\"
\d430.quiho.schedock.emulab.net\"	\"Intel(R) Xeon(TM) CPU 3.00GHz\"	1	1	2	2	\"PowerEdge 2850\"
\d530.quiho.schedock.emulab.net\"	\"Intel(R) Xeon(TM) CPU 3.00GHz\"	1	1	2	2	\"PowerEdge 2850\"
\d710.quiho.schedock.emulab.net\"	\"Intel(R) Xeon(TM) CPU 3.00GHz\"	1	1	2	2	\"PowerEdge 2850\"
\d820.quiho.schedock.emulab.net\"	\"Intel(R) Xeon(TM) CPU 3.00GHz\"	1	1	2	2	\"PowerEdge 2850\"
\dl360.quiho.emulab-net.utahddc.geniracks.net\"	\"Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz\"	8	2	2	32	\"ProLiant DL360e Gen8\"
\dwill.soe.ucsc.edu\"	\"Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz\"	4	1	1	4	\"OptiPlex 790\"
\issdm-41.soe.ucsc.edu\"	\"Dual-Core AMD Opteron(tm) Processor 2212\"	2	2	1	4	\"H8DMR-82i\"
\m510.quiho.schedock-pg0.utah.cloudlab.us\"	\"Intel(R) Xeon(R) CPU D-1548 @ 2.00GHz\"	8	1	2	16	\"ProLiant m510 Server Cartridge\"
\pc2400.quiho.emulab-net.uky.emulab.net\"	\"Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83GHz\"	4	1	1	4	\"Inspiron 530\"
\pc3000.quiho.schedock.emulab.net\"	\"Intel(R) Xeon(TM) CPU 3.00GHz\"	1	1	2	2	\"PowerEdge 2850\"
\pc3300.quiho.emulab-net.uky.emulab.net\"	\"Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz\"	4	1	1	4	\"Inspiron 530\"
\pc3400.quiho.emulab-net.uky.emulab.net\"	\"Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz\"	4	1	1	4	\"Inspiron 530\"
\pc3500.quiho.emulab-net.uky.emulab.net\"	\"Intel(R) Core(TM)2 Quad CPU Q6600 @ 2.40GHz\"	4	1	1	4	\"Inspiron 530\"
\r320.quiho.schedock-pg0.apt.emulab.net\"	\"Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz\"	8	1	2	16	\"PowerEdge R320\"
\r720.quiho.schedock-pg0.apt.emulab.net\"	\"Intel(R) Xeon(R) CPU E5-2450 0 @ 2.10GHz\"	8	1	2	16	\"PowerEdge R320\"
\scruffy.soe.ucsc.edu\"	\"Intel(R) Xeon(R) CPU E5620 @ 2.40GHz\"	4	1	2	8	\"XBDTG-QF\"

Figure 5: List of different type of machines available on Cloudlab.

characteristics can only feasibly[4] be obtained by executing programs and capturing metrics. **[can we show data for this? or add citation]** 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`[5] 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, memory and virtual virtual memory stressors. A *stressor* is a function that loops a for a fixed amount of time (i.e. a microbenchmark), 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. 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. 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 [20] (Fig. 5 shows a summary of their hardware specs). As the figure shows, some

[4]One can get 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).
[5]http://kernel.ubuntu.com/~cking/stress-ng

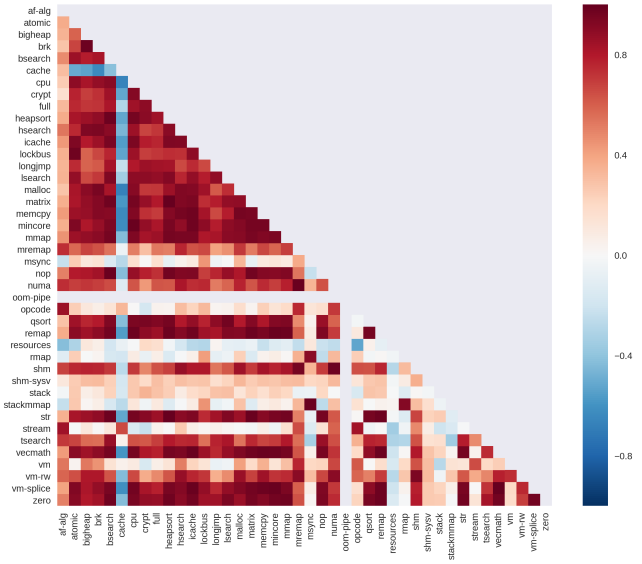

Figure 6: heat-map of Pearson correlation coefficients for performance vectors obtained by executing `stress-ng` on all the distinct machine configurations available in CloudLab.

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).

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 [26]. 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` [27] provides with many of the previously mentioned techniques for building regression models. Another technique available in `scikit-learn` is gradient boosting [28]. 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 [29]. 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 function space by iteratively choosing a function (weak hypothesis) that points in the negative gradient direction. Fig. 7

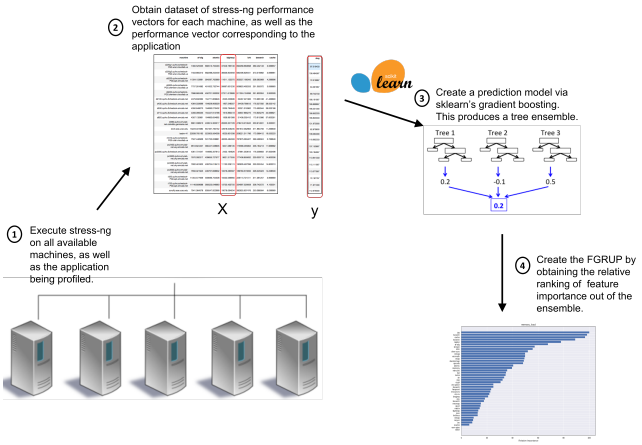

Figure 7: The workflow applied in order to obtain FGRUPs.

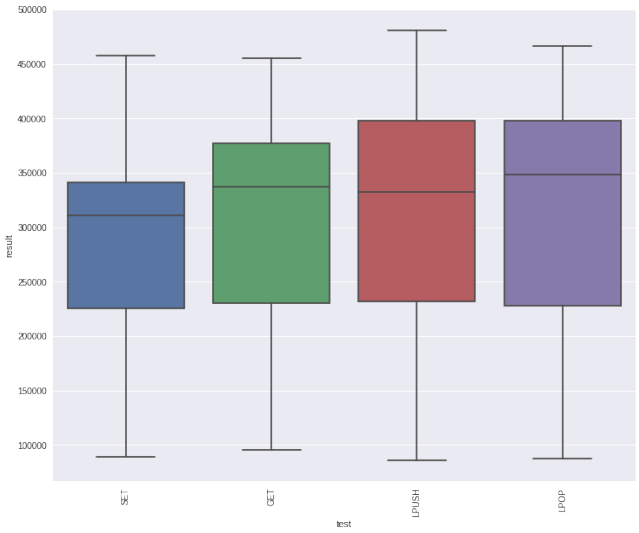

Figure 8: Variability. Y-axis is transactions per second.

shows the process applied to obtain FGRUPs for an application. In the next section we evaluate their effectiveness.

3.3 Using FGRUPs in Automated Regression Tests

As shown in Fig. 1 (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.

TODO: add algorithm

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. 1, 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 three main 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[6] [30], so experiments presented here are available in the repository for this article[7]. 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[8], as long as the reader has an account at Cloudlab (see repo for more details).

4.1 Effectiveness of FGRUPs to capture performance

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

1. Discover relevant performance features using the *quiho* framework.
2. Analyze source code to corroborate that discovered features are indeed the cause of performance differences.

We execute multiple applications for which fine granularity resource utilization characteristics we know in advance. These applications are redis [32], scikit-learn [27], and ssca [33] and others. As a way to illustrate the variability originating from executing these applications on an heterogeneous set of machines, Fig. 8 shows boxplots of the four redis performance tests we execute.

In Fig. 9 we show four profiles side-by-side of four operations on redis, a popular open-source in-memory key-value database. These four tests are `PUT`, `GET`, `LPOP` and `LPUSH`. These benchmarks that test operations that put and get key-value pairs into the DB, and push/pop elements from a list stored in a key, respectively. The resource utilization profiles suggest that `GET` and `PUT` are memory intensive operations (first 3 stressors from each test, as shown in Tbl. **??**). On the other hand, the profiles for `LPOP` and `LPUSH` look different and they seem to have CPU intensive as the most

[6] http://falsifiable.us

[7] http://github.com/ivotron/quiho-popper

[8] **Note to reviewers**: based on the terminology described in the ACM Badging Policy [31] this complies with the *Results Replicable* category. We plan to submit this work to the artifact review track too.

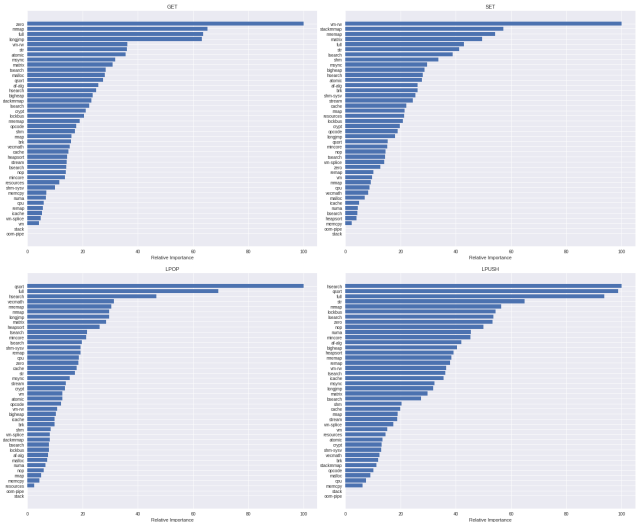

Figure 9: FGRUPs for four redis tests (`PUT`, `GET`, `LPOP` and `LPUSH`). These benchmarks that test operations that put and get key-value pairs into the DB, and push/pop elements from a list stored in a key, respectively.

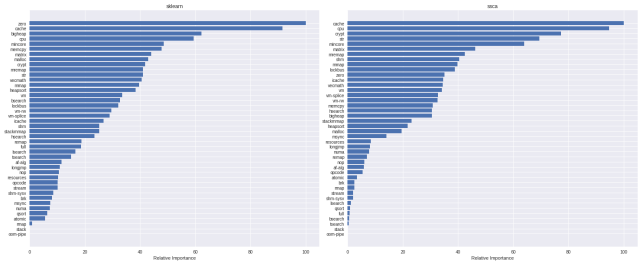

Figure 10: sklearn, ssca.

important feature for this. If we look at the source code of redis, we can see why this is so. In the case of `GET` and `PUT`, these are memory intensive tasks. In the case of `LPOP` and `LPUSH`, these are routines that retrieve/replace the first element in the list, which is cpu-intensive and correlate with cpu-intensive stressors (such as `hsort` and `qsort`).

Fig. 10 shows the profile for one of the sklearn classification algorithm performance test. `scikit-learn` uses NumPy [34] internally, which is known to be memory-bound. SSCA on the other hand known to be CPU-bound.

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 30 performance regressions. For brevity,

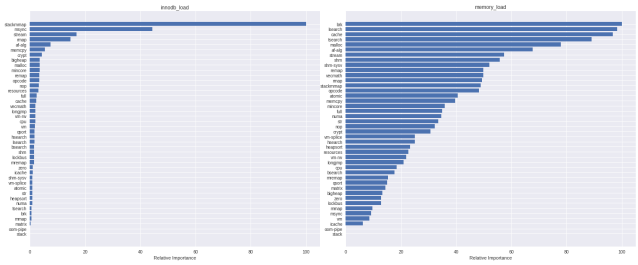

Figure 11: MariaDB with innodb and in-memory backends.

in this section we present results for two applications, MariaDB [35] and the STREAM-cycles.

The MariaDB test is based on the `mysqlslap` utility for stressing the database. In our case we run the load test, which populates a database whose schema is specified by the user. In our case, 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, which we refer to as STREAM-cycles. This version of STREAM introduces a `cycles` parameter that controls the number of times a STREAM operation is executed before reporting the time it took. In terms of the code, this adds an outer loop to each of the four different STREAM operations (`add`, `triad`, `copy`, `scale`), and loops as many times as the `cycles` parameter specifies. All STREAM tests are memory bound, so adding more cycles move the performance test from memory- to being cpu-bound; the higher the value of the `cycles` parameter, the more cpu-bound the test gets. Fig. 12 shows this behavior of all four tests across many machines.

Fig. 13 shows the FGRUPs for the four tests. On the left, we see the "normal" resource utilization behavior of the "vanilla" version of STREAM (which corresponds to a value of 1 for the `cycles` parameter). As expected, the associated features (stressors) to these are from the memory/VM category. To the right, we see FGRUPs capturing the change in utilization behavior when `cycles` goes to its maximum value (20). In general FGRUPs do a good job of catching the simulated regression (which causes this application to be cpu-bound instead of memory-bound).

4.3 Real-world Scenario

We show that *quiho* works with a real software project.

4.4 *quiho* cannot predict performance

We show how *quiho* does not do a good job at predicting performance.

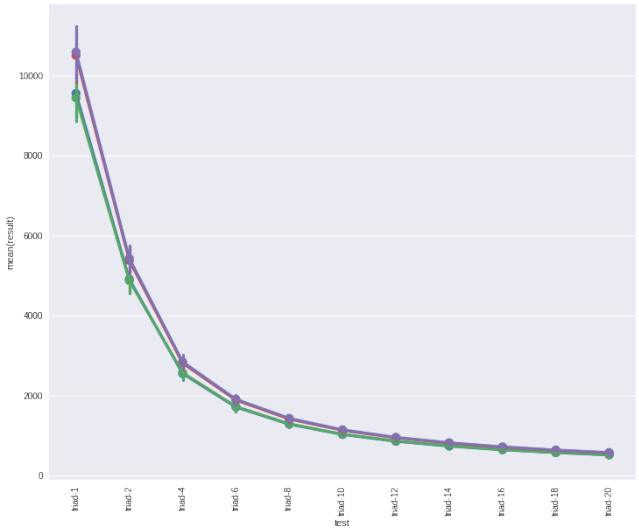

Figure 12: General behavior of the STREAM-cycles performance test. All STREAM tests are memory bound, so adding more cycles move the performance test from memory- to being cpu-bound; the higher the value of the `cycles` parameter, the more cpu-bound the test gets.

5 RELATED WORK

5.1 Anomaly Detection and Bottleneck Identification

It's been used in bottleneck detection [8]. **TODO: mention briefly how it is used**.

5.2 Automated Regression Testing

In [13], they use it to detect regressions using a dataset of performance counters.

6 CONCLUSION AND FUTURE WORK

- Main draw-back of this technique is that we need to run on multiple machines. Time can be saved by carefully avoiding to re-execute stress-ng every time we run a test.
- We used `stress-ng` but this is not the only thing we can use. Ideally, we would extend this battery of tests so that we have more "coverage" of the distinct subcomponents of a system.

In the not-so-distant future:

- multi-node
- minimum number of machines?
- single machine?
- 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.

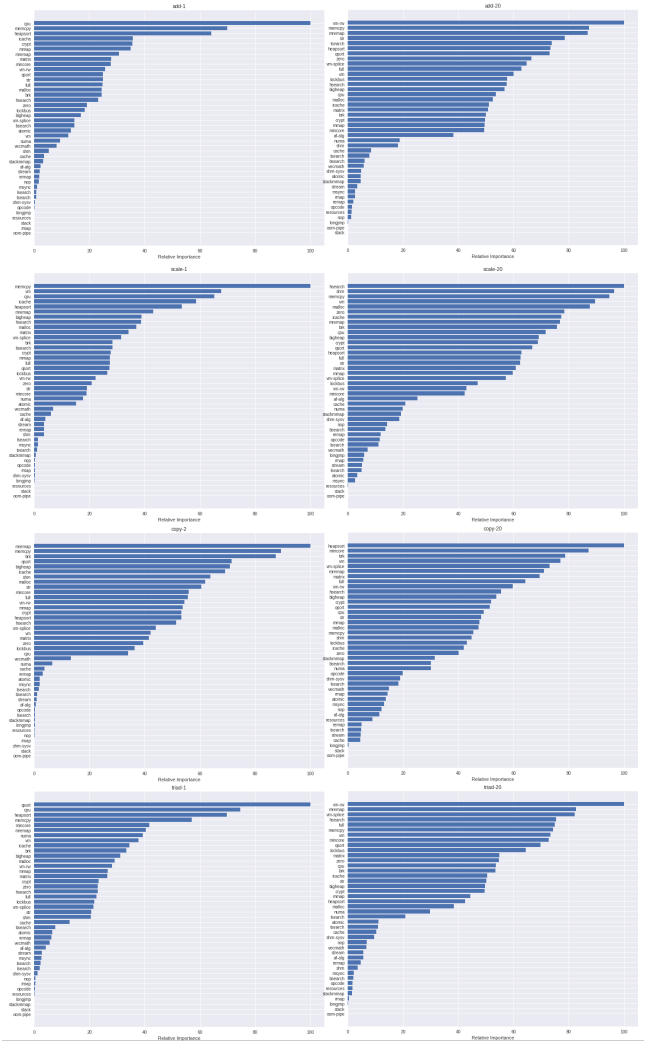

Figure 13: The FGRUPs for the four tests. We see that they capture the simulated regression (which causes this application to be cpu-bound instead of memory-bound).

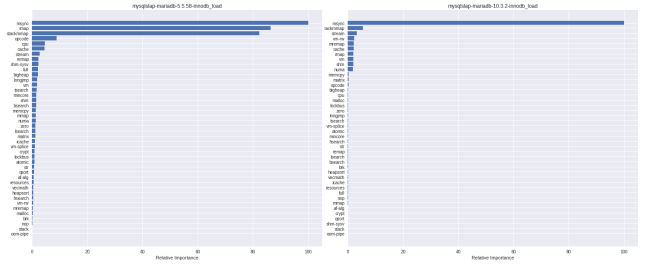

Figure 14: mariadb-10.0.3 vs. 5.5.

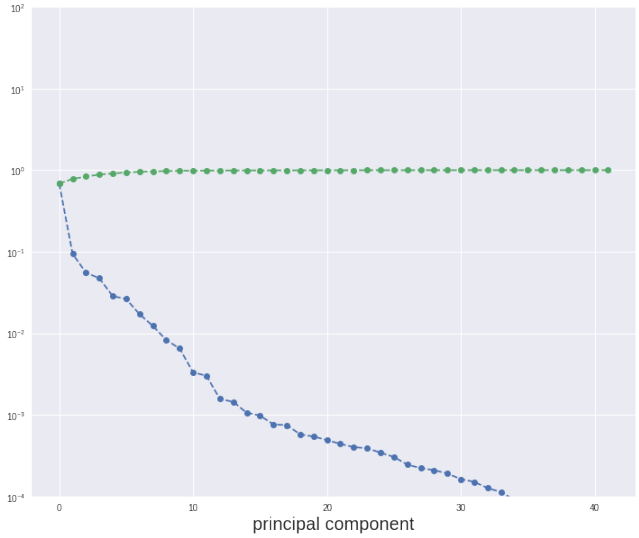

Figure 15: Variability reduction per subcomponent in PCA.

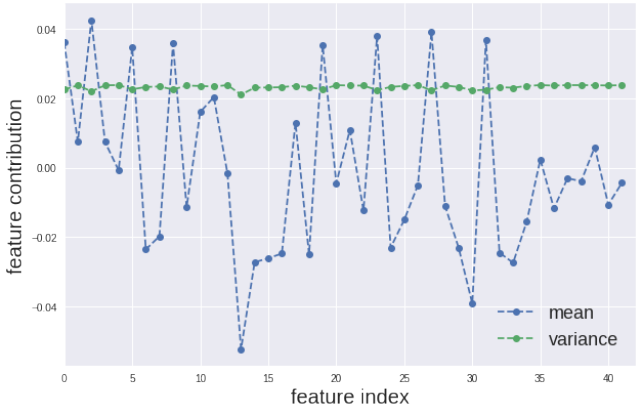

Figure 16: Variability reduction per index.

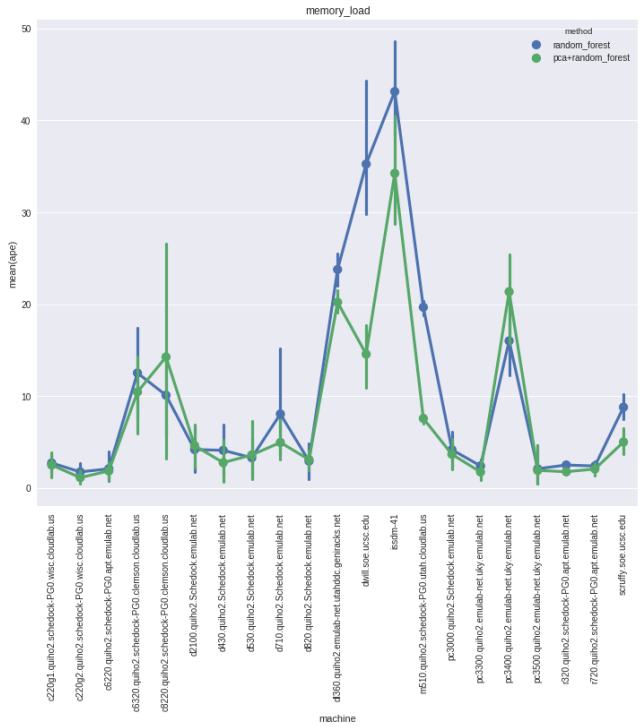

Figure 17: Mean Absolute Percentage Error of cross-validation.

Acknowledgments: This work was partially funded by the Center for Research in Open Source Software[9], Sandia National Laboratories and NSF Awards #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] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Comput Surv*, vol. 48, Jul. 2015.
[9] 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.
[10] 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.
[11] 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.
[12] Z.M. Jiang, "Automated Analysis of Load Testing Results," *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.
[13] W. Shang, A.E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Regression Models on Clustered Performance Counters,"

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

Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, 2015.

[14] 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.

[15] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J.E. Karro, and D. Sculley, *Google Vizier: A Service for Black-Box Optimization*, 2017.

[16] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "Kvm: The Linux virtual machine monitor," *Proceedings of the Linux symposium*, 2007.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] A. Wiggins, "The Twelve-Factor App" Available at: http://12factor.net/. Available at: http://12factor.net/.

[23] I. Jimenez, C. Maltzahn, J. Lofstead, A. Moody, K. Mohror, R. Arpaci-Dusseau, and A. Arpaci-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. 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] D.A. Freedman, *Statistical Models: Theory and Practice*, 2009.

[27] 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.

[28] P. Prettenhofer and G. Louppe, "Gradient Boosted Regression Trees in Scikit-Learn," Feb. 2014. Available at: http://orbi.ulg.ac.be/handle/2268/163521.

[29] J.H. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine," *Ann. Stat.*, vol. 29, 2001.

[30] 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.

[31] 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.

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

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

[34] 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.

[35] 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.