AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses

Yudi Zheng*, Andrea Rosà*, Luca Salucci*, Yao Li†, Haiyang Sun*,
Lubomír Bulej*, Lydia Y. Chen‡, Zhengwei Qi†, and Walter Binder*
*Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland
Email: {yudi.zheng, andrea.rosa, luca.salucci, haiyang.sun, lubomir.bulej, walter.binder}@usi.ch
†Shanghai Jiao Tong University, Shanghai, China
Email: {lastland, qizhwei}@sjtu.edu.cn
‡Cloud Server Technologies Group, IBM Research Lab Zurich, Rüschlikon, Switzerland
Email: yic@zurich.ibm.com

***Abstract*—Researchers often rely on benchmarks to demonstrate feasibility or efficiency of their contributions. However, finding the right benchmark suite can be a daunting task—existing benchmark suites may be outdated, known to be flawed, or simply irrelevant for the proposed approach. Creating a proper benchmark suite is challenging, extremely time consuming, and also—unless it becomes widely popular—a thankless endeavor. In this paper, we introduce a novel approach to help researchers find relevant workloads for their experimental evaluation needs. Our approach relies on the huge number of open-source projects available in public repositories, and on unit testing having become best practice in software development. Using a repository crawler employing pluggable static and dynamic analyses for filtering and workload characterization, we allow users to automatically find projects with relevant workloads. Preliminary results presented here show that unit tests can provide a viable source of workloads, and that the combination of static and dynamic analyses improves the ability to identify relevant workloads that can serve as the basis for custom benchmark suites.**

***Keywords*—Benchmarks; hybrid analysis; unit testing**

I. Introduction

Researchers often rely on experimental evaluation to support their claims and contributions. Experimental evaluation is difficult, as illustrated by the collection of evaluation anti-patterns found in the Evaluate Collaboratory[1]. A properly designed and executed evaluation needs to consider many aspects related to context, workloads, metrics, and analyses. Here, we focus on the workloads, one of the crucial parts of a meaningful experimental evaluation. Traditionally, specific benchmarks are employed, which have been accepted as representative workloads by a research community. However, for many evaluation needs, proper benchmarks are missing [1].

Creating a benchmark suite that aspires to be generally accepted by a research community requires significant effort, both in finding workload candidates and showing that they are good representatives, as well as in solving non-trivial technical challenges associated with creating an easy-to-use "product" for others to use. The design challenges are well demonstrated in the case of DaCapo [2] and ScalaBench [3]. The problem of maintaining a benchmark suite is clearly demonstrated by DaCapo, which was last updated in 2009, and the last mention of a potential update dates back to 2012.

Because manually creating and maintaining a benchmark suite is tedious, we are investigating new ways of constructing benchmark suites for specific tasks, such that researchers can find relevant workloads for their evaluation needs (cf. Section II). Given the success of existing benchmark suites based on open-source software, we focus on open-source software as well. Ultimately, our goal is to (semi-)automatically synthesize a benchmark suite using open-source code found in public repositories, leading to the principal research question:

RQ *Can a fully automated process find open-source workloads that are suitable to serve as benchmarks for specific evaluation needs?*

We cannot hope to provide a simple and resolute answer to such a question in the scope of this paper, but we make the first step towards the goal, and present the general approach and preliminary results that encourage further research.

An obvious hurdle to overcome when mining open-source projects for real-world workloads is obtaining executable code from a completely unknown code base. For this we rely on (unit) testing having become best practice in software development [4], [5], and expect unit tests to provide such code. It is not entirely clear though, whether we can expect a unit test to serve as a benchmark workload—after all, a unit test should only test a single component in isolation. Here, we assume that some developers are using testing frameworks also for more complex tests, which would increase our chances of finding interesting workloads.

In this paper, we therefore investigate the foundational aspects of our benchmark synthesis approach. To this end, we develop AutoBench (cf. Section III), a toolchain combining code repository crawling, pluggable hybrid analyses, and workload characterization techniques to identify and analyze workloads. We use AutoBench to conduct an empirical study to determine whether we can expect to find any unit tests that could serve as benchmark workloads by sifting through a huge number of open-source projects, whether we can identify workloads relevant to a particular research context, and whether these workloads are sufficiently diverse to enable synthesis of a benchmark suite.

[1] http://evaluate.inf.usi.ch/anti-patterns

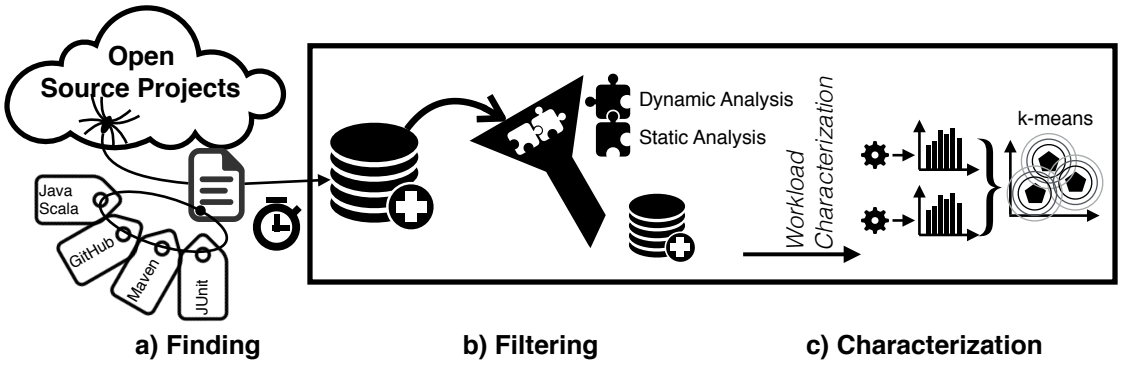

Figure 1. Overview of the proposed approach. The **Finding** procedure crawls public repositories searching for open-source projects with certain properties (labels), and commits to a repository database projects with at least one unit test executing sufficiently long. The **Filtering** procedure employs context-specific pluggable hybrid analyses to filter the committed repositories. The **Characterization** procedure identifies relevant workloads by applying user-defined analyses, such as k-means clustering, on different workload metrics.

II. Motivating Scenarios

To illustrate the evaluation needs of potential users, here we exemplify three use cases, each focusing on a specific context and a user who may experience difficulties in finding suitable benchmarks. These use cases then guide the evaluation of our approach presented in Section IV.

In the first use case (executor), we consider a researcher investigating the usage of task execution frameworks in Java applications, aiming at using the results in different scenarios, ranging from performance modeling and identification of scalability issues, to validating the efficiency of new optimizations. A few concurrent workloads can be found in existing benchmarks [2], but for a meaningful analysis of the use of task execution frameworks, additional concurrent workloads are needed, particularly from recent projects. Apart from the execution frameworks provided by the Java class library, such as `ThreadPoolExecutor` and `ForkJoinPool`, real-world projects could also employ custom frameworks that only rely on basic threading primitives (i.e., the `Thread` class and the `Runnable` or `Callable` interfaces). Our approach allows the researcher to find workloads that use either standard or custom task execution frameworks.

In the second use case (invokedynamic), we consider a Java Virtual Machine (JVM) developer tasked with optimizing the execution of the `invokedynamic` instruction introduced in Java 7. This instruction simplifies execution of dynamic languages hosted on the JVM platform [6], and is also used to implement lambda expressions in Java 8. To evaluate performance improvements due to a hypothetical `invokedynamic`-related optimization in a production environment, the developer is looking for benchmarks based on real-world workloads actually using the `invokedynamic` instruction. In this case, an established benchmark suite such as DaCapo cannot be used, because it is simply too old. Our approach allows the developer to search for relevant, real-world code suitable for the evaluation.

In the third use case (actors), we consider a user looking for realistic workloads utilizing specific actor libraries, with the goal of discovering optimizations (i.e., library-specific programming tricks) that she could learn and apply to her own actor-based application. The actor programming model is becoming increasingly popular, especially for high-performance scientific computations [7], and is provided in different flavors by various libraries. While actor-specific benchmark suites exist [8], they mostly focus on highlighting differences between actor libraries, rather than on representing real applications. Our approach can help the user find real applications employing a specific actor library, such as Akka[2] or JetLang[3].

III. The AutoBench Toolchain

Creating a benchmark suite involves several steps: i) finding the workloads specific to the desired context; ii) classifying the workloads using context-specific metrics; iii) selecting concrete workloads from the candidates; and iv) packaging the workloads into an easy-to-use artifact.

We have developed the AutoBench toolchain to automate most (if not all) of these steps. AutoBench is based on crawling public open-source repositories and applying pluggable user-defined analyses to unit tests[4] found in the discovered projects, thus enabling identification and characterization of candidate workloads. The overall architecture of AutoBench is shown in Figure 1.

We note that completely automating benchmark construction relies on all decisions and choices being automated. This may be difficult to achieve, especially because the step involving choice of concrete benchmarks is likely to require human insight. Also, while AutoBench does not support packaging workloads into a distributable artifact yet, it still offers a simple benchmark harness. We now discuss some of the details related to the steps preceding the choice of concrete workloads.

A. Finding Useful Unit Tests

To discover candidate projects, AutoBench crawls the GitHub[5] repository looking for Java and Scala projects with a Maven-based[6] build system and JUnit[7] as the underlying unit testing framework. We found many projects matching these

[2]http://akka.io

[3]https://github.com/jetlang/core

[4]For practical purposes, we consider a unit test to be any code that can be run unattended using a testing framework.

[5]https://github.com/

[6]https://maven.apache.org/

[7]http://junit.org/

constraints, but if necessary, extending AutoBench to support other repositories, build systems, and testing frameworks is only a matter of additional engineering effort.

AutoBench downloads and compiles the discovered projects, and attempts to execute their tests through the build system. To determine whether a unit test could serve as a workload, AutoBench times the execution of all passing tests using an execution-time profiler—the idea being that only longer-running tests have a chance of containing an interesting workload. The profiler uses instrumentation to measure the net duration of a unit test execution, excluding time spent initializing the JVM and the testing framework.

To decide whether a unit test execution is long enough to be potentially useful, we turned to existing benchmark suites for baseline times. Table I shows the minimum execution times[8] for the first iteration of benchmarks from three established benchmark suites, along with the number of benchmarks in each suite. Based on these times, we require a unit test to execute for at least 1 s to be considered potentially useful. To put together a benchmark suite, we expect to find at least 12 such tests matching context-specific criteria.

Table I. PROPERTIES OF ESTABLISHED BENCHMARK SUITES.

Benchmark Suite	N. of Benchmarks	Minimum Exec. Time
DaCapo-9.12-bach	14	1.1 s
ScalaBench	12	0.8 s
SPECjvm2008	16	0.8 s

We note that these limits are only meant for first-level filtering, and that just meeting these criteria does not turn a unit test into a benchmark workload. We elaborate on this in the following sections.

B. Context-specific Filtering

Because users are expected to have different requirements regarding what the workloads should exercise, AutoBench allows composing a filter pipeline, in which each filter refines the results produced by its predecessor. The filtering pipeline basically serves as a compound predicate with short-circuiting evaluation for efficiency—a unit test passing all the filters is considered to satisfy the context-specific criteria. In general, the filters can be arbitrary, but would typically comprise user-defined static, dynamic, or hybrid analyses, targeting the project source code, class bytecode, or results of dynamic analyses collected during test execution.

To ease the development of dynamic analyses, AutoBench integrates support for the DiSL[9] instrumentation framework, and provides an interface which allows a developer to subscribe to notifications indicating the start and the end of a unit test execution. Additional analysis-specific event triggers may be inserted using DiSL. Thanks to DiSL, the instrumentation can also cover the Java class library. This allows taking library-level behavior of a test into account during filtering.

The short-circuiting evaluation of the filter pipeline suggests that filters should be ordered from coarse-grained and cheap (in terms of execution time) to fine-grained and expensive. A filter pipeline aimed at finding workloads using a specific API would therefore first apply pattern matching (of API invocations, for example) to project source files, and then use DiSL-based instrumentation to discover concrete invocations of the API during test execution.

C. Characterizing Workloads to Facilitate Selection

Selecting concrete workloads from a set of candidates is the most difficult step in benchmark suite construction, one in which we anticipate the need for human insight. Nevertheless, AutoBench supports automating this step, as long as the selection process is executable by a computer.

To facilitate selection of diverse workloads from many candidates, this phase is intended for running user-defined workload characterization analyses that collect comprehensive information about candidate workloads. Unlike in the filtering phase, where short-circuiting evaluation is used, here all the analyses are run to completion. In general, the user of AutoBench is fully responsible for the outcome of the selection procedure—it may be used just to collect workload characterization data, leaving the final selection to the user.

As a proof of concept, we provide an automated selection procedure based on *k-means* clustering [9] applied to workload characterization metrics upon completion of all scheduled analyses. This allows selecting a fixed number $k = 12$ of workloads (to serve as the basis of a benchmark suite, cf. Section III-A) as the final result of AutoBench execution.

The clustering procedure first chooses initial centroid workloads using the *K-means++* algorithm [10], and then iteratively assigns all other workloads to their nearest[10] centroid. To reduce the chance of selecting multiple workloads from the same project, the project name is part of the workload characterization data. When the clustering stops assigning any workloads to a new or to a different cluster, we compute, for each cluster, the nearest workload to the cluster centroid, resulting in 12 workloads with diverse metrics.

To simplify development of general-purpose benchmark suites, we provide generic analyses as a part of our toolchain, consisting of a mix of static (e.g., code complexity analyzer and source code counter) and dynamic analyses (e.g., code coverage analyzer and calling-context profiler JP2 [11]).

IV. EVALUATION

Our evaluation of AutoBench is preliminary, driven primarily by the need to assess the feasibility of our approach. We evaluate AutoBench for the three different use cases presented in Section II. The evaluation consists of multiple steps corresponding to the tasks outlined in Section III, each intended to provide an answer to a specific question:

Q1 Can we expect to find (unit) tests that are long enough to serve as benchmark workload?

[8] Measured on a multicore platform (Intel Xeon E5-2680 2.7GHz with 8 cores, 64 GB of RAM, CPU frequency scaling and Turbo mode disabled, Oracle JDK 1.8.0_66 b17 Hotspot Server VM 64-bit, Ubuntu Linux Server 64-bit version 14.04.3 64-bit).

[9] http://disl.ow2.org/

[10] We use the squared Euclidean distance to compute the distance $d_{ij}(\mathbf{x_i}, \mathbf{c_j})$ between one point $\mathbf{x_i}$ and centroid $\mathbf{c_j}$, $j = 1..k$: $d_{ij} = (\mathbf{x_i} - \mathbf{c_j}) \cdot (\mathbf{x_i} - \mathbf{c_j})$.

Q2 Can dynamic analysis improve identification of workloads matching a specific evaluation context?

Q3 Can we find diversity in the workloads that would allow synthesizing a benchmark suite?

In the first step (cf. Section III-A), we let AutoBench crawl repositories hosted on GitHub looking for suitable projects. At the time of writing, AutoBench discovered 10339 Java or Scala projects which were downloaded and checked for usage of Maven and JUnit. Even though less than 2 % of projects made it past this stage, compiling and measuring the duration of unit tests in these projects would be too inefficient. Unit test filtering is therefore split into two stages. The first (pre-filtering) stage occurs prior to compilation of a candidate project, and is expected to utilize pattern matching or simple static analyses to quickly exclude uninteresting projects. AutoBench then compiles and measures unit test duration for projects that pass the pre-filtering stage. The second-stage filtering is then performed only for tests that execute longer than 1 s.

Table II. TEST EXECUTION TIME AFTER PRE-FILTERING.

Use Case	Total		Execution Time		
	Projects	Tests	≥ 1 s	≥ 10 s	≥ 100 s
executor	133	50704	2869	290	14
invokedynamic	155	52016	2085	196	1
actors	1	3627	326	21	0

Table II summarizes the execution time for tests that passed the pre-filtering stage. We observe that unit test execution time exceeds 1 s in approximately 5 % of the cases. Given enough projects and tests, repository crawling can indeed discover unit tests to be considered as workload candidates, leading us to answer **Q1** in a positive way.

In the second step (cf. Section III-B), AutoBench applies the second-stage filtering to the candidate tests. We use three instrumentation-based profilers that respectively intercept runtime usage of Java executor APIs, of `invokedynamic` bytecodes, and of actor libraries, allowing us to filter out workloads that do not match the behavior desired for the respective use case. For the lack of space, we do not elaborate the technical details of these profilers.

Table III. WORKLOAD CANDIDATES AFTER FILTERING.

Use Case	Unit Tests After Filtering			
	1st Stage		2nd Stage	
executor	2869	5.66 %	493	17.18 %
invokedynamic	2085	4.01 %	4	0.19 %
actors	326	8.99 %	24	7.36 %

Table III summarizes the effect of second-stage filtering. We note that the percentage of tests passing the second filtering stage refers to the number of tests passing the pre-filtering stage. Using dynamic analysis in the second filtering stage significantly helps filter out candidates that are not suited for the given evaluation context. Based on these results, we consider the answer to **Q2** to be also positive. We note that in the case of the invokedynamic use-case, the number of candidates dropped to just four test cases, showing that invokedynamic is rarely used at the moment; we expect this to change in the near future when more projects make use of lambda expressions (introduced in Java 8).

In the third and last step (cf. Section III-C), AutoBench performs workload characterization analyses for all remaining candidate workloads. To obtain results in a timely fashion, we used light-weight instrumentation-based profilers to collect the number of method invocations and the number of object allocations during test execution. While much more comprehensive (and heavier) tools for workload characterization exist [11], the goal of this step was primarily to determine whether there is diversity in the workloads that could be used for the final workload selection. The collected metrics were, together with test execution time and project name, subjected to the *k-means* clustering analysis, with $k = 12$ corresponding to the number of desired workloads. We only performed the clustering analysis for the executor case study, because it still had a number of workload candidates unsuitable for manual selection.

Table IV shows the results of the clustering analysis performed on the 493 candidate workloads for the executor case study. The 12 unit tests suggested by our proof-of-concept selection procedure come from various projects and exhibit significant diversity in their execution times, number of method invocations, and object allocations. Given these results, we consider the answer to **Q3** to be positive.

V. DISCUSSION

The results of the preliminary evaluation of the AutoBench toolchain are encouraging, especially for the executor use case, in which AutoBench managed to sift through thousands of unit tests, and obtain—fully automatically—a selection of potentially exploitable workloads. Before synthesizing a benchmark suite out of these workloads, we would still subject them to more thorough analysis, because here we have only used light-weight workload characterization methods.

The actors use case resulted in 24 workload candidates which could be reviewed manually. AutoBench found only a single relevant project, most likely because the actor model is not so popular in the Java ecosystem. A possible remedy is to extend AutoBench to support, e.g., build systems and unit testing frameworks typical for projects in Scala, which is often the language of choice for actor-based applications.

The invokedynamic use case turned out to require workloads that are rare. A possible reason is that `invokedynamic` is a recent, but rarely used JVM feature intended for compiler writers, which Java only started using internally with the introduction of lambdas in Java 8. Hence it may just take more time for open-source projects to make substantial use of Java features that trigger `invokedynamic` usage.

With respect to the principal **RQ** posed initially, we clearly cannot provide a definitive answer based on a preliminary study. However, we are positive in that the results suggest that our approach is feasible, encouraging further research.

VI. RELATED STUDIES

Automatic benchmark synthesis has been investigated in prior research. Bell et al. [12] propose a technique for generating reduced, shorter-running benchmarks from actual applications. Their approach relies on subjecting the applications to workload characterization to extract dynamic metrics which the synthesized benchmarks try to reproduce. Van Ertvelde et al. [13] propose a technique for generating benchmarks based on existing (proprietary) applications. Their goal is to allow

Table IV. WORKLOADS SUGGESTED BY OUR PROOF-OF-CONCEPT SELECTION PROCEDURE FOR THE EXECUTOR USE CASE.

#	Project	Unit Test	Time [s]	Allocations	Invocations
1	prova	test/ws/prova/test2/ProvaWorkflowsTest.predicate_join	2.523	17938	469320
2	datacube	com/urbanairship/datacube/HBaseBackfillerTest.testMutationsWhileBackfilling	96.806	127961	7628592
3	jdeferred	org/jdeferred/impl/FilteredPromiseTest.testNoOpFilter	2.003	72	755
4	datacube	com/urbanairship/datacube/BackfillExampleTest.test	62.371	78728	4367224
5	cmb	com/comcast/cmb/test/unit/CassandraTest.testCassandraCounters	4.007	54	636
6	svarut	no/kommune/bergen/soa/svarut/ServiceContextTest.init	1.983	12510	205187
7	antlr4	org/antlr/v4/test/runtime/java/TestPerformance.testExpressionGrammar_1	1.212	109350	2595015
8	prova	test/ws/prova/test2/ProvaMessagingTest.ring_parallel	17.961	251119179	3144752537
9	pangool	com/datasalt/pangool/tuplemr/mapred/lib/output/TestMultipleOutputs.test	2.088	241759	12943082
10	prova	test/ws/prova/test2/ProvaFunctionalProgrammingTest.func_reactive_unfoldr_iteration_perf_large	2.690	78180455	928571592
11	graphdb	org/neo4j/backup/TestBackup.fullThenIncremental	2.159	11568	88334
12	prova	test/ws/prova/test2/ProvaMetadataTest.cep005	13.267	30393192	443299688

replicating the CPU-level behavior of the application without revealing proprietary information or requiring the vendor to make the application publicly available. Joshi et al. [14] propose a framework for automatic benchmark synthesis that does not require an existing application to serve as a model. Instead it takes CPU-level workload characteristics as input, and generates a workload with the specified characteristics. In summary, these techniques *generate* short workloads with the desired CPU-level characteristics. In contrast, our approach aims at synthesizing benchmarks by finding *existing* real-world workloads exhibiting user-defined behavior. While some authors point out the importance of unit tests in software projects [4], [5] and use them to increase performance awareness [15], to the best of our knowledge we are the first to consider unit tests as a possible source for benchmarks.

Repository mining [16] has been previous applied to open-source projects with different goals, including bug prediction [17], automatic comment generation [18], code clone detection [19], and authorship determination [20]. Although static analysis is usually preferred for analyzing code in repositories, some researchers utilize also dynamic analyses, e.g., to aid code refactoring [21]. Our approach allows using hybrid analyses to identify workloads suitable for benchmarking. Compared to other open-source repository crawlers [22], AutoBench combines code crawling with pluggable hybrid analyses to find, filter, and characterize workloads of interest.

VII. CONCLUSIONS

In this paper, we explored the feasibility of using unit tests as workloads in custom benchmarks. To this end, we present AutoBench, a toolchain that automatically finds, filters, and classifies workloads found in open-source projects. Preliminary evaluation of AutoBench shows encouraging results for identification of the desired workloads via pluggable hybrid analyses. In future, we plan to extend AutoBench to support additional build systems and testing frameworks, as well as develop a domain specific language to aid in describing the filtering, characterization, and workload selection goals. We plan to make a public release of AutoBench in the near future.

REFERENCES

[1] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications," in *ACM OOPSLA*, 2013, pp. 355–372.

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur *et al.*, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *ACM OOPSLA*, 2006, pp. 169–190.

[3] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder, "Da Capo con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine," in *ACM OOPSLA*, 2011, pp. 657–676.

[4] R. Takasawa, K. Sakamoto, A. Ihara *et al.*, "Do Open Source Software Projects Conduct Tests Enough?" in *Product-Focused Software Process Improvement*, ser. LNCS, 2014, vol. 8892, pp. 322–325.

[5] M. Beller, G. Gousios, and A. Zaidman, "How (Much) Do Developers Test?" in *ICSE*, 2015, pp. 559–562.

[6] F. Ortin, P. Conde, D. Fernandez-Lanvin, and R. Izquierdo, "The Runtime Performance of invokedynamic: An Evaluation with a Java Library," *IEEE Software*, vol. 31, no. 4, pp. 82–90, 2014.

[7] P. Stutz, B. Paudel, M. Verman, and A. Bernstein, "Random Walk TripleRush: Asynchronous Graph Querying and Sampling," in *WWW*, 2015, pp. 1034–1044.

[8] S. M. Imam and V. Sarkar, "Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries," in *AGERE!*, 2014, pp. 67–80.

[9] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, 2nd ed. Springer New York Inc., 2009.

[10] D. Arthur and S. Vassilvitskii, "K-means++: The Advantages of Careful Seeding," in *ACM-SIAM SODA*, 2007, pp. 1027–1035.

[11] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini, "JP2: Call-site Aware Calling Context Profiling for the Java Virtual Machine," *Science of Computer Programming*, vol. 79, pp. 146 – 157, 2014.

[12] R. Bell Jr and L. K. John, "The Case for Automatic Synthesis of Miniature Benchmarks," in *MoBS*, 2005, pp. 4–8.

[13] L. Van Ertvelde and L. Eeckhout, "Benchmark Synthesis for Architecture and Compiler Exploration," in *IEEE IISWC*, 2010, pp. 1–11.

[14] A. Joshi, L. Eeckhout, and L. K. John, "The Return of Synthetic Benchmarks," in *SPEC Benchmark Workshop*, 2008.

[15] V. Horký, P. Libič, L. Marek, A. Steinhauser, and P. Tůma, "Utilizing Performance Unit Tests to Increase Performance Awareness," in *ACM/SPEC ICPE*, 2015, pp. 289–300.

[16] A. Hassan, "The Road Ahead for Mining Software Repositories," in *Frontiers of Software Maintenance*, 2008, pp. 48–57.

[17] X. Xia, D. Lo, E. Shihab, X. Wang, and B. Zhou, "Automatic, High Accuracy Prediction of Reopened Bugs," *Automated Software Engineering*, vol. 22, no. 1, pp. 75–109, 2015.

[18] E. Wong, T. Liu, and L. Tan, "CloCom: Mining Existing Source Code for Automatic Comment Generation," in *IEEE SANER*, 2015, pp. 380–389.

[19] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free Code Clone Detection for A Large-scale Heterogeneous Java repository," in *IEEE SANER*, 2015, pp. 201–210.

[20] X. Meng, B. P. Miller, W. R. Williams, and A. R. Bernat, "Mining Software Repositories for Accurate Authorship." in *IEEE ICSM*, 2013, pp. 250–259.

[21] S. Kimura, Y. Higo, H. Igaki, and S. Kusumoto, "Move Code Refactoring with Dynamic Analysis," in *IEEE ICSM*, 2012, pp. 575–578.

[22] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes *et al.*, "Sourcerer: Mining and Searching Internet-scale Software Repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.