Software Performance Testing in Virtual Time

Extended Abstract

Tony Field
Imperial College London
London, United Kingdom
ajf@imperial.ac.uk

Robert Chatley
Imperial College London
London, United Kingdom
rbc@imperial.ac.uk

David Wei
Imperial College London
London, United Kingdom
dw512@imperial.ac.uk

ABSTRACT

We show how traditional unit testing frameworks can be extended to support the simultaneous testing of behaviour and performance, by embedding performance models in mock objects. Because such models are virtual, and therefore execute in virtual time, performance tests can often be performed substantially quicker than when real resources are involved. Performance models also facilitate testing before some or all of a unit's intended collaborators have been implemented. A key technical challenge is to overcome the impedance mismatch which arises when code that is executing in real time has to communicate with performance models that execute in virtual time. Solutions to this problem naturally facilitate virtual time scaling of both real code and performance models. We also explore potential applications of such time scaling in software performance testing and optimisation.

CCS CONCEPTS

• **Software and its engineering → Software performance**; **Software design techniques**; **Software testing and debugging**; *Agile software development*;

KEYWORDS

Performance, Test-Driven Development

ACM Reference Format:
Tony Field, Robert Chatley, and David Wei. 2018. Software Performance Testing in Virtual Time: Extended Abstract. In *Proceedings of ACM/SPEC International Conference on Performance Engineering (ICPE'18 Companion),* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 2 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Modern software development is now predominantly based around agile development methods [7] which encourage the continuous testing and adaptation of software throughout its evolution. Crucial to the philosophy is fast test-driven feedback, so that errors in design or functionality can be detected quickly and fixed early whenever problems arise.

ICPE'18 Companion, April 9–13, 2018, Berlin, Germany

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

This test-driven development (TDD) has revolutionised software engineering practices and made it easier to ensure that delivered software is functionally correct and meets the needs of its intended users.

However, a key attribute of a software system that is seldom addressed by conventional TDD is performance. Performance testing is typically undertaken late on in the development process, once the software can be integrated and tested as a whole, and often only after the software has been deployed. Furthermore, such testing is usually performed manually, i.e. without using any generic performance testing framework or tooling. A recent study [9] showed that less than 0.4% of over 90,000 open-source GitHub projects used any such framework, let alone one aligned with continuous delivery.

Identifying and resolving performance problems at such a late stage can be expensive, as it may involve redesigning parts of the system, rewriting code or allocating more computing resources to certain components to match requirements [8]. Performance tests are also often slow, or inconvenient, to run, as many components typically have to be tested together. This is at odds with the fast feedback loops associated with test-driven development.

This talk describes our recent work that aims to facilitate continuous performance testing by extending the well-established idea of using *mock objects* [6]. In unit testing frameworks mock objects are used to replace the collaborators of an object under test with alternative implementations that serve only to support the test. The mock objects can be configured to behave in particular ways to simulate different scenarios, and can also be used to verify that the expected messages are exchanged between the various collaborators in a given test scenario.

The idea here is to allow mock objects also to encapsulate *performance models*, whose job is to estimate the time taken for method calls to the mocked object to yield a response, or rather to estimate the response time of a call to a real component in the same scenario. For example, if we are developing a component that interacts with some service then we would like to estimate the response time of our new component, given appropriate assumptions about the performance characteristics of that service.

Crucially, because the approach is model-based, our performance tests execute in *virtual time*, so that performance estimates can be produced without having to wait for the passage of real (wall-clock) time. This leads to fast turnaround times, which is one of the key requirements of effective unit testing.

Performance models have been used extensively to model software systems and services and there are countless examples in the literature, some more recent examples being for NoSQL big data architectures [3], key-value data stores such as Apache Cassandra [5] and distributed data processing frameworks such as Spark [10].

In most performance modelling exercises the modeller is typically concerned with the model structure, its solution, validation and application to a specific question related to the system under study. Much less of a concern is how to put such models in the hands of everyday practitioners, be they software developers, DevOps practitioners, capacity planners, or similar. Our objective is to bridge the gap between these often disparate communities through tooling and to provide a practical realisation of the vision in [11] of early-cycle model-based performance prediction.

Although the idea of mocking performance, in addition to behaviour, is simple in principle, we have to be able to cope with a rich variety of test scenarios, some of which may lead to complex interactions between objects and models. For example, the unit under test may create threads and these may have to compete for shared (virtual) resources in mocked objects, e.g. virtual locks, queues, services etc. within a complex performance model, such as a queueing network. Our mock objects are able to model this contention.

Furthermore the idea requires real code, executing in real time, to interact with mock objects that contain embedded performance models operating in virtual time. This leads to an *impedance mismatch* between real and virtual time and considerable care is required to ensure that both real and mocked objects progress through time in a mutually consistent way.

The talk will describe how the impedance mismatch problem has been solved in our performance mocking framework by implementing threads built by a test object as co-routines – essentially a form of execution-driven simulation. This ensures that the threads under test cannot leap ahead of the collaborating mock object(s) in virtual time, and vice versa.

A generalisation of this is the concept of *virtual time execution* which seeks to align the execution of real and mocked objects by virtualising time within the operating system's, or VM's, scheduler, avoiding the need for co-routines. Once time has been virtualised it possible to perform virtual time scaling on *real* objects, similar to the time scaling that can straightforwardly be done with performance models in mocked objects. This essentially mimics the effect of speeding up or slowing down fragments of real code [2, 4].

In the context of performance-test-driven development virtual time scaling presents some intriguing opportunities for exploring performance problems that might be encountered either during unit testing or as part of some later integration exercise. For example, we can imagine a system test comprising arbitrary collections of real and mocked objects executing, and communicating, entirely in virtual time, much as a test object and its collaborators do in our unit testing framework. If the application as a whole violates some performance constraint then where is the best place to look in order to fix the problem? Virtual time scaling provides the opportunity to explore the effect that optimising some or all of the components will have on the overall performance, but without having to change any of the code. In principle, this can be done dynamically, e.g. by applying time scaling factors for long enough within a single execution for any performance impacts to be quantified.

It is important to remark that the focus of this work is not the models themselves, but rather how established unit testing frameworks can be augmented to incorporate such models through a single interface. The accuracy of any prediction is down to the quality of the model (garbage in, garbage out). If developers are to use off-the-shelf models then considerable care is needed to ensure that such models are fit for this purpose. Possibly a more likely scenario is one where developers build their own models from existing production system logs. In this case, some form of automation of the task, e.g. along the lines of [1], would appear to be essential, as developers typically have little or no expertise in performance modelling.

REFERENCES

[1] M. Awad and D. A. MenascÃĪ. 2016. Performance Model Derivation of Operational Systems through Log Analysis. In *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 159–168. https://doi.org/10.1109/MASCOTS.2016.41

[2] N. Baltas and A.J. Field. 2011. Software Performance Prediction with a Time Scaling Scheduling Profiler. In *IEEE 19th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2011)*. IEEE, 107–116.

[3] Enrico Barbierato, Marco Gribaudo, and Mauro Iacono. 2013. Performance Evaluation of NoSQL Big-Data applications using multi-formalism models. to appear (01 2013).

[4] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 184–197. https://doi.org/10.1145/2815400.2815409

[5] Salvatore Dipietro, Giuliano Casale, and Giuseppe Serazzi. 2017. A Queueing Network Model for Performance Prediction of Apache Cassandra. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'16)*. 186–193. https://doi.org/10.4108/eai.25-10-2016.2266606

[6] Tim Mackinnon, Steve Freeman, and Philip Craig. 2000. Endo-Testing: Unit Testing with Mock Objects. *eXtreme Programming and Flexible Processes in Software Engineering* (2000), 287–301.

[7] Efi Papatheocharous and Andreas S. Andreou. 2014. Empirical evidence and state of practice of software agile teams. *Journal of Software: Evolution and Process* 26, 9 (2014), 855–866. https://doi.org/10.1002/smr.1664 JSME-14-0067.

[8] Connie U. Smith. 1990. *Performance Engineering of Software Systems*. Addison-Wesley.

[9] Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. 2017. Unit Testing Performance in Java Projects: Are We There Yet?. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 401–412. https://doi.org/10.1145/3030207.3030226

[10] K. Wang and M. M. H. Khan. 2015. Performance Prediction for Apache Spark Platform. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 166–173. https://doi.org/10.1109/HPCC-CSS-ICESS.2015.246

[11] Murray Woodside, Greg Franks, and Dorina C. Petriu. 2007. The Future of Software Performance Engineering. In *2007 Future of Software Engineering (FOSE '07)*. IEEE Computer Society, Washington, DC, USA, 171–187. https://doi.org/10.1109/FOSE.2007.32