A DSL-based framework for performance assessment

Hamid El Maazouz[1,2] Guido Wachsmuth[2] Martin Sevenich[2] Dalila Chiadmi[1]
Sungpack Hong[2] Hassan Chafi[2]

[1] Ecole Mohammadia d'ingénieurs, Avenue Ibnsina B.P. 765 Agdal Rabat, Morocco
`emi@emi.ac.ma / contact@emi.ac.ma`,
WWW home page: `https://www.emi.ac.ma`
[2] Oracle Labs, 500 Oracle Parkway Redwood Shores, CA 94065, US
WWW home page: `https://labs.oracle.com`

Abstract. Performance assessment is an essential verification practice in both research and industry for software quality assurance. Experiment setups for performance assessment tend to be complex. A typical experiment needs to be run for a variety of involved hardware, software versions, system settings and input parameters. Typical approaches for performance assessment are based on scripts. They do not document all variants explicitly, which makes it hard to analyze and reproduce experiment results correctly. In general they tend to be monolithic which makes it hard to extend experiment setups systematically and to reuse features such as result storage and analysis consistently across experiments. In this paper, we present a generic approach and a DSL-based framework for performance assessment. The DSL helps the user to set and organize the variants in an experiment setup explicitly. The Runtime module in our framework executes experiments after which results are stored together with the corresponding setups in a database. Database queries provide easy access to the results of previous experiments and the correct analysis of experiment results in context of the experiment setup. Furthermore, we describe operations for common problems in performance assessment such as outlier detection. At Oracle, we successfully instantiate the framework and use it to nightly assess the performance of PGX [12, 6], a toolkit for parallel graph analytics.

1 Introduction

With our increasing reliance on software systems in various domains such as defense and health care, verification and quality assurance have grew fundamental in insuring that functional and non-functional requirements are met. Companies needed to validate features and quality of their products and services, they needed to assure customers and gain their confidence. Researchers needed to provide credible contributions, they needed to build on existing experiments, and assess the progress of their endeavors. This overwhelmed 3^{rd} generation technologies and made development and testing in today's platforms tedious and complex [13].

Performance assessment, for example, involves measurement of metrics such as response time, bandwidth utilization, and memory consumption with regard to system settings, execution environment settings, and workload definitions. System settings are both domain and system specific. Execution environment settings may include different processor architectures such as Intel 64 and Sparc and different types of software such as the Linux Kernel and the Java Runtime Environment. Workload definitions may include datasets, sessions, and requests. This process also involves activities such as result storage, analysis, and visualization. Setting up this process for a software system is challenging because of the big dimensionality present in these settings and the nowadays' complex experimentation requirements [2, 4, 17].

Typical approaches to performance assessment rely on scripting languages as they are weakly typed and allow developers to easily hook different components of the software system. However, this encourages uniformity of code and data, and renders different concepts interchangeable [11]. Consequently, these approaches tend to flatten experiment setups with commands, which often become mixed and duplicate mainly due to different levels of expertise and objectives of involved human resources [3], and their lack of clean code guidelines [10]. Developers, for example, have more technical knowledge about the system than operations engineers which in their turn are more attached to the domain compared to quality assurance engineers. This results in a complex system that makes it hard to extend and maintain experiment setups or reuse features such as result storage and analysis consistently. These scripts are also considered a valuable investment that is unfortunately often thrown away [7].

All of these factors make it challenging to systematically express, extend, reuse, or link experiment setups to results for analysis and visualization [17]. To address these issues, this paper presents a generic approach to performance assessment by introducing a powerful representation of experiment setups. Based on this approach, a DSL-based framework defines a language module for expressing experiment setups explicitly in addition to other modules that help organize and automate performance assessment activities such as result storage and analysis. Our specific contributions in this paper can be summarized in the following:

- A generic DSL-based framework for performance assessment (cf. Section 3).
- A practical approach to implementing DSLs using the Gradle build language (cf. Section 4).
- A successful instantiation of the framework and its evaluation in the context of PGX, a toolkit for parallel graph analytics (cf. Section 5).

Using this approach to performance assessment considerably reduces the complexity of experiment setups, their organization, and association to results as well as their storage for further purposes such as analysis and visualization. This is prominent because it saves developers and experimenters from many repetitive tasks enabling them to focus more on actual development and performance tuning without worrying as much about managing performance related activities.

2 Related works

An experiment in general is an empirical procedure carried out to investigate the validity of a hypothesis. It mainly aims at studying correlations between dependent and independent variables. This is done by manipulating the independent variable and measuring the effect on the dependent variable. Whether the hypothesis is refuted or supported, an experiment brings new knowledge about the variables and their variations. Performance assessment, for example, is an experimental process that involves execution of experiments to assess how different parts of a system perform. Setting up these experiments, expressing them, and managing activities related to performance has been impeding systematic integration of performance assessment as part of the project's regression pipeline [3, 7].

Model-driven engineering (MDE) is a paradigm that came to raise abstraction level at which computer programs are written. In this paradigm, the complexity of platforms and applications is alleviated by considering models as first-class entities in the software development process [13]. For this purpose, MDE technologies aim at building models to capture domain concepts, their relationships, and the constraints associated to these concepts. These models are then used to automatically produce artifacts such as application source code, configuration, and documentation. This automated transformation helps reduce implementation effort and time and ensures consistency and exchange between implementations.

The use of MDE techniques is ubiquitous for many purposes such as augmenting developer productivity, reducing domain complexity [5, 16, 8], and facilitating integration of functional and non-functional requirement testing [3]. For performance assessment, Bernardino et al. [14] adopted a model-based testing approach to automate activities of performance testing in web applications. They proposed a domain specific language (DSL) for modeling performance tests and generating test scenarios and scripts automatically. Barve et al. [1] created a framework for assessing cloud application performance. The proposed framework is derived from a set of meta-models that describe an aspect of the performance assessment pipeline. It also comprises a graphical domain specific modeling language (DSML) for specifying performance experiments which are transformed into low-level scripts for configuring, deploying and measuring metrics of cloud applications on a given platform. The drag-and-drop nature of the language in addition to high level abstractions make it easy for performance engineers to monitor cloud application performance. Bianculli et al. [2] proposed a framework for defining testbeds for service-oriented architecture. This framework consists of a modeling environment that features a DSL (based on a metamodel) for defining a testbed model of a service-oriented architecture (SOA), a set of generators that process an input testbed (using scripts) to produce the actual testbed components (i.e. the mock-ups of the services, the testing clients, and the service compositions to execute) in addition to other artifacts such as deployment descriptors and helper scripts, and a compiler that transforms the generated artifacts into a format supported by the underlying platform that will execute experiments. Ferme et al. [3] introduced and implemented a global approach to integrating performance engineering activities into continuous software integration pipeline (CSI). Their approach features templates for users to set test requirements using a YAML-based DSL to declaratively express performance test configurations (e.g. load functions and workload definitions).

Our approach to performance assessment is domain agnostic and presents a more global framework in which the DSL and Runtime modules are inspired from MDE techniques. The DSL is based on a strong formalism and supports explicit and organized experiment models. Moreover, our unique use of the Gradle build language to implement the DSL is practical for any Gradle project.

3 DSL-based framework

Experimental design generally consists of choosing participants, partitioning them into groups, and assigning the groups to different environments. Both the participants and the environments can have properties where various combinations are possible within an experiment setup.

In the context of performance assessment, the experiment setup space can be similarly organized by deriving participant and environment types from the experimentation requirements. This is possible because they necessarily describe relationships between system settings, execution environment settings, and workload definitions. These relationships help identify similarities in the settings which makes it possible to group them into participant and environment types. This grouping also reduces the dimension of the experiment setup space and leverages good understanding of these relationships.

Performance assessment is additionally responsible for concerns such as experiment setup processing, experiment execution, result storage, analysis, and visualization. Scripting languages were mainly meant for wiring parts of a modular system instead of implementing the system itself [11]. Therefore they tend to render these parts equipotent and do not encourage separation of concerns. Although they can still provide this organization of the experiment setup space for the software system, they cannot protect or maintain it due to their weak typedness and their unrestrictive coding style. In order to address these shortcomings, we extend the DSL design methodology in [15] by further deriving a framework from the DSL to address the concerns of performance assessment.

3.1 Performance assessment DSL

Organization of the experiment setup space requires addressing challenges raised in earlier 3^{rd} generation approaches [13, 15] such as we gathered in the following:

- **Explicitness**: The DSL syntax needs to allow clear and declarative expression of an experiment's intent.
- **Completeness**: The DSL should be capable of expressing any experiment setup.

– **Flexibility**: Requirements evolve and can become complex. The DSL needs to provide flexible control on the granularity of the settings. This means the DSL needs to be resilient, extensible, and maintainable.
– **Reusability**: Settings in the experiment setup space should be reusable. This will allow users to briefly express experiment setups.

Organization of the experiment setup space based on experimental design establishes relationships between participant and environment types. These relationships are hierarchical and can be best formalized by a tree structure such as in Figure 1 where every node (e.g. R) represents a participant type and is fully described with its properties in the form of key-value pairs (e.g. $\forall i : R.prop_i = u_i$).

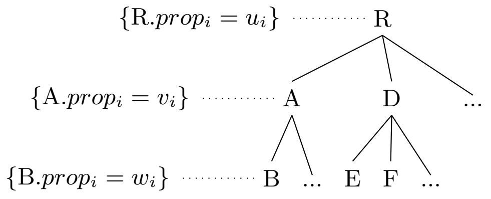

Fig. 1. Example of an annotated tree structure.

We based the DSL on the annotated tree structure because it adequately and completely models the hierarchical relationships between the concepts and allows for explicit expression of these concepts and their properties. Additional syntactic and semantic rules could be enforced on the DSL to allow flexibility and reusability of concepts and/or their properties.

3.2 Performance assessment framework

The framework presented in Figure 2 is composed of 4 main modules. The DSL module allows expressing performance assessment experiment setups systematically. Experiment setups are written in the DSL and are compiled into concrete experiments. The Runtime module runs on the system under test and contains interpreter objects for executing these experiments in addition to APIs and helper utilities for performance metric measurement operations. When an experiment finishes running, both the concrete experiment and results are sent by the Runtime module to the Persistence module. This module is accessed through interfaces and exposes operations such as storage, retrieval, and basic statistical aggregation. The Analysis and Visualization module assess experiment results. It serves for extracting insights about the evolution of the project's performance. A common problem in this assessment is outlier detection. This is addressed by collecting statistics on performance metrics during a period of time in the past and comparing them to current results. Tolerance intervals are defined to detect outliers which are included, among other statistics, in the performance reports which are then easily interpreted by the performance task force.

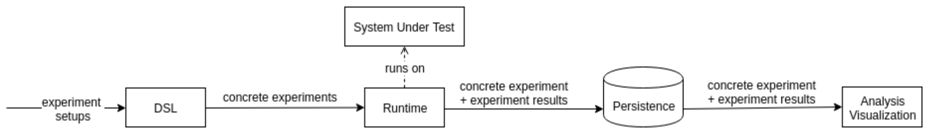

Fig. 2. Architecture of the performance assessment framework.

The DSL and Runtime modules are inspired from MDE techniques and provide a higher abstraction that is capable of closing the gap between the intent of an experiment setup and the expression of the intent [11]. We describe, in the next section, a practical implementation of the DSL module.

4 Gradle based implementation of the DSL module

The annotated tree structure formalism presented in Section 3 supports explicit and well organized experiment models. However, implementing a DSL for each experimentation domain from scratch is costly in terms of additional complexity incurred by the actual project domain because it requires compiler expertise and considerable man-hours to design, develop, and maintain the different language aspects of the DSL.

Embedded DSLs is an implementation approach based on extending a given base language and completely uses its compilation mechanisms [15]. This means the costs of building a DSL compiler or interpreter and maintaining them are completely eliminated. The main bottleneck of this approach is expressiveness of the base language, thus its choice matters most to fit notation requirements of the domain-specific constructs.

An example of such a base language is the Gradle build language. Gradle provides this language itself as an embedded DSL in Groovy or Kotlin for setting up project configuration. We chose to embed our performance assessment DSL in the Gradle build language and provide it as a Gradle plugin. These choices are motivated by the following rationales:

– The Gradle build language provides an explicit and declarative syntax and is also already available, thus we benefit from compilation and editor services for free.
– Gradle supports plugin development. Implementation as a plugin allows for modularity, free and seamless integration into the project's build system as well as into its continuous regression pipeline.
– Gradle is widely adopted by many projects for its ease-of-use and its high degree of automation. This means this DSL implementation is far-reaching and very practical.

Our implementation of the DSL module defines two types of data structures as Plain Old Java Objects (POJOs). The frontend POJOs describe the DSL syntax and are exposed to the project through Gradle extensions to hold experiment setups. The backend POJOs describe the concrete experiments that are generated from compiling the frontend POJOs. This compilation is achieved by methods defined in the frontend POJOs to analyze, optimize, and transform experiment setups into experiment objects.

The plugin defines three main tasks to manage invocation of the performance assessment process. The first task performs analysis on the experiment setups, the second task transforms the analyzed experiment setups into concrete experiments, and the third task invokes the Runtime module and handles parallel execution of the generated concrete experiments.

Both extensions and tasks are injected into the main project's build system by simply applying the plugin in its build script. This means the performance assessment process can be made available for on-demand and continuous invocation for free.

With the embedded languages approach, DSL implementation is reduced to only adding syntactic domain-specific constructs and implementing custom transformation rules. It also does not require much effort from a single developer. In fact, with prior comprehension of the domain idiom [15] and moderate experience in Gradle development, we estimated this effort to be maximum 40 man-hours or equivalent to implementing a fully functional simple Java project.

Although we have designed generic interfaces for the other modules, we chose to omit them in this publication for simplicity and lack of space. Usage of the DSL consists of writing systematic experiment setups and using the produced concrete experiments in the context of a given project. This means implementation of a Runtime module is due to host and execute the produced artifacts. The cost of this implementation depends on the execution semantics of the concrete experiments, the project's own functioning, and the target performance metrics to collect. In the following section, we evaluate the framework on a real academic and industrial project.

5 Evaluation

The research and development team in PGX [12, 6] is responsible for identifying and exploring new technologies that can improve graph analysis in Oracle products. For this purpose, it needs to comprehend nowadays's industrial requirements as well as actively research on how to best address these requirements. In this section, we chose PGX to evaluate our approach to performance assessment.

PGX is a high performance toolkit for graph analysis that supports running algorithms such as PageRank and performing SQL-like pattern matching on graphs [5, 16]. The PGX toolkit includes both single-node in-memory engine and distributed engine for extremely large graphs. Graphs can be loaded from a variety of sources such as the filesystem, a database, and HDFS and in various formats such as adjacency list and edge list.

Performance assessment of PGX mainly requires measurement of execution time and memory consumption of graph workloads such as graph loading, graph algorithms [5], and graph queries [16]. These operations can run in single-node in-memory or distributed engines and on many graph datasets. The single-node in-memory engine has many configurations such as number of threads and thread scheduling strategies. The distributed engine runs on many machines and has many configurations such as buffer size and graph partitioning strategies. Hardware resources are allocated in a cluster and need to be managed properly. Measurements are taken many times and stop conditions are needed to limit resource utilization in the case of lengthy or indeterministic experiments.

Using the DSL module in Section 3, we systematically represented these requirements in the form of the annotated tree structure. Algorithms and graphs illustrated in Figure 1.1 are nodes of the tree, they both have properties and are organized in such a way to fulfill performance assessment idiom. All `algorithms` in the experiment setup space, for example, run on PGX single-node in-memory engine `shared-memory` and are limited to run under a `timeout`. They have mandatory properties such as `source` code and `arguments`, and optional properties such as the `engine` and `timeout`. `"Pagerank"` algorithm runs on `"San Francisco street graph"` and `"Topological Schedule"` runs on `"LiveJournal social network"` and `"Twitter"` graphs. Graph datasets also have mandatory properties such as the `config` file, and optional properties such as `engine` and `timeout`.

To enable briefness, we provided support for default comprehensive configuration globally for the experiment setup space and locally for graph and operation nodes. We also enforced intuitive rules on the tree such as that all properties of a node apply to all of its children. This means users do not have to repeat setting the same settings thus allowing more reusability. Conflicts between the same properties are resolved either by giving priority to the innermost or by merging in the case of operation and graph arguments.

```
algorithms {
  engine "shared-memory"
  timeout "3600 SECONDS"
  "Pagerank" {
    source "./algorithms/pagerank.gm"
    arguments [["tol": 0.001, "damp": 0.85, "norm": false]]
    graphs {
      "San Francisco street graph" {
        timeout "1200 SECONDS"
      }
      // Other graphs ...
    }
  }
  "Topological Schedule" {
    source "./algorithms/topological_schedule.gm"
    arguments [["source": [1, 2, 3]]]
    engine "distributed"
    graphs {
      "LiveJournal social network" {
        arguments = [[source: [59810, 59811, 59812]]]
        timeout "900 SECONDS"
      }
      "Twitter" {
        timeout "1000 SECONDS"
      }
    }
  }
  // Other algorithms ...
}
```

Listing 1.1. Experiment setup examples for performance assessment of Green-Marl algorithms.

Invocation of the performance assessment framework is done through Gradle commands. With no parameters, it will run the entire experiment setup space. To allow selection of targeted experiment setups, we devised filters with the help of Gradle project properties illustrated in Listing 1.2. These commands compile experiment setups and invoke the Runtime module on the generated experiments. Listing 1.3 shows a fragment of printed logs showing progress of execution. Results along with concrete experiments are optionally saved through REST calls to a database or simply saved on the local filesystem.

```
./gradlew :qa_framework:gmBenchmark -Poperations="Pagerank"
-Pgraphs="San Francisco street graph" -Pruntimes="sm"
```

Listing 1.2. Gradle command to only run PageRank algorithm on San Francisco street graph using PGX SM engine.

```
Benchmarking GM algorithms for PGX SM
Starting engine with 72 threads and 'ENTERPRISE_SCHEDULER'
Loading graph 'San Francisco street graph' ...
Graph 'San Francisco street graph' loaded in 0 MINUTES
Preprocessing graph for GM algorithm 'Pagerank'

Resizing thread pool to 1 threads for 'ENTERPRISE_SCHEDULER'
Running algorithm 'Pagerank' 5 times or within 1200 SECONDS
Starting BASIC memory listener
Measurement 1/5 took 196 ms, time left: 1199803 ms
Measurement 2/5 took 45 ms, time left: 1199757 ms
Measurement 3/5 took 42 ms, time left: 1199714 ms
Measurement 4/5 took 40 ms, time left: 1199674 ms
Measurement 5/5 took 33 ms, time left: 1199640 ms
Stopping BASIC memory listener

Resizing thread pool to 4 threads for 'ENTERPRISE_SCHEDULER'
Running algorithm 'Pagerank' 5 times or within 1200 SECONDS
Starting BASIC memory listener
Measurement 1/5 took 218 ms, time left: 1199781 ms
Measurement 2/5 took 110 ms, time left: 1199671 ms
Measurement 3/5 took 30 ms, time left: 1199640 ms
Measurement 4/5 took 16 ms, time left: 1199624 ms
Measurement 5/5 took 16 ms, time left: 1199608 ms
Stopping BASIC memory listener
```

Listing 1.3. Log excerpt from the framework invocation by the command in Listing 1.2.

6 Conclusion

Performance assessment is ubiquitous in many industrial and academia projects. Moreover, evolution of projects necessitates a high degree of automation and ease-of-use in the performance assessment process. In this paper, we described our approach to performance assessment by introducing a DSL to express experiment setups systematically from which we also derived a framework to automate performance assessment activities such as result storage, analysis, and visualization. Experimental design inspired us to group settings into participant and environment types, which greatly helped in reducing the complexity and improved understanding and organization of the experiment setup space. Moreover, our choice of the Gradle build language as the base language for the DSL made it seamless to automate activities in the performance assessment process. We believe the presented DSL is powerful, inexpensive to implement, practical, and could be instantiated to address experimentation requirements of other projects.

We envision as future work to support more advanced result analysis and visualization. Response time and memory consumption were the main performance metrics addressed in this work, and we plan to support detailed memory profiling as well as processor and cache performance. Resources for executing experiments such as individual machines or clusters have to be closely monitored and used optimally by the performance assessment framework. This relates to the number of measurements of a given experiment setup as it determines warmup and effective measurement phases. These two settings make sure no experiment runs indefinitely or no idle resource is held. Moreover, a fixed number of measurements for some experiments may not be enough for the numbers to eventually stabilize or may be excessive thus resulting in unnecessary resource utilization. Although we have flexibility in expressing these settings, and that we have useful estimations for PGX operations, we believe that there needs to be proper online support in the Runtime module for when measurements need to stop. We also consider to grow the framework and include stress testing [9].

References

1. Yogesh Barve, Shashank Shekhar, Shweta Khare, Anirban Bhattacharjee, and Aniruddha Gokhale. Upsara: A model-driven approach for performance analysis of cloud-hosted applications. In *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, pages 1–10. IEEE, 2018.

2. Domenico Bianculli, Walter Binder, and Mauro Luigi Drago. Soabench: Performance evaluation of service-oriented middleware made easy. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 301–302. IEEE, 2010.
3. Vincenzo Ferme and Cesare Pautasso. Towards holistic continuous software performance assessment. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 159–164. ACM, 2017.
4. Tor-Morten Gr ø nli and Gheorghita Ghinea. Meeting quality standards for mobile application development in businesses: A framework for cross-platform testing. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5711–5720. IEEE, 2016.
5. Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-marl: a dsl for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News*, 40(1):349–362, 2012.
6. Sungpack Hong, Siegfried Depner, Thomas Manhardt, Jan Van Der Lugt, Merijn Verstraaten, and Hassan Chafi. Pgx. d: a fast distributed graph processing engine. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 58. ACM, 2015.
7. Lennart CL Kats, Rob Vermaas, and Eelco Visser. Integrated language definition testing: enabling test-driven language development. *ACM SIGPLAN Notices*, 46(10):139–154, 2011.
8. Lennart CL Kats and Eelco Visser. The spoofax language workbench: rules for declarative specification of languages and ides. In *ACM sigplan notices*, volume 45, pages 444–463. ACM, 2010.
9. Martin L Kersten, Alfons Kemper, Volker Markl, Anisoara Nica, Meikel Poess, and Kai-Uwe Sattler. Tractor pulling on data warehouses. In *Proceedings of the Fourth International Workshop on Testing Database Systems*, page 7. ACM, 2011.
10. Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.
11. John K Ousterhout. Scripting: Higher level programming for the 21st century. *Computer*, 31(3):23–30, 1998.
12. Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. Pgx. iso: parallel and efficient in-memory engine for subgraph isomorphism. In *Proceedings of Workshop on GRAph Data management Experiences and Systems*, pages 1–6. ACM, 2014.
13. Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.
14. Maicon Bernardino da Silveira et al. Canopus: a domain-specific language for modeling performance testing. 2016.
15. Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.
16. Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. Pgql: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 7. ACM, 2016.
17. Johannes Wienke, Dennis Wigand, Norman Koster, and Sebastian Wrede. Model-based performance testing for robotics software components. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 25–32. IEEE, 2018.