Automated Benchmarking of Java APIs

Michael Kuperberg, Fouad Omri, Ralf Reussner
{firstname.lastname}@kit.edu

Paderborn, Feb. 25nd, 2010
Motivation for API Benchmarking

- Performance evaluation in software engineering:
  - Quantify response time, resource utilisation, scalability
  - Important for both required and offered APIs

- Performance Prediction Models
  - Benchmark results as input parameter
  - Calibration of prediction models
  - Validation of prediction models
  - Prediction of performance of software on different platforms

- Comparisons
  - Comparing the performance of similar methods
  - Comparing the performance of different platforms
Outline

Motivation

- Challenges
- API Benchmarking in Java
- Evaluation
- Related Work
- Conclusion
Challenges

- Size:
  - 1000s of methods
  - Examples: Java platform API, JScience API, etc.

- Virtual machines: managed environments
  - try to speed up bytecode interpretation at runtime

- Just-In-Time compilation (JIT) finds "hot" code sections, compiles them to machine code – hard to predict!
  - Method performance highly sensitive to invocation context
  - Just-In-Time compilation causes on-stack replacement etc.
  - Requires statistical consideration and measurement setup
  - JIT cannot be controlled externally

R. Reussner et al.: 'Automated Benchmarking of Java APIs'
SE2010, Paderborn, Germany
Challenges: Parameter Generation

- Method parameters are subject to constraints, e.g. `String.substring(int index)`:
  - Constraints: `index ≥ 0` and `index < string's length()`
  - Constraints clear to humans – unavailable for machines
  - Violation of constraints → exceptions at runtime

- Input parameter values have to be generated in accordance to the declared static types
  - Difficult: Generation of reference/generic types, e.g. interface types
  - E.g. `cs` in `String.contentEquals(CharSequence)`

- API is a "black box" → assumption: source code of API implementation unavailable
APIbenchJ: Overview of the approach

1. Identify and sort benchmarkable methods
2. Create benchmarking scenarios for each method
3. Satisfy and test preconditions for method invocation (parameters, …)
   - Successful?
     - yes
     - 4a. Save successful preconditions for reuse
     - 4b. Analyse exceptions, recommend new preconditions
     - no (e.g. runtime exception)
   - 5. Generate individual method microbenchmarks
   - 6. Run microbenchmark suite on the target platform, evaluate benchmarking results

Steps 3-5: performed for each API method
Steps 1-5: only 1x per API, on any platform

Uses heuristics [KOR09] to find parameters, or human-specified values
APIbenchJ: Microbenchmark Generation

- Executable microbenchmarks: *generated* bytecode
  - Microbenchmarks address measurement details, e.g.
    - timer resolution [KKR09]
    - JVM optimisations: JIT etc.
    - warmup; outlier detection
    - statistical validity
    - result: probability histogram, not just an average value
  - Each microbenchmark is a stand-alone Java class
    → APIbenchJ requires no specific infrastructure for execution
- The **microbenchmark suite**: additional infrastructure for collecting results and evaluating them
Dealing with JVM optimisations

- Need to ensure that JIT does not “optimise away” the benchmarked operations
  - especially for deterministic methods: different parameters must be passed, record returned values!
  - one of used solutions to “outwit” the JIT compiler:
    - use array elements as input parameters
    - reference the $i$th element of the arguments array $\text{arg}$ in a special way: $\text{arg}[i \% \text{arg}.\text{length}]$

- Our solution prevents the JIT compiler from applying undesirable constant folding, identity optimisation and global value numbering optimisations
Benchmarking Methodology
Considering Timer Accuracy and Invocation Cost

/* \( N \) Starting number of iterations set as default value to 100 in APIBENCHJ */
/* \( O \) The timer invocation cost in nanoseconds */
/* \( R \) The timer accuracy in nanoseconds */
/* \( PT \) The precision threshold, i.e., the degree to which the timing measurements will be influenced from the timer. \( PT \) is set in APIBENCHJ to 0.01. */

Data: \( N \), initial number of loop iterations
Result: Elapsed time by calling the invokable

1 \( \text{START} \leftarrow \text{Start timing}; \)
2 for \( i = 1 \) to \( N \) do
3 \quad execute invokable;
4 end
5 \( \text{STOP} \leftarrow \text{Stop timing}; \)
6 \( D \leftarrow (\text{STOP} - \text{START}); \)
7 if \( (D \times PT) \leq (R + O) \) then
8 \quad \( NI \leftarrow (NI \times 2) + 1; \)
9 \quad return measure(\( NI \));
10 end
11 return \( D / NI; \)
Evaluation: Quantify Parameteric Dependency

- Linear dependency detected
Evaluation (1)

- **Four metrics:** Precision, Accuracy, Coverage, Effort
- **Precision:** repeatability of the benchmarked values:
  → distribution function
- **Accuracy:** The only reference available for comparison is "best-effort" manual work
- The method `java.lang.String.substring(...)`
  - `substring`: Performance-intensive, used often, parametric
  - Example: `String length 14, beginIndex 4, endIndex 8`
  - Manual benchmarking: **9 ns**
  - `APIbenchJ`: ∅ of **8.555 ns**
  - Distribution: CPU scheduling etc.
Evaluation (2)

- **Effective coverage**: Java platform API

<table>
<thead>
<tr>
<th>Package</th>
<th>#Public non-abstract classes</th>
<th>#Public non-abstract methods</th>
<th>#Executed methods w/o exceptions</th>
<th>Execution success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util</td>
<td>58</td>
<td>738</td>
<td>668</td>
<td>90.51%</td>
</tr>
<tr>
<td>java.lang</td>
<td>76</td>
<td>861</td>
<td>790</td>
<td>91.75%</td>
</tr>
</tbody>
</table>

- **Effort** (examples, see paper for details):

<table>
<thead>
<tr>
<th>Package</th>
<th>Heuristic parameter generation</th>
<th>Benchmarking duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.util</td>
<td>~6 min</td>
<td>~101 min</td>
</tr>
</tbody>
</table>

- due to extensive warmup for inducing JIT optimisations
- The generation of microbenchmarks: very fast
  - For the method `String.contains(CharSequence s)` < 10 ms
- `String.contains(CharSequence s)`, benchmarking: ~5000 ms (348 repetitions to reach the confidence interval of 0.95)
Related work

- Zhang, Seltzer [ZS]: only 30 methods
  - benchmarked manually, JIT not regarded

- Java benchmarks ([Cor08,Sta08,Pie] etc.)
  - Small, insufficient API coverage, JIT not in focus
  - Execution durations for individual methods unavailable

- Profilers [VTune,JP]; performance testing frameworks
  - Need "test cases", in particular parameters for methods
  - Do not account for JIT, timer accuracy etc.
Future Work

- Generate parameters w.r.t. quantification of parameter performance dependencies
  - several input sets per signature (sensitivity analysis)
  - vary invocation target objects

- Enhance the heuristical parameter generation
  - by incorporating machine learning techniques
  - using search-based software engineering techniques
  - use recorded parameters (e.g. with ByCounter [11])

- Integrate into benchmarking/perf. prediction
- Port to .NET CLR and other environments
Conclusion

- **Benchmarking API methods**: evaluate the performance of required and provided interfaces
- **Automated benchmark generation** and execution
- **Modular automated approach** for any Java API; evaluated (>1000 methods) on the Java platform API
- **Addressed challenges**: impact of JIT compilation, parameter generation, statistic validity, etc.

- Thank you for your attention! Questions?
References


[Cor08] SPEC Corp. SPECjvm2008 Benchmarks http://www.spec.org/jvm2008/.


