Application Performance Management

Frühling 2023

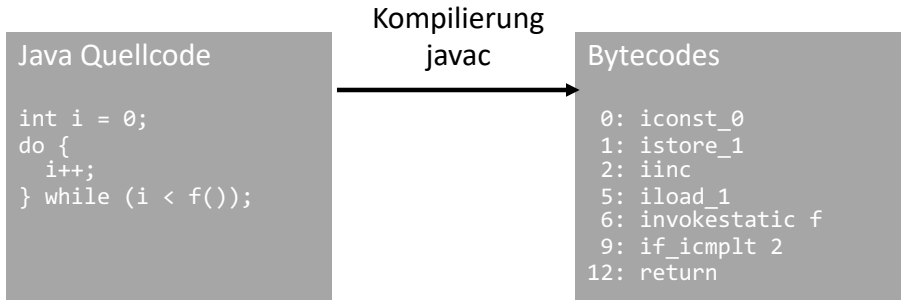
JIT-Kompilierung

Zoltán Majó

Agenda

Im Fokus heute: Kompilierung in der VM

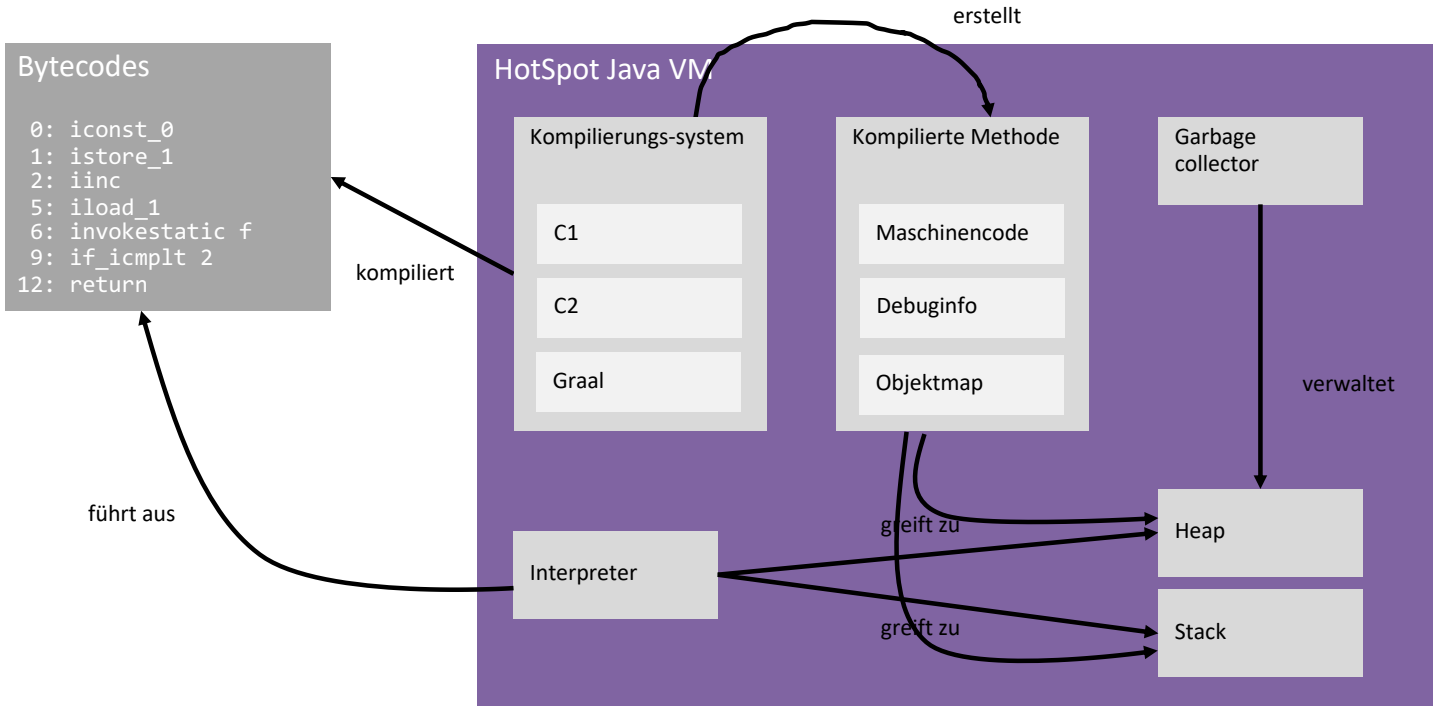
Kompilierung *für* die Java VM



Bemerkungen

- Kompilierung passiert «ahead-of-time»
- Bytecodes: Instruktionen für eine abstrakte Maschine (die JVM)
- Details der tatsächlichen Ausführung (auf einer physischen Maschine) sind der JVM überlassen

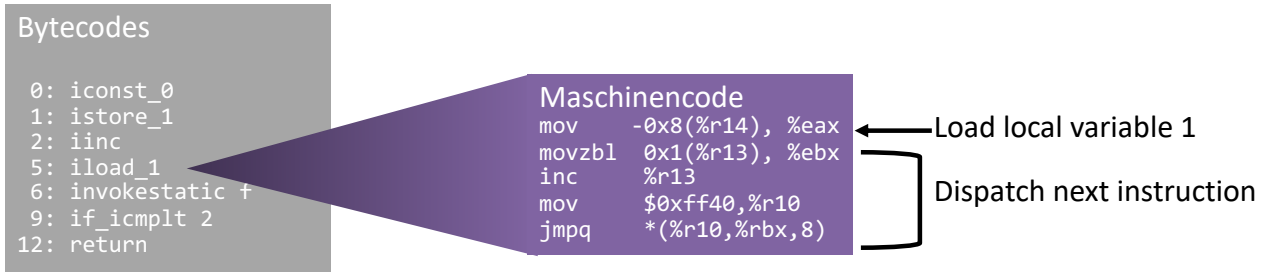
Kompilierung *in* der JVM



Interpretierung

Template-based interpreter

- Zuordnung Bytecode-Instruktion Maschinencode-Snippet



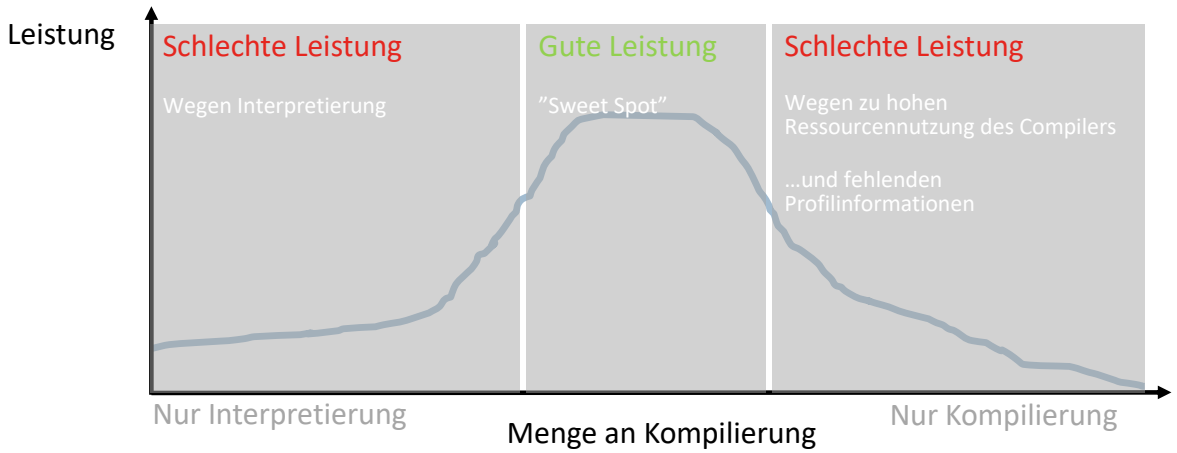
Kompilierungssystem: Leistungsverbesserung von etwa 100X

Kompilierung: Just-in-time (JIT)

JIT-Kompilierung passiert während Programmausführung

Kompromiss zwischen der Ressourcennutzung des Compilers
und der
Leistung des generierten Codes

Kompromiss


Leistung
Schlechte Leistung
Wegen Interpretierung
Gute Leistung
”Sweet Spot”
Schlechte Leistung
Wegen zu hohen Ressourcennutzung des Compilers
…und fehlenden Profilinformationen
Nur Interpretierung
Menge an Kompilierung
Nur Kompilierung

Übung 1

1. **Testprogramm im Ordner `code` kompilieren: `javac Consumer.java`**
   - Ich habe das Experiment mit Java 8 und 17
   - Evtl. funktionieren andere Versionen auch.
   - Wichtig: Java Version >= 8
2. **Führen Sie das Programm `Consumer` aus. Was ist die Laufzeit des Programmes?**
   - Sie können den Output des Programmes ignorieren
3. **Was ist die Laufzeit des Programmes nur mit Interpretierung? Sie können den Switch `-Xint` verwenden um Kompilierung auszuschalten.**
4. **Was ist die Laufzeit des Programmes wenn alle Methoden kompiliert werden? Sie können den Switch `-Xcomp` verwenden um alle Methoden zu kompilieren.**

Wie kommt man in den «Sweet Spot»

Zwei Mechanismen

1. Auswahl kompilierter Methoden
2. Auswahl Compileroptimierungen

1. Auswahl kompilierter Methoden

Nur oft ausgeführte Methoden werden kompiliert

- «Heisse Methoden»

Profilieren der Methodenausführung

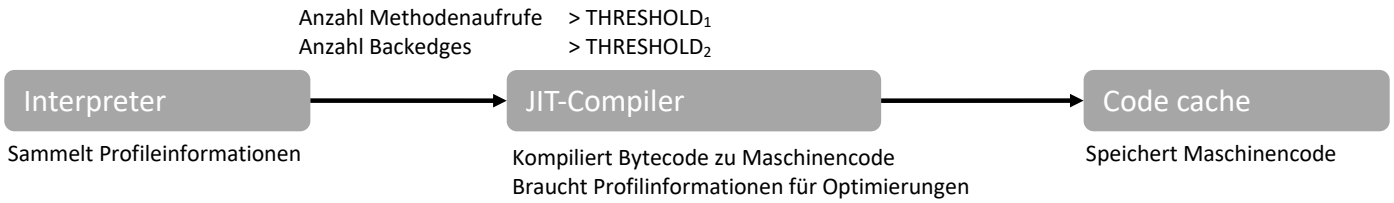
- Anzahl Methodenaufrufe
- Anzahl «Backedges» (relevant auch für On-Stack Replacement)

Viel mehr vom «Verhalten» einer Methode wird erfasst

- Profiling ist eine Voraussetzung für die meisten Kompileroptimierungen
- Mehr dazu später

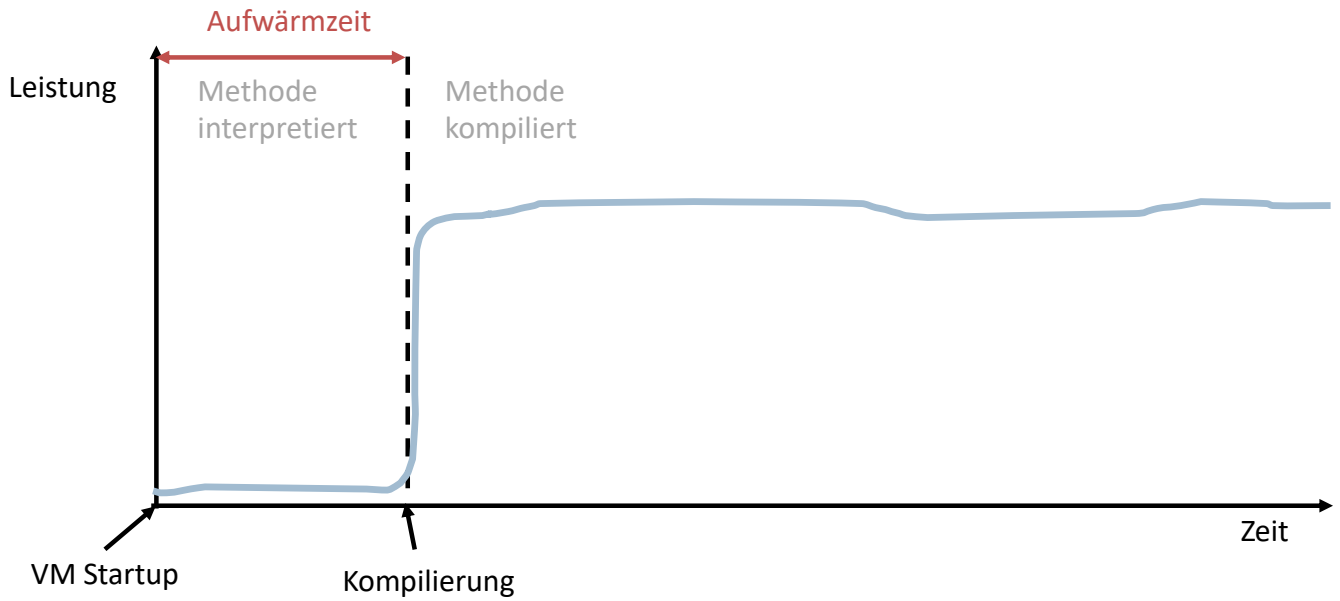
Das Leben einer Methode...

...in der HotSpot Java VM



Auswirkung auf die Leistung

Aus der Perspektive einer Methode



Wie kommt man zum «Sweet Spot»

Zwei Mechanismen

1. Auswahl kompilierter Methoden
2. Auswahl Compileroptimierungen

2. Auswahl Compileroptimierungen

1. C1 Compiler

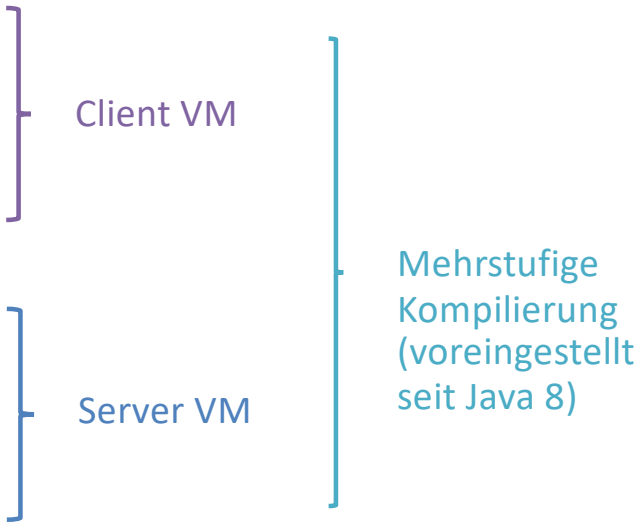
- Wenige Optimierungen
- Schnelle Kompilierung
- Niedriger Speichergebrauch

2. C2 Compiler

- Aggressiv optimierender Compiler
- Hohe Ressourcennutzung
- Hohe Leistung des generierten Codes

3. Graal Compiler

- Verfügbar als Ersatz von C2 in der Graal VM
- Details später



Mehrstufige Kompilierung

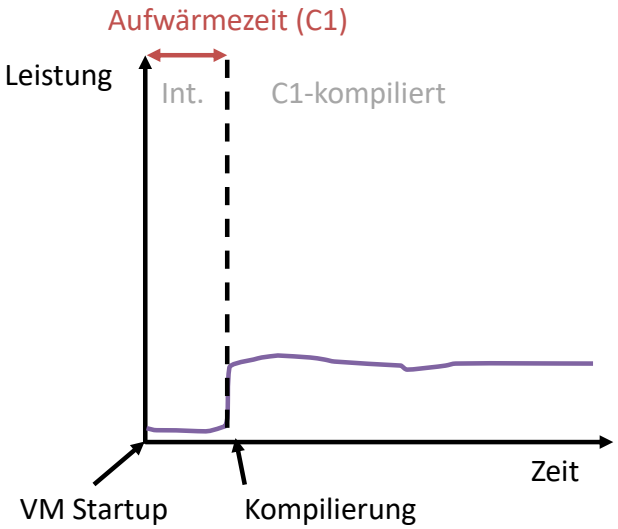
Englisch: Tiered Compilation

Kombiniert das Nutzen vom

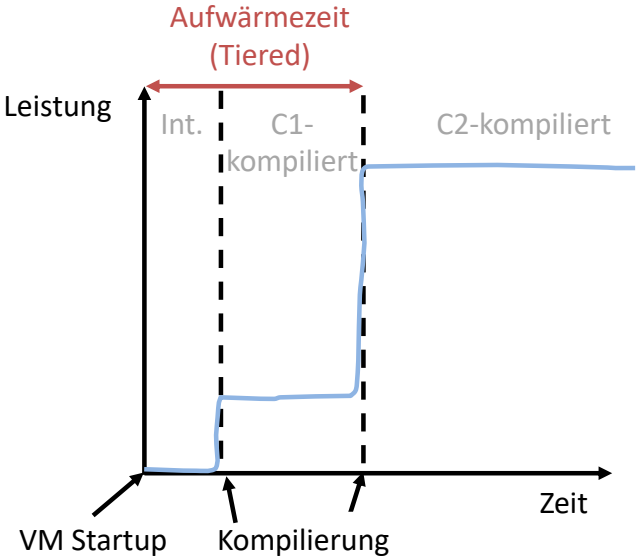
- Interpreter: schneller VM Start — Tier 0
- C1 Compiler: schnelle Kompilierung — Tier 1
- C2 Compiler: hohe Leistung — Tier 2

Nutzen der mehrstufigen Kompilierung

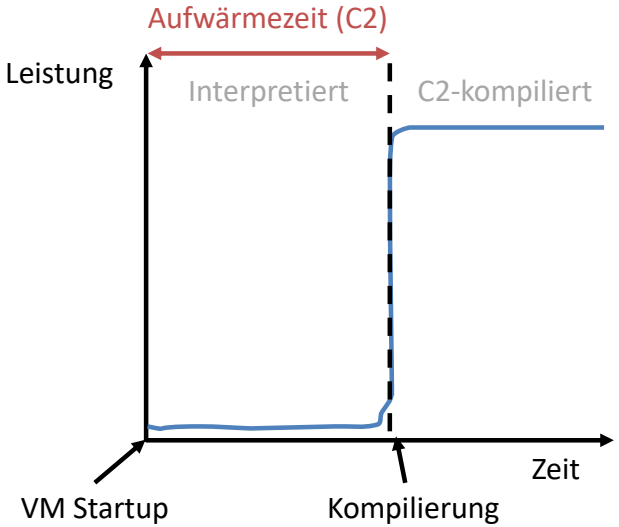
Client VM (C1)



Tiered VM

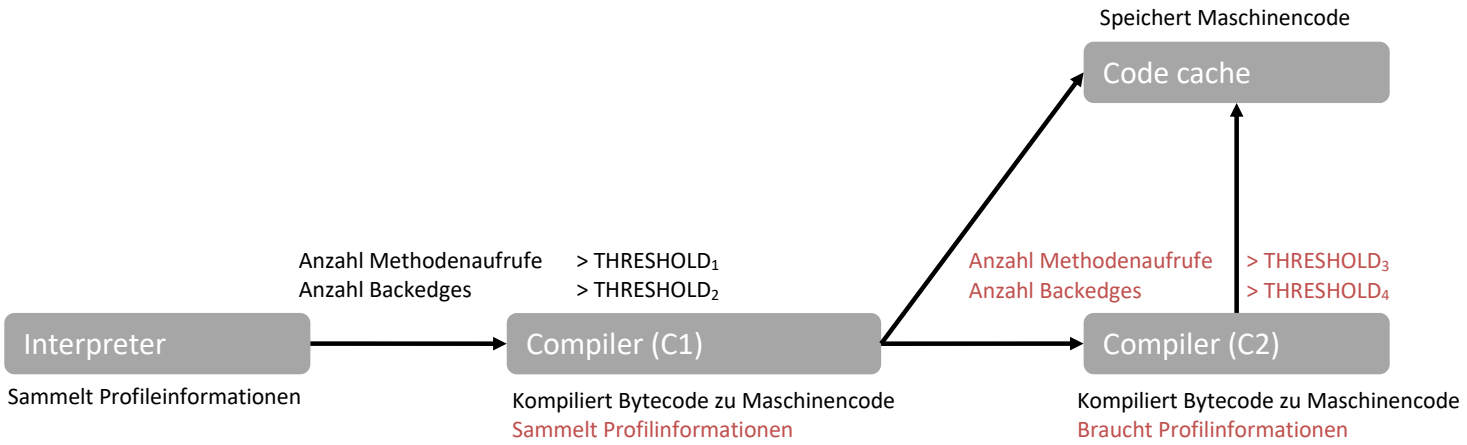


Server VM (C2)



Das Leben einer Methode mit Tiered Compilation

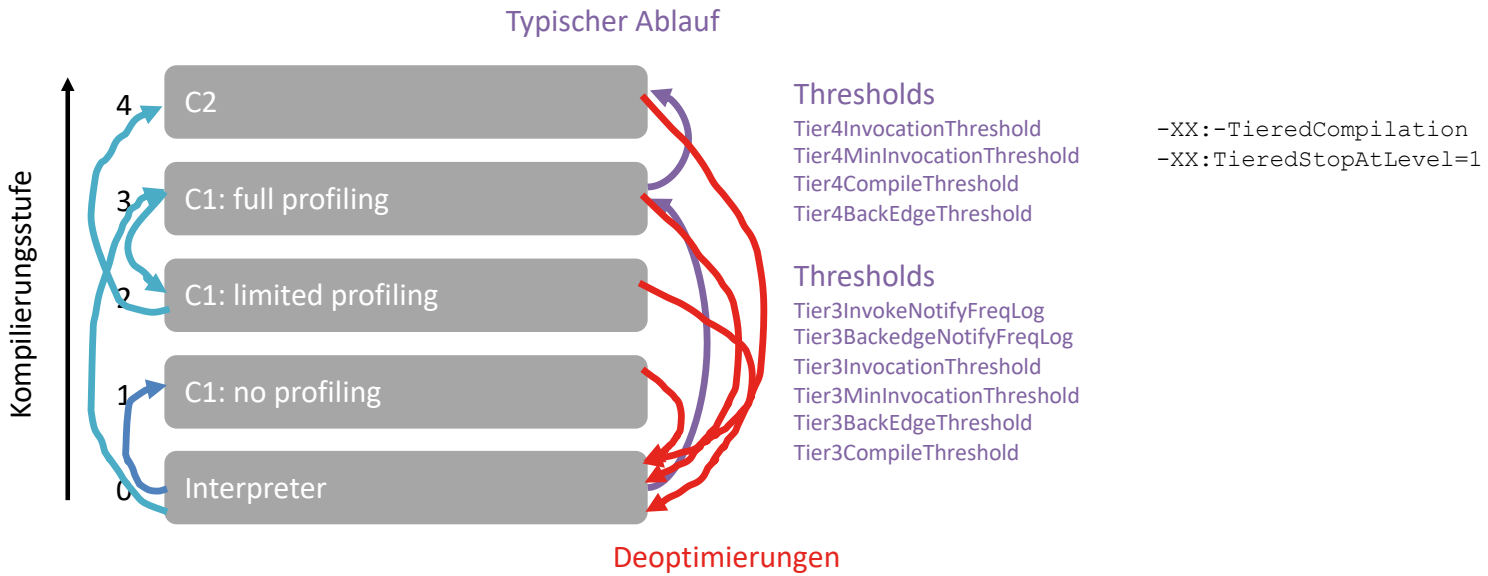
(Geschichte von vorher ergänzt)



Bemerkung

Aufwärmzeit (C1) < Aufwärmzeit (Tiered) ≤ Aufwärmzeit (C2)

Kompilierungsstufen (detaillierte Ansicht)



Übung 2

5. **Was ist die Laufzeit des Programmes mit C1? Sie können den Switch `-XX:TieredStopAtLevel=3` verwenden um nur C1-Kompilierungen zu erlauben.**
6. **Was ist die Laufzeit des Programmes mit C2? Sie können den Switch `-XX:-TieredCompilation` verwenden um nur C2-Kompilierungen zu erlauben.**
7. **Vergleichen sie die Messungen bei den Punkten 5 und 6 mit der Messung ohne spezielle Einstellungen (Punkt 2 von vorher). Wie ist das Verhältnis zwischen den Messwerten?**

Zusammenfassung: Tiered Compilation

Kombiniert Nutzen vom Interpreter, C1 und C2

- Bessere Leistung der VM

Nachteile

- Komplexe Implementierung und Konfigurierung
- Mehr Druck auf den Codespeicher

Deoptimierungen

Wir schauen uns den Effekt von Deoptimierungen durch ein praktisches Beispiel an

Beispiel

Weiterhin `Consumer.java`, jedoch mit einer detaiillierteren Sicht

```
public class Consumer {
    // Omitted
    public static void main(String[] args) {
        Producer producer = new Producer();
        Consumer consumer = new Consumer();
        System.out.println("===============");
        System.out.println("Run 1");
        System.out.println("===============");
        consumer.consume(producer, 500);
        System.out.println("===============");
        System.out.println("Run 2");
        System.out.println("===============");
        consumer.consume(producer, 500);
    }
}
```

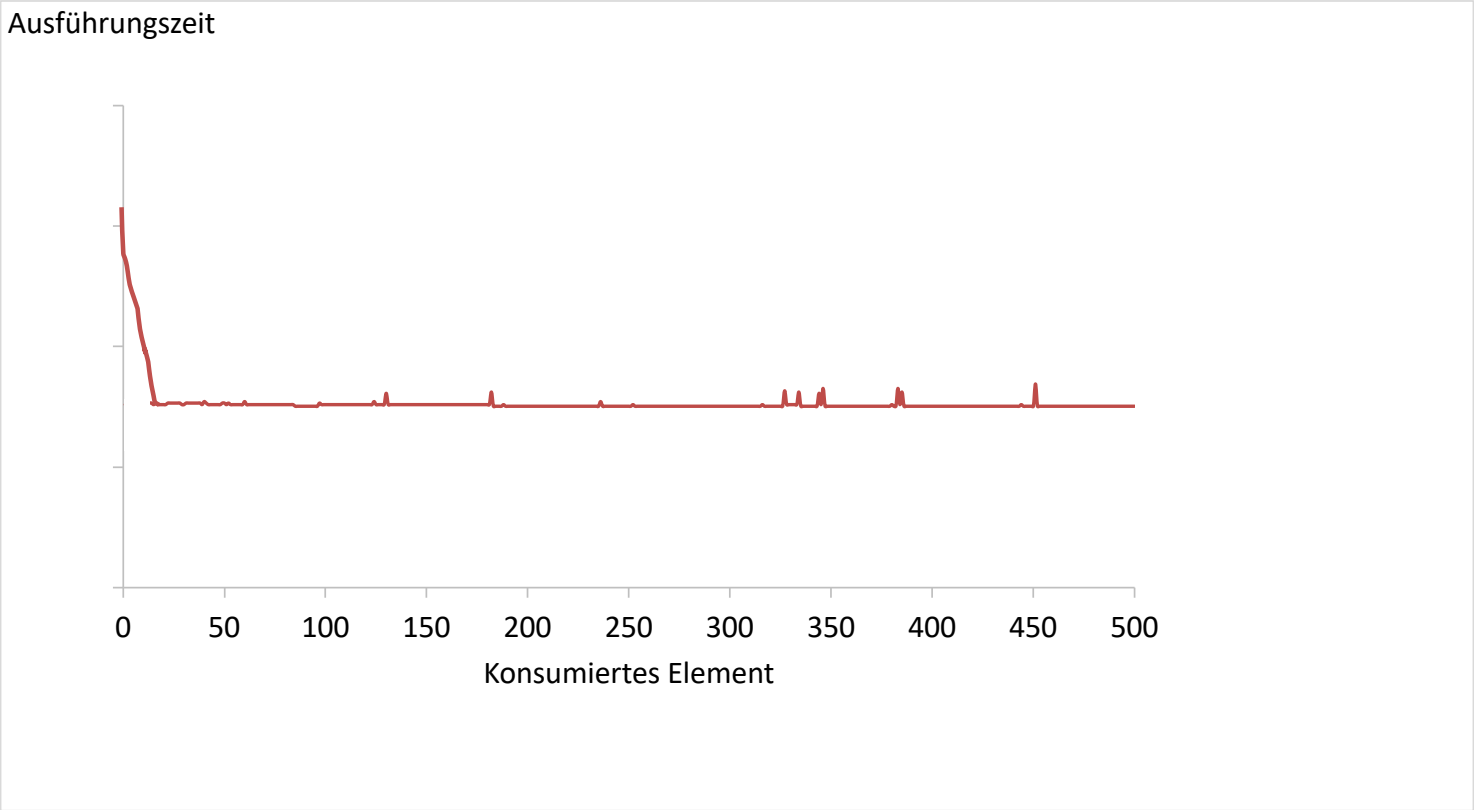
Run 1: Konsumiert 500 Elemente

Die Dauer jeder Konsum-Operation wird gemessen und in der Konsole gezeigt

Run 2: Dito

(Kümmert uns aktuell nicht)

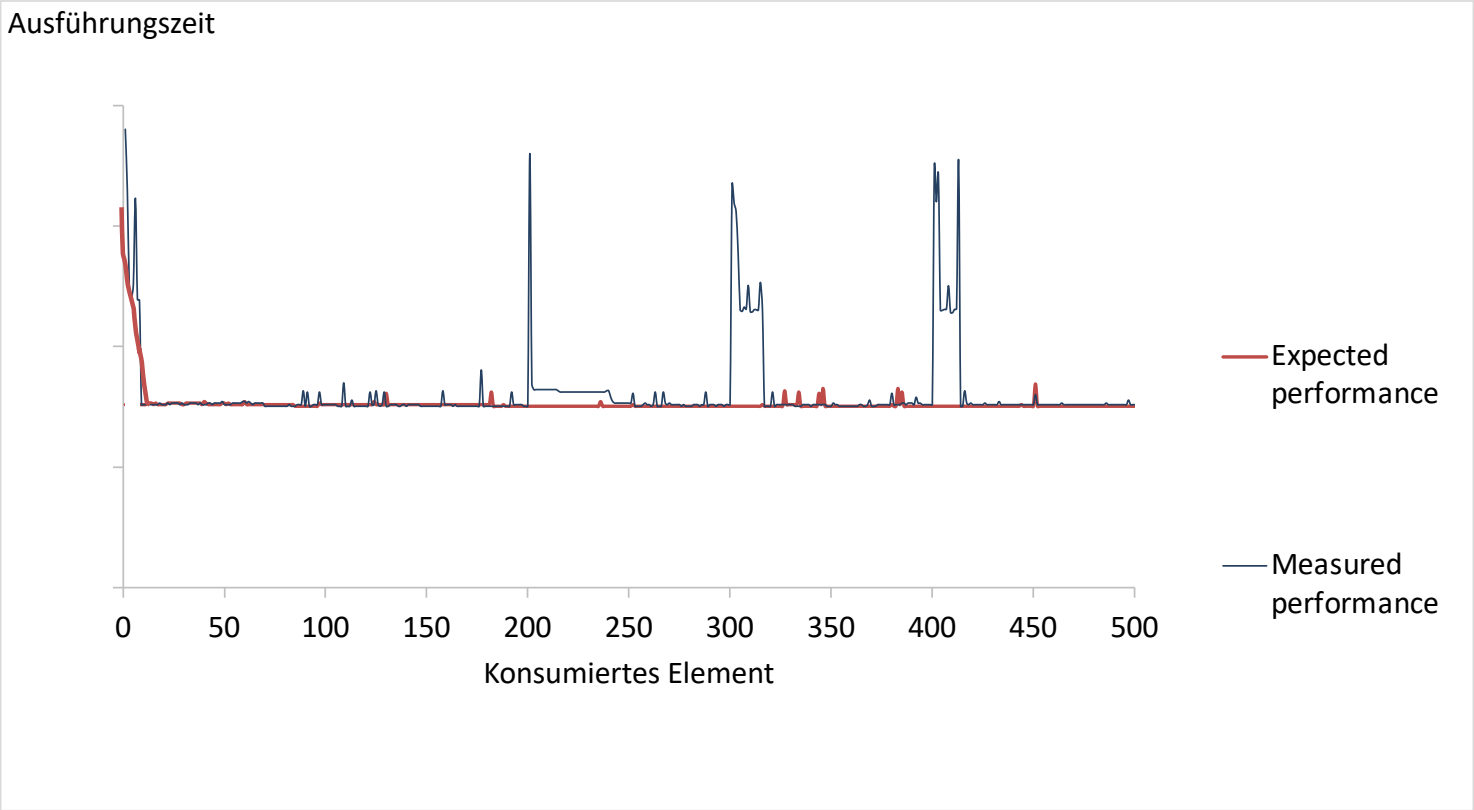
Erwartete Leistung (Trend)



Übung 3

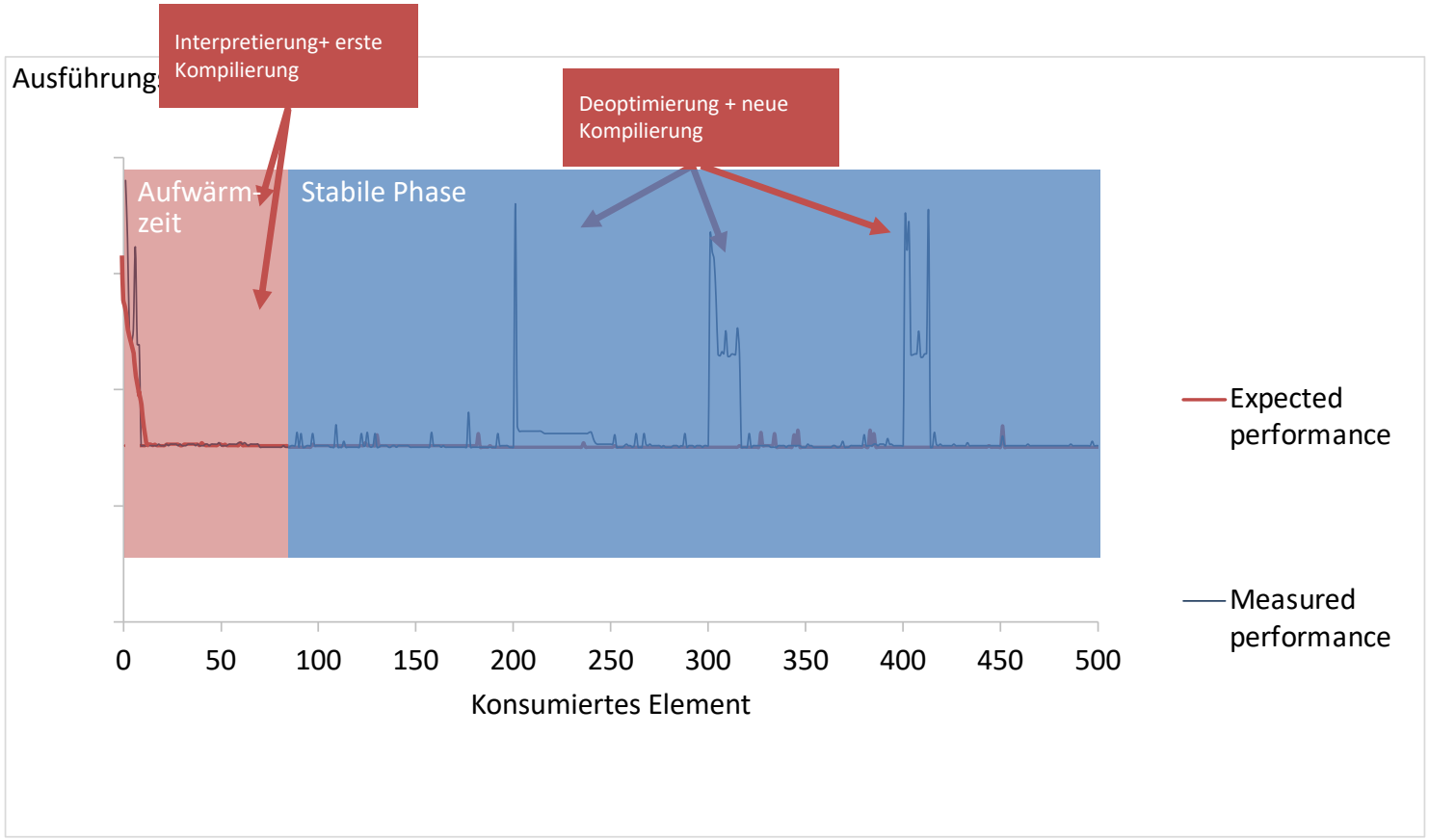
8. **Was sind die Laufzeiten der ersten 500 Aufrufe der Methode** `produce()` **(Run 1)?**

Gemessene Leistung



Wieso?

Gemessene Leistung



Deoptimierungen

Compileroptimierungen basieren sich auf optimistische Annahmen

Annahmen basiert auf Profilinformationen

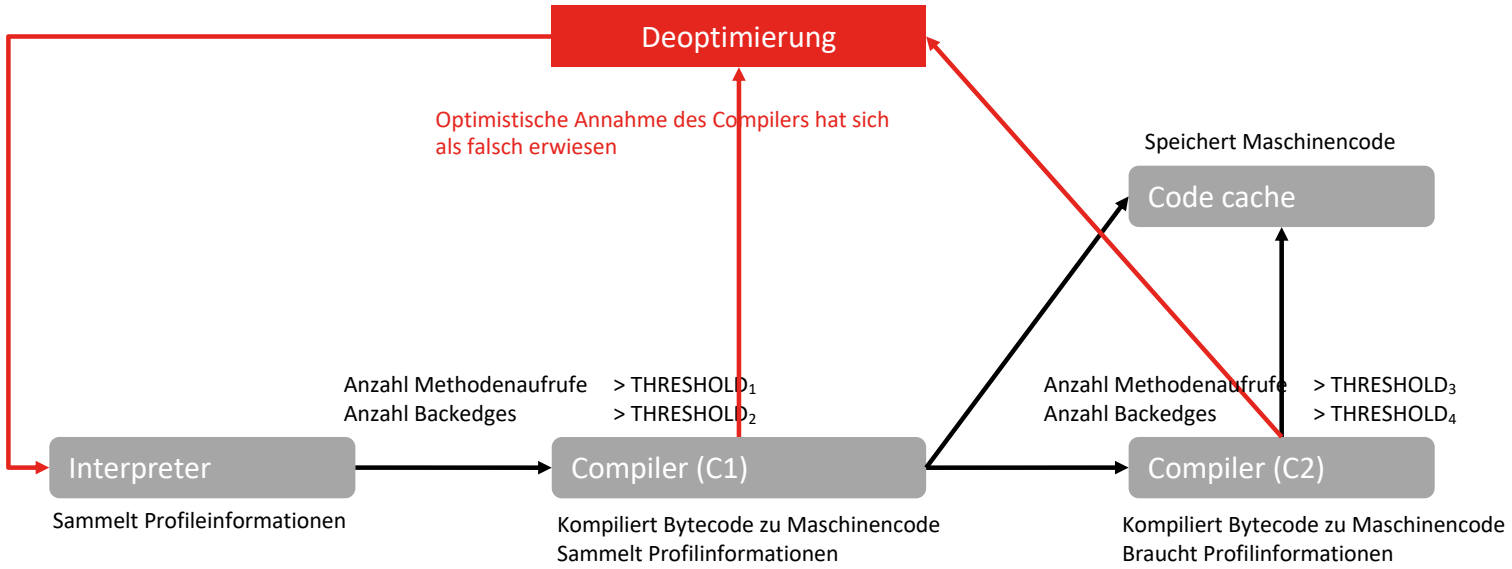
- Anzahl Methodenaufrufe und „Backedges“
- Ausgeführte Pfade in der Methode
- Typen bei Methodenaufrufe
- Typen der Methodenparameter
- Klassenhierarchie
- Und noch mehr anderes

Grundlegendes Prinzip: Vergangenheit = Zukunft

Falls optimistische Annahme des Compilers nicht mehr gilt: Deoptimierung

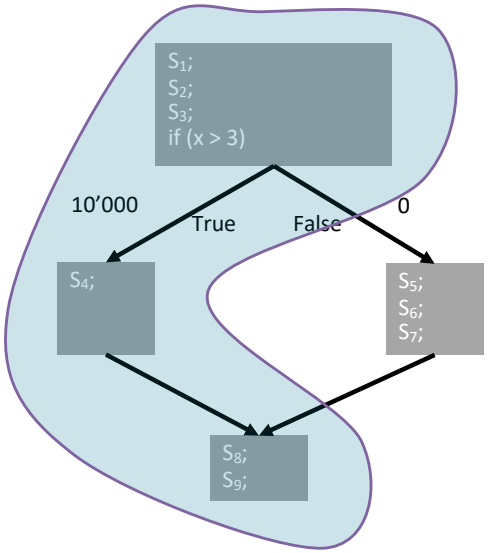
- Kompilierter Code weggeworfen
- Methode wird neu profiliert und erneut kompiliert (mit weniger optimistischen Annahmen)

Das Leben einer Methode: Komplette Geschichte



Beispieloptimierung: “Hot path”-Kompilierung

Kontrollflussgraph



Generierter Code

Zurück zu unserem Beispielprogramm

```
public long produce(int item)
  long result = 0;
  for (int i = 0; i < 100_000; ++i) {
    long[] pattern = {i, i + 1, i + 2, i + 3};
    if (item == 200) {
      result += pattern[0];
    } else if (item == 300) {
      result += pattern[1];
    } else if (item == 400) {
      result += pattern[2];
    } else {
      result += pattern[3];
    }
  }
  return result;
}
```

Wieso passieren Deoptimierungen?

Producer

```
public long produce(int item)
  long result = 0;
  for (int i = 0; i < 100_000; ++i) {
    long[] pattern = {i,     i + 1,
                      i + 2, i + 3};
    if (item == 200) {
      result += pattern[0];
    } else if (item == 300) {
      result += pattern[1];
    } else if (item == 400) {
      result += pattern[2];
    } else {
      result += pattern[3];
    }
  }
  return result;
}
```

Abgedeckte Pfade

- Kompilierung #1

Wieso passieren Deoptimierungen?

Producer

```
public long produce(int item)
  long result = 0;
  for (int i = 0; i < 100_000; ++i) {
    long[] pattern = {i,     i + 1,
                      i + 2, i + 3};
    if (item == 200) {
      result += pattern[0];
    } else if (item == 300) {
      result += pattern[1];
    } else if (item == 400) {
      result += pattern[2];
    } else {
      result += pattern[3];
    }
  }
  return result;
}
```

Abgedeckte Pfade

- Kompilierung #1
- Kompilierung #2

Wieso passieren Deoptimierungen?

Producer

```
public long produce(int item)
  long result = 0;
  for (int i = 0; i < 100_000; ++i) {
    long[] pattern = {i,     i + 1,
                      i + 2, i + 3};
    if (item == 200) {
      result += pattern[0];
    } else if (item == 300) {
      result += pattern[1];
    } else if (item == 400) {
      result += pattern[2];
    } else {
      result += pattern[3];
    }
  }
  return result;
}
```

Abgedeckte Pfade

- Kompilierung #1
- Kompilierung #2
- Kompilierung #3

Wieso passieren Deoptimierungen?

Producer

```
public long produce(int item)
  long result = 0;
  for (int i = 0; i < 100_000; ++i) {
    long[] pattern = {i,     i + 1,
                      i + 2, i + 3};
    if (item == 200) {
      result += pattern[0];
    } else if (item == 300) {
      result += pattern[1];
    } else if (item == 400) {
      result += pattern[2];
    } else {
      result += pattern[3];
    }
  }
  return result;
}
```

Abgedeckte Pfade

- Kompilierung #1
- Kompilierung #2
- Kompilierung #3
- Kompilierung #4

Übung 4

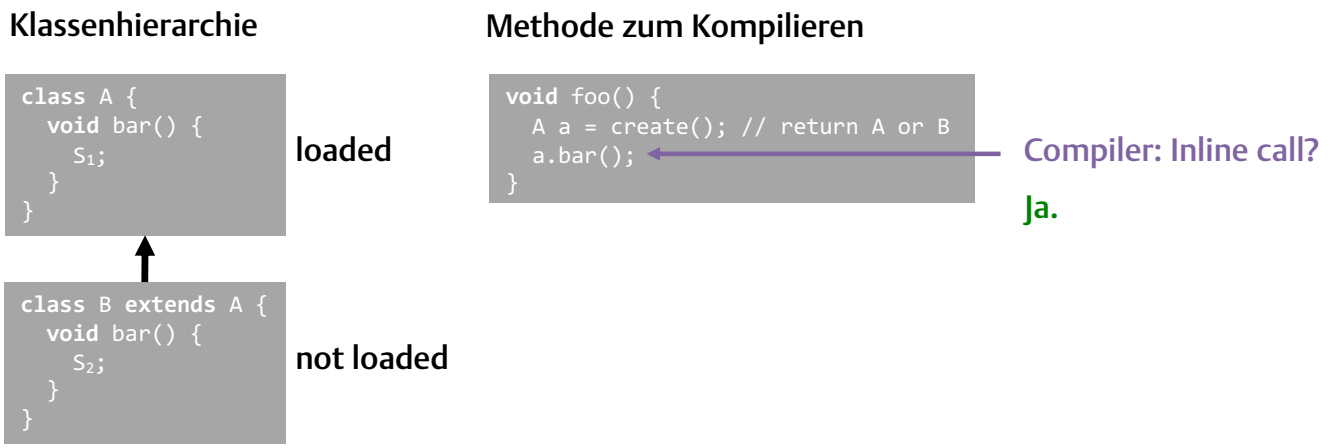
9. Was sind die Laufzeiten der 500 Aufrufe der Methode `produce()` während Run 2?

```
public class Consumer {
    // Omitted
    public static void main(String[] args) {
        Producer producer = new Producer();
        Consumer consumer = new Consumer();
        System.out.println("===============");
        System.out.println("Run 1");
        System.out.println("===============");
        consumer.consume(producer, 500);
        System.out.println("===============");
        System.out.println("Run 2");
        System.out.println("===============");
        consumer.consume(producer, 500);
    }
}
```

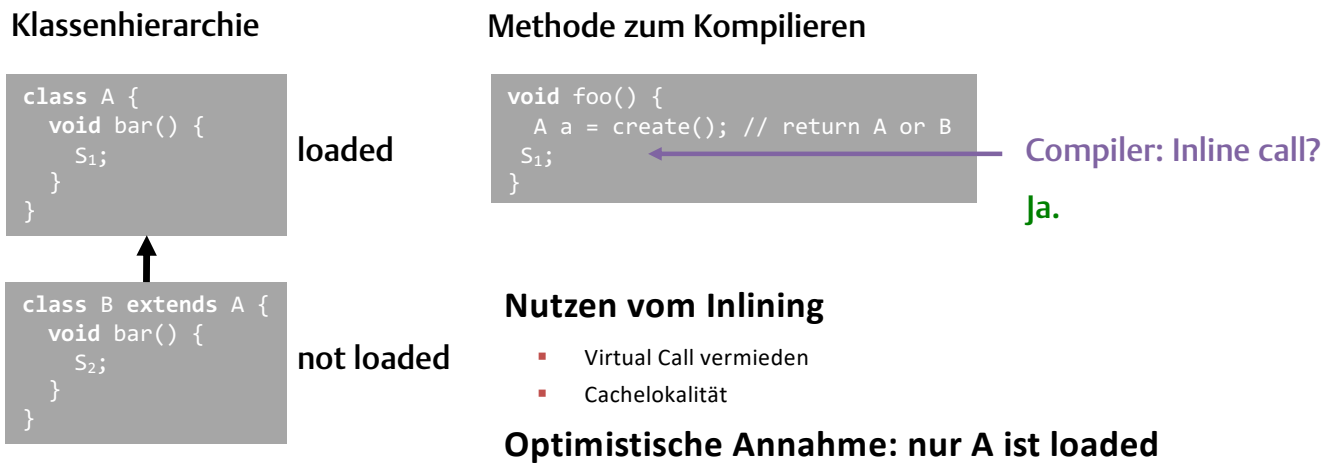
Run 2:

Gibt es denn andere Gründe für Deoptimierungen?

Beispieloptimierung 2: Virtual Call Inlining



Beispieloptimierung 2: Virtual Call Inlining



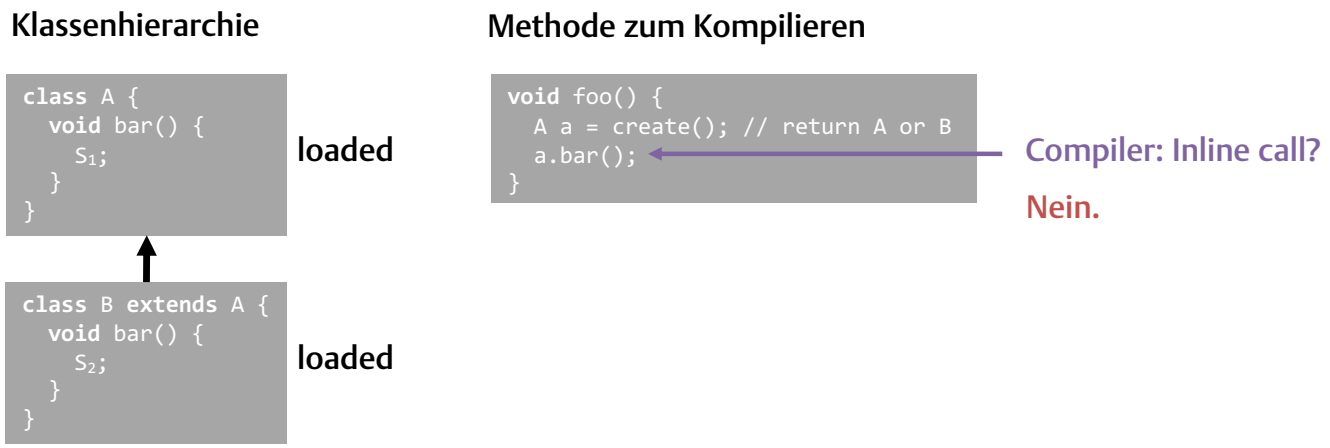
Nutzen vom Inlining

- Virtual Call vermieden
- Cachelokalität

Optimistische Annahme: nur A ist loaded

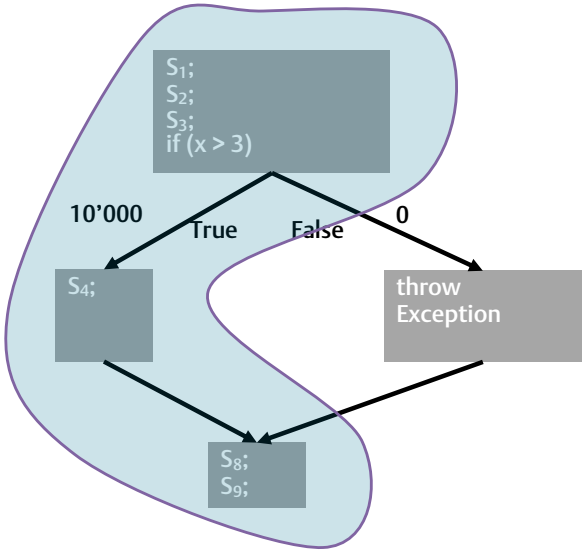
- Compiler merkt Abhängigkeit von Klassenhierarchie
- Wenn Hierarchie verändert: Deoptimierung

Beispieloptimierung 2: Virtual Call Inlining



Beispieloptimierung 3: Exception Handling

Kontrollflussgraph



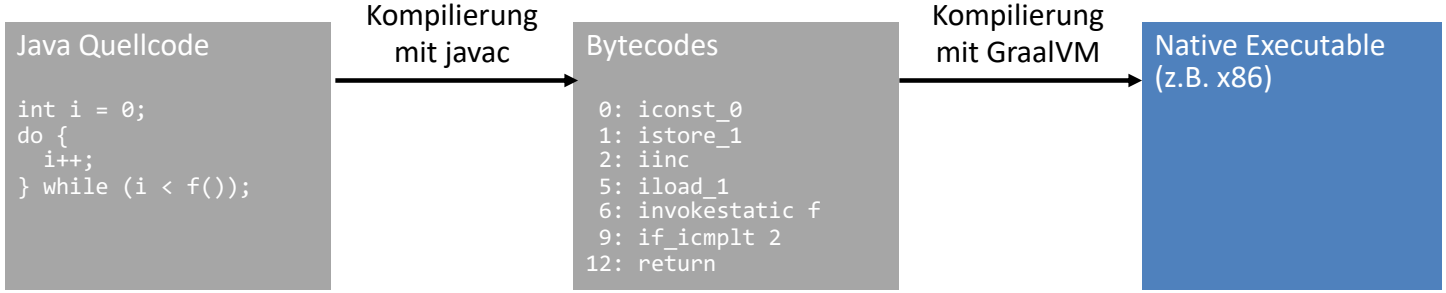
Generierter Code

Anzahl Ausführungen Guard > Schwellwert → Rekompilierung (evtl. mit anderen Optimierungen)

siehe z.B. Flag OmitStackTraceInFastThrow

Weitere interessante Experimente:
https://www.baeldung.com/java-exceptions-performance

Ahead-of-time Kompilierung für Java (mit Graal VM)

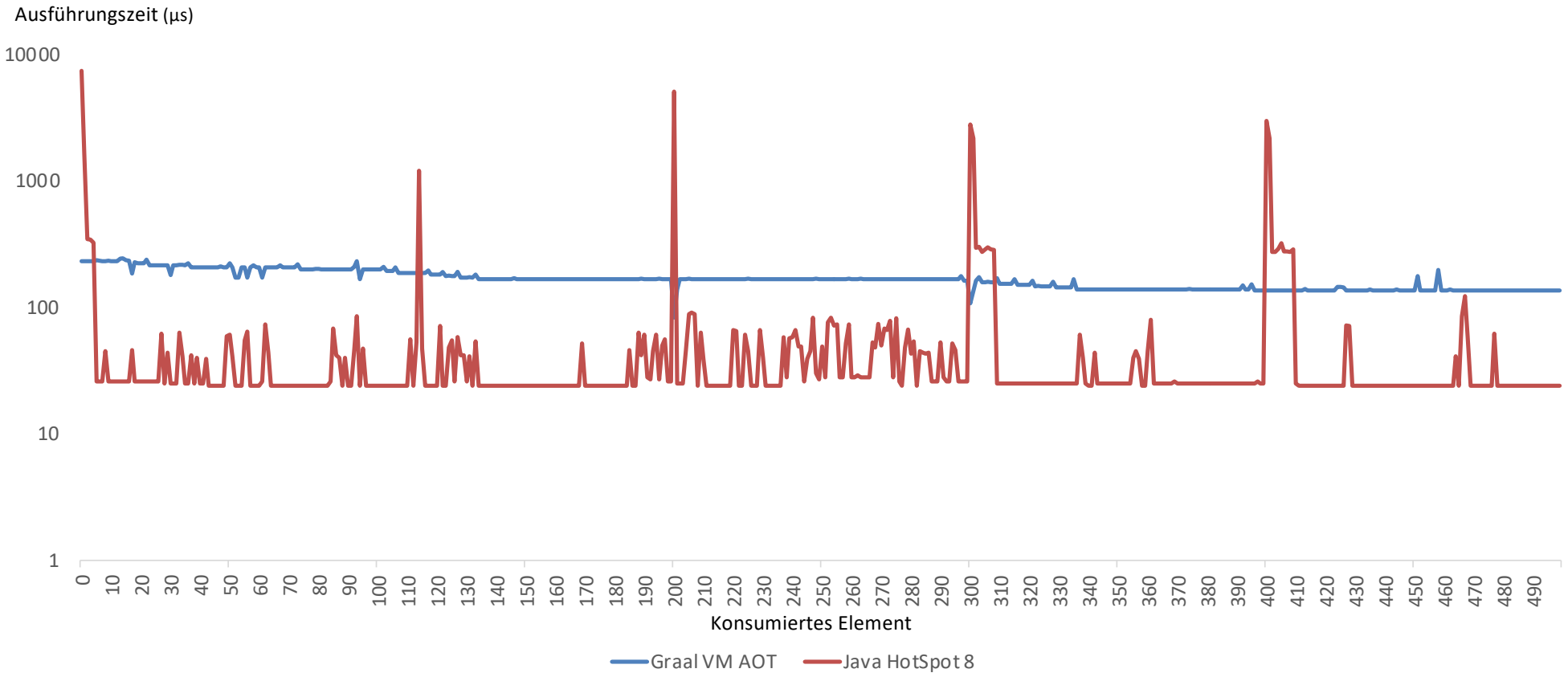


Demo

```
native-image Consumer
========================================================================================================================
GraalVM Native Image: Generating 'consumer' (executable)...
========================================================================================================================
[1/7] Initializing...                                                                                    (7.2s @ 0.19GB)
 Version info: 'GraalVM 22.3.0 Java 19 CE'
 Java version info: '19.0.1+10-jvmci-22.3-b08'
 C compiler: cc (apple, x86_64, 14.0.0)
 Garbage collector: Serial GC
[2/7] Performing analysis...  [*****]                                                                    (6.5s @ 0.65GB)
   2,894 (73.96%) of  3,913 classes reachable
   3,428 (50.70%) of  6,762 fields reachable
  12,917 (43.36%) of 29,791 methods reachable
      25 classes,     0 fields, and   330 methods registered for reflection
      57 classes,    57 fields, and    52 methods registered for JNI access
       4 native libraries: -framework Foundation, dl, pthread, z
[3/7] Building universe...                                                                               (0.9s @ 1.12GB)
[4/7] Parsing methods...      [*]                                                                        (0.7s @ 1.76GB)
[5/7] Inlining methods...     [***]                                                                      (0.3s @ 0.64GB)
[6/7] Compiling methods...    [**]                                                                       (4.5s @ 1.48GB)
[7/7] Creating image...                                                                                  (1.5s @ 1.91GB)
   4.14MB (36.64%) for code area:     7,307 compilation units
   6.95MB (61.51%) for image heap:   93,547 objects and 5 resources
 213.84KB ( 1.85%) for other data
  11.30MB in total
------------------------------------------------------------------------------------------------------------------------
Top 10 packages in code area:                               Top 10 object types in image heap:
 646.60KB java.util                                          925.31KB java.lang.String
 327.01KB java.lang                                          918.47KB byte[] for code metadata
 269.35KB java.text                                          829.89KB byte[] for general heap data
 221.24KB java.util.regex                                    635.74KB java.lang.Class
 192.96KB java.util.concurrent                               552.32KB byte[] for java.lang.String
 149.42KB java.math                                          226.09KB com.oracle.svm.core.hub.DynamicHubCompanion
 118.25KB java.util.stream                                   217.68KB java.lang.Object[]
 115.30KB java.lang.invoke                                   196.31KB java.util.HashMap$Node
 114.76KB com.oracle.svm.core.genscavenge                    163.64KB java.lang.String[]
  92.94KB java.util.logging                                  157.27KB java.util.concurrent.ConcurrentHashMap$Node
   1.89MB for 122 more packages                                1.58MB for 780 more object types
------------------------------------------------------------------------------------------------------------------------
                        0.4s (1.6% of total time) in 17 GCs | Peak RSS: 3.20GB | CPU load: 8.22
------------------------------------------------------------------------------------------------------------------------
Produced artifacts:
 /Users/zmajo/Documents/fhnw/git-reps/students/apm-fs22/week-06/code/consumer (executable)
 /Users/zmajo/Documents/fhnw/git-reps/students/apm-fs22/week-06/code/consumer.build_artifacts.txt (txt)
========================================================================================================================
Finished generating 'consumer' in 22.9s.

./consumer
```

Kurzevaluation



Bemerkungen

Performanz könnte mit Profilinformationen besser sein → Enterprise Graal

Annahme einer “geschlossenen Welt” (engl.: close-world assumption)

Zusammenfassung

Im Fokus heute: Kompilierung in der VM

- Einführung in Just-in-time Kompilierung
- Tiered Compilation
- Deoptimierungen
- Ahead-of-time Kompilierung