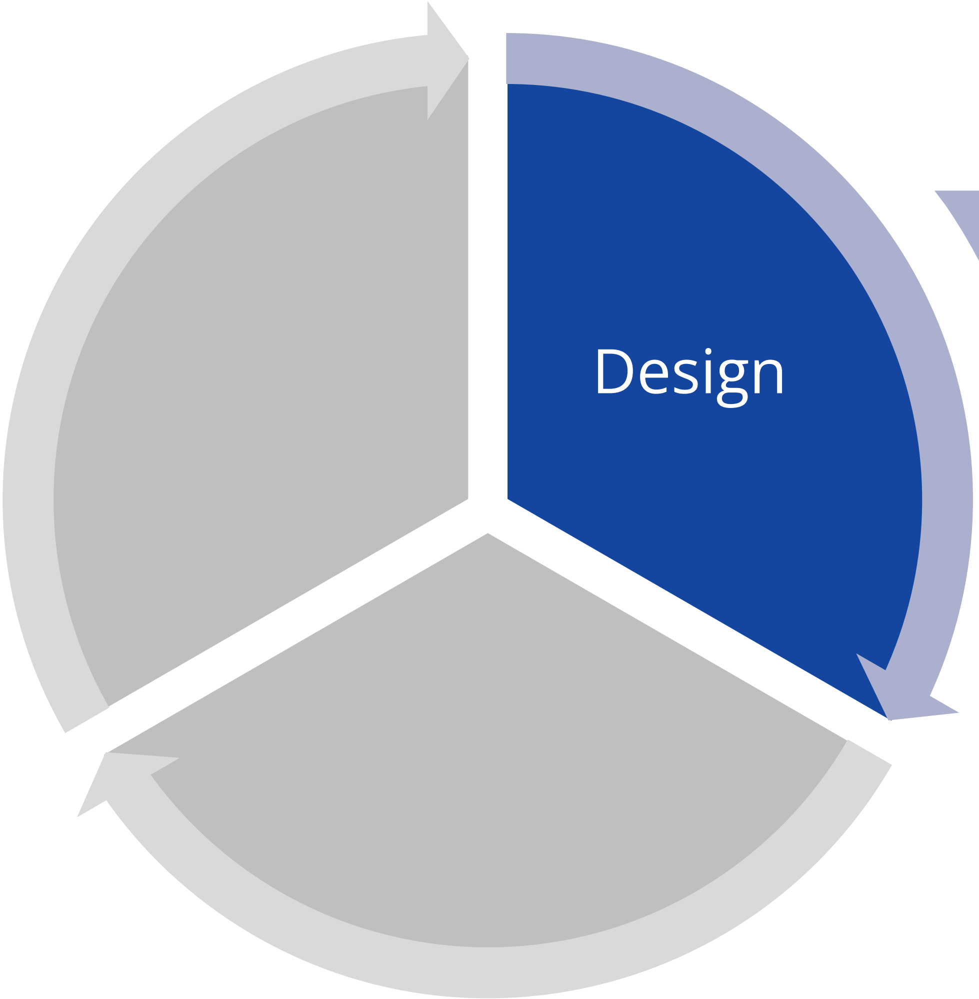
Performance Benchmarking

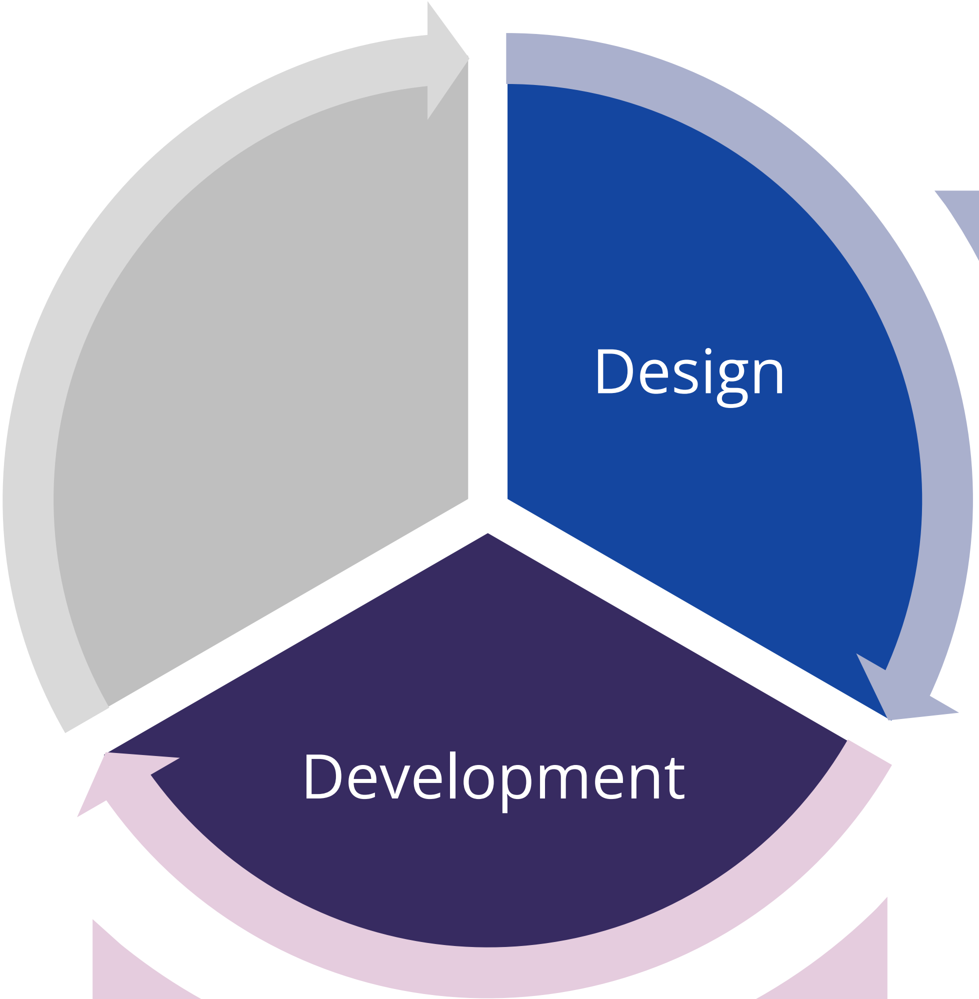
MŰEGYETEM 1782
BME Faculty of Electrical Engineering and Informatics
ftsrg Critical Systems Research Group

Overview



- Graph-Based Modeling
- Textual Modeling
- Code Generation
- Model Intelligence
- Model Checking

Overview

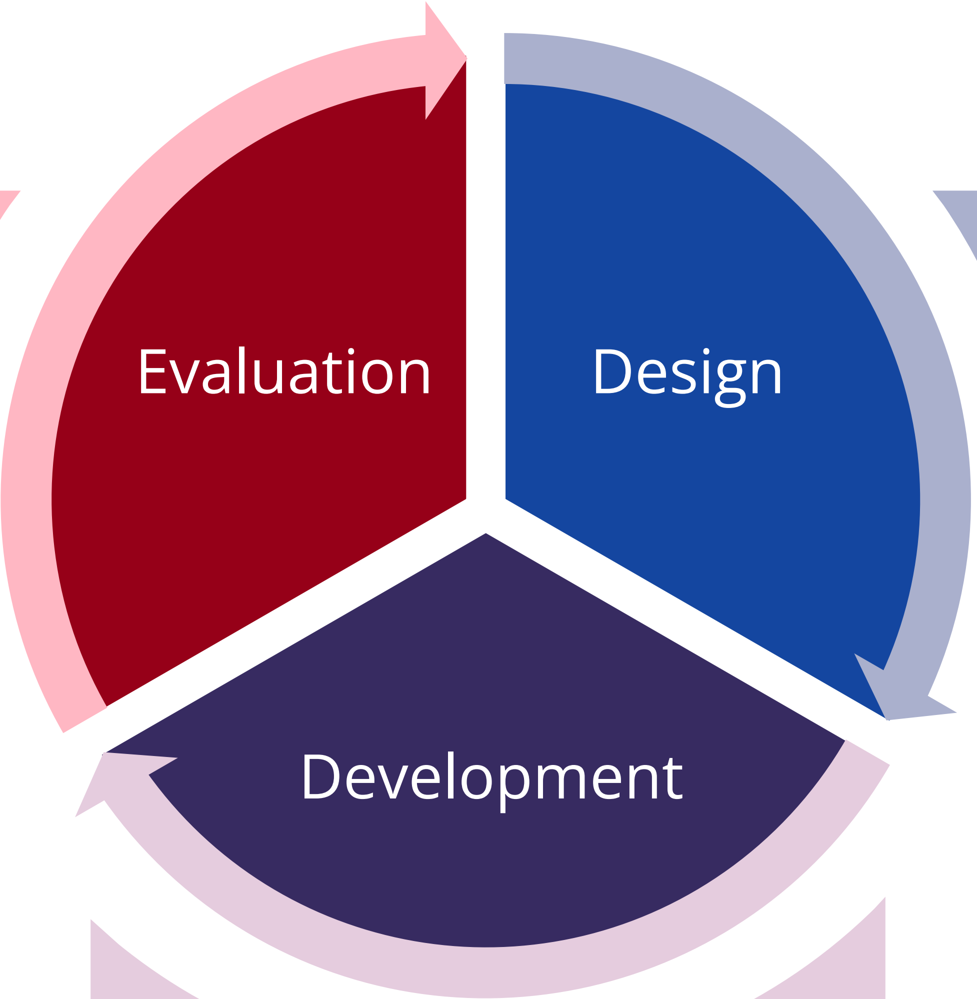


- Graph-Based Modeling
- Textual Modeling
- Code Generation
- Model Intelligence
- Model Checking

- Build Automation
- Code Generation
- Code Intelligence

Overview

- Performance evaluation
- Data Analysis
- Code Quality
- Static Analysis
- Testing & Coverage



- Graph-Based Modeling
- Textual Modeling
- Code Generation
- Model Intelligence
- Model Checking

- Build Automation
- Code Generation
- Code Intelligence

Performance Benchmark Overview

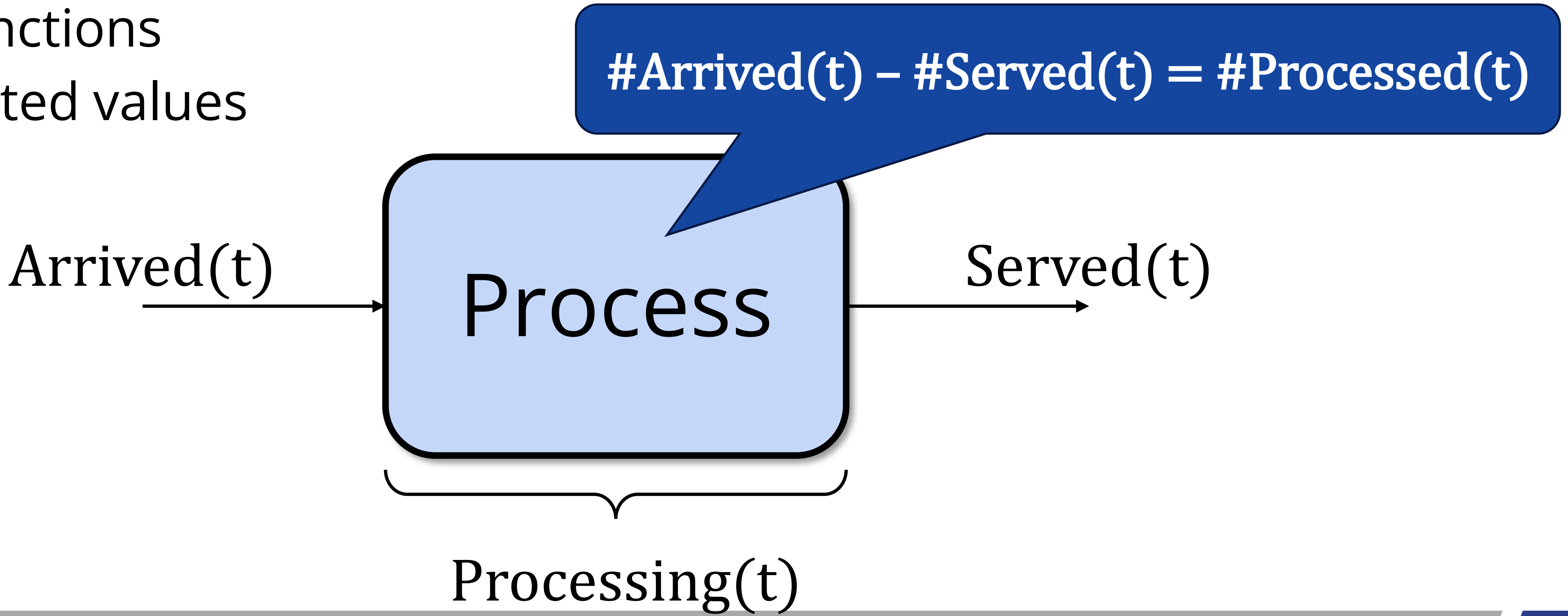
- Generic overview
- Micro-benchmark
- Macro-benchmark

- How to create a benchmark?
- How to write an evaluation section?

Benchmarking Overview

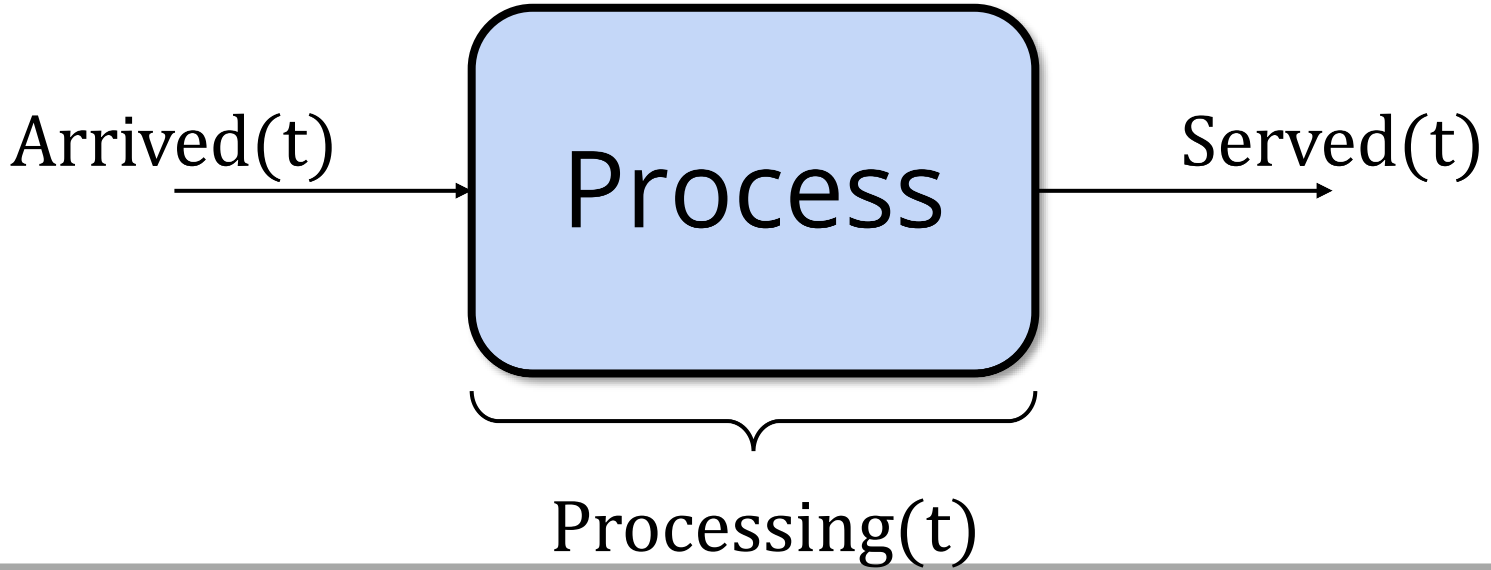
Basic Model

- Executing a process multiple times
- Objectives: measure the time aspect of the behavior
- Describe the behavior with
  - Time functions
  - Aggregated values



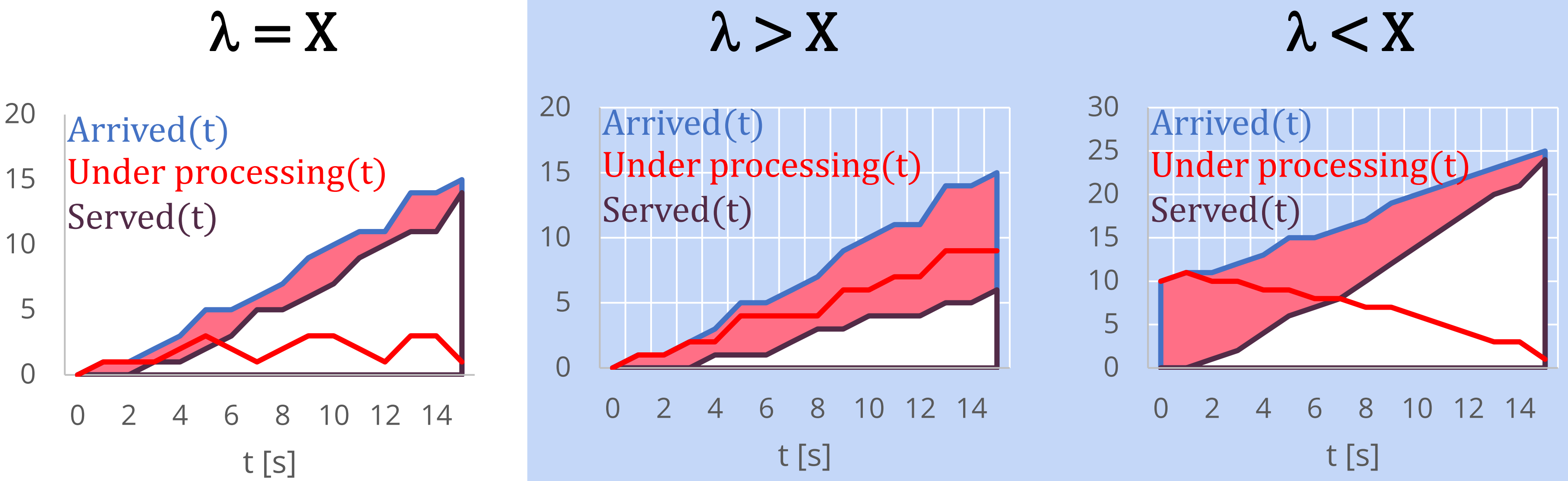
Definition: Arrival Rate, Throughput

- **Arrival rate**: number of **arriving** requests during a specific unit of time. $\lambda = \frac{Arrived(t)}{t}$ $[\lambda] = \frac{1}{s}$
- **Throughput** (XPUT): number of requests **processed** during a specific unit of time. $X = \frac{Served(t)}{t}$



Definition: Stable State

- **Stable state:** *Under processing(t)* is approximately constant
  - Average values can be applied in such state!
  - A system is in balance, if: $$\lambda = X$$



Definition: Stable State

- **Stable state:** *Under processing(t)* is approximately constant
  - Average values can be applied in such state!
  - A system is in balance, if:

In stable state:
Same number of logins and logouts per minute

Limited Capacity – DoS

- N is not infinite in real life
- So, what is then?

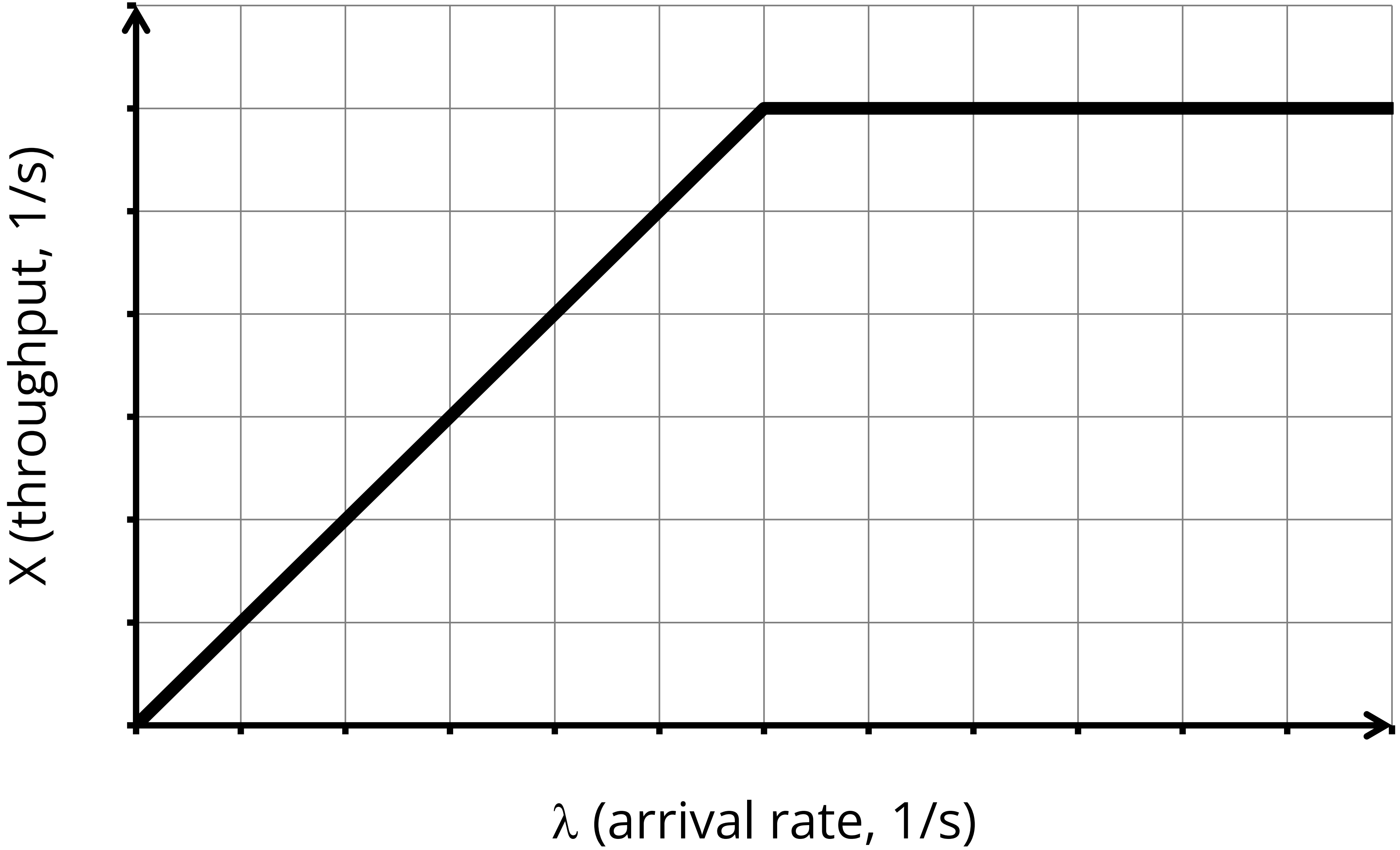
Denial of Service Attack

Thursday, August 6, 2009 | By Biz Stone (@biz) 08/06/2009 - 15:00

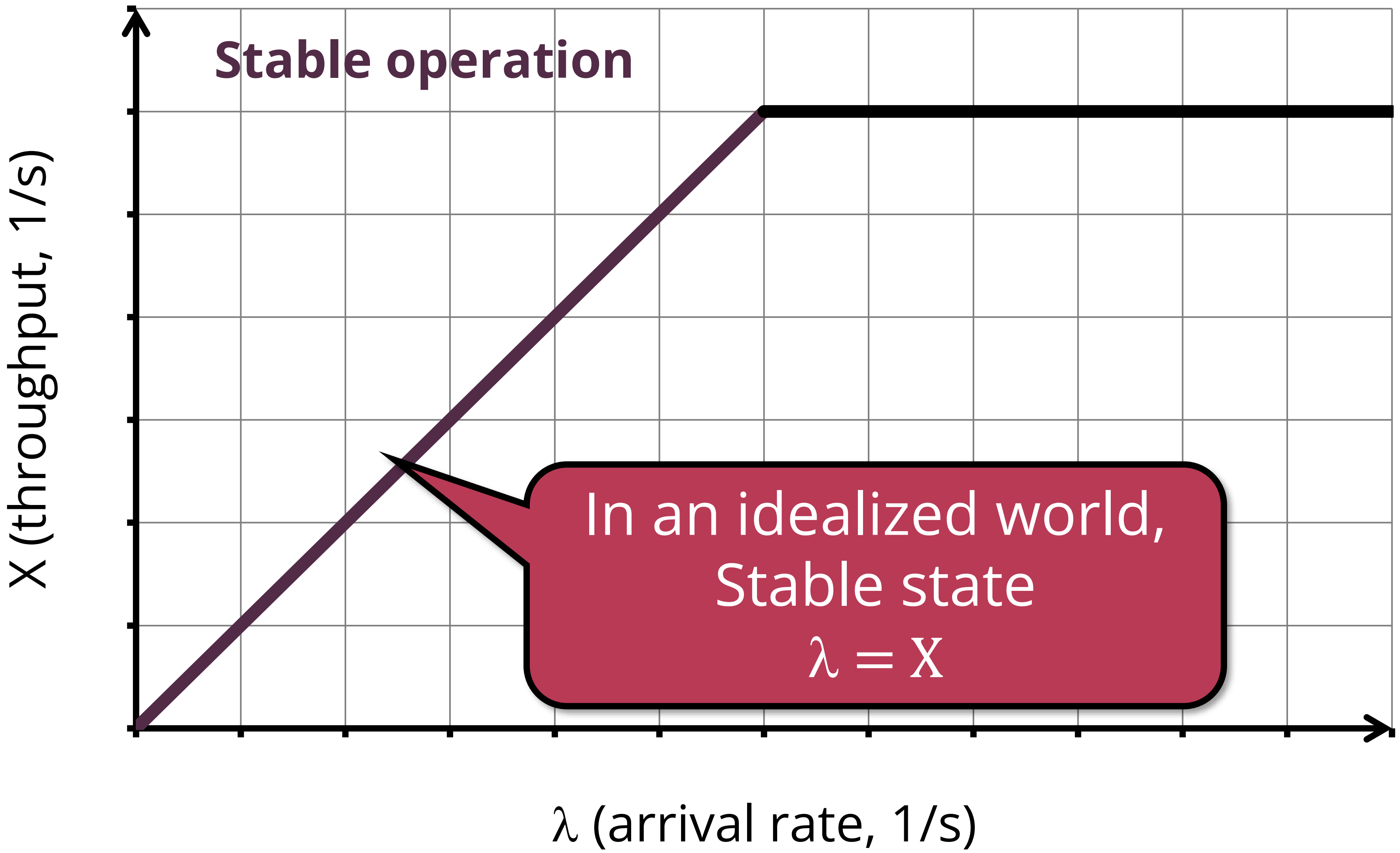
Tweet

On this otherwise happy Thursday morning, Twitter is the target of a denial of service attack. Attacks such as this are malicious efforts orchestrated to disrupt and make unavailable services such as online banks, credit card payment gateways, and in this case, Twitter for intended customers or users. We are defending against this attack now and will continue to update our status blog as we continue to defend and later investigate.

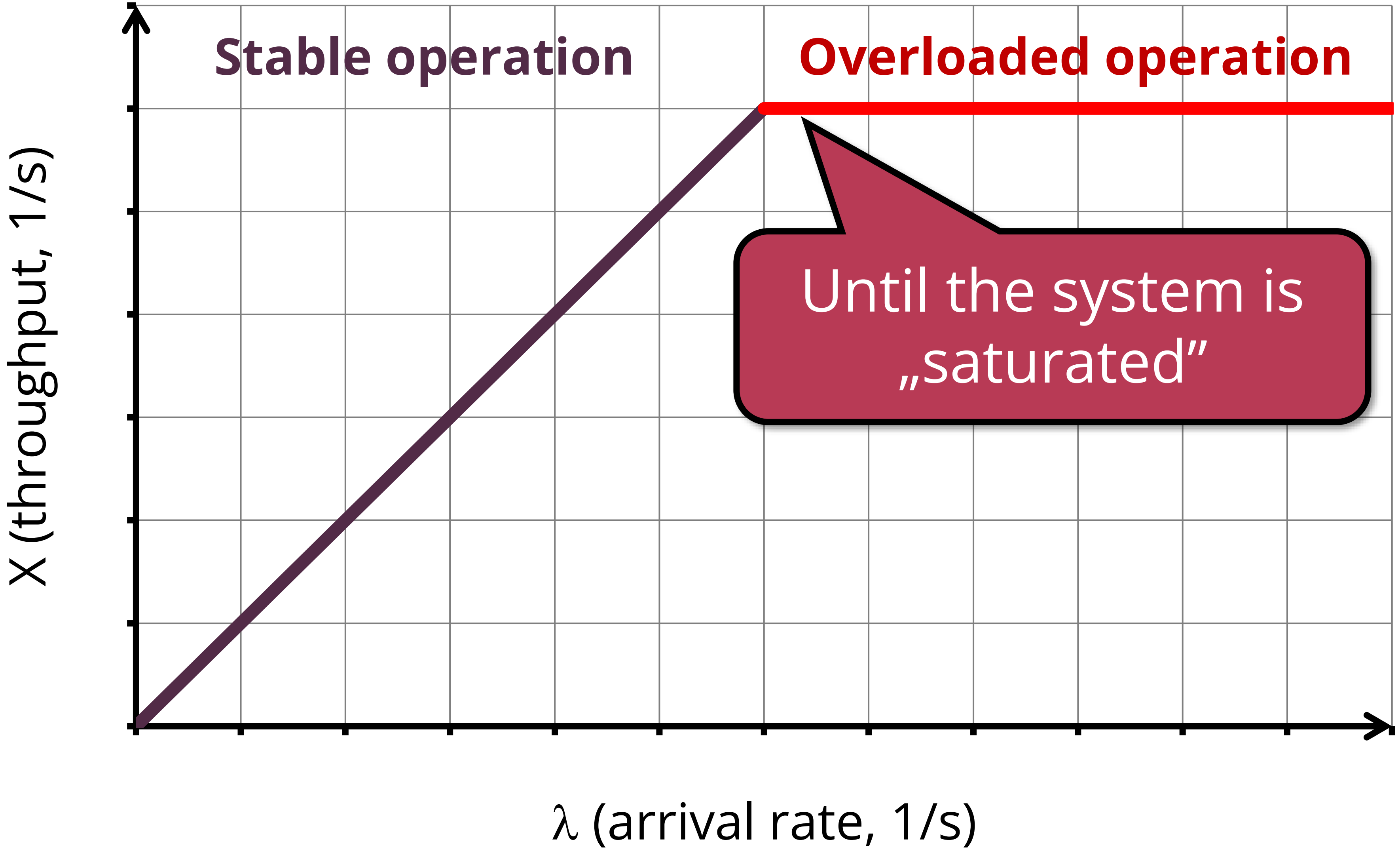
Load Diagram



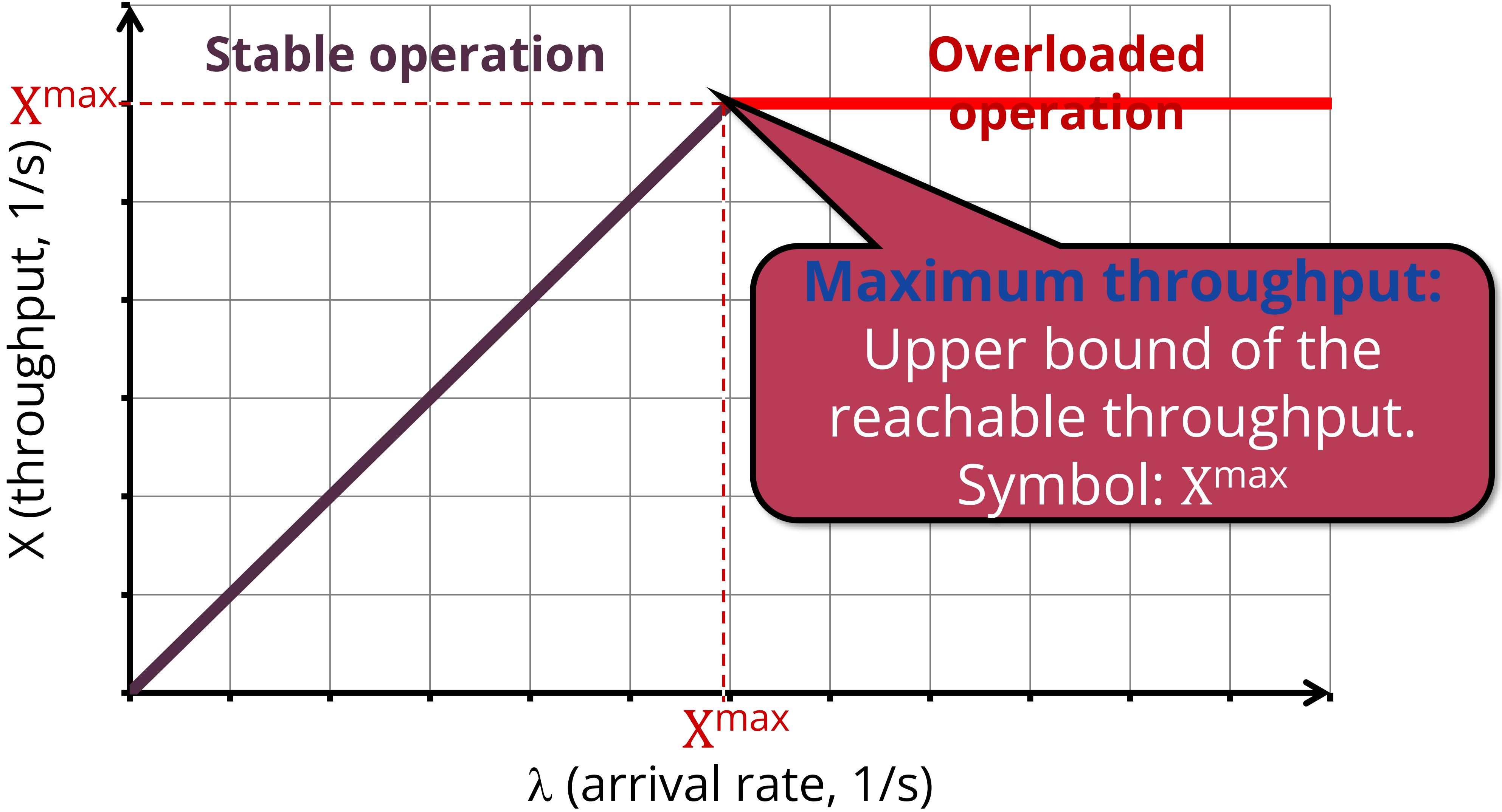
Load Diagram



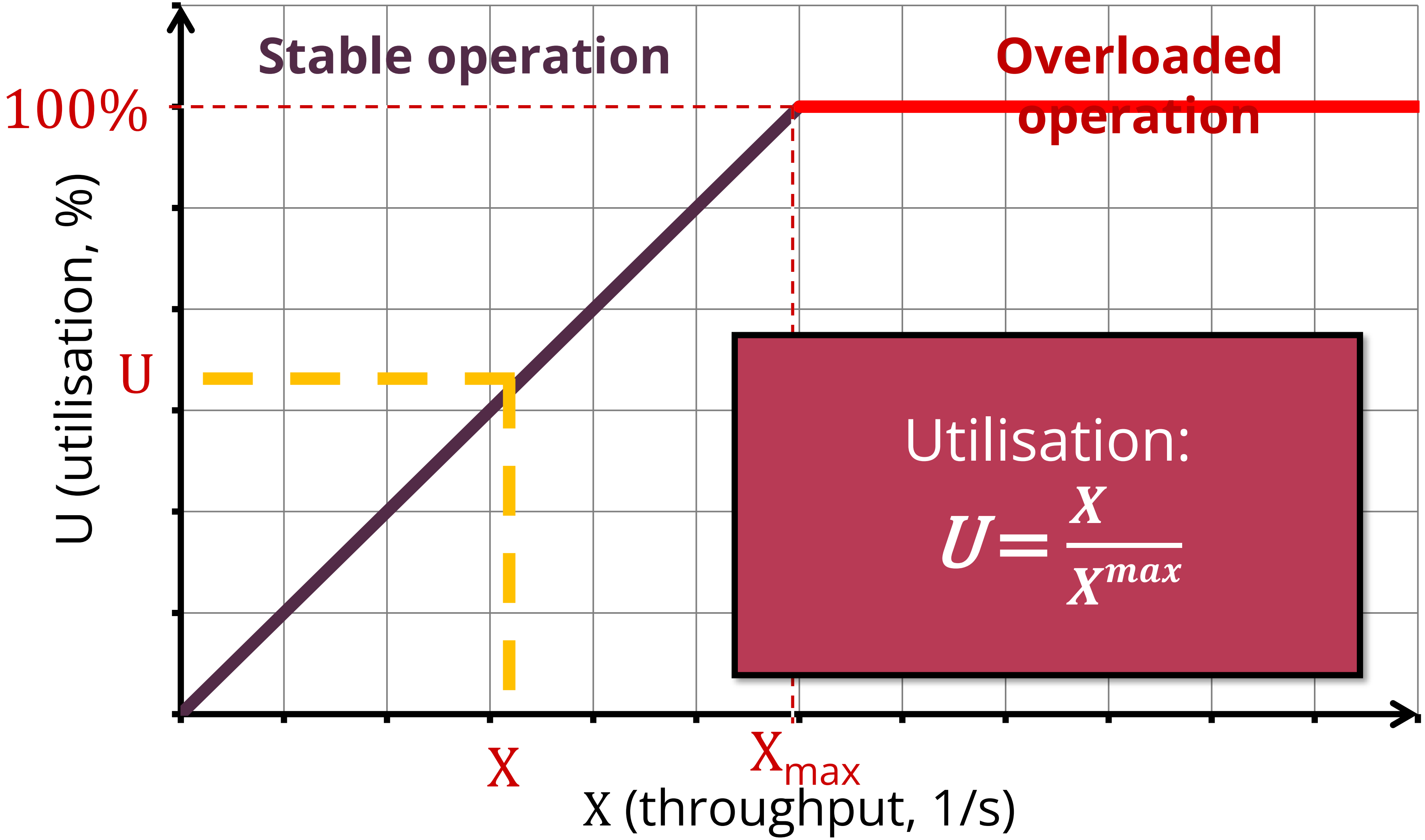
Load Diagram



Maximum Throughput



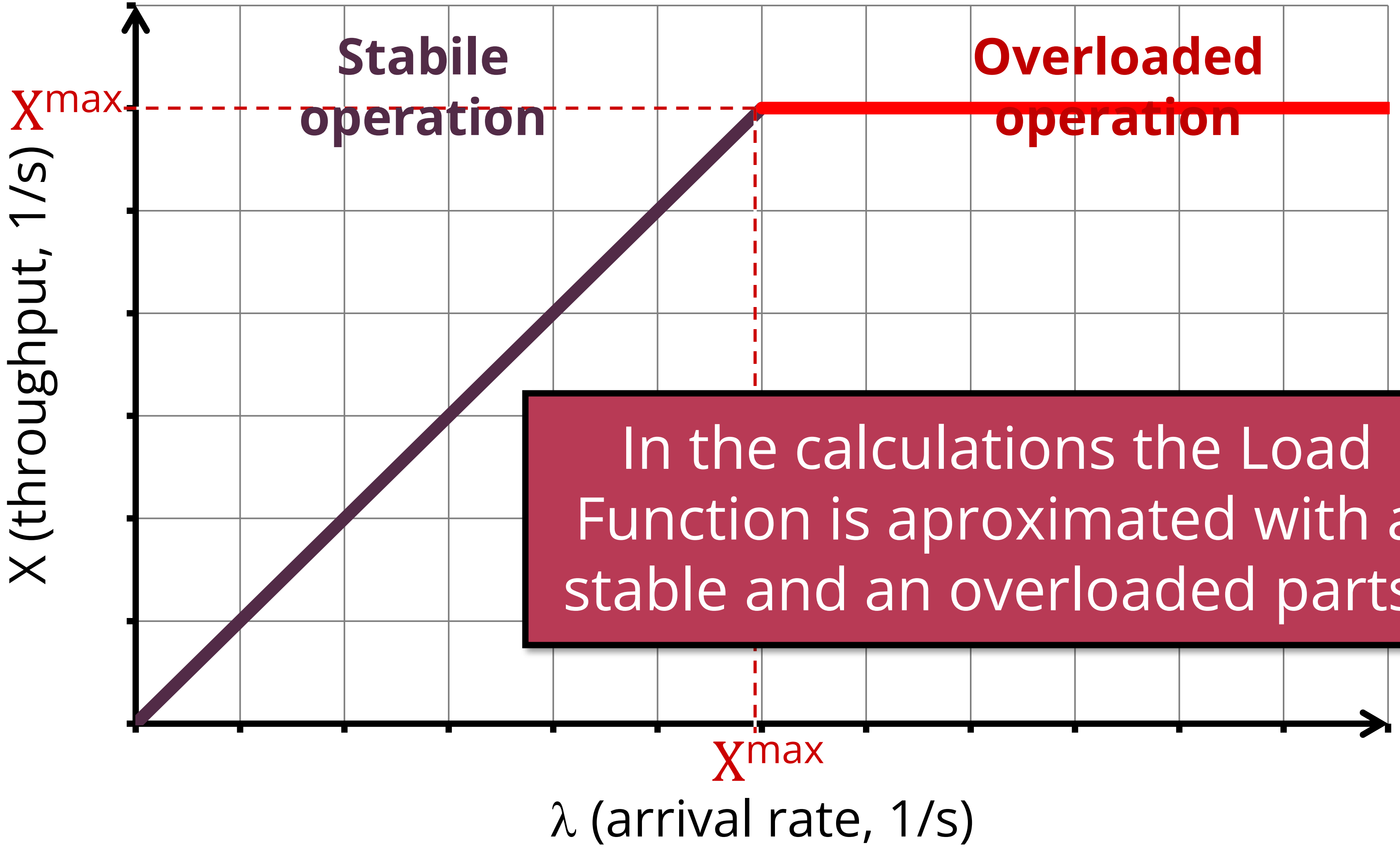
Utilisation



Utilisation:

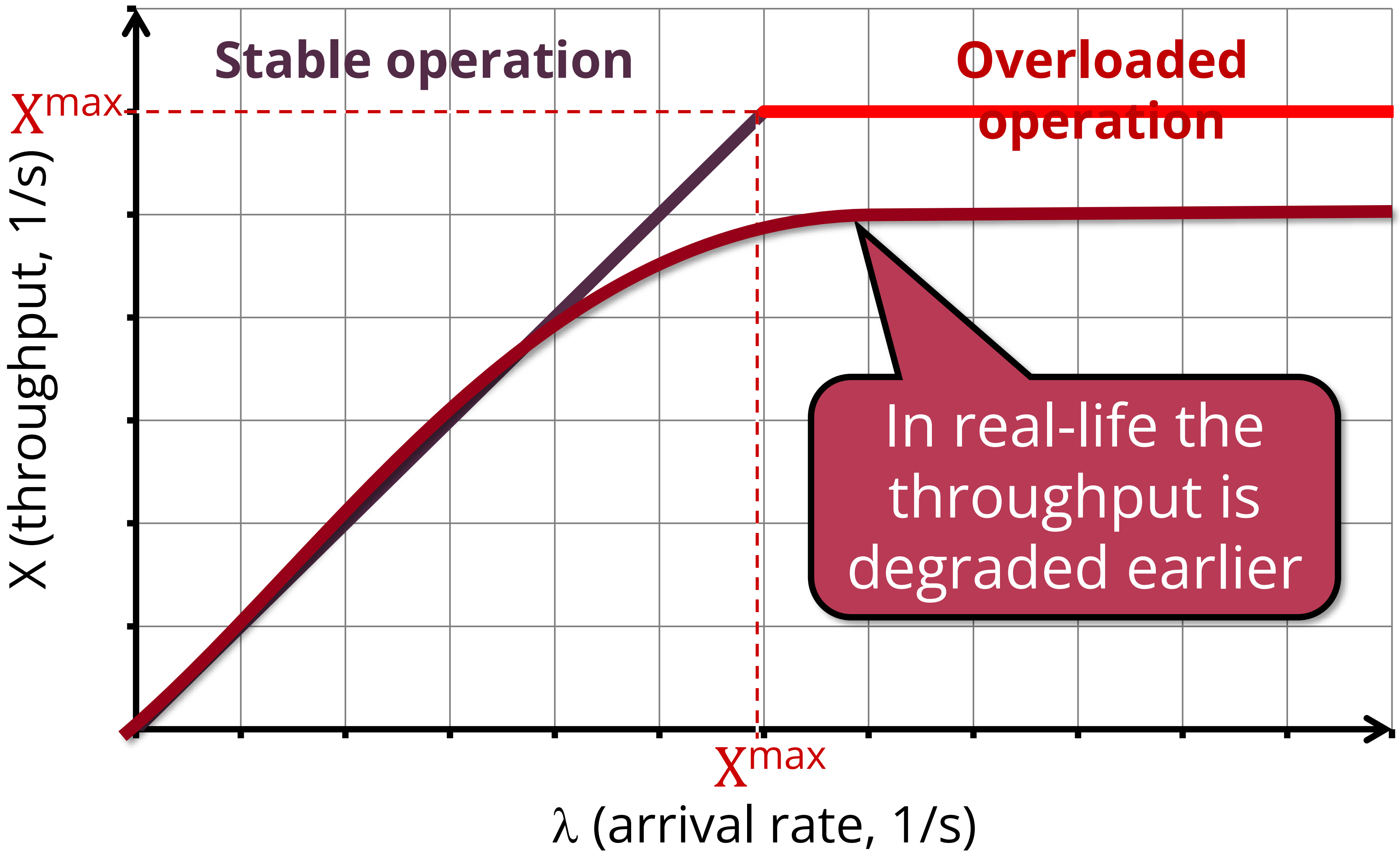
$$U = \frac{X}{X^{max}}$$

Approximative Load Function

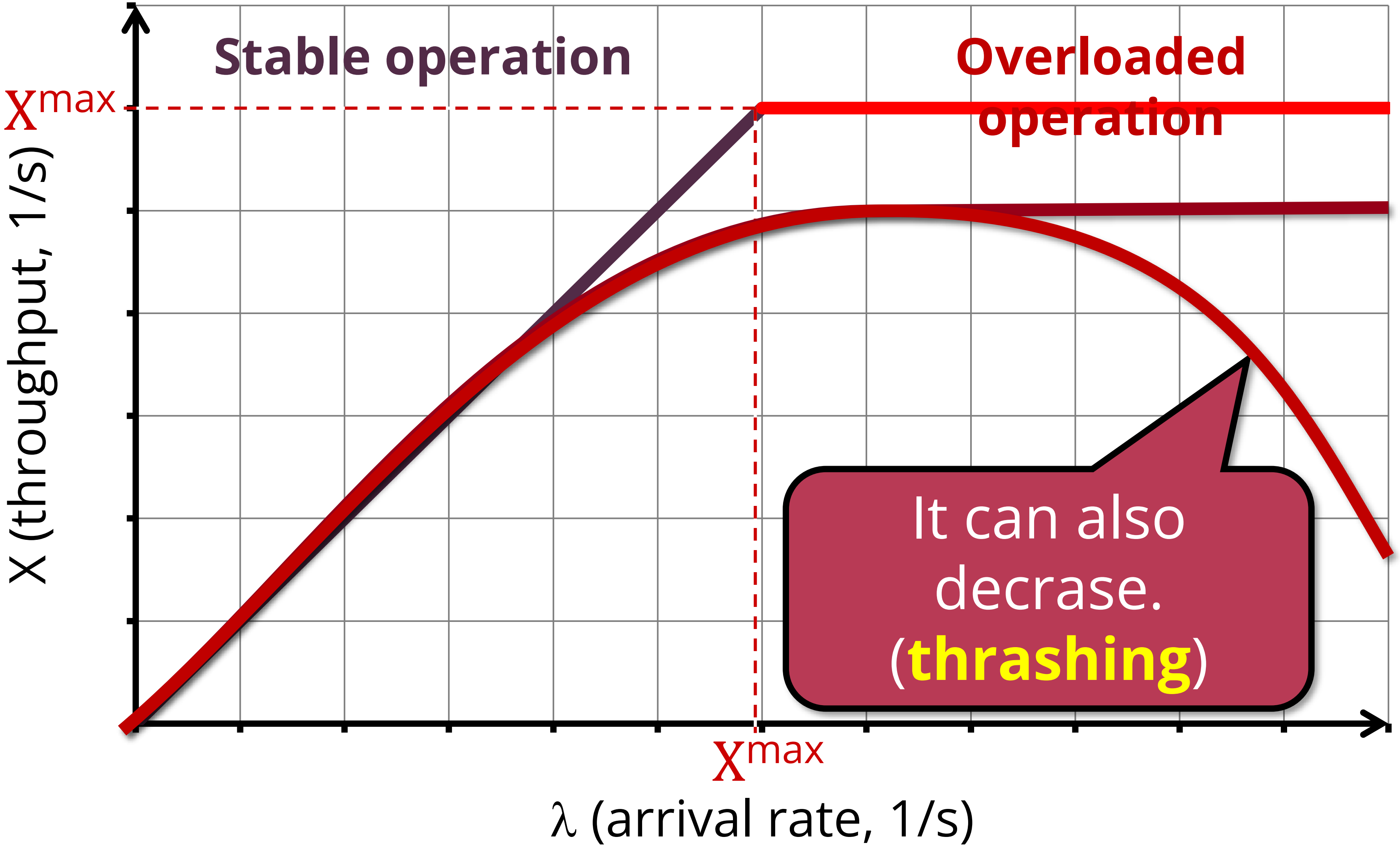


In the calculations the Load Function is aproximated with a stable and an overloaded parts.

Real Load Diagram



Real Load Diagram



Definitions

Maximum throughput: **maximal reachable** throughput

– Symbol: X^{max}

Utilisation: **ratio** of the actual and the maximum throughput

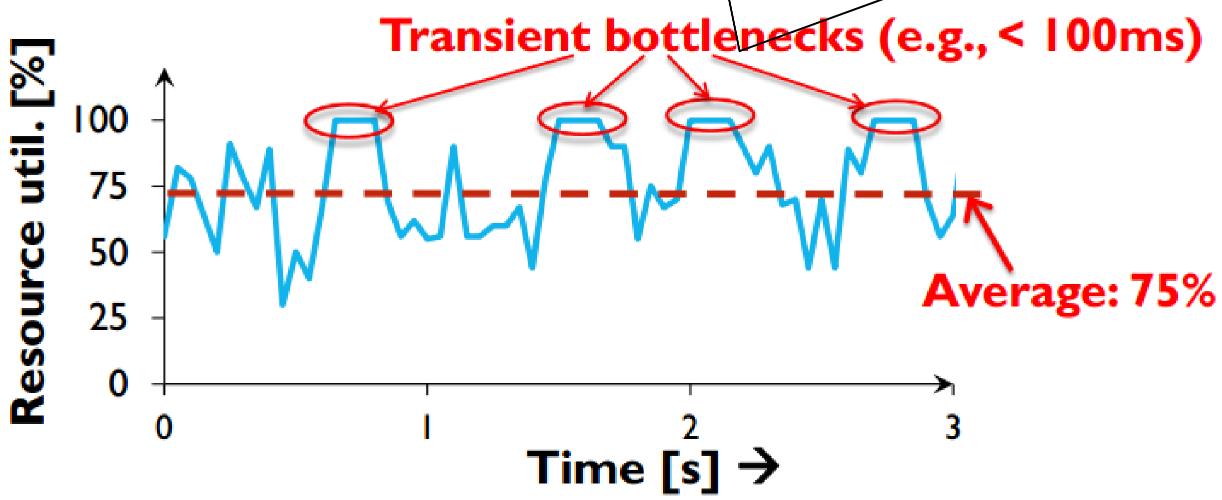
– Symbol: $U = \frac{X}{X^{max}}$

Thrashing: throughput decreases during overloaded operation

Effects of Load Fluctuation

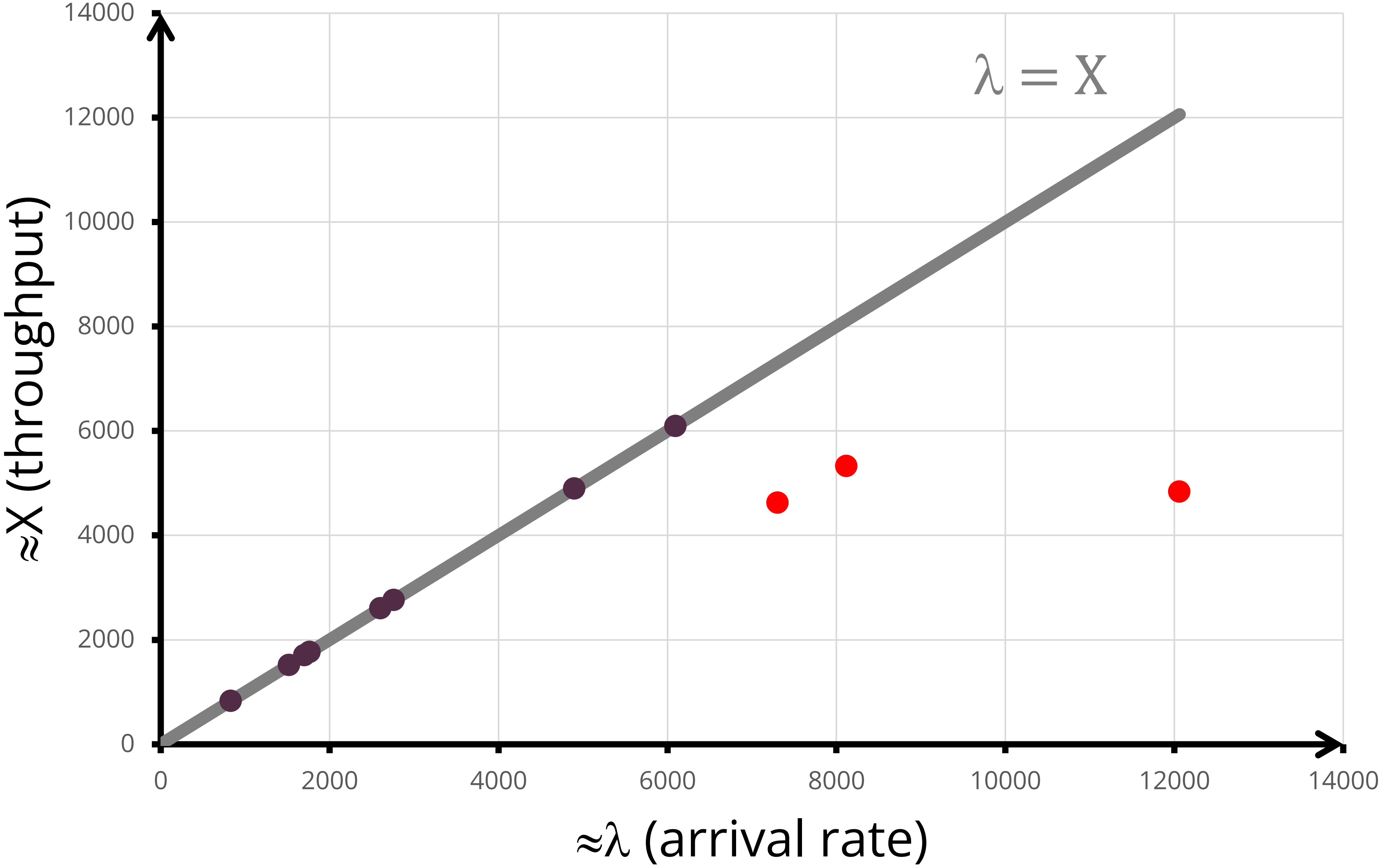
- Average values vs. **real load**

In case of an overloaded system the waiting time can **2-3 magnitude higher**



Qingyang Wang, Yasuhiko Kanemasa, Jack Li, Deepal Jayasinghe, Toshihiro Shimizu, Masazumi Matsubara, Motoyuki Kawaba, Calton Pu, "Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis", In Proc. of the 33rd International Conference on Distributed Computing Systems (**ICDCS'13**), Philadelphia, Pennsylvania, July 2013.

Outlook: Data Visualization



Micro-Benchmark

Micro-Benchmark Overview

- **Goal:** evaluate the performance of **small code fragments**
- Small but **important** piece of code

We should forget about small efficiencies, say about 97% of the time:
***premature optimization** is the root of all evil"*
Donald Knuth

- In a highly controlled environment
  - Avoid optimization by compilers

Examples (from Refinery)

- We know that a hash function is critical in our application
  - Some hash functions are **slower** but more effective
  - Others are **quicker** but they clash more
  - How to evaluate the performance of each hash function?

- Which Map implementation should I use?

  Google Guava IntObjectMap **vs** *good old java.util.HashMap*?

- New data structure, how to evaluate its performance?

How to execute those measurement precisely?

JMH: Java Microbenchmark Harness

- Library to execute precise Microbenchmarks
- It provides:
  - Measurement environment
  - Annotations to design measurements
  - Automated measurement execution

- It provides strict framework
- To avoid typical errors

Benchmark types

- We can annotate methods as benchmarks

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
public void method() { /* Do something */ }
```

- Supported modes:
  - **Throughput:** Operations per unit of time
  - **SampleTime:** Time distribution, percentile estimation
  - **SingleShotTime:** Time the single execution
- Example output:

```
Benchmark    Mode   Samples      Score-Idea      Score-Term   (MAX - MIN)/MAX   Units
measure     thrpt        25   341919594.645   344548681.629             0.007   ops/s
```

https://github.com/artyushov/idea-jmh-plugin/blob/develop/research/results.md

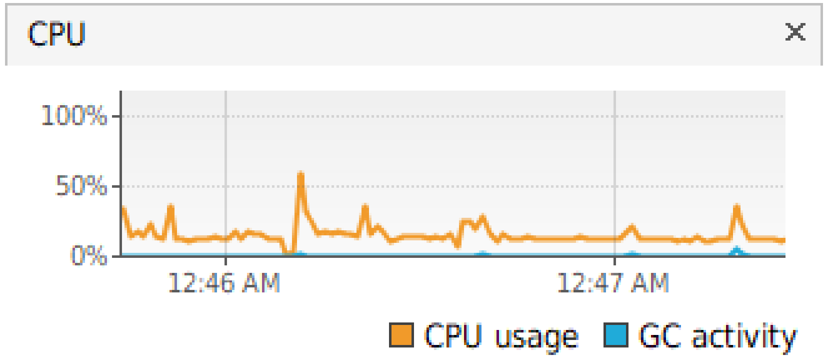
Warmup

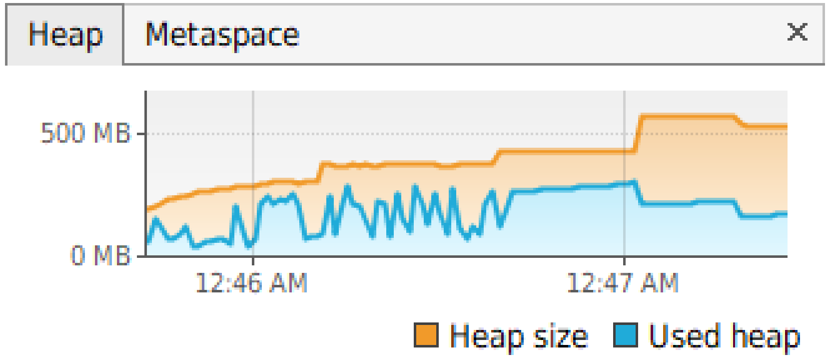
- In managed environment (e.g. Java), the runtime of a method **changes over time**
- We want to measure **peak performance**
- **Class loading:** for the **first time**, a lot of classes class need to be loaded (e.g. Refinery is 102MB of pure *.class* files).
- **Code optimization:** managed environments (e.g., Java JIT) collects statistics during runtime and optimizes code
  → better performance **after some iterations**

```
@Warmup(iterations = 20)
@Warmup(time = 1, timeUnit = TimeUnit.SECONDS)
```

Garbage collection

- In managed environments, GC is an automated process, which runs in the background.
- We need to avoid any interference.
- We need to measure the peak performance →
  - measure the system when there is enough memory, GC is not running
  - measure the system when in stable state, GC running in the background, evenly





How to measure memory

- How to measure memory requirements in a GC environment?
- Performance ⬄ Memory interaction.

- Lower the memory limit until the application crashes ☺.
  - **-Xmx16g** ☑
  - **-Xmx8g** ☑
  - **-Xmx4g** ☑
  - **-Xmx2g** ☒

  Answer: memory requirement is between 2GB and 4GB
- This is a separate measurement.

Eliminating compiler optimization

- Calling the same function with the same parameters multiple times

```
public int calculateHash(Node node) { … }
```

```
for (int i = 0; i < iterations; i++) {
        calculateHash(Node node);
}
```

- Compiler optimization, caching value →
  After the second calculation, the runtime is ~0 ms.
- Solution: “black hole” object, that prevents caching

```
Blackhole blackhole = …
for (int i = 0; i < iterations; i++) {
        blackhole.consume(calculateHash(Node node));
}
```

Comparing multiple variants

- Run multiple measurements with multiple variations
- Like parametric tests

```
@State(Scope.Benchmark)
public class ImmutablePutExecutionPlan {

        @Param({ "100", "10000" })
        public int nPut;

        @Param({ "32", "1000", "100000" })
        public int nKeys;

        @Param({ "2", "3" })
        public int nValues;
        …
}
```

Macro-Benchmark

Macro-Benchmark Overview

- **Goal:** evaluate the performance of **an algorithm**
- The **whole application** is evaluated.

- **Sub-Goal 1:** discover and identify bottlenecks

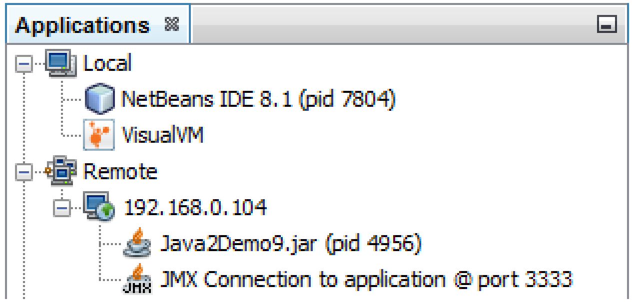
The same relevant quote:

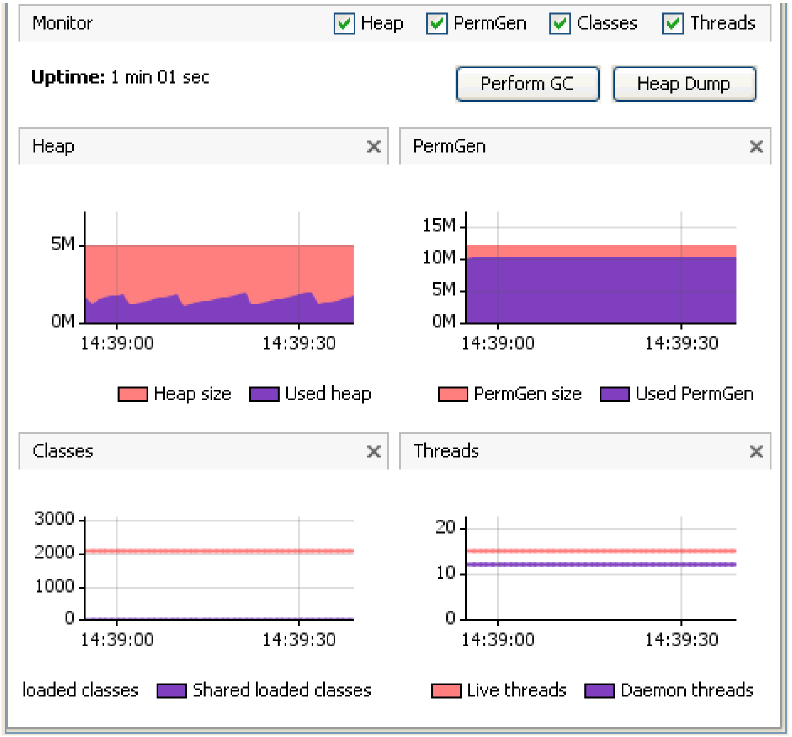
We should forget about small efficiencies, say about 97% of the time:
***premature optimization** is the root of all evil“*
Donald Knuth

Sub-Goal 2: check the performance in increasing problem size

Profiling: identifying bottlenecks

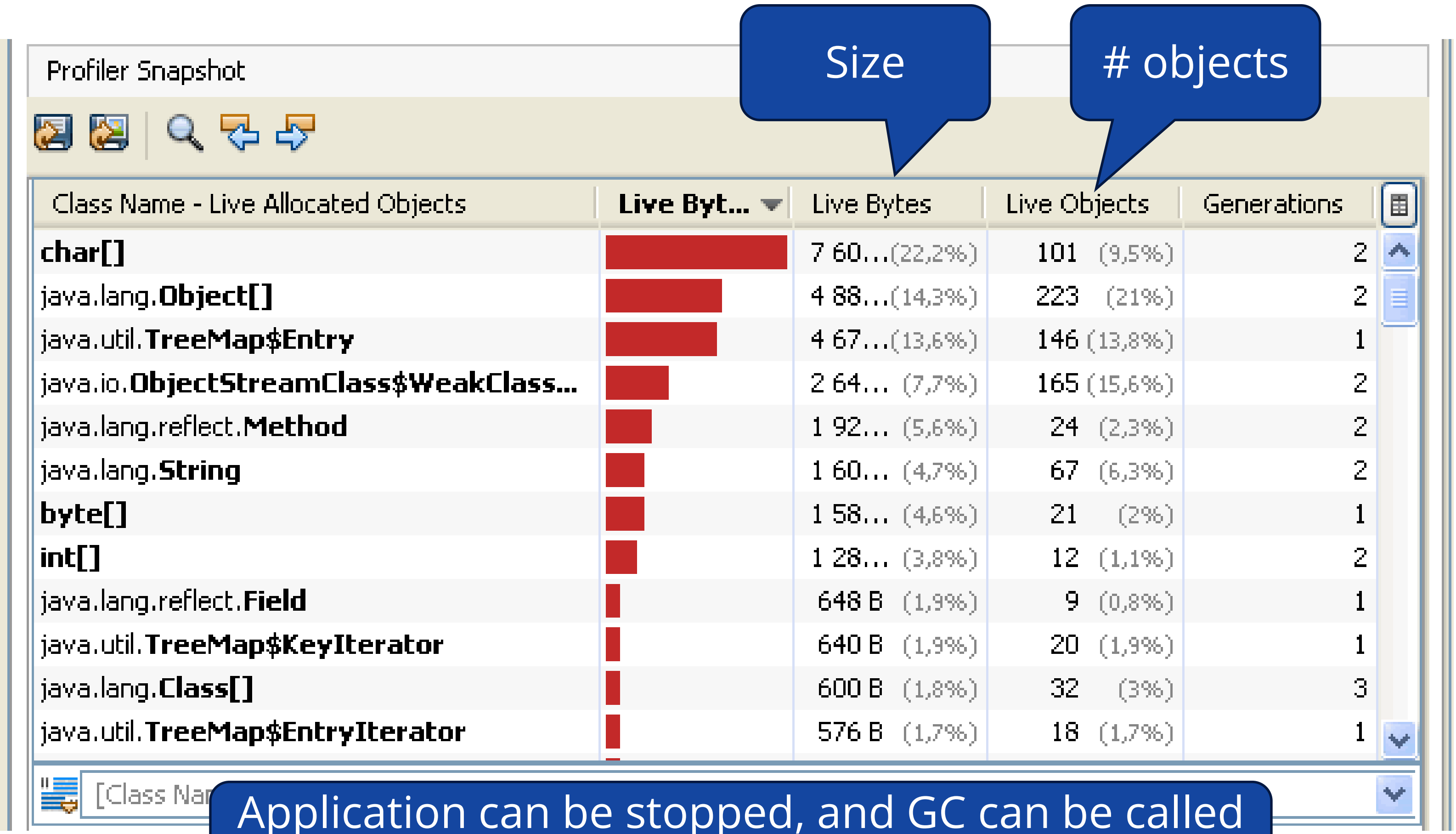
- VisualVM is a lightweight profiling tool
- Attaches itself to an application, and measures





- Saves and visualizes measurements

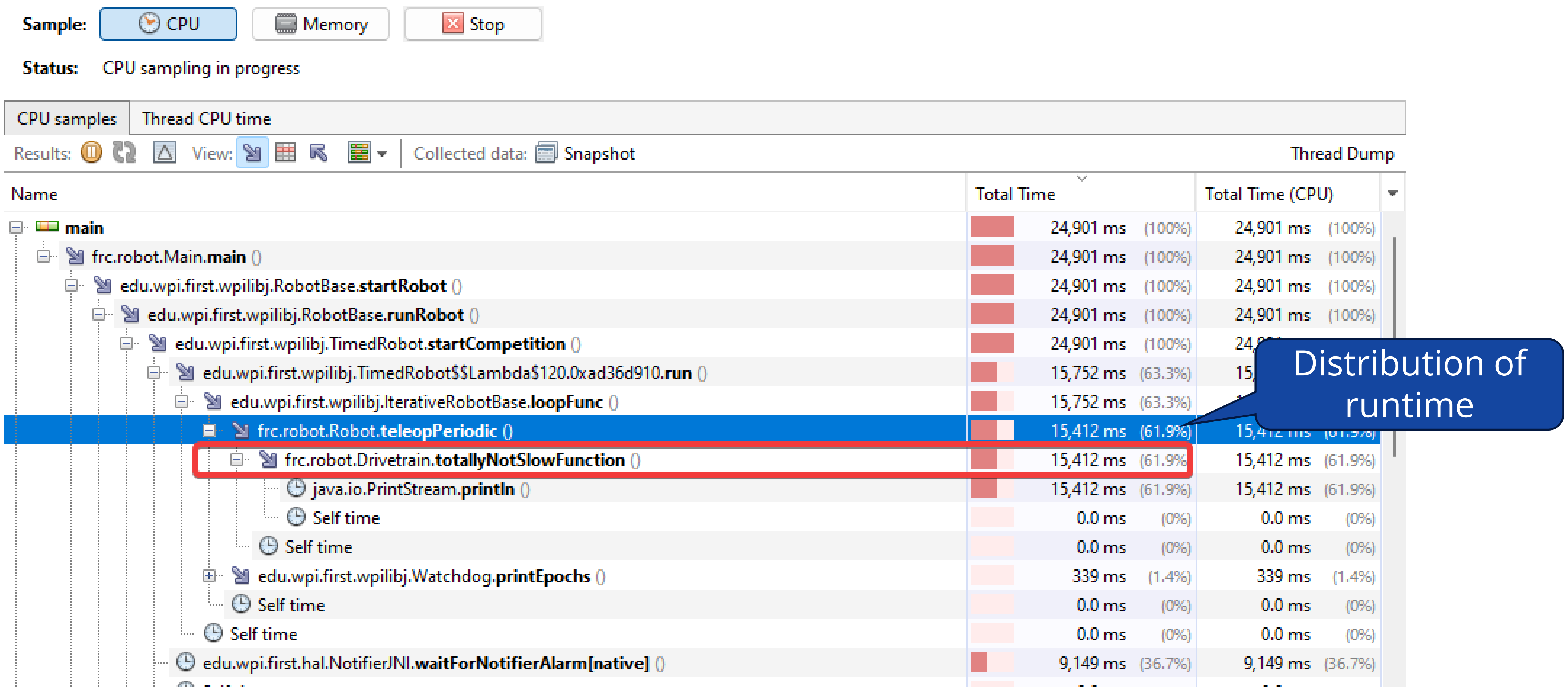
Memory profiling



Class Name - Live Allocated Objects	Live Byt...	Live Bytes	Live Objects	Generations
char[]		7 60...(22,2%)	101 (9,5%)	2
java.lang.Object[]		4 88...(14,3%)	223 (21%)	2
java.util.TreeMap$Entry		4 67...(13,6%)	146 (13,8%)	1
java.io.ObjectStreamClass$WeakClass...		2 64... (7,7%)	165 (15,6%)	2
java.lang.reflect.Method		1 92... (5,6%)	24 (2,3%)	2
java.lang.String		1 60... (4,7%)	67 (6,3%)	2
byte[]		1 58... (4,6%)	21 (2%)	1
int[]		1 28... (3,8%)	12 (1,1%)	2
java.lang.reflect.Field		648 B (1,9%)	9 (0,8%)	1
java.util.TreeMap$KeyIterator		640 B (1,9%)	20 (1,9%)	1
java.lang.Class[]		600 B (1,8%)	32 (3%)	3
java.util.TreeMap$EntryIterator		576 B (1,7%)	18 (1,7%)	1

Application can be stopped, and GC can be called manually

Performance profiling



https://docs.wpilib.org/en/stable/docs/software/advanced-gradlerio/profiling-with-visualvm.html

Two supported measurement method

- **Sampling: periodically** checks the stack and **makes snapshots**
  - Statistics are calculated on the snapshots
  - Hides fast methods, statistical guarantees
  - But realistic distribution
  - Typically the good solution

- **Profiling: changes** the code and **reports each call**.
  - Precisely captures the runtimes
  - But changes the code → we are not measuring the target application
  - Huge performance decrease, not realistic runtime.

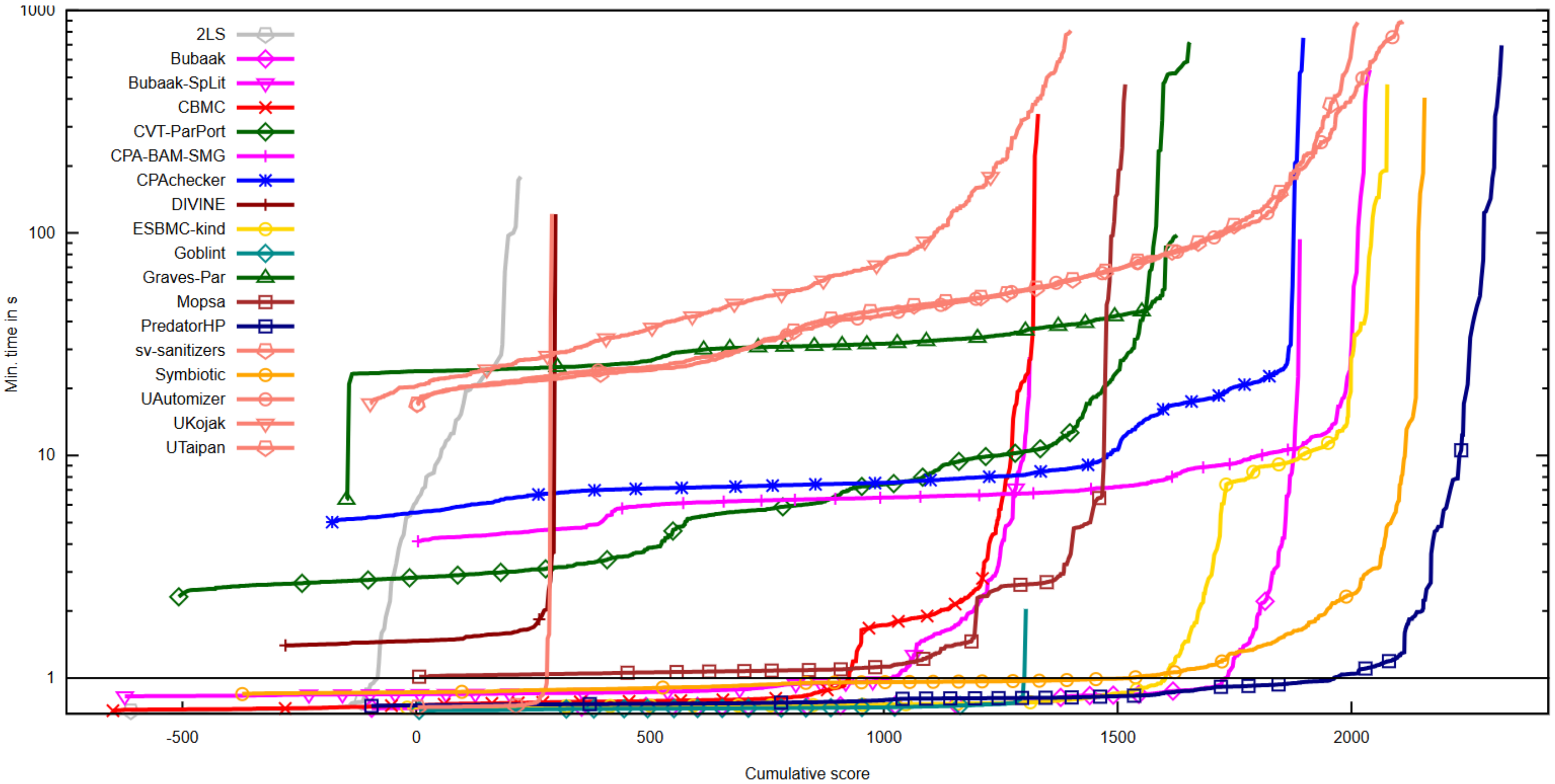
How to create a Benchmark?

Important aspects of benchmark

- **Relevant:** realistic data
- **Independent:** multiple implementations can use it
- **Scalable:** small problem ⇔ big problem
- **Portable & Reproducible:** the measurement can be executed

Example: BenchExec

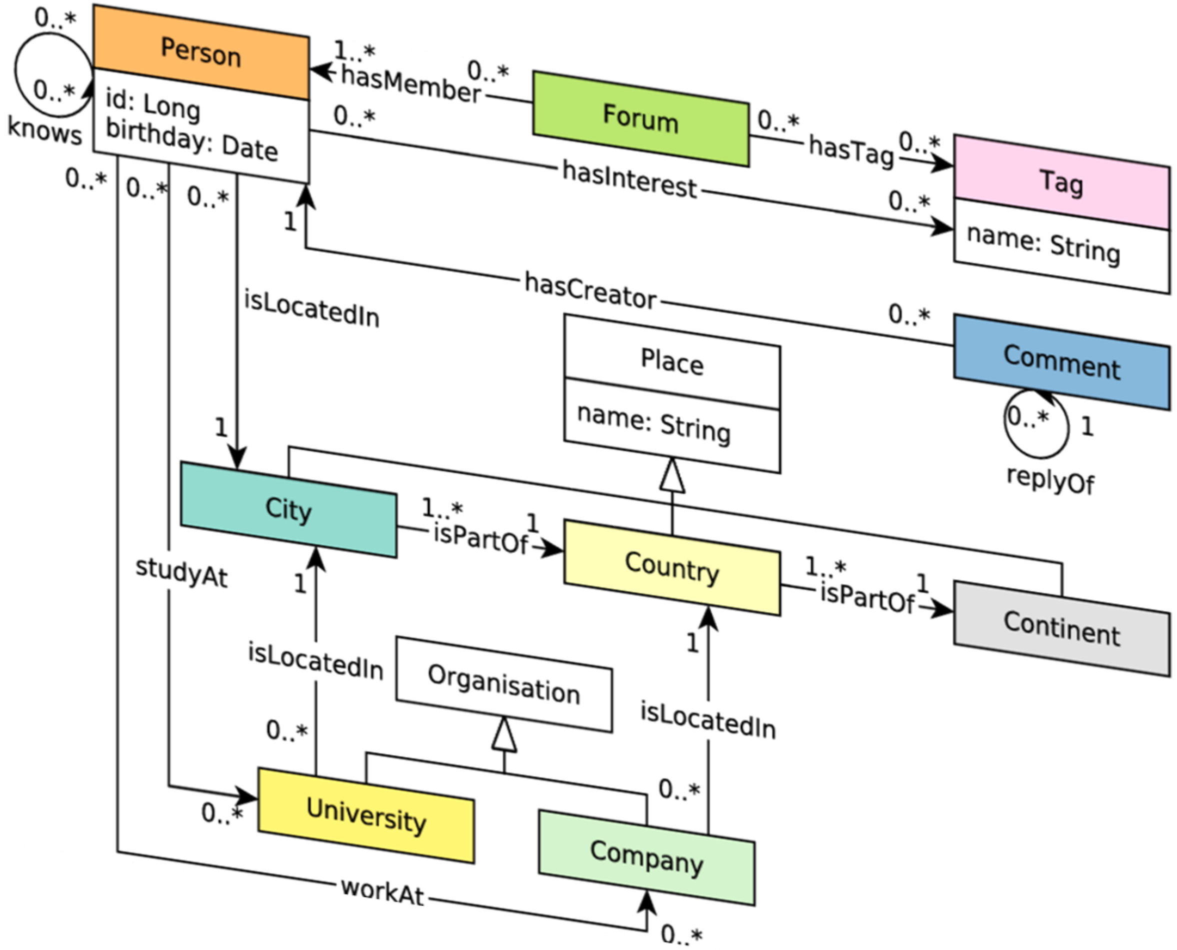
- SV-COMP 2024: model checking benchmark
- Automated evaluation on large set of problems, competitors



LDBC benchmark

- Graph query benchmark
- Bigger and bigger graphs vs
- Complex queries

- Audited benchmark ($)



How to Document?

Typical documentation of evaluation

- This can be a research report, TDK, thesis work, research paper
- There are typical sections

1. **Research Questions.**

Simple, straightforward question. Not yes-no.

RQ1 How does our graph solver scale (in time and model size) when generating consistent models of increasing size?
RQ2 How does our approach scale (in time and model size) compared to the widely used model finder Alloy [21]?
RQ3 How do the different steps of the exploration influence performance of the graph solver?

Measurement environment

2. Selected domains

Selected example domains should be independent and representative.

3. Benchmarking setup

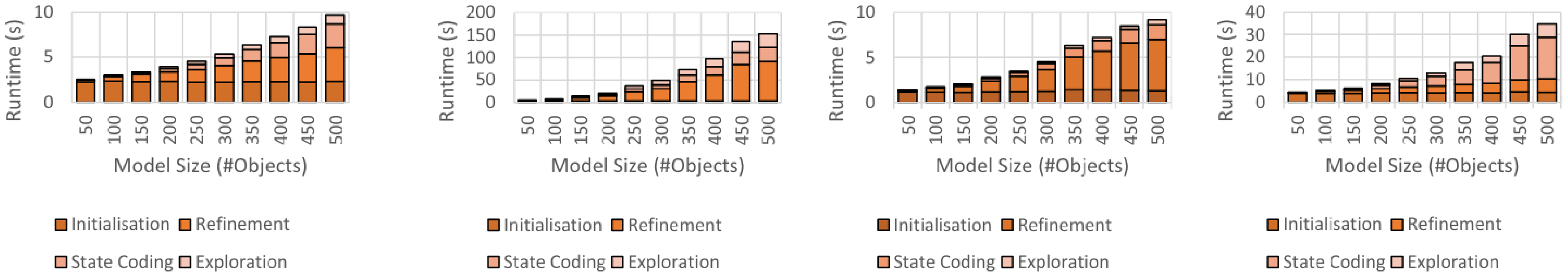
How is the evaluation executed?

Machine specification, software configuration, steps.

Communicating the results

4. Experimental results

Figures are explained here. Just simple explanation.



5. Research results

RQ2. *According to the results (see Figure 8e–8j and Figure 8k), our approach scales much better as it generates models 1-2 orders of magnitude larger than Alloy could handle regardless of the back-end SAT solver which only had little impact on scalability.*

Threats to validity

6. Threats to validity

- **Internal:** confidence in measurements
- **External:** confidence in generalization
- **Construct:** confidence in the relevance of measured metrics

Summary

Summary

- Generic overview
- Micro-benchmark
- Macro-benchmark

- How to create a benchmark?
- How to write an evaluation section?