

# **Application Performance Management**

## **Frühling 2022**

### **JIT-Kompilierung**

*Zoltán Majó*

# Agenda

## Im Fokus heute: Kompilierung in der VM

- Mehrstufige Kompilierung
- Segmentierung des Codespeichers
- Caching von Programmprofilen

# Agenda

## Im Fokus heute: Kompilierung in der VM

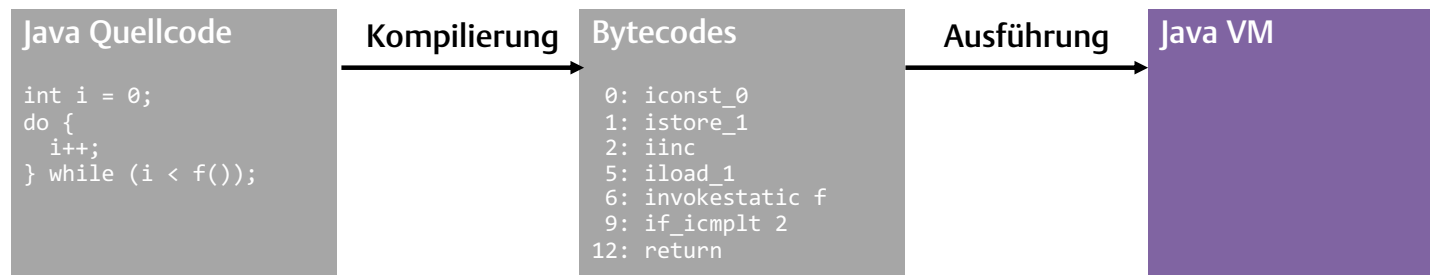
- Mehrstufige Kompilierung – Tiered compilation
- Segmentierung des Codespeichers – Segmented code cache
- Caching von Programmprofilen – Profile caching

ab Java 8

ab Java 9

in der Zukunft  
(vielleicht)

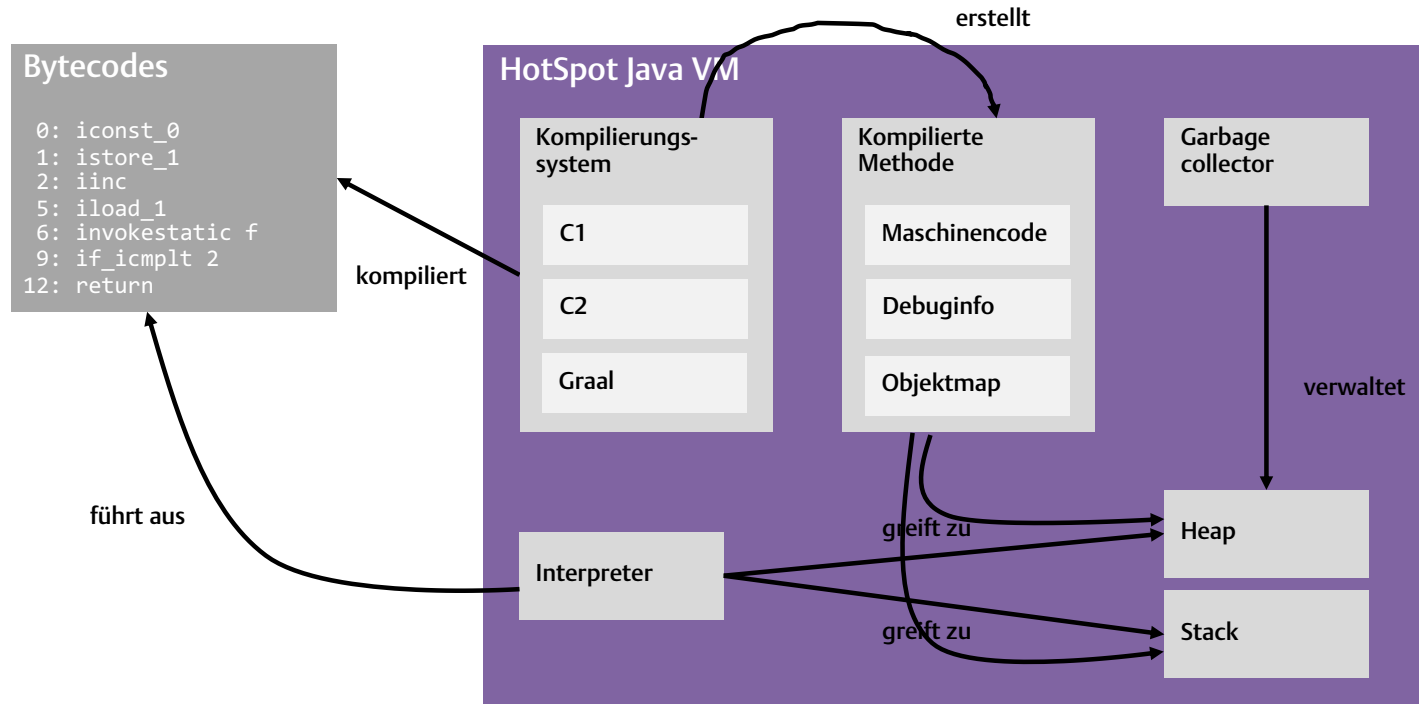
# Kompilierung für die Java VM



## Bemerkungen

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

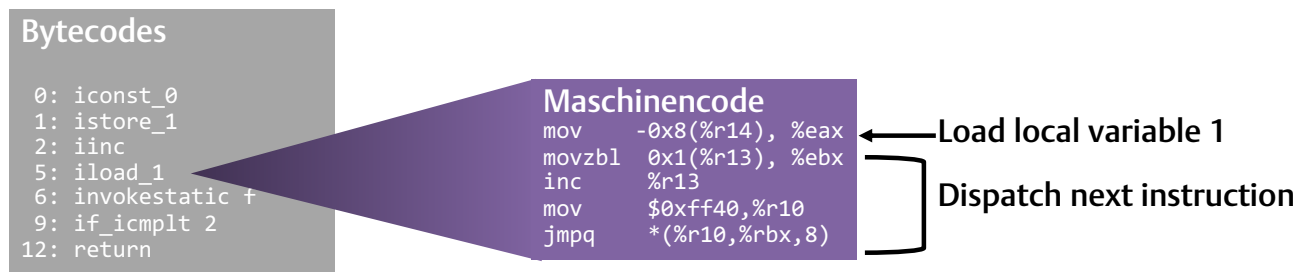
# Kompilierung in der JVM



# Interpretierung

## Template-based interpreter

- Zuordnung Bytecode-Instruktion Maschinencode-Snippet



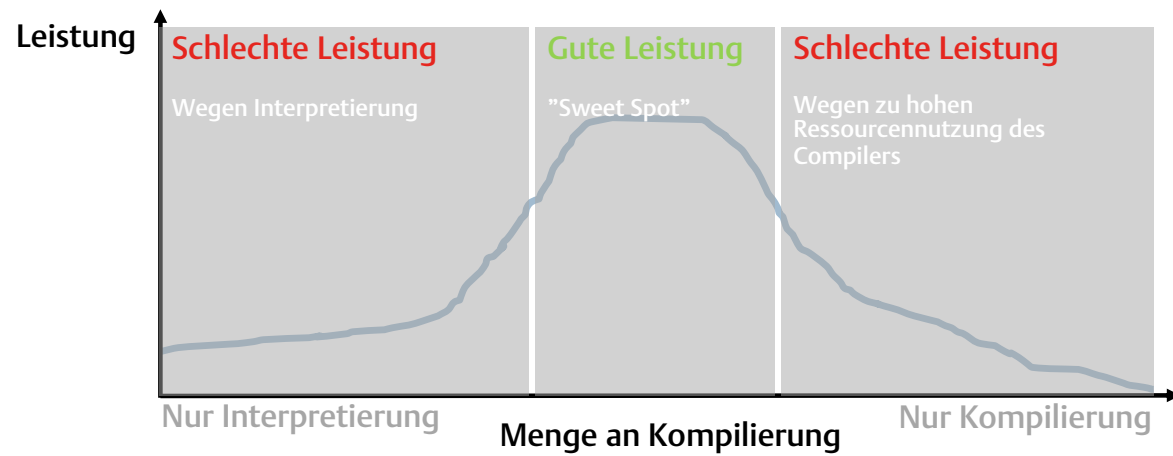
**Kompilierungssystem: Leistungsverbesserung von etwa 100X**

# Kompilierung: Just-in-time (JIT)

JIT-Kompilierung passiert während Programmausführung

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

# Kompromiss





# Wie man in das „Sweet Spot“ kommt

## Zwei Mechanismen

- ➔ 1. Auswahl kompilierter Methoden
- 2. Auswahl Compileroptimierungen

# 1. Auswahl kompilierter Methoden

**Nur oft ausgeführte Methoden werden kompiliert**

- „Heisse Methoden“

**Profilieren der Methodenausführung**

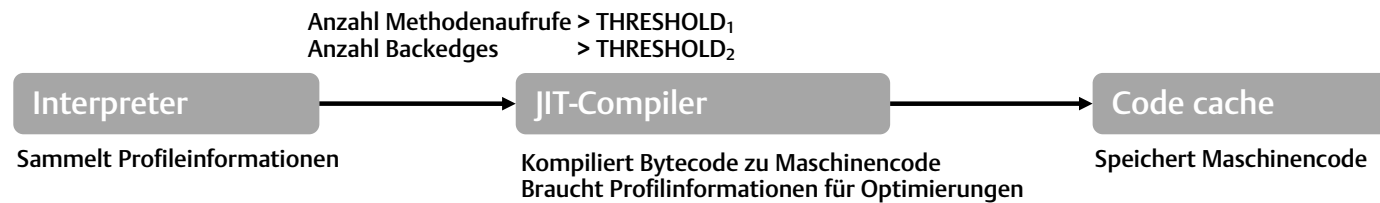
- Anzahl Methodenaufrufe
- Anzahl „Backedges“ (relevant auch für On-Stack Replacement)

**Viel mehr vom „Verhalten“ einer Methode wird erfasst**

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

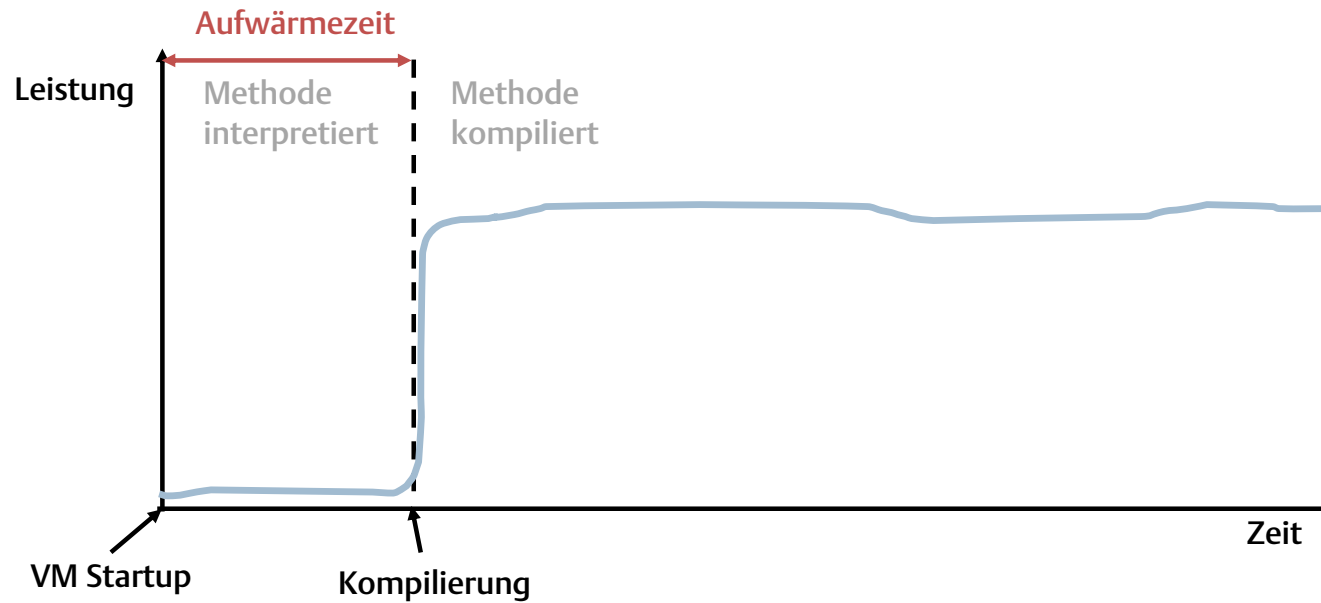
# Das Leben einer Methode...

...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

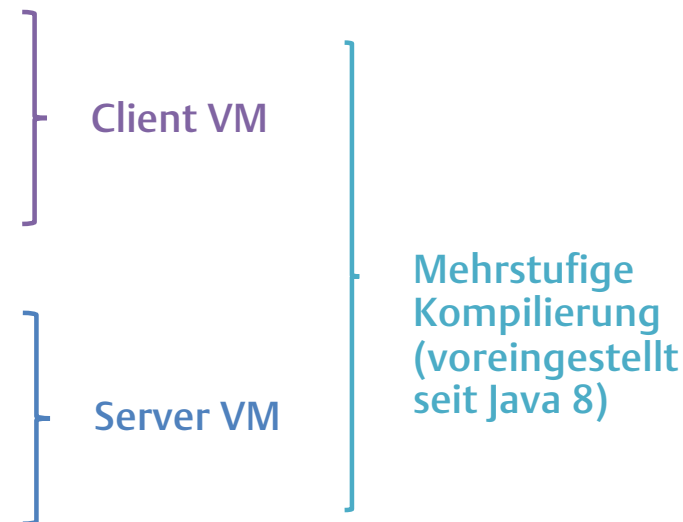
- Wenige Optimierungen
- Schnelle Kompilierung
- Niedriger Speichergebrauch

### 2. C2 Compiler

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

### 3. Graal Compiler (heute out of scope)

- Experimentell ab Oracle JDK 10
- Verfügbar als Ersatz von C2 in der Graal VM



# Mehrstufige Kompilierung

## „Tiered Compilation“

### Kombiniert das Nutzen vom

- Interpreter: schneller VM Start
- C1 Compiler: schnelle Kompilierung
- C2 Compiler: hohe Leistung

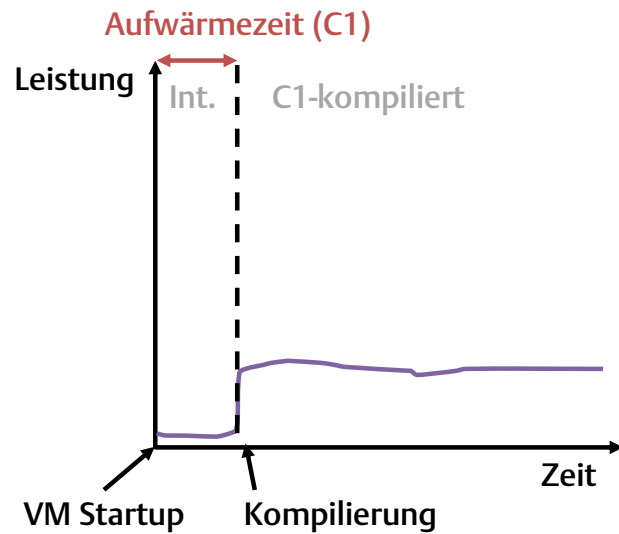
Tier 0

Tier 1

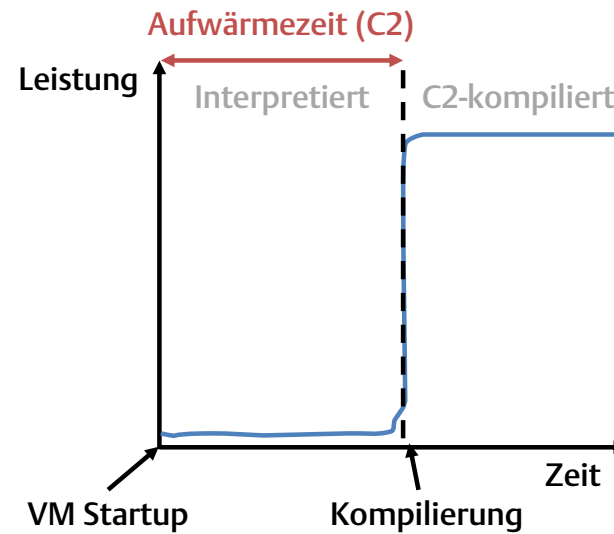
Tier 2

# Nutzen der mehrstufigen Kompilierung

Client VM (C1)



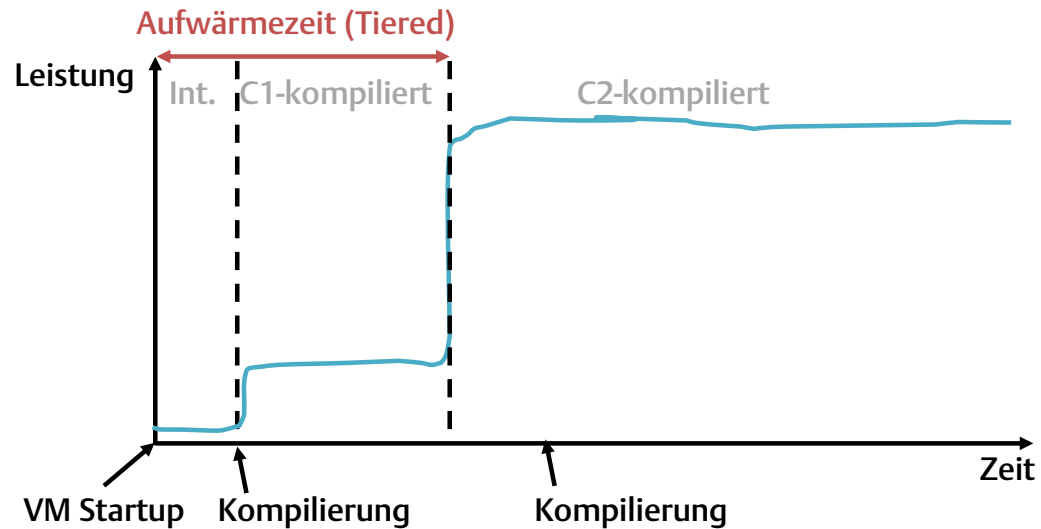
Server VM (C2)





# Nutzen der mehrstufigen Kompilierung (Forts.)

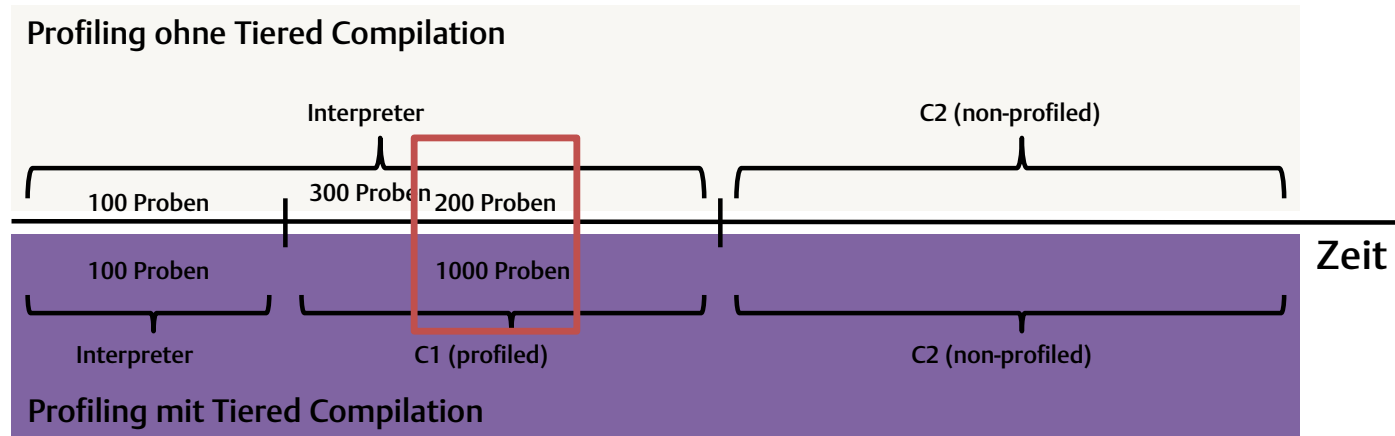
## Mehrstufige Kompilierung



## Bemerkung

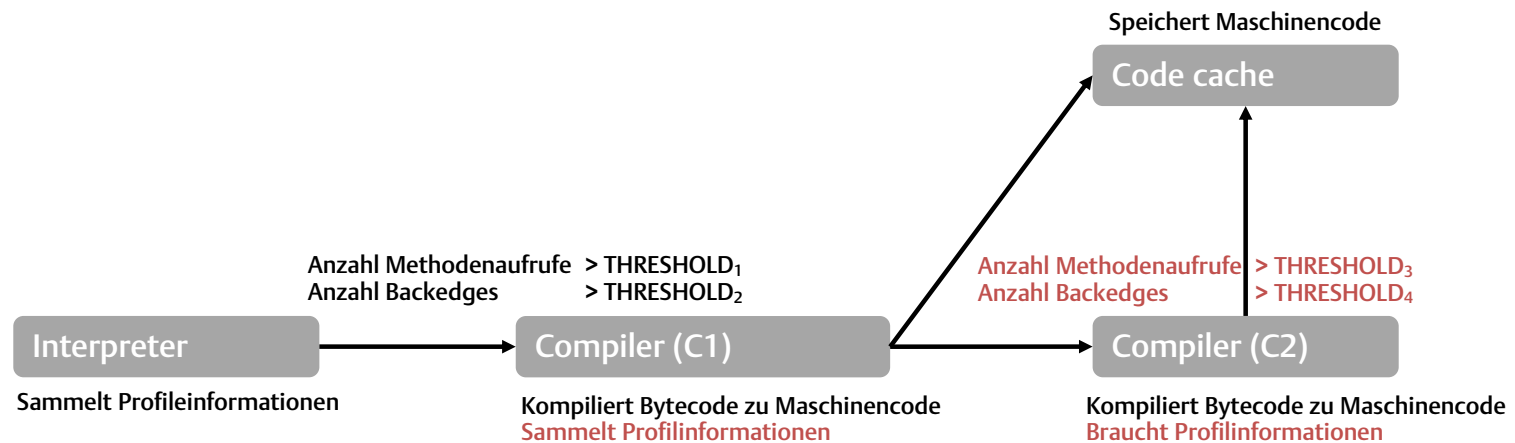
$$\text{Aufw\u00e4rmezeit (C1)} < \text{Aufw\u00e4rmezeit (Tiered)} \leq \text{Aufw\u00e4rmezeit (C2)}$$

# Schnelleres Profiling

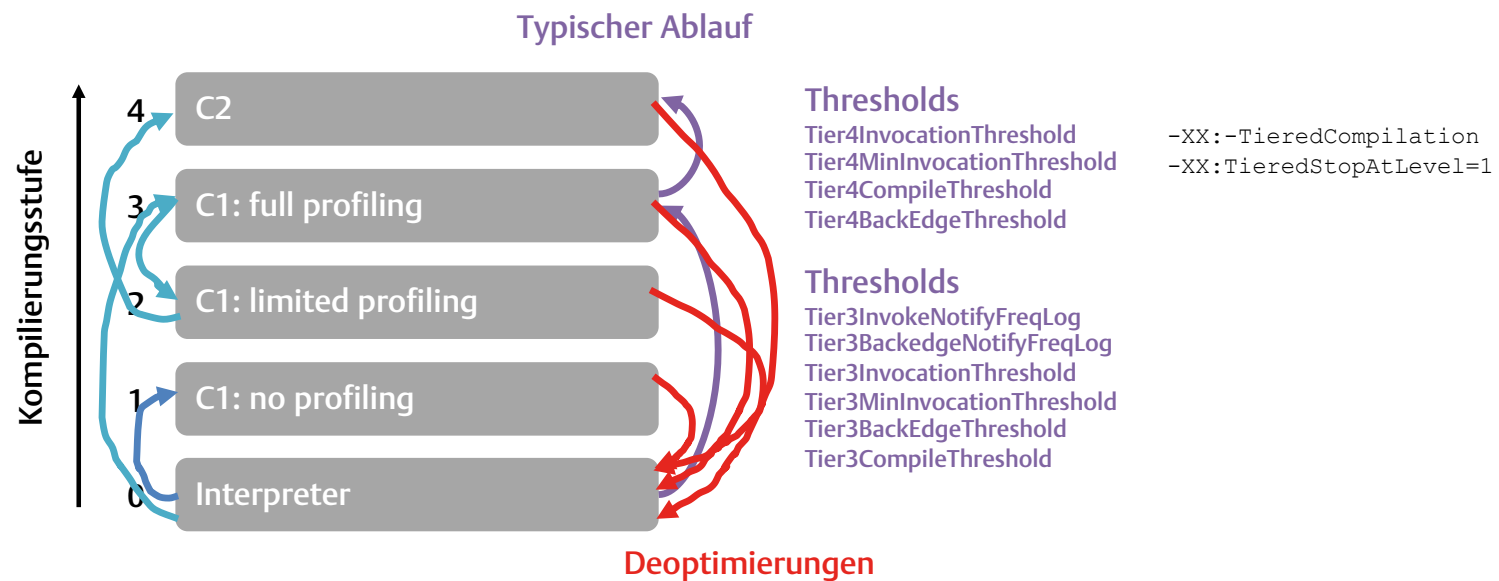


# Das Leben einer Methode mit Tiered Compilation

(Geschichte von vorher ergänzt)



# Kompilierungsstufen (detaillierte Ansicht)



# Zusammenfassung: Tiered Compilation

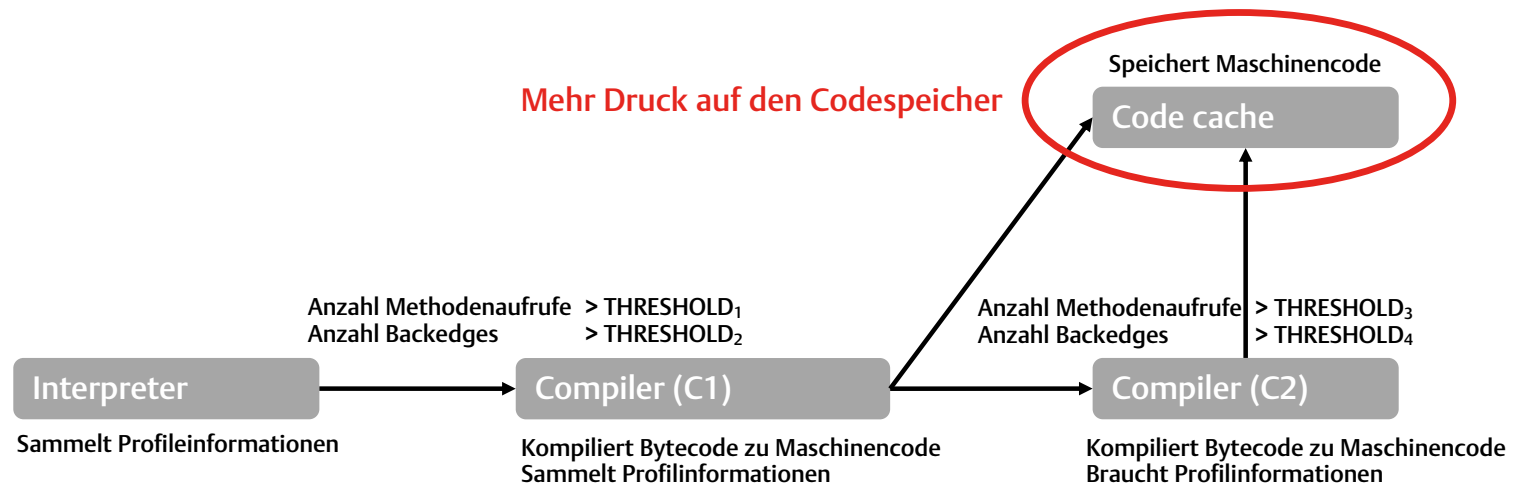
## Kombiniert Nutzen vom Interpreter, C1 und C2

- Bessere Leistung der VM

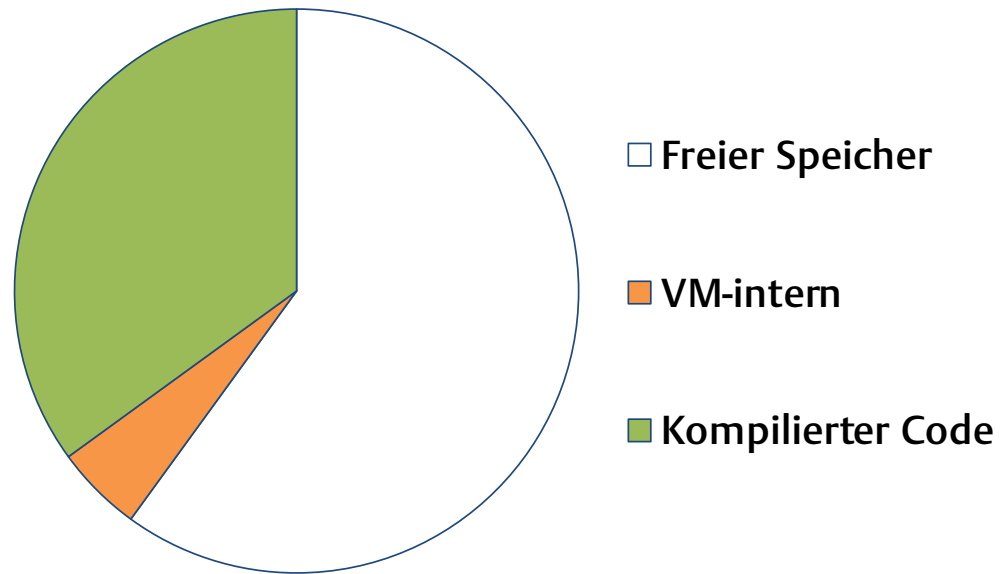
## Nachteile

- Komplexe Implementierung und Konfigurierung
- Mehr Druck auf den Codespeicher

# Das Leben einer Methode mit Tiered Compilation

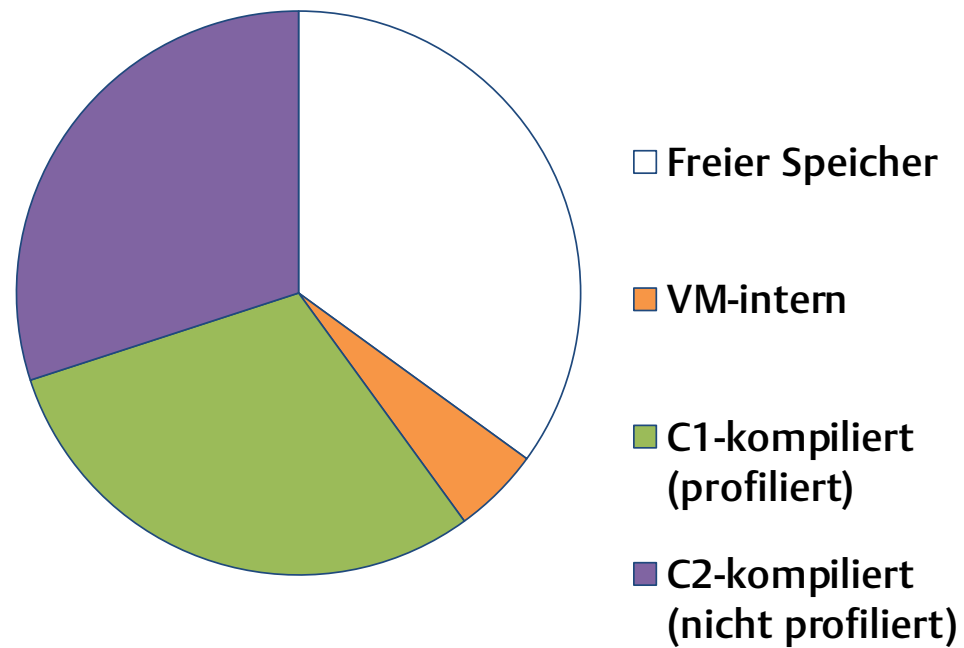


## Gebrauch des Code Cache (JDK 6 und 7)





## Gebrauch des Code Cache (ab JDK 8 mit Tiered Compilation)

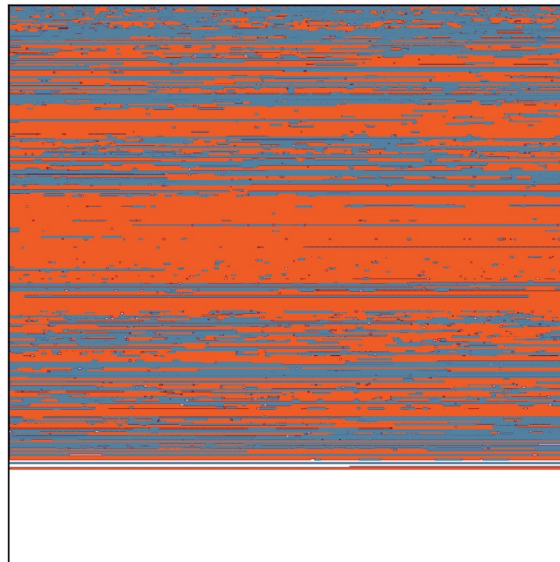


# Herausforderungen

**Mit Tiered Compilation VM generiert 4X mehr Code als ohne**

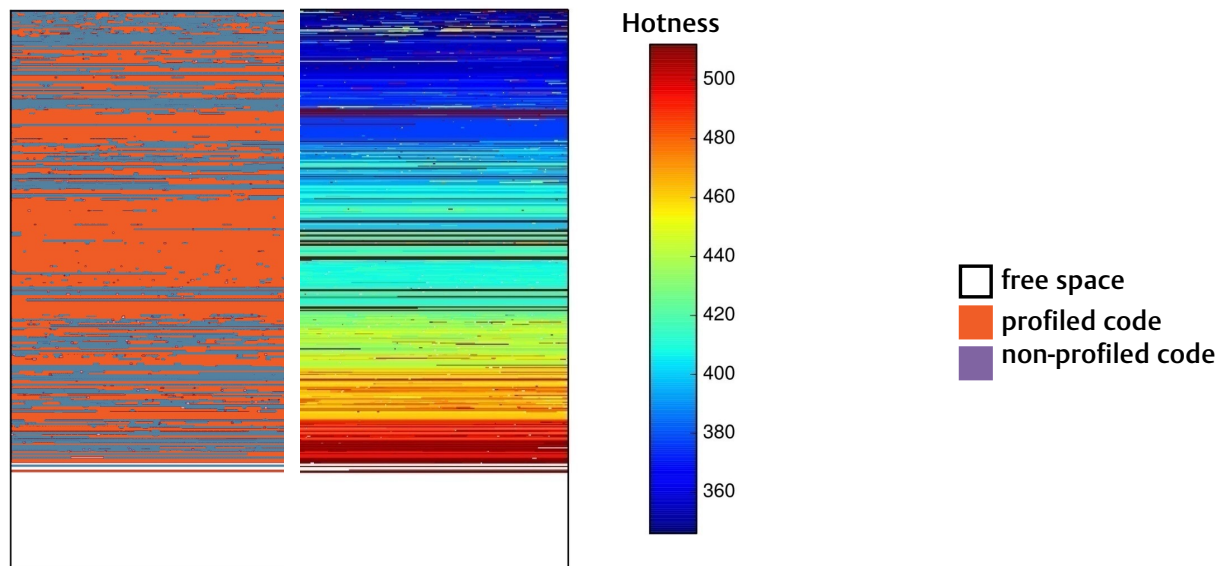
**All Code ist in einem Code Cache gespeichert**

- Hohe Fragmentierung
- Schlechte Lokalität



□ free space  
■ profiled code  
■ non-profiled code

**Gebrauch des Code Cache**



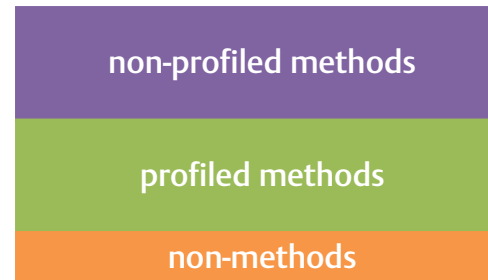
**Gebrauch des Code Cache**

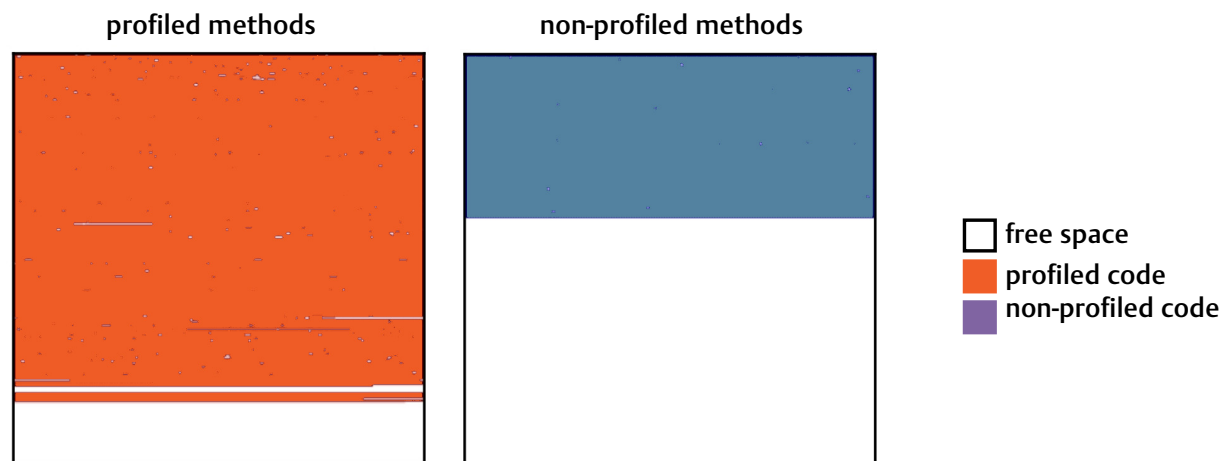
# Design

## Ohne Segmented Code Cache

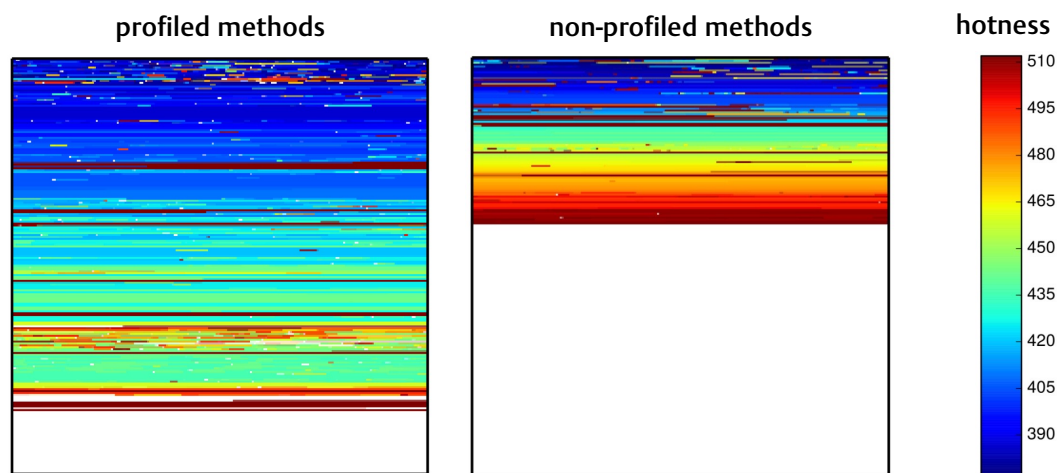


## Mit Segmented Code Cache





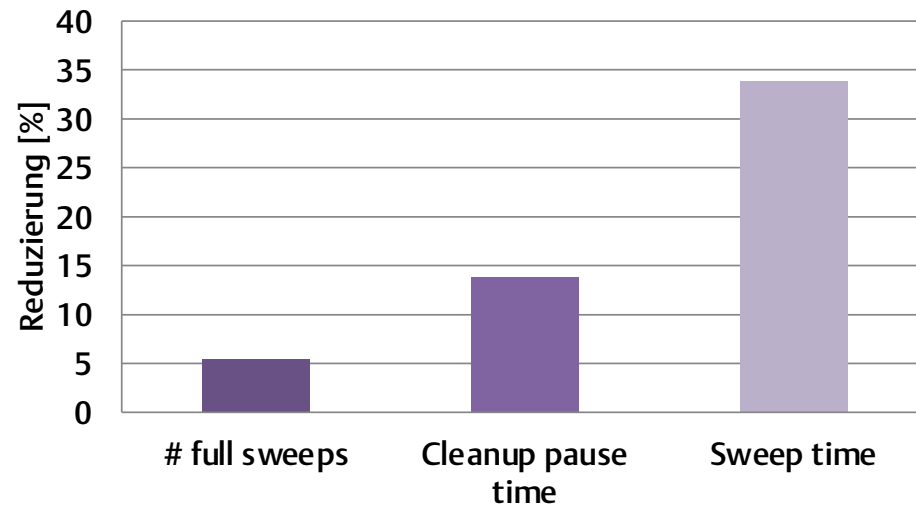
**Segmented Code Cache: Resultate**



**Segmented Code Cache: Resultate**

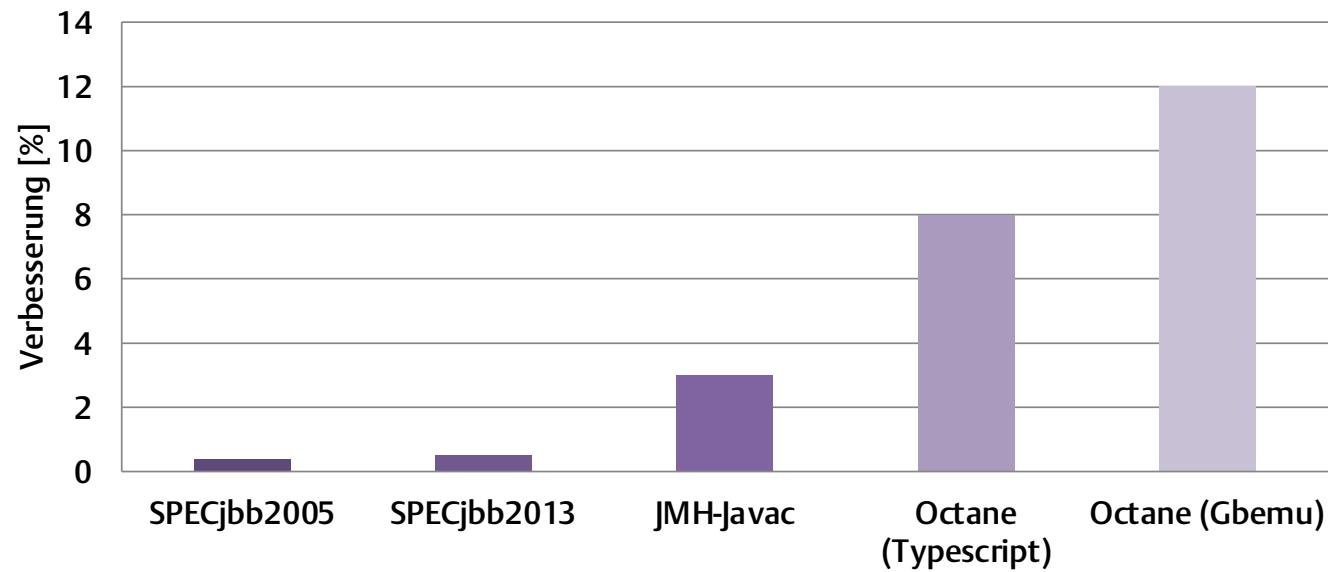
# Evaluation: Reaktionsgeschwindigkeit

## Sweeper (GC für kompilierten Code)





# Evaluation: Leistung



# Segmented Code Cache (Zusammenfassung)

Segmented Code Cache verbessert **Leistung** und **Reaktionsgeschwindigkeit**

- Reduziert Fragmentierung
- Reduzierter Mehraufwand des Sweepers
- Verbesserte Codelokalität

## Verfügbar mit Java 9

-XX:+SegmentedCodeCache  
-XX:-SegmentedCodeCache

# Agenda

## Im Fokus heute: Kompilierung in der VM

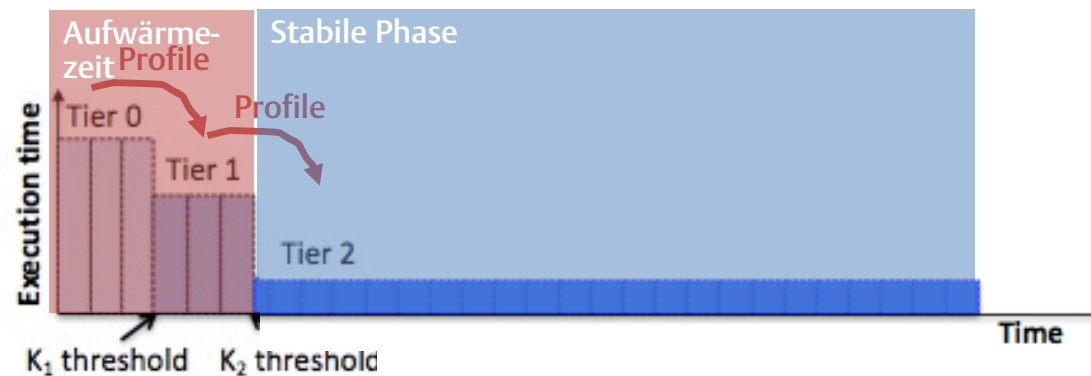
- Tiered Compilation
- Segmented Code Cache
- Profile Caching

# Erwartete Leistung mit Tiered Compilation

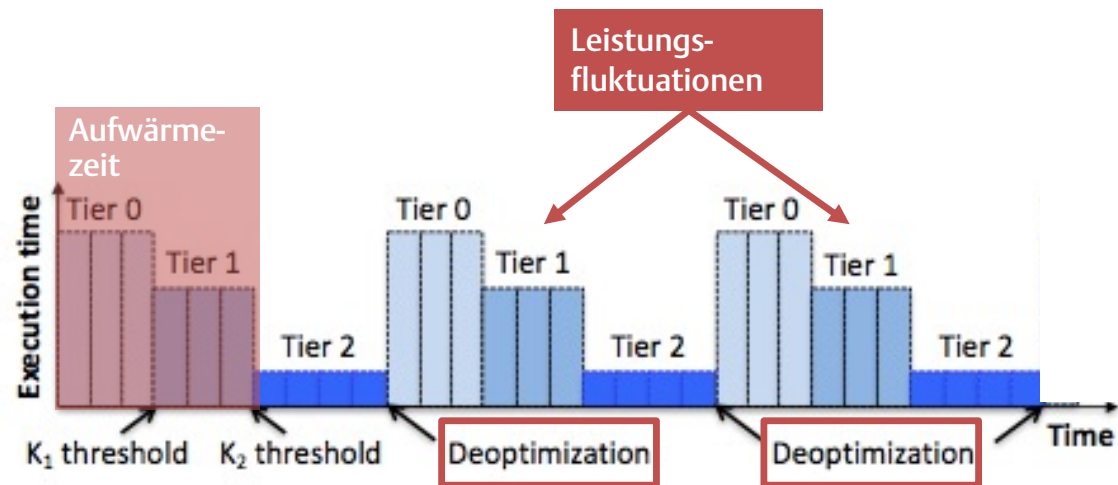
## Aus der Perspektive der Ausführung einer Methode

- Methode wird mit dem selben Zustand und Parameter aufgerufen

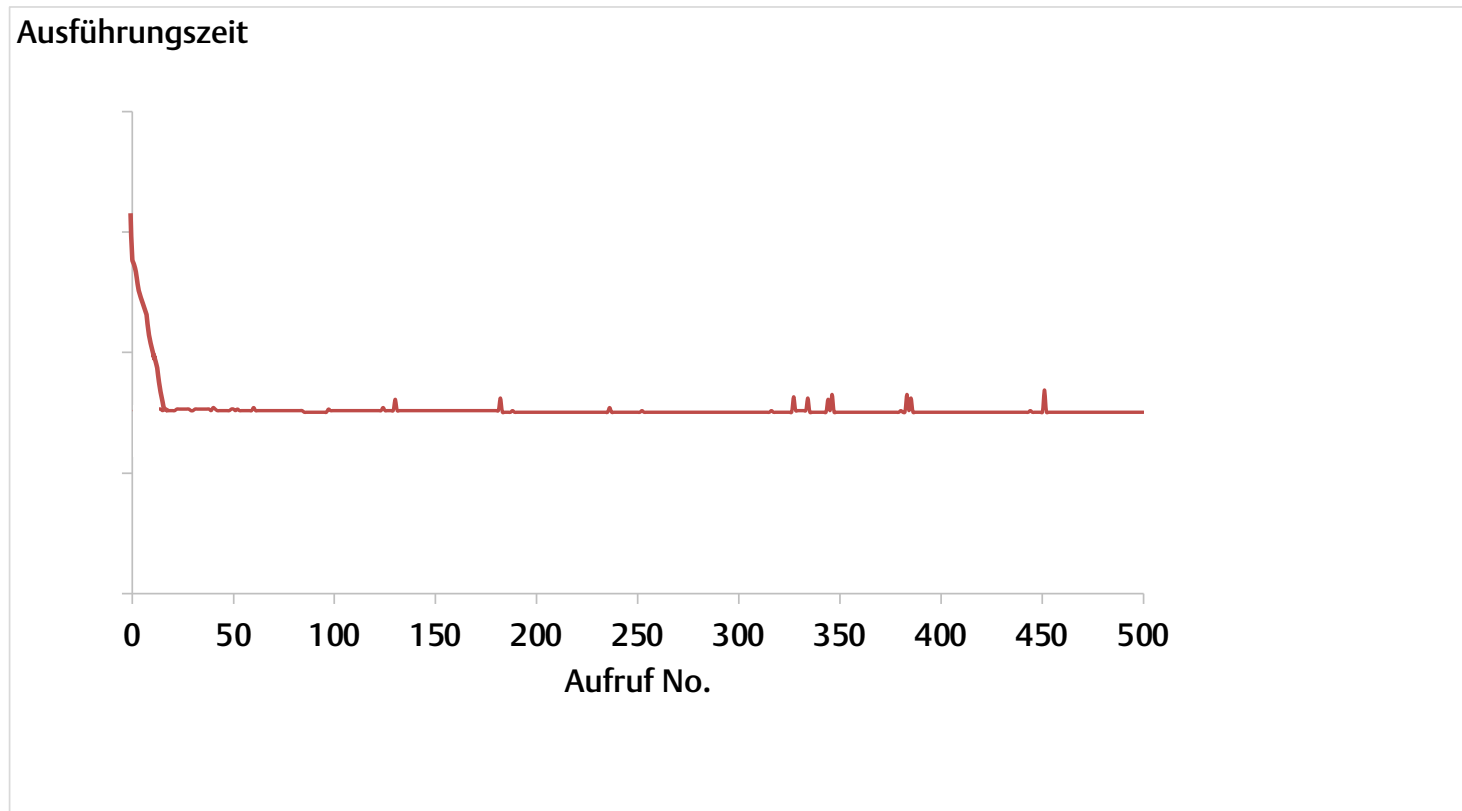
## Erwartete Leistung



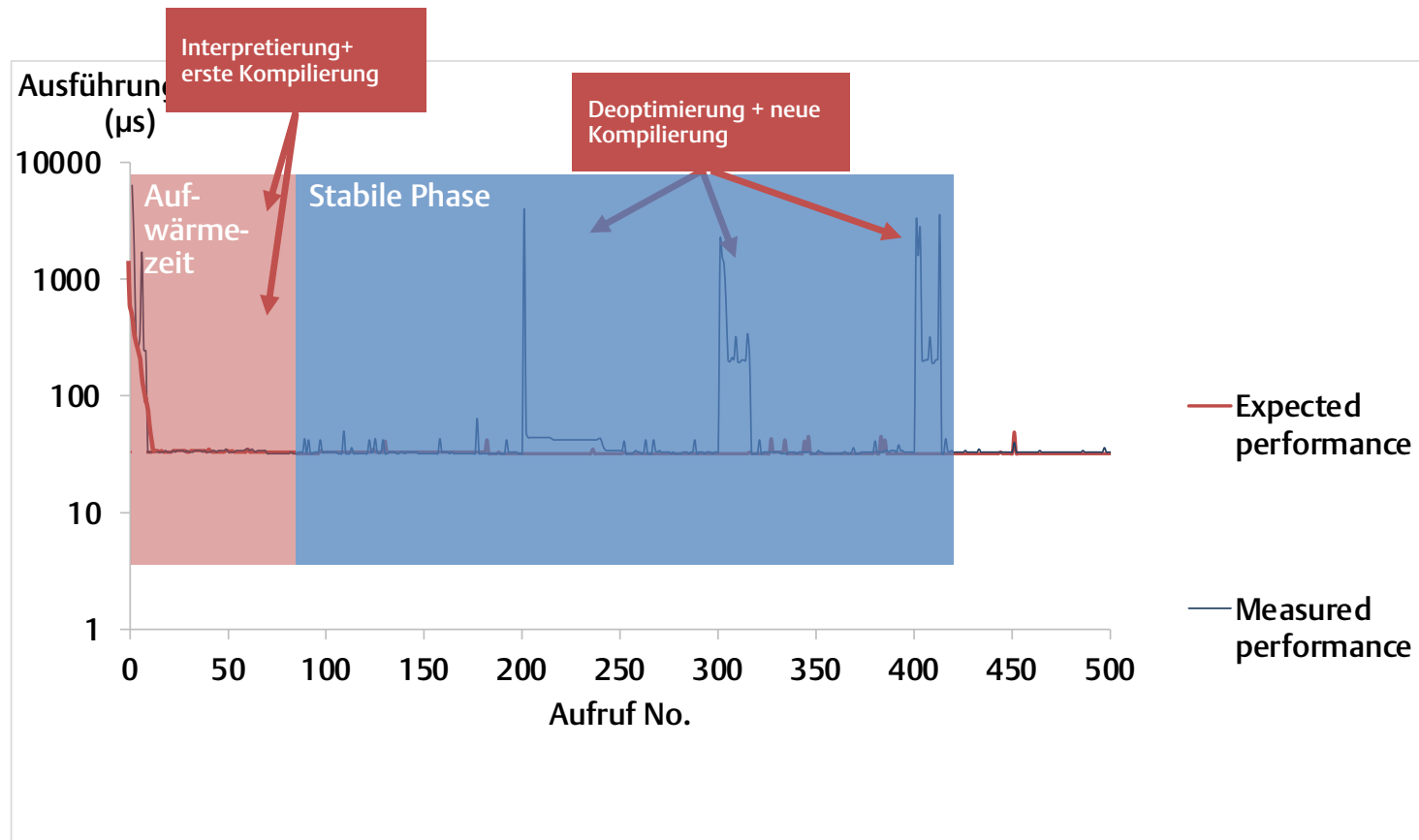
# Eigentliche Leistung mit Tiered Compilation



# Erwartete Leistung



# Gemessene Leistung



# Deoptimierungen

Compileroptimierungen basieren sich auf **optimistische Annahmen**

**Annahmen basiert auf Profilinformatoren**

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

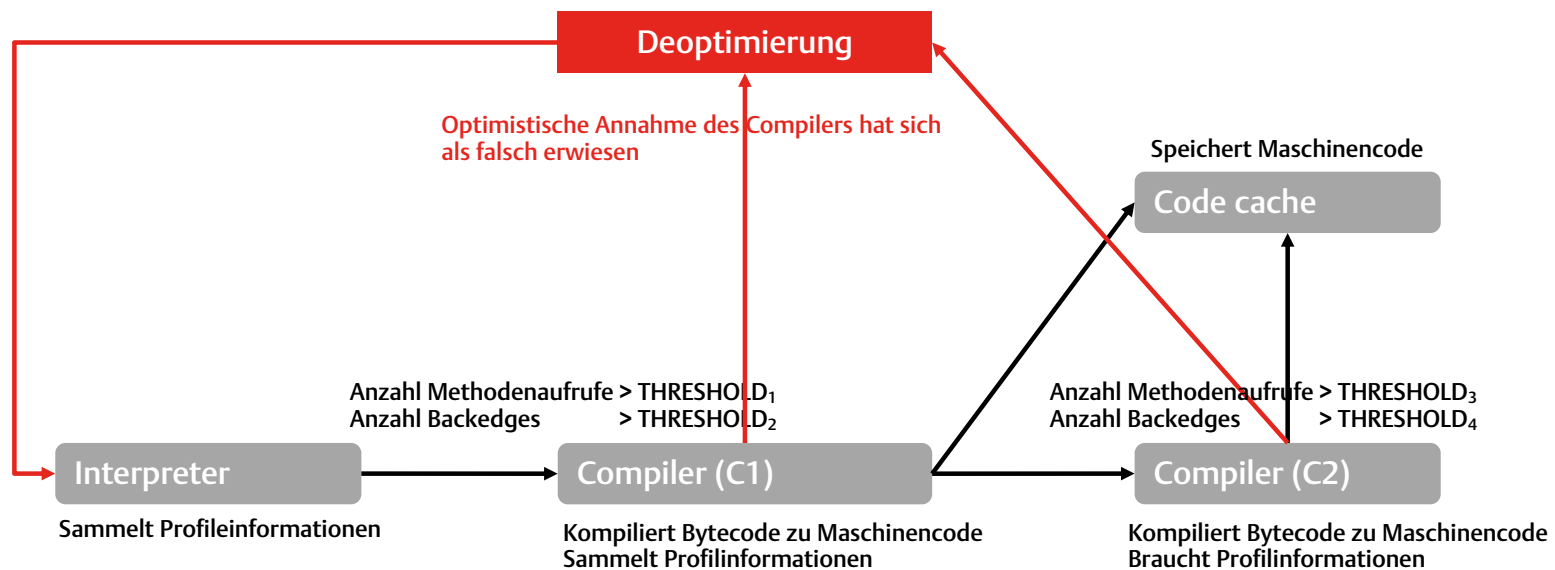
**Grundlegendes Prinzip: Vergangenheit = Zukunft**

**Falls optimistische Annahme des Compilers nicht mehr gilt: Deoptimierung**

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

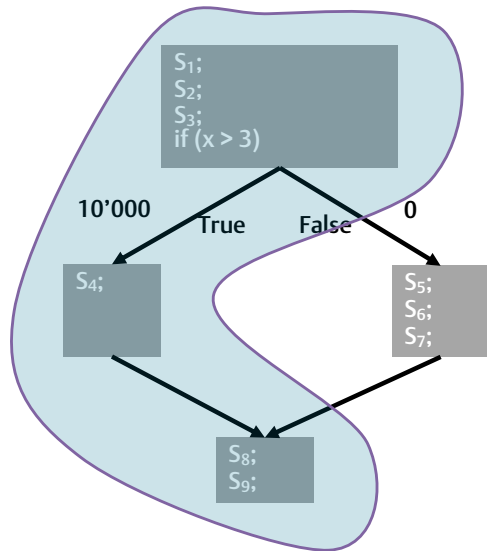


# Das Leben einer Methode: Komplette Geschichte



# Beispieloptimierung: “Hot path”-Kompilierung

## Kontrollflussgraph



## Generierter Code



# Beispiel: Producer-Methode

```
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;
}
```

# Synthetic producer-consumer workload

## 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;
}
```

## Consumer

```
public void consume(Producer prod) {
    for (int item = 0; item <= 500; ++item) {
        long result = prod.produce(item);
        // Do something with '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



# Beispieloptimierung 2: Virtual Call Inlining

## Klassenhierarchie

```
class A {  
    void bar() {  
        S1;  
    }  
}
```

loaded

```
class B extends A {  
    void bar() {  
        S2;  
    }  
}
```

not loaded

## Methode zum Kompilieren

```
void foo() {  
    A a = create(); // return A or B  
    a.bar();  
}
```

Compiler: Inline call?

Ja.

# Beispieloptimierung 2: Virtual Call Inlining

## Klassenhierarchie

```
class A {  
    void bar() {  
        S1;  
    }  
}
```

loaded

```
class B extends A {  
    void bar() {  
        S2;  
    }  
}
```

not loaded

## Methode zum Kompilieren

```
void foo() {  
    A a = create(); // return A or B  
    S1;  
}
```

Compiler: Inline call?

Ja.

## 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

## Klassenhierarchie

```
class A {  
    void bar() {  
        S1;  
    }  
}
```

loaded



```
class B extends A {  
    void bar() {  
        S2;  
    }  
}
```

loaded

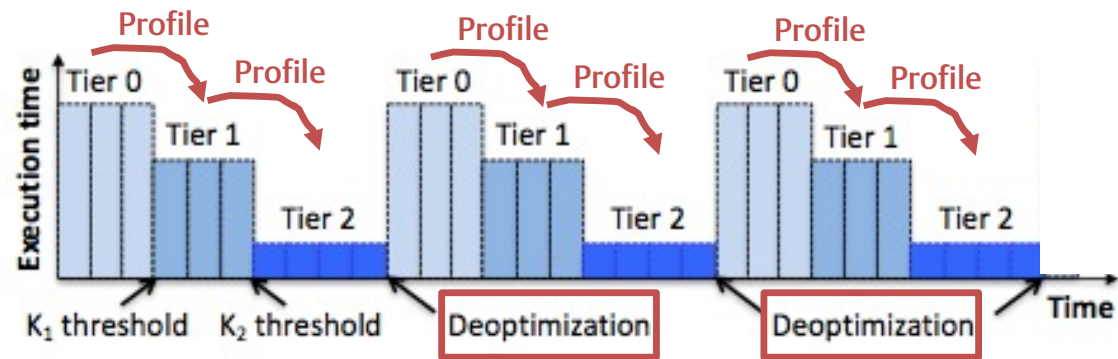
## Methode zum Kompilieren

```
void foo() {  
    A a = create(); // return A or B  
    a.bar();  
}
```

Compiler: Inline call?

Nein.

# Profilieren mit Deoptimierungen



# Wie erreicht man Leistungsstabilität?

**Kompiler muss wissen: Alle Pfade werden erreicht**

**Einige (vielleicht nicht so gute) Ideen:**

**1. Ausschalten** von "Hot path"-Kompilierung

Allgemein: Verbot optimistischer Annahmen

Mögliches Resultat: Schlechte Leistung

**2. Aufwärmen** der VM vor produktivem Einsatz

Wann ist die VM aufgewärmt? Nach 20 Minuten? Wenn die Leistung gut genug ist?

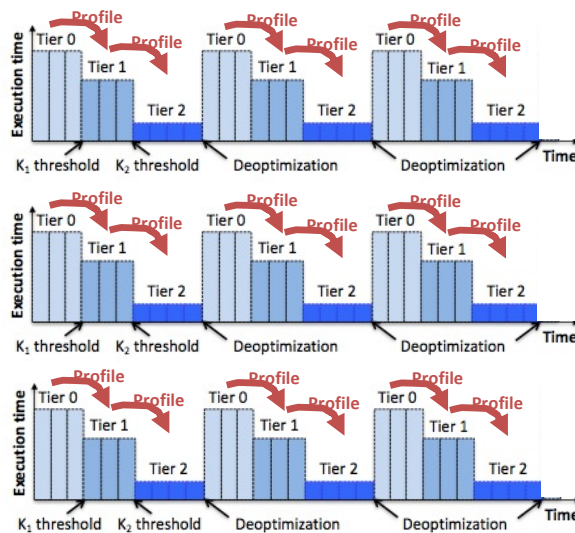
**3. Training** der VM vor dem produktiven Einsatz

Kritische Methoden werden mit entsprechenden Parameter und Zustand aufgerufen

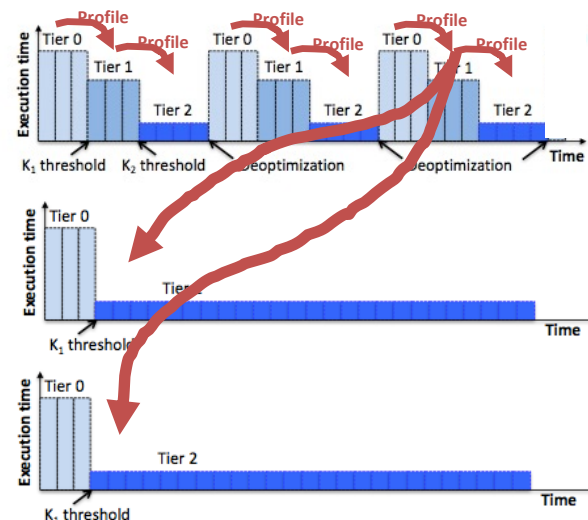
Kompliziert: Es geht nicht nur um Programmpfade, sondern auch um Typen, Inlining, etc.

# Bemerkung: Profile sind weggeworfen

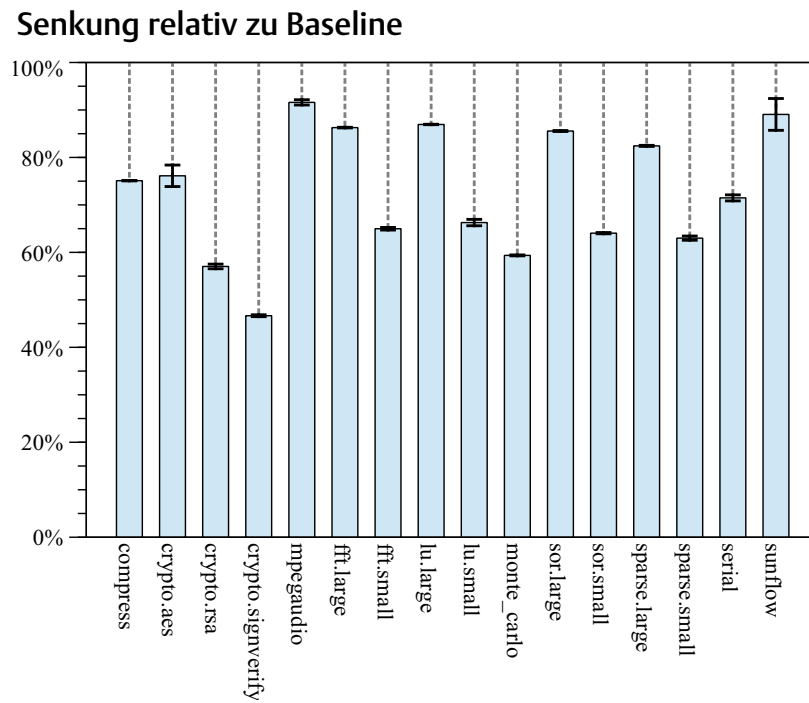
## Baseline HotSpot



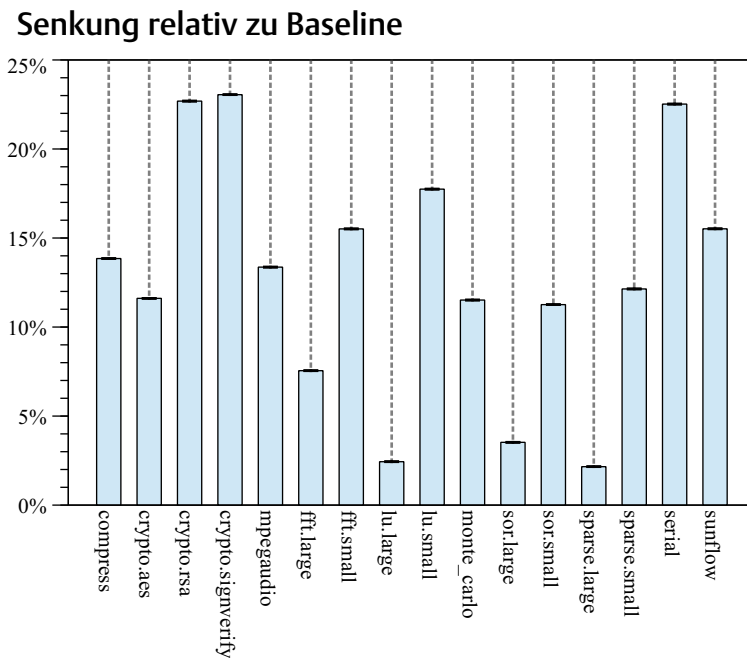
## Profile Caching



# Deoptimierungen



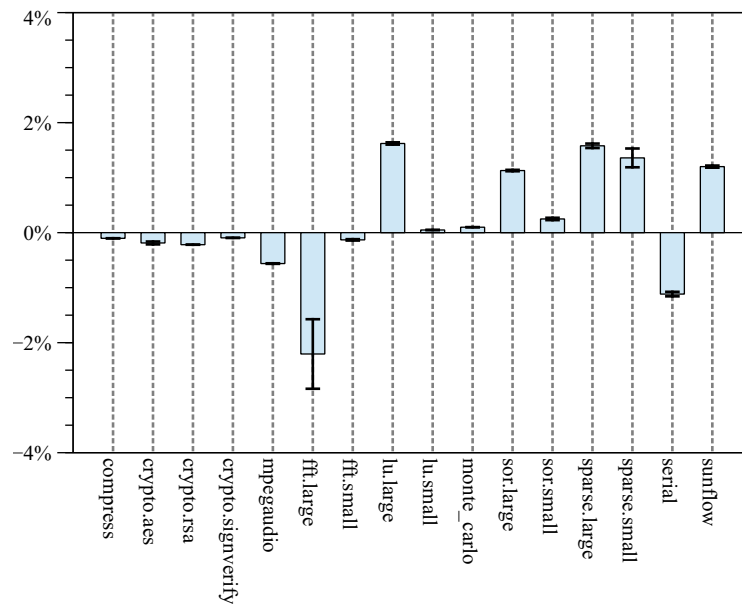
# Anzahl Kompilierungen





# End-to-end Leistung

Senkung relativ zu Baseline



# Profile Caching (Zusammenfassung)

## Reduziert Leistungsfluktuationen

- Weniger Deoptimierungen
- Weniger Kompilierungen

**End-to-end Laufzeit ist nicht deutlich beeinträchtigt**

# Zusatzmaterial

## Zum Thema JIT-Kompilierung

- Fachartikel über Profile Caching: [w11/manlang\\_2017.pdf](#)
- Detaillierte Erklärung von Deoptimierungen und des Producer-Consumer Beispiels

# Zusammenfassung

## Kompilierung in der VM

- Mehrstufige Kompilierung
- Segmentierung des Codespeichers
- Deoptimierungen und Caching von Programmprofilen