Performance Unit Testing

Vojtěch Horký Petr Tůma

Charles University in Prague

Saturday 22nd March, 2014

In Lieu Of Motivation

Developers Care About Performance

Otherwise following comments would not exist...

```
SelectorUtils {
  Vector tokenizePath(String path) {...}

  // Same as tokenizePath but hopefully faster.
  String[] tokenizePathAsArray(String path) {...}

}
```
[1]

[1] Apache Ant 1.9.3: org.apache.tools.ant.types.selectors.SelectorUtils

Developers Care About Performance

. . . or we would not find this in commit logs

> *The Re-Work includes a major performance upgrade because FilterList and FilterListIterator are now 'lazy' . . .*
>
> *This should make things like for-each type loops much faster to start up, and isEmpty() and random-access-type get() calls much faster too.*[2]

[2] JDOM XML manipulation library commit log (932cfb2)

We Want To Do Better

. . . than to just write a comment with our assumption.

We do not do this for a functional requirement:

```
// Hopefully foo is not NULL
foo->bar = 42;
```

We write it explicitly:

```
assert(foo != NULL);
foo->bar = 42;
```

Performance Unit Tests

- Something like functional unit tests
- Time-checking alternative of `assert()`

Challenges

Function `foo()` runs under 2 ms.

Function `bar()` is in version 2.0 faster than in 1.0.

1. How to embed this in the code?
2. How to automatically evaluate this?
   - Robust measurement is not about single run...
3. How to cooperate with VCSs?
4. Anyway, where did you get these 2 ms from?
   - Platform matters...

Our Approach

Formalism called Stochastic Performance Logic [1]

- Compare performance of individual functions
- Hypothesis testing
- Embedded in the code

Formalism

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

$$\forall n \in \{1024, 16384, 65536\}$$

$$Perf^{\text{encrypt}}(n) <_{(id,200)} Perf^{\text{memcopy}}(n)$$

Formalism

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

$$\forall n \in \{1024, 16384, 65536\}$$

$$Perf^{\text{encrypt}}(n) <_{(id,200)} Perf^{\text{memcopy}}(n)$$

Formalism

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

$$\forall n \in \{1024, 16384, 65536\}$$

$$Perf^{\text{encrypt}}(n) <_{(id,200)} Perf^{\text{memcopy}}(n)$$

Formalism

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

$$\forall n \in \{1024, 16384, 65536\}$$

$$Perf^{\text{encrypt}}(n) <_{(id,200)} Perf^{\text{memcopy}}(n)$$

Formalism

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

$$\forall n \in \{1024, 16384, 65536\}$$

$$Perf^{\text{encrypt}}(n) <_{(id,200)} Perf^{\text{memcopy}}(n)$$

Implementation for Java

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

```
@SPL(
  methods = "memcopy=java.lang.System#arraycopy",
  formula = "for ( i { 1024, 16384, 65536 }) "
    + "SELF(i) <=(1, 200) memcopy(i)"
)
public long[] encrypt(long[] data) {
  /* ... */
}
```

Implementation for Java

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

```
@SPL(
  methods = "memcopy=java.lang.System#arraycopy",
  formula = "for ( i { 1024, 16384, 65536 }) "
    + "SELF(i) <=(1, 200) memcopy(i)"
)
public long[] encrypt(long[] data) {
  /* ... */
}
```

Implementation for Java

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

```
@SPL(
  methods = "memcopy=java.lang.System#arraycopy",
  formula = "for ( i { 1024, 16384, 65536 }) "
    + "SELF(i) <=(1, 200) memcopy(i)"
)
public long[] encrypt(long[] data) {
  /* ... */
}
```

Implementation for Java

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

```
@SPL(
  methods = "memcopy=java.lang.System#arraycopy",
  formula = "for ( i { 1024, 16384, 65536 }) "
    + "SELF(i) <=(1, 200) memcopy(i)"
)
public long[] encrypt(long[] data) {
  /* ... */
}
```

Implementation for Java

Encryption function is for input of size 1024, 16384 and 65536 at most 200 times slower than memory copying (base line operation).

```
@SPL(
  methods = "memcopy=java.lang.System#arraycopy",
  formula = "for ( i { 1024, 16384, 65536 }) "
    + "SELF(i) <=(1, 200) memcopy(i)"
)
public long[] encrypt(long[] data) {
  /* ... */
}
```

Workload

Separate function prepares the arguments for the tested method.

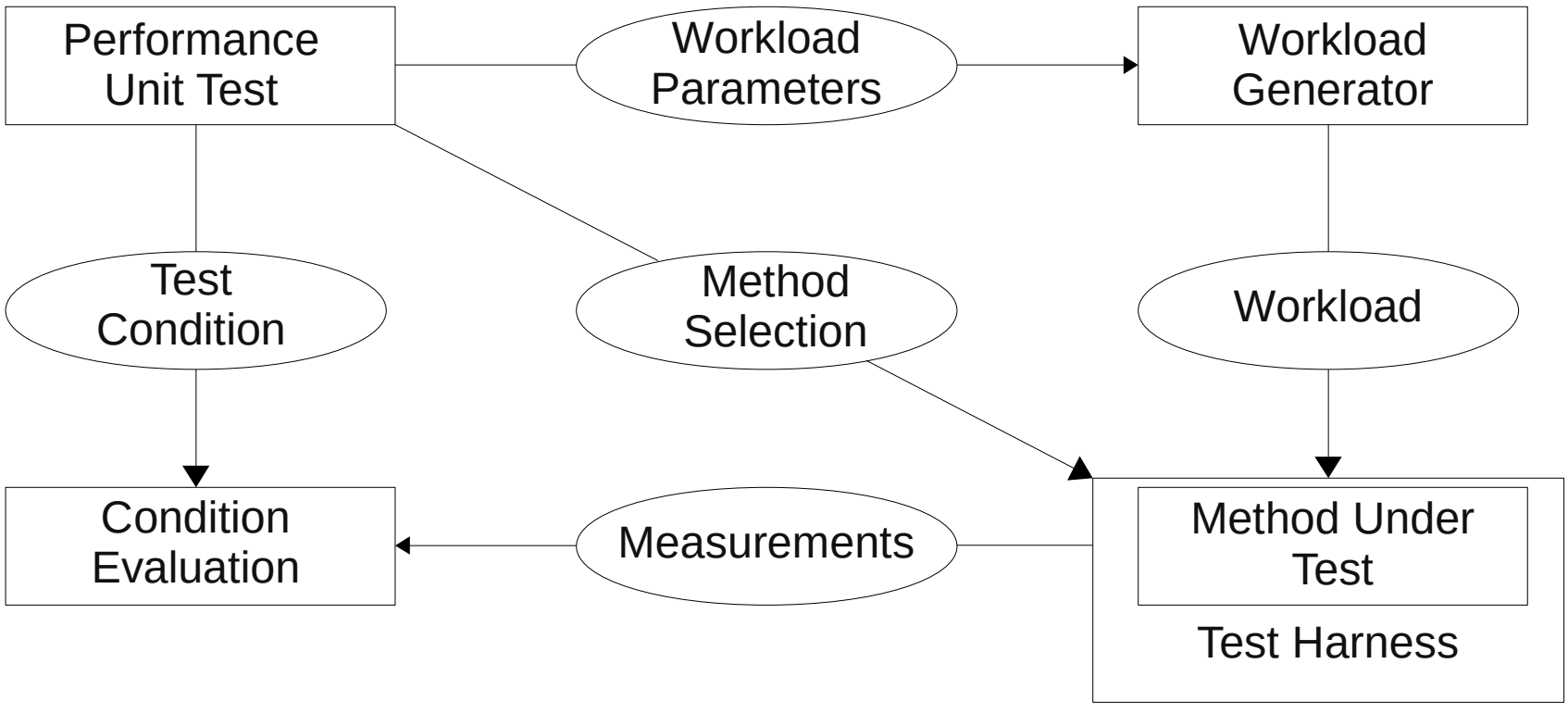
```
Object[] encryptionWorkload(int size) {
    long[] data = new data[size];
    /* Prepare the data to encrypt. */

    return new Object[] { data };
}
```

Measurement just calls the method with this workload.

```
testedMethod.invoke(encryptionWorkload(1024));
```

Performance Unit Testing



Our Tool

- Isolated method execution
- JUnit-like output + raw measurement data
- Eclipse plugin
- Hudson plugin
- Testing on a remote machine
- Support for Git and Subversion
- Open source under 3-clause BSD license [3]

Case Study

- JDOM XML manipulation library[3]
- Retroactively test that performance assumptions were correct [2]

[3] http://www.jdom.org/, https://github.com/hunterhacker/jdom

Case Study: Results

- 46 commits across 10 years of development
- We found regression earlier
- Confirmed assumptions
- Tested on different machines
  - Different absolute times
  - Relative comparisons work well
- t-test works even with violated assumptions

Hands-on Tutorial

Tutorial Scenario

We have a CSV of purchases from different customers and we want to compute the total sum for each customer.

We made several changes to the code that tried to improve its performance.

Let's see whether our assumptions were correct.

Tutorial Resources

https://github.com/d3s-tools/spl-perf-unit-testing-tutorial

- Branch `master` contains the main parts
- Branch `solution` adds the SPL-based performance unit testing

http://d3s.mff.cuni.cz/software/spl-java

- Other resources (Eclipse plugin, manuals, case study)

Tutorial Contents

- Find performance related commits
- Add regression performance unit tests
- Launch the tests and evaluate them

Performance Related Commits

- *Use HashMap to speed-up the lookup*
  - Use associative array instead of iterating through a linked list
- *Purchase summary: cache last entry*
  - Last used `Map` item as an extra variable
- *Use compiled pattern for splitting*
  - Compile regular expression just once

Test Example

- `test/spl` subdirectory
- `PurchaseReaderSplTest.java`
  - Workload generator and a simple performance test
- Running the tests: `ant test-spl`
  - Downloads SPL tool JAR
  - Executes the test on local machine
  - Creates HTML output in `out/spl-wd/evaluation/`

Test Example Zoomed In

```
@SPL(
  generators = {
    "gen=PurchaseReaderSplTest#generator()"
  },
  methods = {
    "read=PurchaseReader#read"
  },
  formula = {
    "for (count {10, 100}) "
    + "read[gen](count) = read[gen](count)"
  }
)
```

Test Example Zoomed In: Workload Generator

```
static Iterable<Object[]> generator(int items) {
  StringBuilder emulatedFile = new StringBuilder();

  for (int i = 0; i < items; i++) {
    String line = String.format("customer%d;...\n",
        i + 1);
    emulatedFile.append(line);
  }

  PurchaseAggregator agg = new DummyAggregator();

  List<Object[]> result = new ArrayList<>(1);
  result.add(new Object[] { agg,
    new StringReader(emulatedFile.toString()) });
  return result;
}
```

Test Configuration

- Project
  - Repository (path)
  - Build command
  - Classpaths
  - Aliases to interesting commits
- Eclipse editor

Converting the Example Into a Real Test

Add alias for the interesting commits

- `init` → `9d62`
- `regexp` → `4045`

Converting the Example Into a Real Test (cont.)

Extend the formula

```
methods = {
  "string=SELF@init:cz...PurchaseReader#read",
  "regexp=SELF@regexp:cz...PurchaseReader#read"
},
formula = {
  "for (count {10, 100, 500, 1000}) "
  + "string[gen](count) > regexp[gen](count)"
}
```

Converting the Example Into a Real Test (cont.)

Create a new project for generators for better separation

```
generators = {
  "gen=GENERATORS@reader:cz.cuni.mff.d3s.spl."
   + "tutorial.PurchaseReaderSplTest#generator()"
}
```

Result Interpretation

- Pass/fail summary
- Statistical details
- Run-plots and histograms

Next Steps

- Aadd aliases for other interesting commits
- Create generator for the `PurchaseSummary` class

```
new Object[] {
  String.format("customer%d", random.nextInt(100)),
  "itemXY",
  new Date(),
  random.nextInt(10) + 1, // max 10 items
  random.nextDouble() * 100, // price 0..100
}
```

- Create formula testing the improvements
  - `LinkedList` $\rightarrow$ `Map` $\rightarrow$ `Map` + caching

Demo

For Completeness: Were Our Assumptions Correct?

Use HashMap to speed-up the lookup

- The effect is clearly visible for > 50 customers

Purchase summary: cache last entry

- Visible only for large chunks
- Probably not worth the hassle

For Completeness: Were Our Assumptions Correct?

Use compiled pattern for splitting

- Wrong assumption
- `String.split()` has a fast-path when the pattern consists of a single character
- Added penalty
  - Compilation of the pattern
  - Regular expression automaton vs. `String.indexOf()`

Conclusion

Performance Unit Testing with SPL

- Test small pieces of code
- Robust & automatic evaluation
- Tools for Java
  - Eclipse, Hudson

References

[1] Lubomír Bulej, Tomáš Bureš, Jaroslav Keznikl, Alena Koubková, Andrej Podzimek and Petr Tůma. Capturing Performance Assumptions using Stochastic Performance Logic. *In proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012).* ACM 2012.

[2] Vojtěch Horký, František Haas, Jaroslav Kotrč, Martin Lacina and Petr Tůma. Performance Regression Unit Testing: A Case Study. *In proceedings of the 10th European Workshop (EPEW 2013).* Springer 2013.

[3] http://d3s.mff.cuni.cz/software/spl-java

Thank You

http://d3s.mff.cuni.cz/software/spl-java

horky@d3s.mff.cuni.cz