Performance Awareness in Agile Software Development

Vojtěch Horký

Department of Distributed and Dependable Systems, Charles University
Context & Motivation
Context: Agile Software Development

Code is frequently built, tested and shipped.

Shorter release cycles, build pipeline automation.
Context: Agile Software Development

Code is frequently built, tested and shipped.

Shorter release cycles, build pipeline automation.

Functionality of the software is tested on many levels (unit, system, integration, ...).
Context: Agile Software Development

Code is frequently built, tested and shipped.

Shorter release cycles, build pipeline automation.

*Functionality* of the software is tested on many levels (unit, system, integration, ...).

What about *performance testing*?
Performance Testing in Software Development

Seems that performance testing is not happening at all...
Performance Testing in Software Development

Seems that performance testing is not happening at all...

General public would definitely agree that this is true for enterprise systems... (HealthCare.gov, ABS Census website, Czech Vehicle Registration)
Performance Testing in Software Development

Seems that performance testing is not happening at all...

General public would definitely agree that this is true for enterprise systems... (HealthCare.gov, ABS Census website, Czech Vehicle Registration)

Open source is not much better. One third of Java projects on GitHub has functional unit tests. Only 0.2% has some kind of a performance test.

Stefan, Horký, Bulej, and Tůma. “Unit Testing Performance in Java Projects: Are We There Yet?” (ICPE '17).
Why There Are No Performance Tests?

```java
int add(int a, int b) {
    return a + b;
}
```
int add(int a, int b) {
    return a + b;
}

assertEquals(10, add(7, 3))
assertEquals(99999, add(12345, 87654))

All is nice for functional tests...
int add(int a, int b) {
    sleep(a); return a + b;
}

Why There Are No Performance Tests?
int add(int a, int b) {
    sleep(a); return a + b;
}

assertEquals(10, add(7, 3))
assertEquals(99999, add(12345, 87654))

All is still nice for functional tests...
Why There Are No Performance Tests?

int add(int a, int b) {
    sleep(a); return a + b;
}

assertEquals(10, add(7, 3))
assertEquals(99999, add(12345, 87654)

All is still nice for functional tests...

But what about performance?
Challenges

How to express performance assumption?
   “Fast enough” is simply not good enough.

How to write (performance) tests?
   Functional tests are often stateful and focus on corner cases.

How to document performance?
   Big-O is useful but we never sort infinitely large arrays.

How to compare measured data?
   Distinguishing noise from real change is hard.

Can we measure software in production environment?
   Overhead prevents us to measure everything.
Goal of the Thesis

Make developers aware of the performance during all phases of development lifecycle.
Selected Contributions

Performance testing
Change detection
Runtime monitoring
Expressing (Unit) Test Assumptions

\[ \text{encrypt}_{\text{key}=17}^{\text{data}=[0, 0, 0]} = 42 \]
Expressing (Unit) Test Assumptions

\[ \text{encrypt}_{\text{key}=17}^{\text{data}=[0, 0, 0]} = 42 \]

```java
test void encryptionWorks() {
    assertThat(encrypt({0, 0, 0}, 17), is(42))
}
```
Expressing (Unit) Test Assumptions

encrypt_{data=[0, 0, 0], key=17} = 42

```java
@Test void encryptionWorks() {
    assertThat(encrypt({0, 0, 0}, 17), is(42))
}
```

We want the same simplicity for performance too . . .

. . . to test performance of the building blocks . . .

. . . and to build from performance tested components.
Expressing Test Assumptions

\[ \forall n \in \{1024, 2048, 4096\} : \]
\[ \text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n) \]
Expressing Test Assumptions

\[ \forall n \in \{1024, 2048, 4096\} : \]

\[ \text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n) \]

\[ \leq \text{ – intuitive meaning of “faster than”} \]

encrypt, memcpy – compare with baseline operation

randomBytes – workload generator (test input data)

1024, 2048, 4096 – typical workload sizes in our application
Expressing Test Assumptions

\[ \forall n \in \{1024, 2048, 4096\} : \]

\[ \text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n) \]

\[ \leq \text{ – intuitive meaning of “faster than”} \]

\text{encrypt, memcpy} \text{ – compare with baseline operation}

\text{randomBytes} \text{ – workload generator (test input data)}

1024, 2048, 4096 \text{ – typical workload sizes in our application}
Expressing Test Assumptions

\[ \forall n \in \{1024, 2048, 4096\} : \]
\[ \text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n) \]

\( \leq \) – intuitive meaning of “faster than”

encrypt, memcpy – compare with baseline operation

randomBytes – workload generator (test input data)

1024, 2048, 4096 – typical workload sizes in our application
Expressing Test Assumptions

\[ \forall n \in \{1024, 2048, 4096\} : \]

\[ \text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n) \]

\[ \leq - \text{intuitive meaning of “faster than”} \]

encrypt, memcpy – compare with baseline operation

randomBytes – workload generator (test input data)

1024, 2048, 4096 – typical workload sizes in our application
Code Example

```java
@PerformanceTest("for n in { 1024, 2048, 4096 }: encrypt[randomBytes](n) <= 10 * builtin.memcopy[randomBytes](n)")
byte[] encrypt(byte[] data, int key) {...}

/** Generate random byte array with n elements. */
Parameters<byte[], int> randomBytes(int n) {...}
```
@PerformanceTest("  
for n in \{ 1024, 2048, 4096 \}:  
    encrypt[randomBytes](n) <=  
        10 * builtin.memcopy[randomBytes](n)  
")  
byte[] encrypt(byte[] data, int key) {...}

/** Generate random byte array with n elements. */  
Parameters<byte[], int> randomBytes(int n) {...}
Code Example

```java
@PerformanceTest("for n in { 1024, 2048, 4096 }: encrypt[randomBytes](n) <= 10 * builtin.memcopy[randomBytes](n)")
byte[] encrypt(byte[] data, int key) {...}

/** Generate random byte array with n elements. */ Parameters<byte[], int> randomBytes(int n) { ... }
```
Toolchain for Performance Unit Testing

For Java: automated execution, JUnit-style HTML report, measurement on a remote machine, support for Git and Subversion, Eclipse plugin, Hudson plugin.

Evaluation summary

<table>
<thead>
<tr>
<th></th>
<th>✓ Satisfied</th>
<th>☢ Failed</th>
<th>🌍 Undecidable</th>
<th>⚠ Not parsed</th>
<th>⋄ All</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formulas</td>
<td>47</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>58</td>
</tr>
</tbody>
</table>

Evaluated annotations

<table>
<thead>
<tr>
<th>Name</th>
<th>✓</th>
<th>✓</th>
<th>🌍</th>
<th>⚠</th>
<th>⋄</th>
</tr>
</thead>
<tbody>
<tr>
<td>cz.cuni.mff.spl.casestudy.annotations</td>
<td>47</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>58</td>
</tr>
<tr>
<td>MeasuredAnnotations1</td>
<td>12</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>failInitAndCurrentVerifier()</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>satisfyElementImprAndActualSAXBuilder()</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>satisfyUpdatedDOMBuilder()</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>unstableSetAttributeDOMBuilder()</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MeasuredAnnotations2</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>failFilterListDOMBuilder()</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>failFilterListSAXBuilder()</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Toolchain for Performance Unit Testing

For Java: automated execution, JUnit-style HTML report, measurement on a remote machine, support for Git and Subversion, Eclipse plugin, Hudson plugin.

Reusing Performance Tests

Because of the decoupled structure of the whole test, we can reuse parts of it . . .

– to track historical performance
– test condition (formula) can read data from other sources such as runtime monitoring probes
– workload generators can be used to create performance-enhanced API documentation
Reusing Tests for Documentation

```java
public boolean contains(java.lang.Object o)

Specified by:
    contains in interface java.util.Collection<T>

Specified by:
    contains in interface java.util.List<T>

Overrides:
    contains in class java.util.ArrayList<T>

Performance:
    • Generator: Unsuccessful search
```

Unsuccessful search in a collection

Configuration:

Collection size **1 to 100000**

Graph

Evaluation of Performance Unit Testing

We selected an existing project (XML library JDOM) with interesting performance characteristics (commits with optimizations, known performance bugs, . . . ).

We modified its history to include performance unit tests.

Would they improve the development process?
Evaluation of Performance Unit Testing

We selected an existing project (XML library JDOM) with interesting performance characteristics (commits with optimizations, known performance bugs, . . . ).

We modified its history to include performance unit tests.

Would they improve the development process?

- simple tests of selected classes only
- found several bugs earlier than in real history
- in several cases, developers introduced a regression in a commit stating performance improvement

Selected Contributions

Performance testing
Change detection
Runtime monitoring
How is “≤” Interpreted?

\[ \forall n \in \{1024, 2048, 4096\} : \]

\[ \text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n) \]

Structure formalization to simplify reasoning.
How is “≤” Interpreted?

∀ \( n \in \{1024, 2048, 4096\} \):

\[
\text{encrypt}_{\text{randomBytes}}(n) \leq 10 \cdot \text{memcpy}_{\text{randomBytes}}(n)
\]

Structure formalization to simplify reasoning.

What is so difficult about “≤”?  

If both sides were one number, it would be easy...
Comparing Performance Data Sets

We rarely get a single number from a benchmark.

Usually, the “score” has some variance (noise) that becomes apparent when restarting the benchmark.

Thus, we do not compare 6.7 and 6.9 but:

9.3, 6.7, 6.8, 6.6, 6.7, …

9.0, 6.9, 6.5, 6.4, 6.6, …
Comparing Performance Data Sets

We rarely get a single number from a benchmark.

Usually, the “score” has some variance (noise) that becomes apparent when restarting the benchmark.

Thus, we do not compare 6.7 and 6.9 but:

- 9.3, 6.7, 6.8, 6.6, 6.7, …
- 9.0, 6.9, 6.5, 6.4, 6.6, …

…a textbook example for statistical tests.

But the data rarely fits the assumptions of such tests.
Case Study: False Alarms

Run benchmark multiple times in the same configuration. The scores are the same.

For multiple benchmarks: compute rate of false alarms. How often would the test consider the same data different?
Case Study: False Alarms

Run benchmark multiple times in the same configuration. The scores *are the same*.

For multiple benchmarks: compute rate of false alarms. How often would the test consider the same data different?

Case Study: False Alarms

Run benchmark multiple times in the same configuration. The scores are the same.

For multiple benchmarks: compute rate of false alarms. How often would the test consider the same data different?
Sensitivity of “≤” Interpretations

Experiment II: run benchmark with increased workload size.

How soon can we detect the difference?
Sensitivity of “≤” Interpretations

Experiment II: run benchmark with increased workload size.

How soon can we detect the difference?

Ideal method

Probability

Correct decision
False alarm

Actual performance difference [%]

Sensitivity of "≤" Interpretations

Experiment II: run benchmark with increased workload size.

How soon can we detect the difference?

![Graph showing sensitivity of "≤" interpretations]

Sensitivity of “≤” Interpretations

Experiment II: run benchmark with increased workload size.

How soon can we detect the difference?

- Ideal method
- Welch’s t-test
- Bootstrap–based (ours)

Selected Contributions

Performance testing
Change detection
Runtime monitoring
Precision of Measurements in Production Environment

Performance in testing environment is inherently skewed.

Rarely the workload matches the real one, there are fewer outside effects . . .
Precision of Measurements in Production Environment

Performance in testing environment is inherently skewed.

Rarely the workload matches the real one, there are fewer outside effects . . .

We should measure the real application . . .

. . . and apply the performance formulas to data from production environments.
Issues of Runtime Performance Monitoring

Each measurement introduces overhead, measuring everything (e.g. each method) is not practical.

Managed environments (e.g. Java) support dynamic monitoring – measurement probes are inserted/removed on demand, code is JIT recompiled.
Issues of Runtime Performance Monitoring

Each measurement introduces overhead, measuring everything (e.g. each method) is not practical.

Managed environments (e.g. Java) support dynamic monitoring – measurement probes are inserted/removed on demand, code is JIT recompiled.

What is the overhead of dynamic probes?

How precise data can we collect with these probes?

Is probe insertion/removal a transparent operation?
Experiment Setup

Extend application with probes collecting the baseline performance. Normally, one cannot afford that because of the overhead.

Use dynamic monitoring to collect observed performance.

Our scenario

- SPECjbb2015 at constant load
- Measured over 1200 methods
- One method observed at a time
- Collected several TBs of data
What is the difference between performance collected by static (baseline) and dynamic (observed) probes?

Ideally, none. Practically, constant overhead would be nice.
Accuracy of Runtime Performance Monitoring

Effects of Measurement Probes

Dynamic monitoring of one place means (notice that baseline performance is always collected)

1. Application is running without dynamic probes
2. Insert the dynamic probe
3. Collect the observed performance [WITH]
4. Remove the dynamic probe
5. Application continues its execution [AFTER]

Baseline performance of [WITH] and [AFTER] should not differ.

Effects of Measurement Probes

Baseline performance: ratio of mean execution times [WITH] and [AFTER] observation

Effects of Measurement Probes

Baseline performance: ratio of mean execution times [WITH] and [AFTER] observation

Conclusion
Contributions of the Thesis

Make developers aware of the performance during all phases of development lifecycle.

- Performance unit testing
- Robust performance regression detection
- Performance documentation
- Analysis of limits of runtime performance monitoring
Acknowledgements

D3S team: Petr Tůma, Lubomír Bulej, Tomáš Bureš, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Rima Al Ali, Vlastimil Babka, Martin Blicha, Jakub Daniel, Martin Děcký, Ilias Gerostathopoulos, David Hauzar, Pavel Jančík, Pavel Ježek, Peter Libič, Jaroslav Keznikl, Michal Kit, Filip Krijt, Alena Koubková, Petr Kubát, Lukáš Marek, Vladimír Matěna, Petra Novotná, Antonín Steinhauser, Dominik Škoda, Jiří Vinárek

Students: František Haas, Martin Lacina, Jaroslav Kotrč, Jakub Náplava, Petr Stefan

and also all the reviewers
Other Activities

– teaching, theses supervision
– (sub)reviewer for several conferences
– SPEC RG release manager