Automated Refactoring of Performance and Concurrency Anti-Patterns

Patrycja Wegrzynowicz

CTO, Yonita, Inc.

About Me

- Programmer at heart
- Researcher in mind
- Speaker with passion
- Entrepreneur by need

@yonlabs

YONITA

Agenda

- Motivation and performance issues
- The concept
  - Yonita Optimizer
- 3 anti-patterns
  - Performance
  - Concurrency
  - Stats and demo
- Hibernate Core Refactored

YONITA

On most Intels Java is as fast or faster than C

William O'Mullane, European Space Astronomy Centre

Jazoon 2010

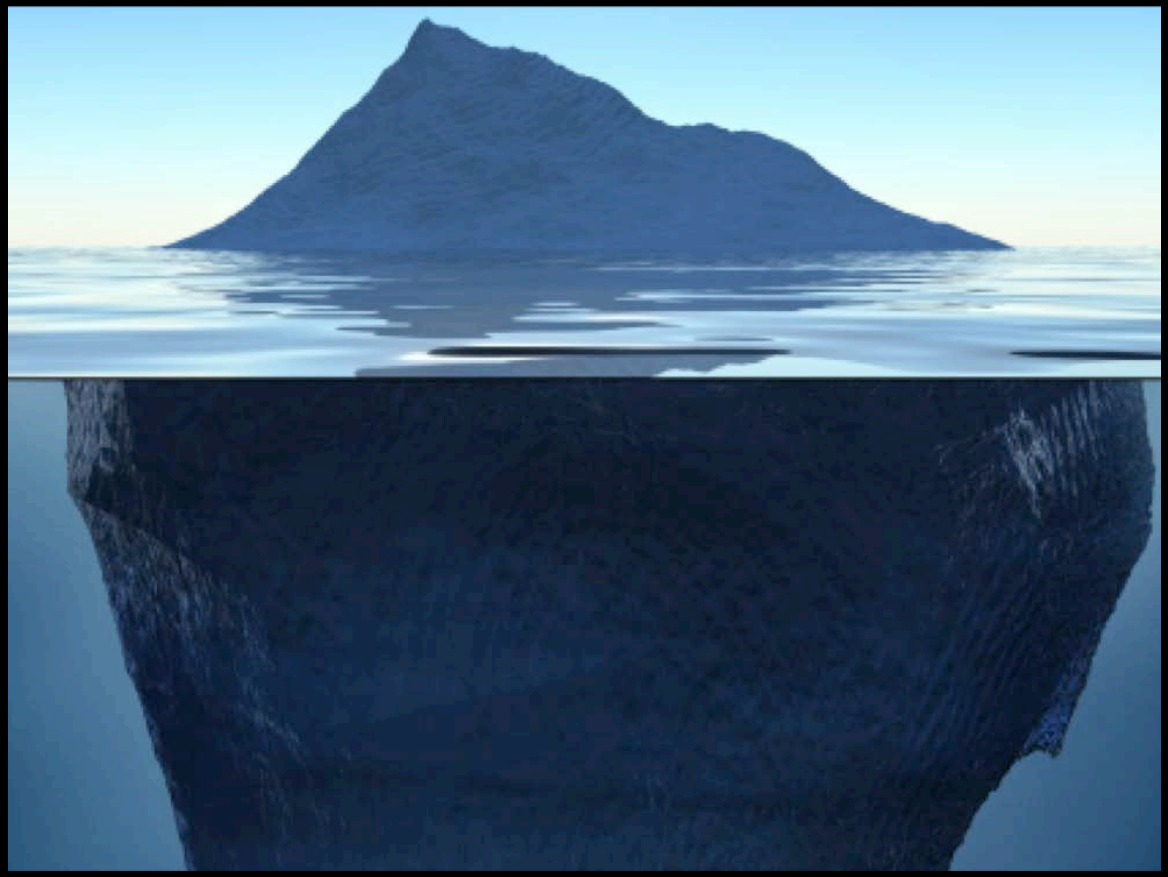
Where is the Problem?

- Root causes
  - Bad architecture, design, or code
  - Wrong configuration (app/db server, OS, RDBMS)
  - Rarely insufficient hardware
- Bottlenecks usually unnoticed until happen
  - Special conditions to happen: size of data, number of users
  - The later discovered the more expensive to solve

Hello World in cloud is involve 1 load balancer, 3 web server and 2 database server

DevOps_Borat, Twitter

YONITA



YONITA

YONITA

Think? Why think! We have computers to do that for us.

Jean Rostand

Yonita Optimizer

- Pre-compile source to source optimizer
- Refactors inefficient code structers into more efficient ones
  - At the moment supports subset of variants of 5 anti-patterns
- Not a profiler!
  - Automated
  - Sources
  - Does not look for bottlenecks, only optimizes what it's able to optimize

Anti-Patterns

1, 1, 2, 3, 5, 8, 13, 21,…

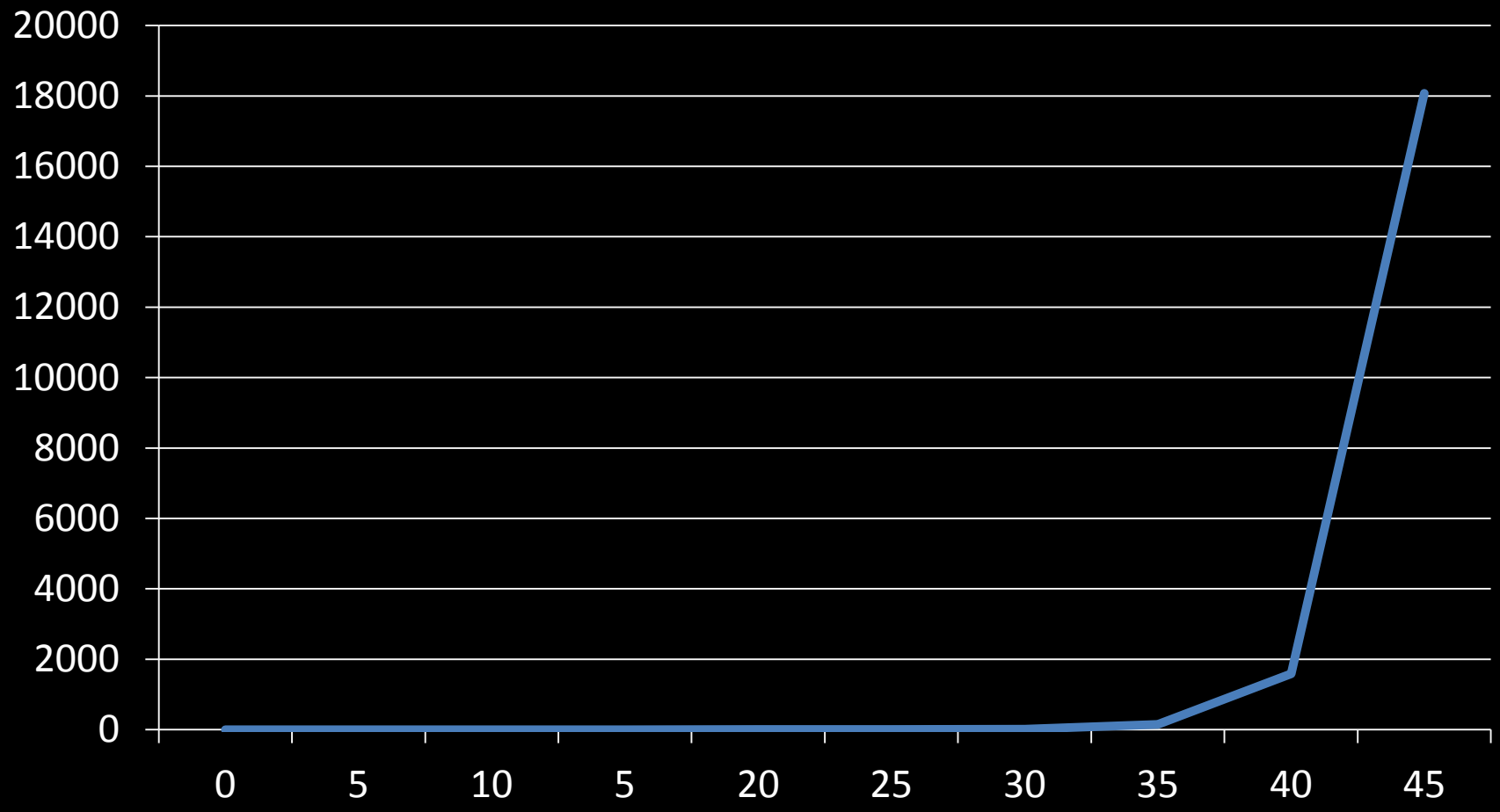
Fibonacci Sequence

- 1, 1, 2, 3, 5, 8, 13, 21,…
- Fib(n) = Fib(n-1)+Fib(n-2)
- Fib(1) = Fib(0) = 1

Fibonacci Sequence: Java

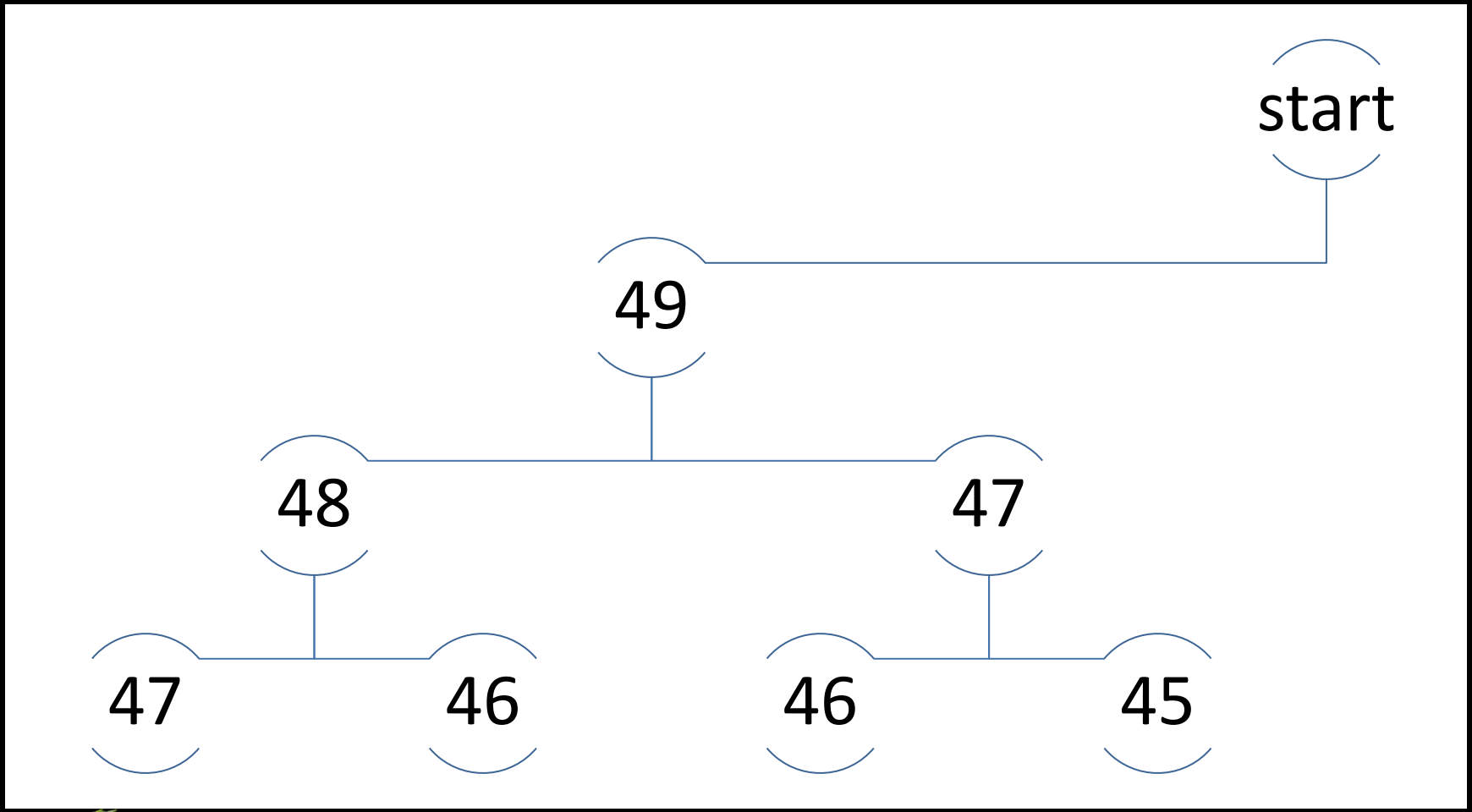
```
public long fibonacci(long n) {
        if (n <= 1) return 1;
        return fibonacci(n-1) + fibonacci(n-2);
  }
```

Fibonacci Sequence: The Problem



YONITA

Fibonacci Sequence: Computational Tree



Anti-Pattern #1: Redundant Work

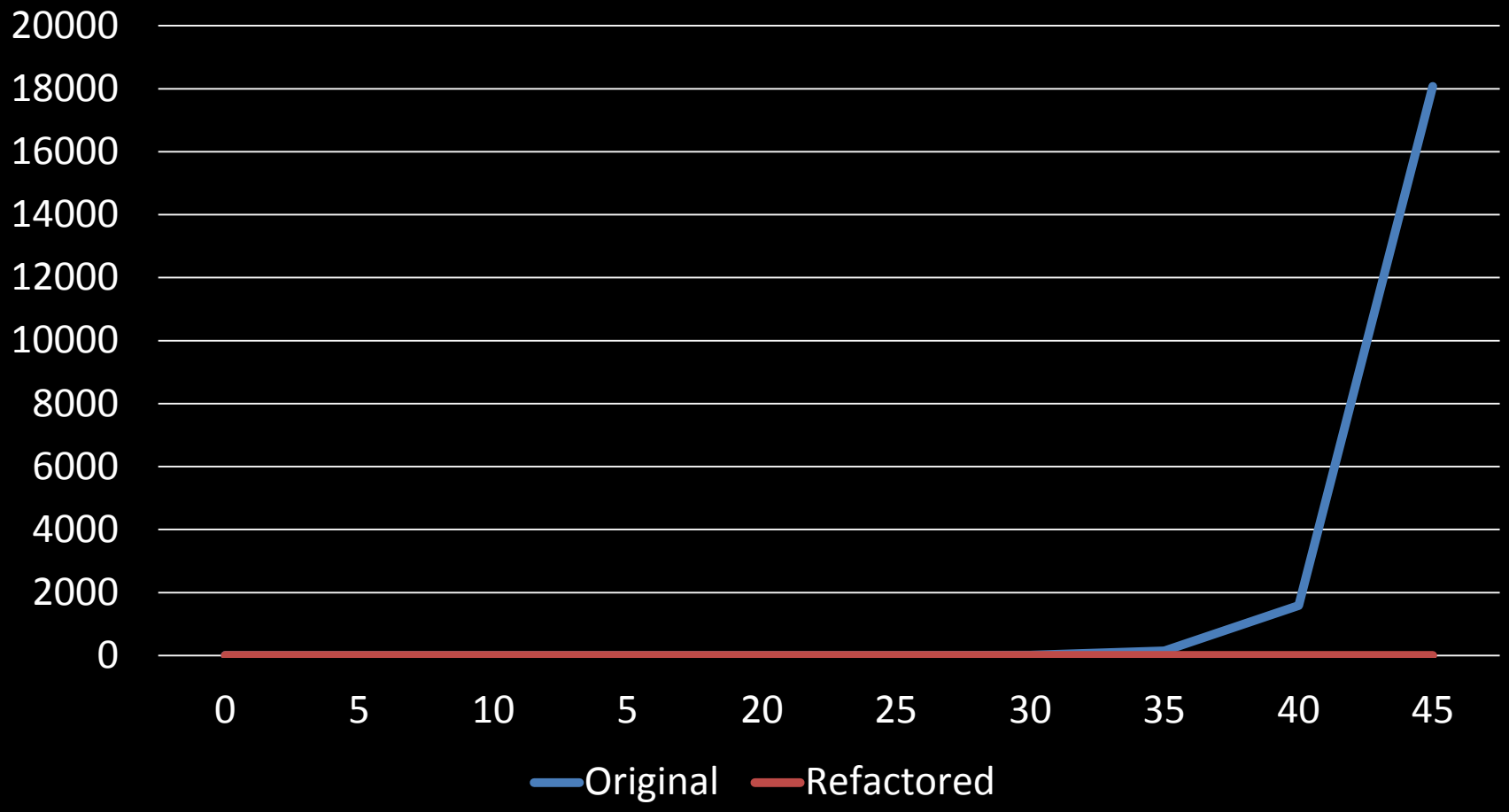
- Description
  - A time-consuming method computes the same many times in a single execution path
- Consequences
  - A slower execution time since the time-consuming operation is performed multiple times.
- Refactored solution
  - Call the heavy method only once and store the result for further re- use

Fibonacci Refactored

```
Map<Long, Long> __cache1 = new HashMap<Long, Long>();

long fibonacci(long n) {
  if (__cache1.containsKey(n))
    return __cache1.get(n);
  if (n == 0 || n == 1) {
    long __var1 = 1;
    __cache1.put(n, __var1);
    return __var1;
  }
  long __var2 = fibonacci(n-1) + fibonacci(n-2);
  __cache1.put(n, __var2);
  return __var2;
 }
```

Fibonacci Sequence: The Problem



YONITA

Refactoring of Redundant Work

- Cache instance
  - A cache map instance per the anti-pattern method
  - Keys: arguments
  - Values: computed return values
- Refactored method
  - At the begining: check if the cache contains a value for given parameters
  - At th end: store computed value in the cache

Formalization of Redundant Work Preconditions of Refactoring

- Variant 1
  - Recursive functions with at least 2 recursive invocations
  - No random/temporal/external source dependencies
- Variant 2
  - At least 2 invocations in a ‘straight’ single execution path
  - The same object, the same arguments
  - No modication of the object dependent substate (used fields) between invocations
  - No side-effects
  - No random/temporal dependencies

Redundant Example: Variant 2

```java
public class UserBean {
    // ...
    public User getUser() {
        return userManager.getUser(userId);
    }
```

```
<h:outputText value="#{userBean.user.username}" />
<h:outputText value="#{userBean.user.firstName}" />
<h:outputText value="#{userBean.user.lastName}" />
<h:outputText value="#{userBean.user.email}" />
<h:outputText value="#{userBean.user.street}" />
<h:outputText value="#{userBean.user.city}" />
<h:outputText value="#{userBean.user.country}" />
...
```

Search

```
private ArrayList<Item> list = new ArrayList<Item>();

List<Item> findGreaterThan(int value) {
        List<Item> ret = new ArrayList<Item>();
        for (Item item : list) {
                if (item.isGreaterThan(value)) {
                        ret.add(item);
                }
        }
        return ret;
}
```

Anti-Pattern #2: One by One Processing

- Description
  - Overused linear search/processing
- Consequences
  - Slower performance
- Solution
  - Use smarter algorithms and/or data structures (binary search, sorted collections, map with precomputed search predicates)

Search Refactored

```
private List<Item> list = new ArrayList<Item>();
private List<Item> __var1 = new SortedList<Item>(new
__Comparator());

List<Item> findGreaterThan(int value) {
        return subList(__var1, value);
}
```

Password Cracking

```
static List<String> passwordsToCheck;

// launch 100 threads

// each thread

void run() {
    while (!passwordsToCheck.isEmpty()) {
                synchronized (passwordsToCheck) {
                    if (!passwordsToCheck.isEmpty()) {
                                String pwd = passwordsToCheck.remove(0);
                                checkPassword(pwd);
                    }
                }
    }
}
void checkPassword() {
}
```

YONITA

Anti-Pattern #3: Long Critical Section

- Desciption
  - Unnessesary code performed in a critical section
- Consequences
  - More like single-thread model
- Solution
  - Remove the code outside the critical section

Password Cracking Refactored

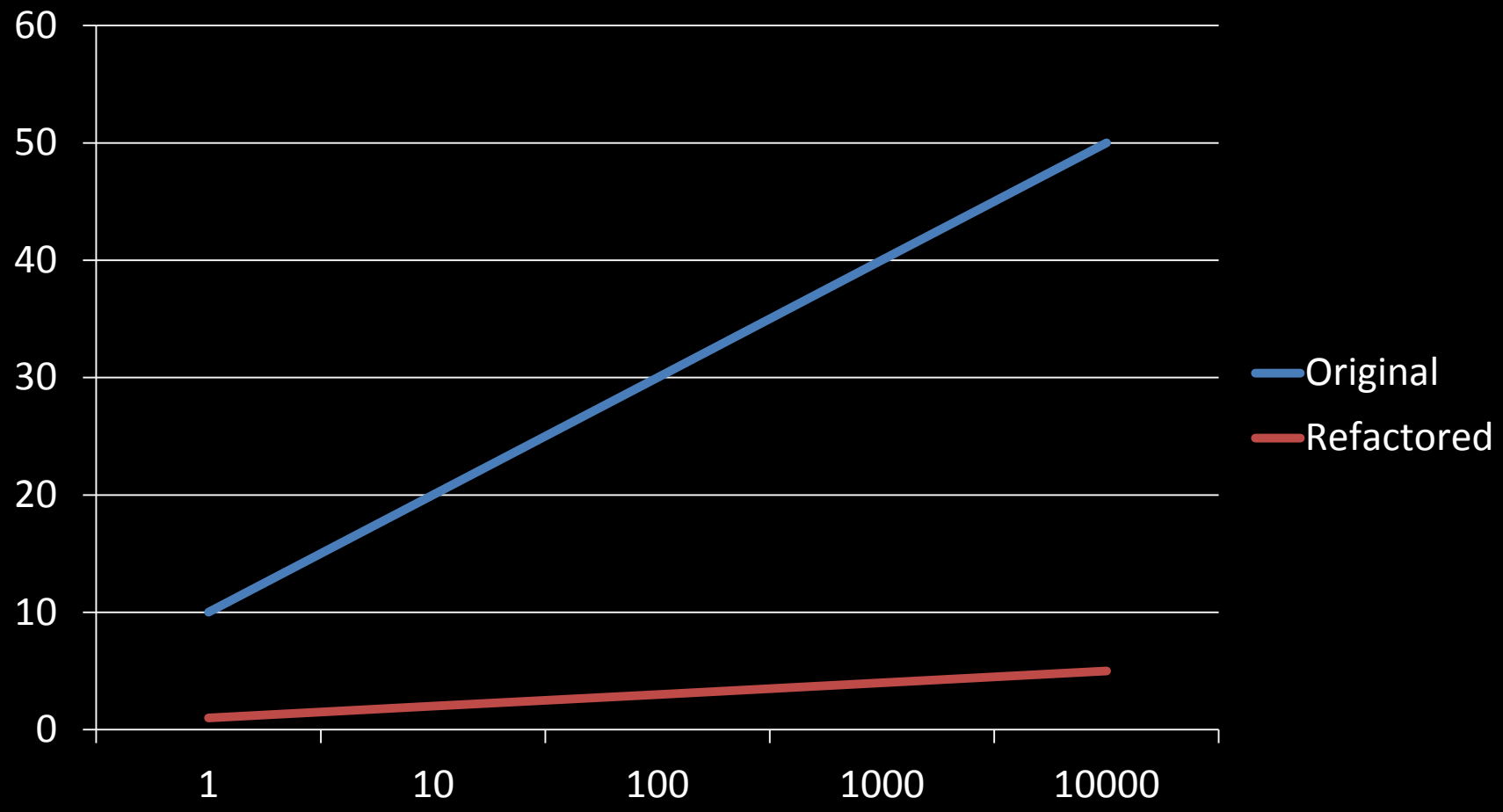
```
static List<String> passwordsToCheck;

// launch 100 threads

// each thread

void run() {
    while (!passwordsToCheck.isEmpty()) {
                synchronized (passwordsToCheck) {
                    if (!passwordsToCheck.isEmpty()) {
                                String pwd = passwordsToCheck.remove(0);
                    }
                }
                checkPassword(pwd);
    }
}
void checkPassword() {
}
```

Password Cracking Refactored

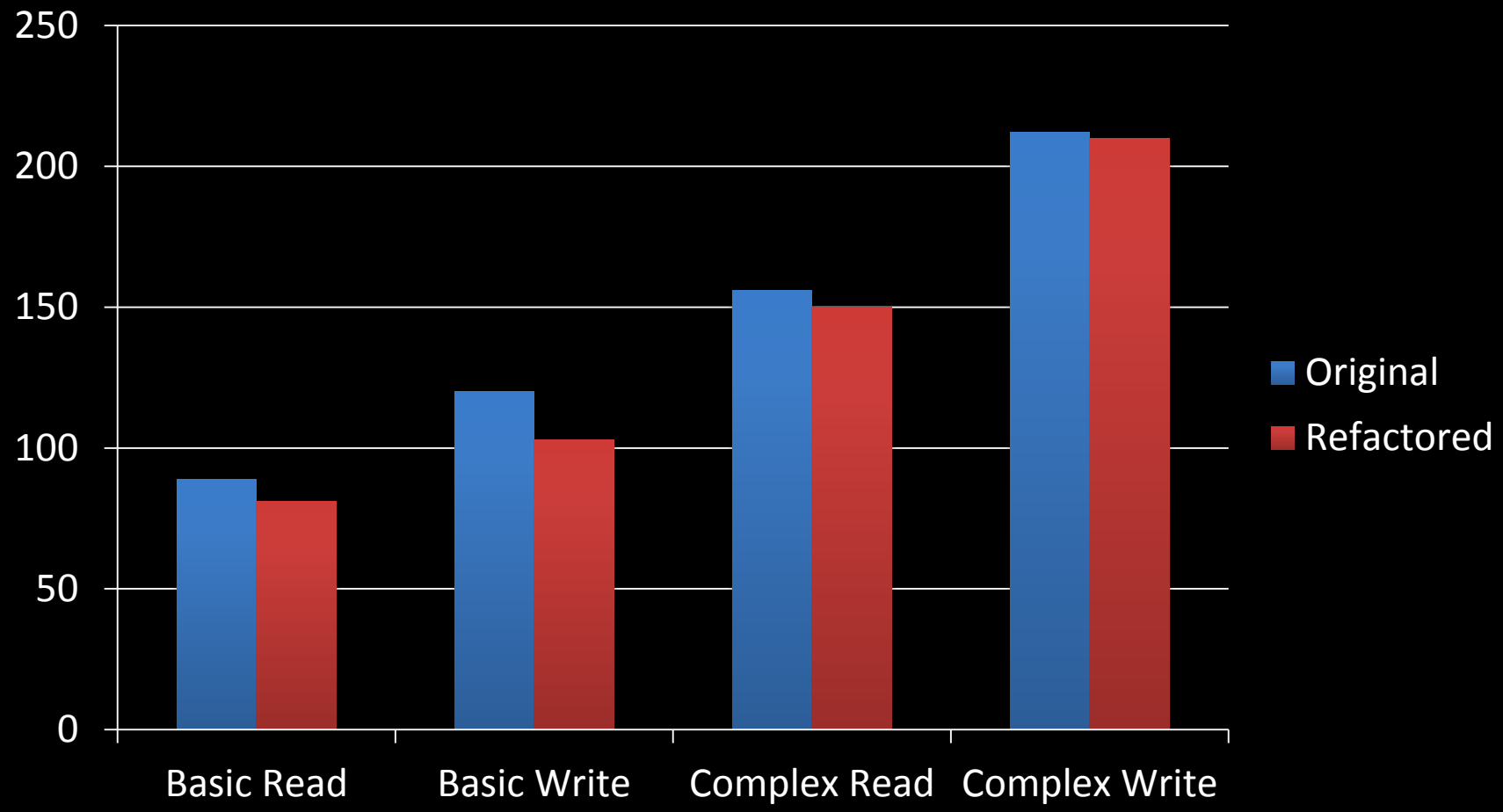


YONITA

Yonita Optimizer and Hibernate Core

- Only one refactoring – linear search
- Took 8 hours (sic!) to refactor
- Required manual assistance
  - Problems with parsing (Recoder library)
  - Finally, managed to get it working ☺

Hibernate Core Refactored Execution Times



YONITA

Current State

- Early prototype
  - Not alpha yet ☹
  - Want to give it a try?
- Problems
  - Static analysis
    - Front-end compiler
    - Reflection
    - Libraries and XML/file configs
  - Warranty of correctness
    - Thread safety
  - Hints from programmers?
    - @LargeCollection, @HeavyMethod

patrycja@yonita.com
@yonlabs