

Benchmark this!



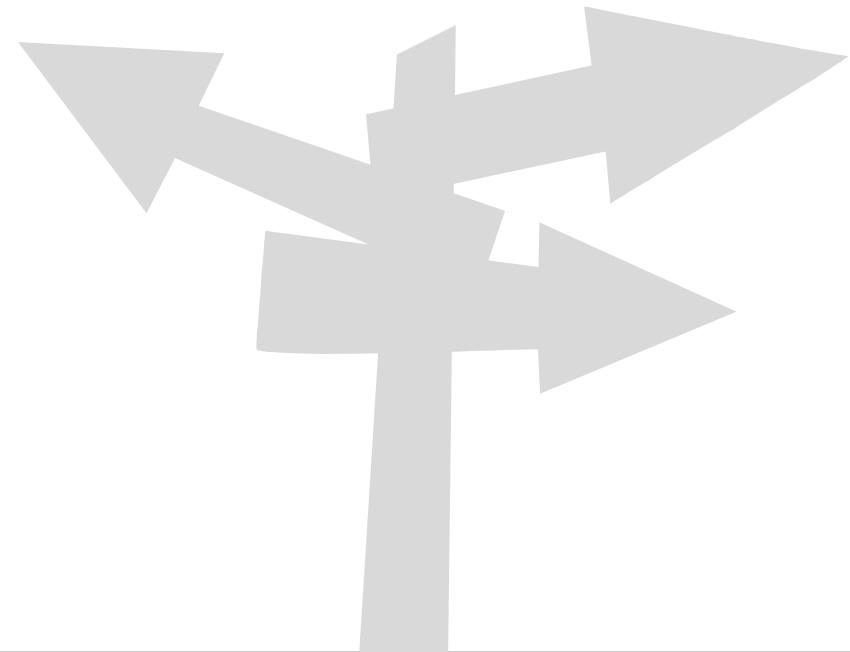
Bildquelle: http://campar.in.tum.de/twiki/pub/Students/SepKatjaUli/Klassendiagramm_2406.png

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



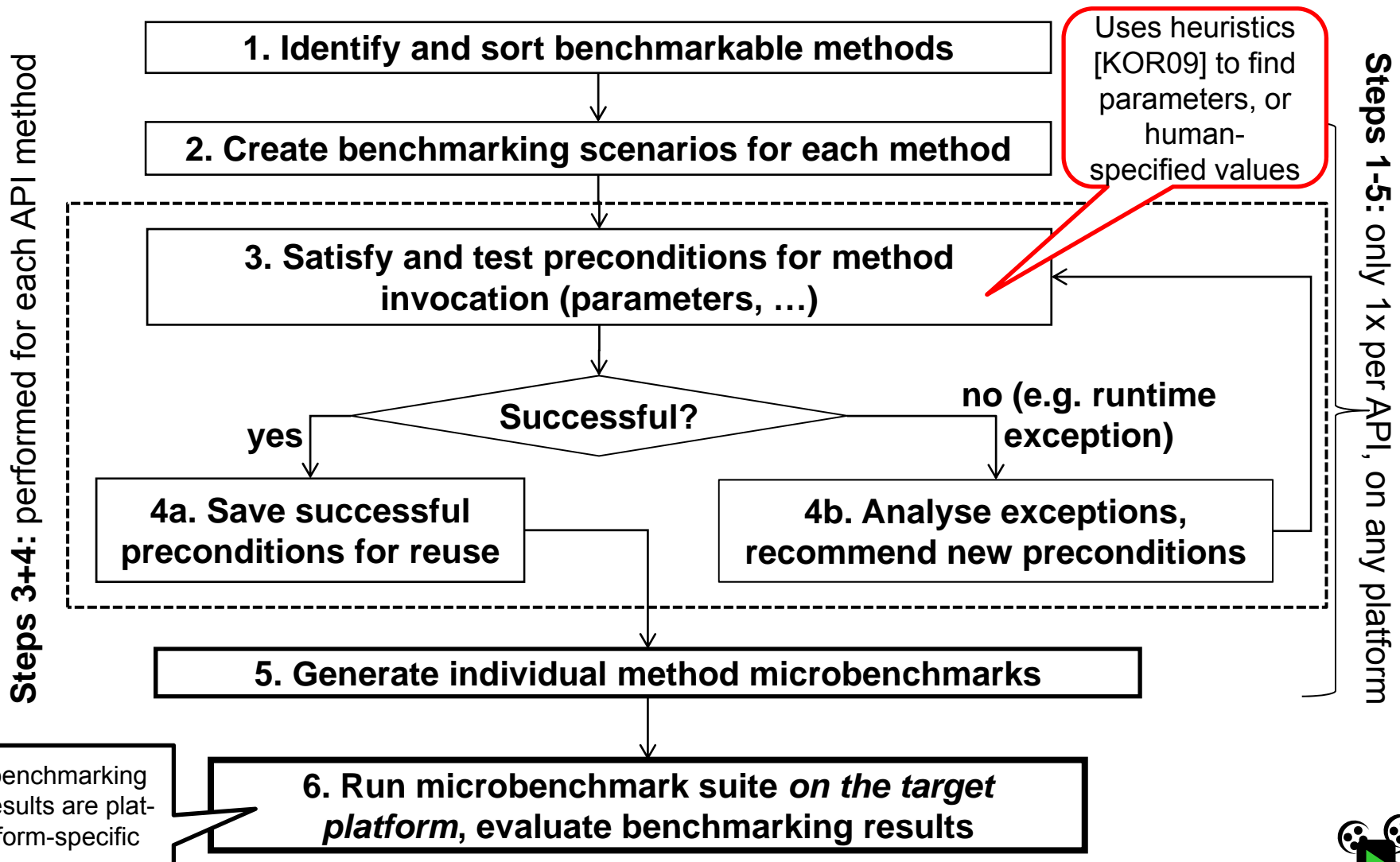
Challenges

- 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

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



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 `arg` in a special way: `arg[i%arg.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

```
/*  $\mathcal{N}\mathcal{I}$  Starting number of iterations set as default value to 100  
   in APIBENCHJ */  
/*  $\mathcal{O}$  The timer invocation cost in nanoseconds */  
/*  $\mathcal{R}$  The timer accuracy in nanoseconds */  
/*  $\mathcal{P}\mathcal{T}$  The precision threshold, i.e, the degree to which the  
   timing measurements will be influenced from the timer.  $\mathcal{P}\mathcal{T}$   
   is set in APIBENCHJ to 0.01. */
```

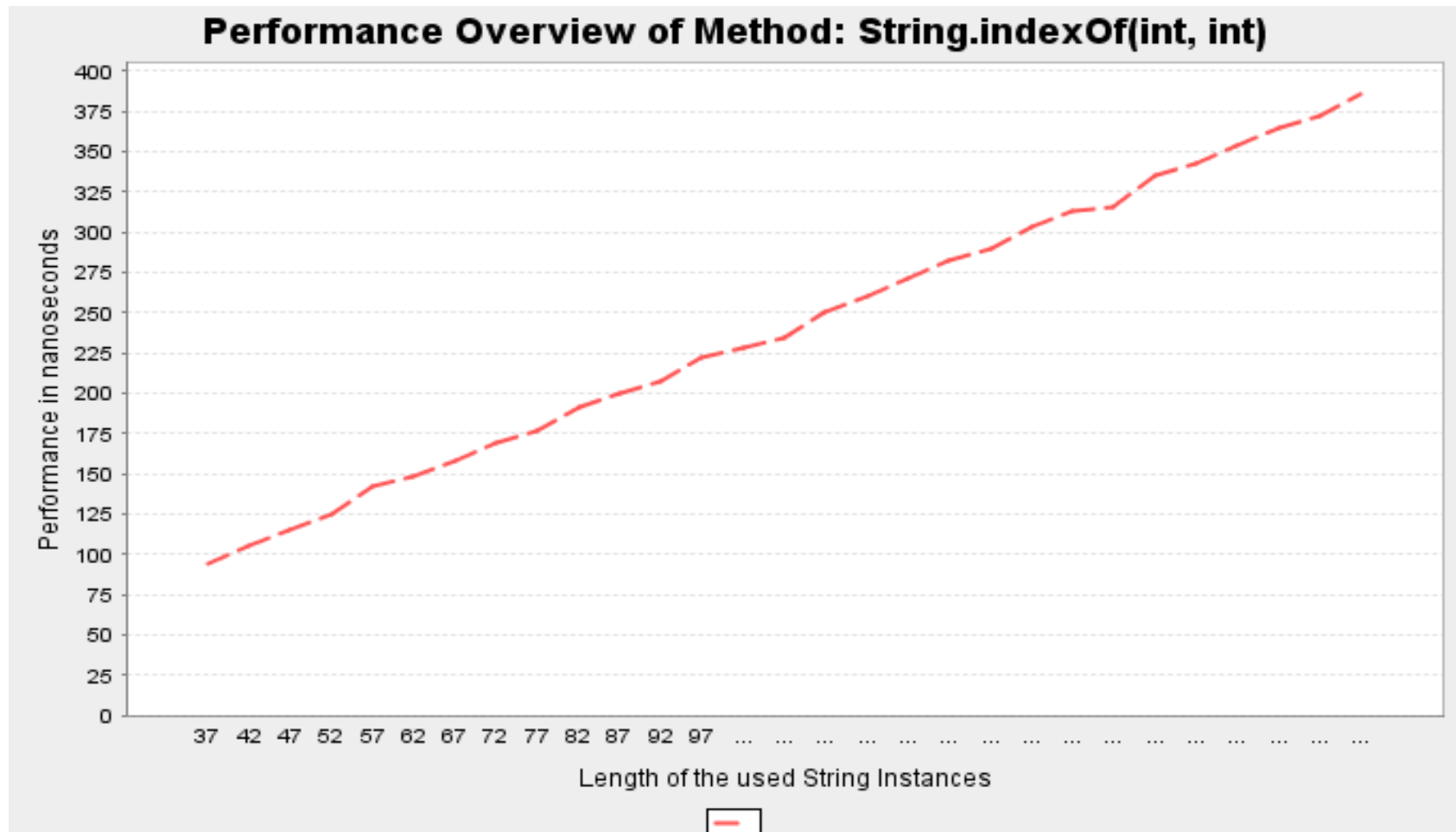
Data: $\mathcal{N}\mathcal{I}$, initial number of loop iterations

Result: Elapsed time by calling the invocable

```
1  $START \leftarrow$  Start timing;  
2 for  $i = 1$  to  $\mathcal{N}\mathcal{I}$  do  
3   | execute invocable;  
4 end  
5  $STOP \leftarrow$  Stop timing;  
6  $\mathcal{D} \leftarrow (STOP - START)$ ;  
7 if  $(\mathcal{D} * \mathcal{P}\mathcal{T}) \leq (\mathcal{R} + \mathcal{O})$  then  
8   |  $\mathcal{N}\mathcal{I} \leftarrow |(\mathcal{N}\mathcal{I} * 2) + 1|$ ;  
9   | return measure( $\mathcal{N}\mathcal{I}$ );  
10 end  
11 return  $\mathcal{D} / \mathcal{N}\mathcal{I}$ ;
```

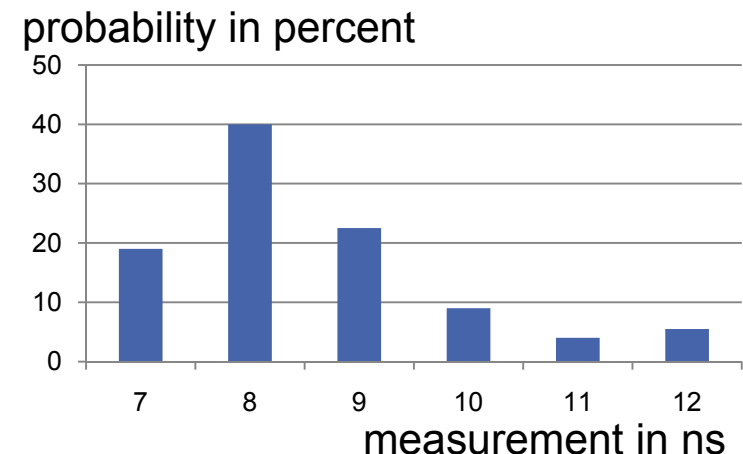
Evaluation: Quantify Parameteric Dependency

- 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: \emptyset of **8.555 ns**
 - Distribution: CPU scheduling etc.



Evaluation (2)

■ Effective coverage: Java platform API

Package	#Public non-abstract classes	#Public non-abstract methods	#Executed methods w/o exceptions	Execution success rate
java.util	58	738	668	90.51%
java.lang	76	861	790	91.75%

■ Effort (examples, see paper for details):

Package	Heuristic parameter generation	Benchmarking duration
java.util	~6 min	~101 min

- 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

- [PCM] Palladio Component Model: <http://www.palladio-approach.net>
- [JP] JProfiler, 2009. <http://www.ej-technologies.com/products/jprofiler/overview.html>
- [KOR09] Michael Kuperberg, Fouad Omri, and Ralf Reussner. Using Heuristics to Automate Parameter Generation for Benchmarking of Java Methods. In Proceedings of the 6th International Workshop on Formal Engineering approaches to Software Components and Architectures, York, UK, 28th March 2009 (ETAPS 2009, 12th European Joint Conferences on Theory and Practice of Software), 2009.
- [KKR09] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. *ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations*. In BYTECODE'08, 2008.
- [Cor08] SPEC Corp. SPECjvm2008 Benchmarks <http://www.spec.org/jvm2008/> .
- [Pie] Darryl L. Pierce. J2ME Benchmark and Test Suite. <http://sourceforge.net/projects/j2metest/> .
- [Sci07] Java SciMark 2.0, 2007. URL: <http://math.nist.gov/scimark2/> .
- [Sta08] Standard Performance Evaluation Corp. SPECjAppServer2004 Benchmarks, 2008. <http://www.spec.org/jAppServer2004/> .
- [VTune] Intel VTune Performance Analyser, 2009. <http://software.intel.com/en-us/intel-vtune/> .
- [ZS00] Xiaolan Zhang and Margo Seltzer. *H Bench: Java: an application-specific benchmarking framework for Java virtual machines*. ACM 2000 conference on Java Grande, pages 62–70, New York, NY, USA, 2000.