Continuous Performance Testing in Virtual Time

Robert Chatley
Department of Computing
180 Queen's Gate
London, SW7 2AZ, United Kingdom
rbc@imperial.ac.uk

Tony Field
Department of Computing
180 Queen's Gate
London, SW7 2AZ, United Kingdom
ajf@imperial.ac.uk

David Wei
Department of Computing
180 Queen's Gate
London, SW7 2AZ, United Kingdom
dw512@imperial.ac.uk

***Abstract*—We introduce the notion of performance unit testing which allows developers to explore performance characteristics and detect potential performance problems continuously throughout the development of a software system. Our ideas are embodied in *PerfMock*, which extends a well-established object mocking framework so that each mock object can be configured with a performance model for predicting the time taken to process each message it receives. *PerfMock* executes tests in *virtual* time. This allows performance to be evaluated much more quickly than running a full system performance test, making it possible to test performance continuously, as part of a unit test suite. We demonstrate the core features of *PerfMock* and show how it can be used to support a process of iterative refinement, whereby models can be improved when more about the actual performance of the objects being mocked becomes known, e.g. by building models from production data. We show that even very simple performance models used early on in the development process can provide useful information for estimating both absolute execution times and the effects of changes in functionality and/or design. The iterative approach we support has the pleasing property that as the system evolves, more decisions are made and more data is collected meaning that we can refine our models, and predicted and actual performance gradually converge.**

I. Introduction

The widespread adoption of agile methods [1] and continuous delivery [2] of software has resulted in development processes that are dependent on the use of rapid feedback from automated testing. In order to obtain fast feedback, developers often concentrate on testing small units in isolation, typically with pure unit tests [3]. This sort of testing has proved valuable in practice, but it cannot tell the developer everything. For example, unit tests do not indicate whether the system works as a whole to achieve a business goal, nor are they capable of evaluating the user experience. Of particular interest to this paper is the fact that they do not address performance.

Performance testing is typically done later on in development once a complete version of the software system can be deployed, instead of a primary concern that drives the software development process [4]. Performance issues are therefore usually not exposed until the system is integrated and tested as a whole. Resolving performance problems at this 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 [5]. Performance tests are also typically slow or inconvenient to run, which is at odds with the fast feedback loops associated with test-driven development (TDD), i.e. the 'red, green, refactor' [6] loop, gaining confidence of correctness after every change.

The objective of this paper is to extend existing TDD techniques so that performance-related properties can be continuously verified throughout the software development process. For example, we may wish to establish that a class A will meet its required performance characteristics given that its collaborator B has a performance profile that matches X. The key idea is to capture X, using a *performance model*.

Our approach builds on the well-established idea of using *mock objects* to conduct unit testing in isolation [7]. Mock objects are used to replace the real collaborators of an object under test with implementations that serve only to support the test. 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. We extend this technique to allow mock objects to be configured with embedded performance models that are responsible for predicting the time that the object being mocked will take to respond to a message. This enables *performance unit tests* to be written that make assertions about performance as well as behaviour.

A performance model is any piece of code that is capable of estimating a time delay, e.g. by straightforward distribution sampling, the solution of a mathematical model such as a Markov Process or product-form queueing network, or by running a discrete-event simulation alongside the unit under test. The choice of model is up to the developer and our approach supports arbitrary models types, including the above. Whatever model is chosen from the beginning, the intention is that it can be refined iteratively as more becomes known about the actual behaviour of the object being modelled, e.g. through ongoing integration testing or data from production deployment – see Section III-A.

A key point is that performance models work entirely in *virtual* time, which means that performance estimates can be produced without having to wait for the passage of real time. This leads to fast turnaround times, which is one of the key requirements of effective automated testing, and enables large suites of performance tests to be included in a pre-commit build or a continuous delivery pipeline, without significantly increasing the build time.

The ability to do early-stage continuous performance testing is a realisation of the software performance engineer-

ing methodology articulated in [5] and [8]. This promotes model-based performance testing from the inception of a project through to its completion and, in particular, enables performance aspects of a system to be explored before key components and any necessary supporting infrastructure have been developed or acquired. The key difference we explore here is the idea of executing code and models side-by-side.

The main contributions of the paper are as follows:

- We introduce the concept of performance unit testing, and demonstrate our framework **PerfMock** that extends a well-established mocking framework for Java, *jMock2*, to support such tests (Section III).
- We demonstrate the continuous performance testing development method through a progressive case study, using an example web application (Sections IV-V). The focus here is on the method itself, rather than on the construction of high-fidelity performance models, which is a discipline within its own right.
- We show that performance unit testing yields fast turnaround times and provides useful guidance to developers, even when the performance of the collaborator being mocked is not precisely known (Section VI).

II. Test-driven development using mock objects

Unit tests focus on testing the behaviour of one object (a single unit) and its interactions with its neighbouring collaborators [6]. For example, if we are testing an `EmailClient` object, and tell it to send an email, we may well want to test that a collaborating `EmailServer` object receives a message. The test would need to detect any outward flowing messages, which can be done by replacing collaborators that would receive messages with a test "double", also known as a *mock object* [3]. Test doubles are special implementations that are used in place of real collaborators during tests [7]. Mock objects can be configured with *expectations* of the messages they should receive during a test scenario, and also return canned values if necessary. The mocks record the messages that they actually receive and the test passes if these match the prescribed expectations. While it is simple enough to code this sort of test infrastructure by hand, by using a mock object framework we can easily generate and manage mock objects that implement any given interface. In this paper we will show examples using jMock2 [9], but other comparable libraries are available in many languages and follow similar ideas.

Mock objects are often thought of in the context of isolating tests from dependencies [10], allowing business logic to be tested without relying on real external services and even before such services have been implemented. It has been argued that, in addition to supporting early testing, this can also lead to better design [11], [12].

A. Example

To demonstrate performance unit testing, we apply PerfMock to the development of a micro-blogging web application using a Model-View-Controller pattern [13] called Tweeter. In this section we summarise how traditional unit testing might be applied to test one component of this application.

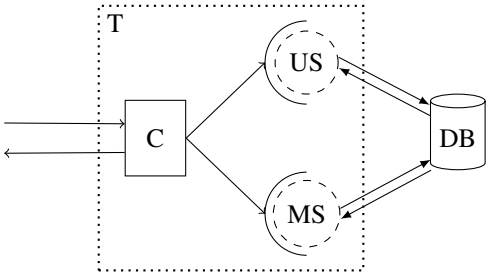

Fig. 1. Diagram showing the architecture of Tweeter, a Model-View-Controller (MVC) web application. C = controller, US = UserService, MS = MessageService, DB = database, T = unit test. The arcs denote interfaces that are implemented by a particular concrete class.

```
1  public class TweeterControllerTest {
2    @Rule
3    public JUnitRuleMockery ctx = new JUnitRuleMockery();
4    UserService users = ctx.mock(UserService.class);
5    MessageService msgs = ctx.mock(MessageService.class);
6
7    @Test
8    public void rendersUserTimeline() {
9      TweeterController ctlr = new TweeterController(...);
10     User alice = new User("Alice");
11
12     ctx.checking(new Expectations() {{
13       exactly(1).of(users).getByUsername("Alice");
14           will(returnValue(alice));
15       exactly(1).of(msgs).getUserTimeline(alice);
16     }});
17
18     ctlr.userTimeline("Alice", new ModelMap());
19   }
20 }
```

Fig. 2. A unit test for the controller method `userTimeline`, which does not exist at this point. This example is written using JUnit 4 as the test framework and jMock2 as the mock object framework.

Figure 1 shows an overview of its architecture, where a controller (C) handles a request from a client by using two collaborators `UserService` (US) and `MessageService` (MS), which both interact with a database (DB). The controller then prepares a model containing the data needed by the view to render the response, and returns it to the client.

Figure 2 shows a possible unit test (T) for the controller method, using jMock2/JUnit 4 syntax. In TDD we write the test first, i.e. before the implementation, so we use mocks of `UserService` and `MessageService` (seen as dashed circles in Figure 1) and assert that the correct interactions occur between them and the controller in the test scenario.

The test class manages mock objects and their expectations via the jMock2 context object, `JUnitRuleMockery` (line 3). We pass in the mocks when creating an instance of `TweeterController` (line 9) and declare expectations of the methods that should be invoked on the mock objects by our new controller method (lines 12-16). In this example, we expect there to be one call to `getByUsername` in `UserService` and one call to `getUserTimeline` in `MessageService`. We may also configure these methods to

returned canned values when invoked (line 14).

```
1  @Controller
2  public class TweeterController {
3    private final UserService users;
4    private final MessageService msgs;
5
6    public TweeterController(UserService users,
7                             MessageService msgs) {
8      this.users = users; this.msgs = msgs;
9    }
10
11   @RequestMapping("/u/{username}")
12   public String userTimeline(
13       @PathVariable String username, Model model) {
14     User userProfile = users.getByUsername(username);
15     model.addAttribute("username", username);
16     List<Message> userMsgs =
17         msgs.getUserTimeline(userProfile);
18     model.addAttribute("messages", userMsgs);
19     return "timeline";
20   }
21 }
```

Fig. 3. A basic implementation of the controller method `userTimeline` that will pass the unit test in Figure 2.

Assuming a skeleton method for `userTimeline` such that the test compiles, we can run the test, but it will fail because there is not yet an implementation of this method. The next step is to write a minimal implementation needed to make the test pass, an example of which is shown in Figure 3. We may then refactor the code and repeat the cycle for the next requirement. This red (write a failing test), green (make it pass) and refactor cycle forms the core of the TDD process.

III. Performance unit tests

In order to integrate performance testing with continuous development processes that emphasise tight feedback loops, we introduce the concept of *performance unit tests*. Where a unit test checks the behaviour of a single object under test, a performance unit test additionally checks its performance characteristics, given suitable models for the performance of collaborators. Our framework, PerfMock, supports such tests.

Performance can be characterised by different metrics such as throughput, response time and scalability with respect to varying workloads. At the unit testing level, response time is the most relevant, since it can be measured directly by micro-benchmarking code [14]. However, the execution time of the mock object itself will be very different from that of the object it represents because a mock contains minimal internal logic. By embedding a performance model within the mock object, we have a way of estimating the actual execution time of the real object whose behaviour is being mocked.

The key idea behind performance unit tests is that they help developers reason about the performance impact of changes as they are being made. Dependencies of the object under test may not yet exist, so these tests can help determine how they must perform in order for the application overall to perform within requirements. Unlike traditional performance testing, these tests do not require the application to be deployed, and integration with unit testing frameworks enable them to be executed early and often, even before code is committed.

A. Performance models

The nature of performance models used in performance unit tests may vary substantially, depending on an application's current stage of development. When not much is decided or known about the final implementation of the object or system being mocked, a model might be as simple as an aggregate expected delay (a *deterministic* model) or a sample from a probability distribution that is believed, or known, to represent well the spread of time delays over a large number of requests. Such estimates may represent prescribed 'budgets' set during early stages of the design, or be based on 'received wisdom' based on prior experience or observations from similar software systems.

More sophisticated models that are capable of describing the internal structure of an object or service, including the interactions between its subcomponents, can equally be used. However, their effectiveness depends on finding parametrisations that reflect the hardware and workload imposed by the object under test, including other background load that will affect the mocked object's response time [15]. Examples include models of big-data systems such as Apache Hadoop [16], key-value data stores e.g. Apache Cassandra [17] and distributed data processing frameworks e.g. Apache Spark [18].

PerfMock supports all of the above model types, including complex discrete-event simulations which model explicitly the passage of (virtual) time triggered by events internal to some structural model. It achieves this by managing an explicit virtual time line which is used to ensure that 'real' events initiated by the unit under test, such as the sending of a message to a collaborator, and 'virtual' events, such as the end of a random time delay, are all correctly ordered with respect to a global virtual clock. In this respect PerfMock essentially performs an *execution-driven* simulation [19].

Later, once the system is deployed in some environment that is representative of production (or production itself), we have the opportunity to measure the observed behaviour of objects whose performance might previously have been estimated or modelled based on assumptions. The idea is to use such measurements to build better performance models and/or provide better parameterisations of existing models. We envisage this type of model refinement to be a continuous process throughout the lifetime of a software system, enabling predicted response times from models to converge to actual performance measured in a production environment.

IV. Using performance unit tests

We now demonstrate how to apply continuous performance testing by using the Tweeter application introduced in Section II. The focus of this paper is on the model-based approach to performance testing, rather than on the models themselves. For this reason we illustrate the application of PerfMock assuming relatively simple performance models based on probability distribution sampling, rather than Markov Processes or discrete-event simulations. In practice we anticipate that simple models will be favoured by developers anyway, as they require no specialist modelling expertise.

We start by applying a simple transformation to the existing unit test in Figure 2, with the result shown in Figure 4. The original `JUnitRuleMockery` has been replaced with `PerformanceMockery` (lines 3-4). The `mock` method for creating mock objects now accepts a performance model that describes the performance characteristics of the service being mocked as a second parameter (lines 5-8).

Practically, models are stored in a `PerformanceModels` class accessed via static methods, as illustrated in lines 6 and 8 in Figure 4. When it is time to implement these objects/services, it is often useful to compare the models used upstream, as here, with the measured performance of a downstream implementation; we return to this in Section V.

In this example, we initially use exponential distributions to model the performance of the two services, but any model implementing a `PerformanceModel` interface can be used. The mean delays in this case are both set to 1.5 ms, which might correspond to a given time budget or a best guess of the ultimate mean response time of the two services.

The stochastic nature of distribution sampling means that it is not sufficient to run a performance unit test just once. PerfMock therefore allows a *test kernel* to be executed repeatedly as a single performance *experiment* (lines 15-22) in order to determine point estimates and confidence intervals for the mean and percentiles of the response time distribution. Running tests in virtual time means that even with thousands of trials, the test suite still only takes a fraction of a second to run in real elapsed time.

To determine performance requirements, developers will typically use overall systems requirements, e.g. service level agreements (SLAs), to allocate budgets for individual components of the system. For example, if we want a page to be fully rendered within 100 ms, we might allocate a budget of 40 ms to one component, and split the remaining 60 ms amongst those to be developed later. PerfMock supports various types of performance assertion. In our example we specify that the 80th percentile predicted response time for the controller should be under 15 ms (lines 24-25).

With a `TweeterController` implementation already in place, as in Figure 3, the mean estimated execution time is 3.01 ± 0.01 ms and the 80th percentile is estimated to be 4.51 ± 0.02 ms, which is well within our requirements. The test therefore passes.

A. Adding new features

Let us now suppose that the next stage of development requires implementing the ability to *reply* to messages. This means that on a user's timeline, we will need to render all the replies to a given message.

As before, we start by writing a performance unit test for this new behaviour, which is shown in Figure 5 and this requires a new method, `getReplies` to be added to the existing `MessageService`. We configure Alice's timeline to have 10 messages, and therefore expect 10 calls to `getReplies` (lines 14-15). The overall performance target is unchanged, so we maintain the assertion from the previous test. We then implement the feature in the controller method `userTimeline`, shown in Figure 6. For each message on the timeline, we make a call to `getReplies` to retrieve a list of replies and add them to the message (lines 11-14).

```
1  public class TweeterControllerTest {
2    @Rule
3    public PerformanceMockery ctx =
4        new PerformanceMockery();
5    UserService users = ctx.mock(UserService.class,
6        PerformanceModels.userServiceModel());
7    MessageService msgs = ctx.mock(MessageService.class,
8        PerformanceModels.messageServiceModel());
9
10   @Test
11   public void rendersUserTimeline() {
12     TweeterController ctlr = new TweeterController(...);
13     User alice = new User("Alice");
14
15     ctx.repeat(2000, () -> {
16       ctx.checking(new Expectations() {{
17         exactly(1).of(users).getByUsername("Alice");
18             will(returnValue(alice));
19         exactly(1).of(msgs).getUserTimeline(alice);
20       }});
21       ctlr.userTimeline("Alice", new ModelMap());
22     });
23
24     assertThat(ctx.runtimes(),
25         hasPercentile(80, lessThan(15.0)));
26   }
27 }
```

Fig. 4. A performance unit test for the controller method `userTimeline`, written using JUnit 4 and PerfMock. The test kernel calling `userTimeline` is repeated 2,000 times and a performance assertion is made against the aggregated results.

```
1  public class TweeterControllerTest {
2    @Test
3    public void rendersUserTimelineWithReplies() {
4      TweeterController ctlr = new TweeterController(...);
5      User alice = new User("Alice");
6      List<Message> TEN_MSGS = ...
7
8      ctx.repeat(2000, () -> {
9        ctx.checking(new Expectations() {{
10         exactly(1).of(users).getByUsername("Alice");
11             will(returnValue(alice));
12         exactly(1).of(msgs).getUserTimeline(alice);
13             will(returnValue(TEN_MSGS));
14         exactly(10).of(msgs).getReplies(
15             with(any(Message.class)));
16       }});
17       ctlr.userTimeline("Alice", new ModelMap());
18     });
19
20     assertThat(ctx.runtimes(),
21         hasPercentile(80, lessThan(15.0)));
22   }
23 }
```

Fig. 5. A performance unit test for adding the replies feature to the controller method `userTimeline`.

At any point in development a performance unit test may fail, and that is what happens in this example:

```
java.lang.AssertionError: Expected: percentile
80 to be a value less than <15.0> but:
<22.172782855636996> (95% CI +/- 0.04)
```

When a performance test fails there are three different approaches we can take to fix it:

```
1  @Controller
2  public class TweeterController {
3    @RequestMapping("/u/{username}")
4    public String userTimeline(
5        @PathVariable String username, Model model) {
6      User profileUser = users.getByUsername(username);
7      model.addAttribute("profileUserName", username);
8      List<Message> userMsgs =
9          msgs.getUserTimeline(profileUser);
10     model.addAttribute("messages", userMsgs);
11     for (Message m : userMsgs) {
12       List<Reply> replies = msgs.getReplies(m);
13       m.addReplies(replies);
14     }
15     return "timeline";
16   }
17 }
```

Fig. 6. An implementation of the controller method `userTimeline` with the replies feature, fetching and attaching the list of replies for each message.

1) Relax the performance assertions: The performance assertions themselves can be adjusted, and whether this is acceptable or not depends on how they were defined in the first place, for example whether or not they are not derived from hard requirements such as SLAs. For example, we could choose to change the assertion (lines 24-25) from 15 ms to 25 ms, although this may have a knock-on effect for a time budget elsewhere in the application in order that any overall SLAs can continue to be met.

2) Adjust the models: The parameters of a performance model can be adjusted, which means changing how a collaborator needs to perform in order to meet the performance requirements of the object under test. For example, in the Tweeter application both `UserService` and `MessageService` rely on a database, so adjusting their model parameters downwards, e.g. from 1.5 (mean response time) to 1.0, will require correspondingly increased performance from whatever database system is ultimately chosen.

3) Optimise the object under test: The object under test itself can be optimised, e.g. to use better algorithms, better data structures, make fewer calls to collaborators, or if possible, make these calls in parallel.

Before moving on let us make the assumption that we address the failed test above by relaxing the performance assertion, as suggested above.

V. Moving the lens

Once the implementation of `TweeterController` is complete, we might next focus on the downstream `UserService` and `MessageService`, using the same TDD process. We first choose a database implementation, which for the purposes of the case study, will be taken to be Apache Cassandra. We then select a behaviour to implement, and begin with `getByUsername` which is used in the controller method `userTimeline` (Figure 3). The code for this `CassandraUserService` is not shown.

We start by writing a performance unit test for this behaviour. The Spring framework conveniently provides a `CassandraOperations` interface, with typical create/read/update/delete operations. This enables us to build a mock of the Cassandra database in the performance unit test, as shown in Figure 8. The model used in the mock object defines the estimated time taken to access the database itself, which we assume to be circa 80% of the overall time assumed by the `userServiceModel` in Figure 4. The `cassandraOpsModel` is therefore set to be exponential with mean 1.2 ms on average, i.e. 80% of the 1.5 ms assumed earlier. The rest of the time (a *real* time) is assumed to come from the `UserService` itself, which is now the unit under test – see Figure 7.

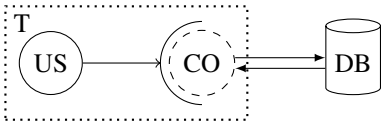

Fig. 7. Diagram showing the lens in the form of a performance unit test (T) being moved onto UserService (US). CassandraOperations (CO) is being mocked and represents the database (DB).

```
1  public class CassandraUserServiceTest {
2    @Rule
3    public PerformanceMockery ctx =
4        new PerformanceMockery();
5    CassandraOperations db = context.mock(
6        CassandraOperations.class,
7        PerformanceModels.cassandraOpsModel());
8
9    @Test
10   public void getUserAliceFromCassandra() {
11     UserService users = new CassandraUserService(db);
12
13     ctx.repeat(2000, () -> {
14       ctx.checking(new Expectations() {{
15         exactly(1).of(db).selectOne(
16           "SELECT * FROM users WHERE username='Alice';",
17           User.class);
18       }});
19       users.getByUsername("Alice");
20     });
21
22     assertThat(ctx.runtimes(), matchMean(
23       PerformanceModels.userServiceModel()));
24   }
25 }
```

Fig. 8. A performance unit test for the method `getByUsername` in `CassandraUserService`.

The performance unit tests for `TweeterController` passed given certain assumptions about the performance characteristics of `UserService` (Section IV). The next question is how well the Cassandra `UserService` performance matches the distribution that was previously assumed, i.e. exponential with mean 1.5 ms. PerfMock provides methods, `matchMean` and `matchDistribution` that respectively check whether the mean and distribution of the measured response time distribution of an implemented downstream service match that of the distribution assumed in an upstream test. In Figure 8 we work with the means, as we are currently only concerned with modelling aggregate time budgets. In this case the `matchMean` method uses a one-sample t statistic, as the `UserServiceModel` has a known mean of 1.5 ms; appropriate overloading will ensure that a two-sample test will be used if both are based on empirical data.

With the performance unit test in place, we implement `getByUsername` which makes the CQL queries to Cassandra and the test passes, which suggests that the assumed 80%:20% split above might be a reasonable working assumption. At this point in the development we might now repeat the process for the other required behaviours in `UserService` and `MessageService`. For the purposes of the paper, we omit the details.

A. Model refinement

Having selected a database implementation, we now seek to use empirical measurements to refine the distribution models used in Figure 8, specifically `cassandraOpsModel`. PerfMock provides various additional utilities, one of which takes a log file of measured response times and generates a Java class that uses inverse transform sampling to compute samples from the ecdf of those times. This process requires almost no extra work if the system being modelled already has logs from production, and is straightforward for developers to use and understand.

As we have chosen Cassandra for the case study, we use Cassandra's built-in load testing tool, `cassandra-stress`, to generate the data needed to create an empirical model. To do this we set up a test Cassandra database, with a similar configuration to the one intended for production, and running on similar hardware. We then populate the database with randomly generated data that is representative of expected real data and configure expected workloads by specifying a mix of representative queries. The number of concurrent clients can be changed to model different utilisation levels.

Replacing the `cassandraOpsModel` in Figure 8 (exponential with mean 1.2 ms) with the resultant empirical model in this case causes the performance assertion to fail (lines 22-23). Cassandra is actually a bit slower than we assumed. In order to make the test pass we can update `PerformanceModels.userServiceModel` to have a larger mean, and as a consequence, this change would propagate to other tests using the same model, and might cause these tests to fail, which is useful information, allowing us to fix performance problems early. Even if the test passed, we might take the opportunity to refine `userServiceModel` based on our latest data, because in general we always want to be using the most up-to-date model. The assumption here is that refined models will have higher fidelity than the models they replace, as more will be known about a system's performance as the software evolves.

In this example we could build such a model from the set of `ctx.runtimes()` in Figure 8. This would be an empirical model that includes the measured (real) time for the user service and the modelled (virtual) time for Cassandra. Alternatively, we might fit the same data to a mathematical distribution – it turns out in this case that the data fits well to a log-normal distribution. This is not, perhaps, surprising, as many network services exhibit a heavy tail [20].

The updating of performance models continuously, in order to reflect the most up-to-date observations of downstream performance, is an important aspect of the performance unit testing approach.

VI. Discussion

In this section we evaluate the prediction accuracy and the turnaround times of performance unit tests in the context of the Tweeter application.[1] In order to do this we built, deployed and measured the application running on a VM instance on a private Apache CloudStack cloud, provisioned with 2x 2.0 GHz CPU and 4 GB RAM. The Cassandra database was also deployed on the same platform, configured with four nodes, each provisioned with 2x 2.0 GHz CPU, 8 GB RAM, and hard disk storage.

A. Turnaround times

We compared the amount of (wallclock) time it took to run a performance unit test in PerfMock and a performance test against a real environment, each testing the same *scenario*, with results shown in Table I. Fig 4 is a scenario that renders the timeline for a specific user. Fig 5 is similar, except replies to messages are also rendered. T_{unit} denotes the mean time taken to execute 2,000 performance unit tests, which were run 100 times on a typical developer machine using Gradle as the test runner and profiler. To replicate the two scenarios in a real environment, we populated the Cassandra database with synthetic data that matched the performance unit tests; namely, each user had ten messages and each message had one reply. T_{actual} specifies the mean time taken to start up Tweeter and issue 2,000 requests to it using Apache Benchmark as a load generator, computed using ten identical runs. The results show that running a performance unit test is substantially faster than the equivalent full-system performance test, even for simple application like Tweeter, and are fast enough (under one second) for them to be useful in a TDD environment.

TABLE I
Mean wallclock time taken to execute two test scenarios, one in PerfMock and one against a real environment. 95% confidence intervals are shown in brackets.

Test	T_{unit} / s	T_{actual} / s
Figure 4	0.261 (±0.011)	29.85 (±0.61)
Figure 5	0.644 (±0.022)	50.09 (±0.49)

B. Prediction accuracy

We compared the predicted response times of each scenario against their measured values, with results shown in Table II. T_p^{exp} denotes the mean predicted response time using initial exponential distribution models for the two services. T_p^{emp} is similar, except using log-normal distributions for the two services, as detailed in Section V-A. T_m shows the measured response times of each scenario via performance interceptors placed around the two iterations of the controller method `userTimeline`.

[1] A replication package for these performance experiments is available online at: https://www.github.com/spikeh/perfmock-cse-qudos-19

TABLE II
MEAN PREDICTED AND MEASURED RESPONSE TIMES FOR TEST SCENARIOS INVOLVING TWO ITERATIONS OF THE CONTROLLER. 95% CONFIDENCE INTERVALS ARE SHOWN IN BRACKETS.

Test	T_m / ms	T_p^{exp} / ms	T_p^{emp} / ms	T_p^{imp} / ms
Fig 4	3.15	3.01 ($\pm$0.01)	1.48 ($\pm$0.001)	3.08 ($\pm$0.05)
Fig 5	13.37	18.04 ($\pm$0.02)	8.98 ($\pm$0.006)	13.56 ($\pm$0.07)

T_p^{exp} have relative errors of 4.44% and 34.92% against measured values T_m for the two scenarios respectively. These models were based on initial intuition, so it is unsurprising if their predictions do not quite match measured values. T_p^{emp} exhibits relative errors of 53.02% and 32.83% respectively, which are larger than expected given these predictions were based on empirical models built from stress testing a real Cassandra database. This is due to the `cassandra-stress` tool, which submits all requests at the start of a stress test and only measures *service time* on the Cassandra side, as opposed to response time. In addition, the driver on the Java side does non-negligible work marshalling objects, especially when fetching many rows for messages.

Having deployed the system we now have the option to refine the `userServiceModel` further by measuring the actual performance of the Cassandra `UserService`. This gives T_p^{imp} in Table II, with relative errors of 2.22% and 1.42% respectively. While empirical models from `cassandra-stress` could be built from just a test Cassandra deployment and an idea of the schema, the most accurate improved empirical models needed to measure a real deployment; this is the classic trade-off between accuracy and modelling detail. As these models evolve over time we would expect predicted and actual performance to converge, which we have demonstrated here.

C. Relative performance predictions

With performance unit tests in place, we also look at using them to predict *relative* differences before and after a code change. That is, whether performance unit tests can correctly predict the percentage slowdown or speedup of a code change. T_p^{emp} predicts a 6.07x slowdown when implementing the replies feature, whereas in reality performance slowed by 4.24x, which is a relative error of 43.0%. This is perhaps not surprising given the limitations of the Cassandra stress testing tool. For T_p^{imp} the predicted performance slowdown is 4.40x, with a relative error of 3.73%. Even if models are suboptimal, as they are likely to be early in development, we would nonetheless expect predicted performance changes that arise from implementing new features to be useful. Also, as we have seen, these errors can be expected to reduce as development of the system and refinement of the models proceed iteratively.

VII. FUTURE WORK

We are now in the process of evaluating our approach using PerfMock as part of the development of a substantial software application, from its inception to its completion. As part of this we would like to explore the role of more complex performance models, e.g. based on discrete-event simulation, in addition to the relatively straightforward models we have used here. PerfMock supports such models though its own internal event-driven execution engine and it will be interesting to evaluate how effective they are when used in performance unit tests of real software systems.

REFERENCES

[1] E. Papatheocharous and A. S. Andreou, "Empirical evidence and state of practice of software agile teams," *Journal of Software: Evolution and Process*, vol. 26, no. 9, pp. 855–866, 2014.

[2] J. Humble and D. Farley, *Continuous Delivery*. Addison-Wesley Professional, 2010.

[3] G. Meszaros, *xUnit Test Patterns*. Pearson Education, 2011.

[4] L. Chen, "Continuous Delivery: Huge benefits, but challenges too," *IEEE Software*, vol. 32, no. 2, pp. 50–54, 2015.

[5] C. U. Smith, *Performance Engineering of Software Systems*. Addison-Wesley, 1990.

[6] K. Beck, *Test Driven Development. By Example*. Addison-Wesley Professional, 2003.

[7] T. Mackinnon, S. Freeman, and P. Craig, "Endo-Testing: Unit testing with mock objects," *Extreme Programming Examined*, pp. 287–301, 2000.

[8] M. Woodside, G. Franks, and D. C. Petriu, "The future of software performance engineering," in *Future of Software Engineering (FOSE '07)*. IEEE, 2007, pp. 171–187.

[9] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "jMock: Supporting responsibility-based design with mock objects," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*. ACM, 2004, pp. 4–5.

[10] M. Feathers, *Working Effectively with Legacy Code*. Pearson Education, 2004.

[11] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock roles, not objects," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*. ACM, 2004, pp. 236–246.

[12] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*. Pearson Education, 2009.

[13] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 2013.

[14] V. Horký, P. Libič, A. Steinhauser, and P. Tůma, "DOs and DON'ts of conducting performance measurements in java," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE '15)*. ACM, 2015, pp. 337–340.

[15] M. Awad and D. A. Menasce, "Deriving parameters for open and closed QN models of operational systems through black box optimization," in *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE '17)*. ACM, 2017, pp. 127–138.

[16] E. Barbierato, M. Gribaudo, and M. Iacono, "Performance evaluation of NoSQL big-data applications using multi-formalism models," *Future Generation Computer Systems*, vol. 37, pp. 345–353, 2014.

[17] S. Dipietro, G. Casale, and G. Serazzi, "A queueing network model for performance prediction of Apache Cassandra," in *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS '16)*. ACM, 2017, pp. 186–193.

[18] K. Wang and M. M. H. Khan, "Performance prediction for apache spark platform," in *Proceedings of the 17th International Conference on High Performance Computing and Communications, 7th International Symposium on Cyberspace Safety and Security, and 12th International Conference on Embedded Software and Systems (HPCC-CSS-ICESS '15)*. IEEE, 2015, pp. 166–173.

[19] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair, "The rice parallel processing testbed," *SIGMETRICS Perform. Eval. Rev.*, vol. 16, no. 1, pp. 4–11, May 1988. [Online]. Available: http://doi.acm.org/10.1145/1007771.55596

[20] D. Ersoz, M. S. Yousif, and C. R. Das, "Characterizing network traffic in a cluster-based, multi-tier data center," in *Proceedings of the 27th International Conference on Distributed Computing Systems (ICDCS '07)*. IEEE, 2007.