Application Performance Management

Frühling 2021

JIT-Kompilierung

Zoltán Majó

Kurzfassung

Im Fokus heute: Kompilierung in der VM

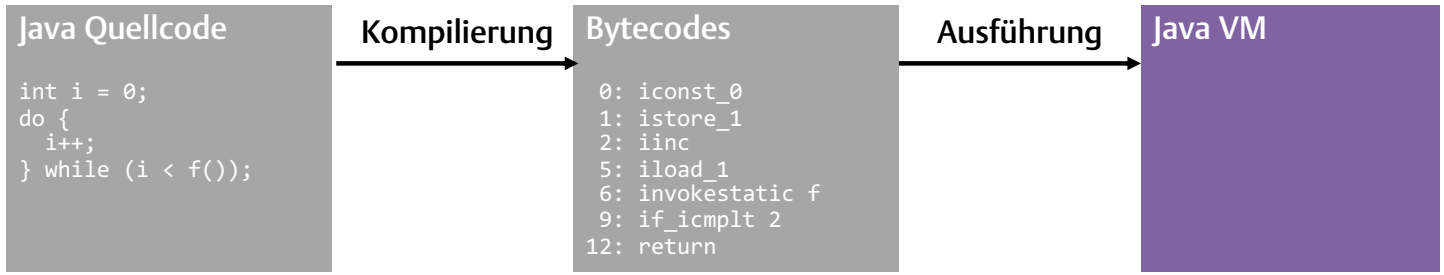
- Mehrstufige Kompilierung
- Segmentierung des Codespeichers
- Caching von Programmprofilen

Kurzfassung

Im Fokus heute: Kompilierung in der VM

- Mehrstufige Kompilierung – Tiered compilation — Java 8
- Segmentierung des Codespeichers – Segmented code cache — Java 9
- Caching von Programmprofilen – Profile caching — in der Zukunft (vielleicht)

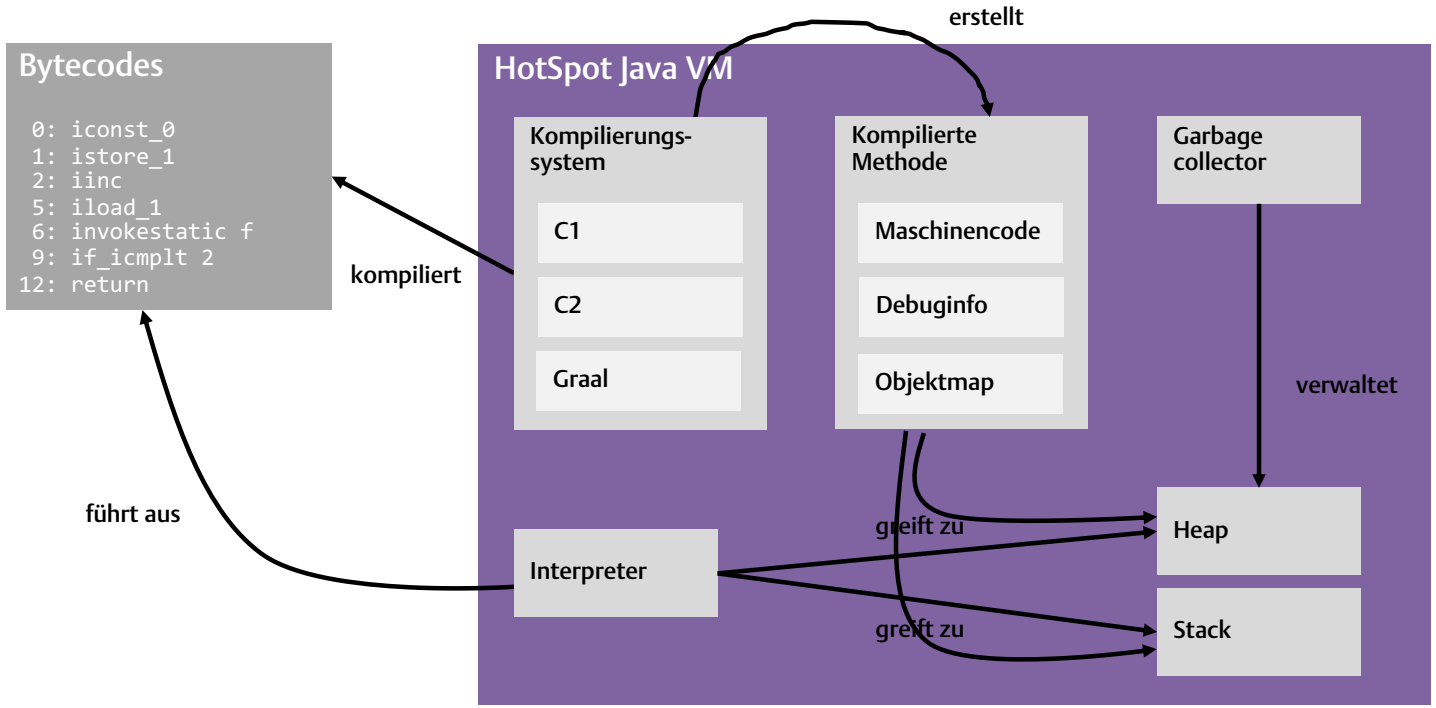
Kompilierung *für* die Java VM



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

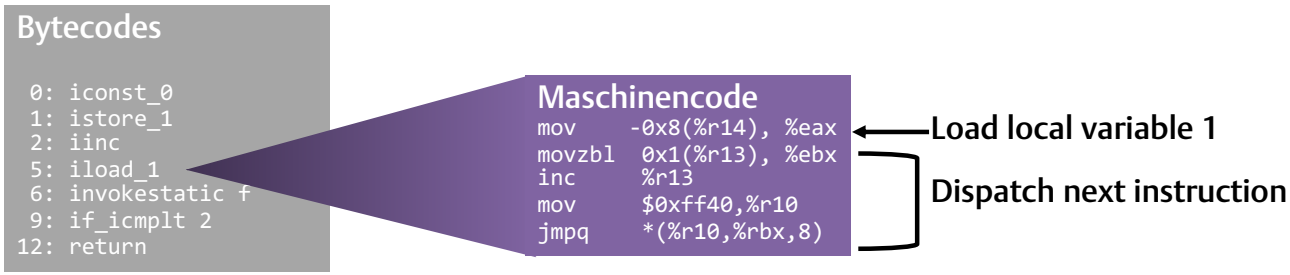
Kompilierung *in* der JVM



Interpretierung

Template-based interpreter

- Zuordnung Bytecode-Instruktion Maschinencode-Snippet



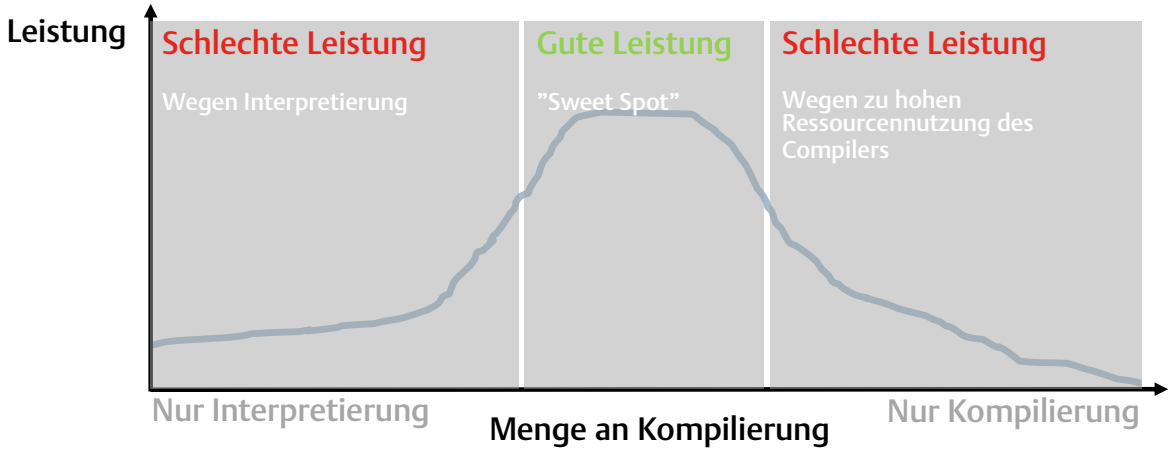
Kompilierungssystem: Leistungsverbesserung von etwa 100X

Kompilierung: Just-in-time (JIT)

JIT-Kompilierung passiert während Programmausführung

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

Kompromiss


Leistung
Schlechte Leistung
Wegen Interpretierung
Gute Leistung
"Sweet Spot"
Schlechte Leistung
Wegen zu hohen Ressourcennutzung des Compilers
Nur Interpretierung
Menge an Kompilierung
Nur Kompilierung

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

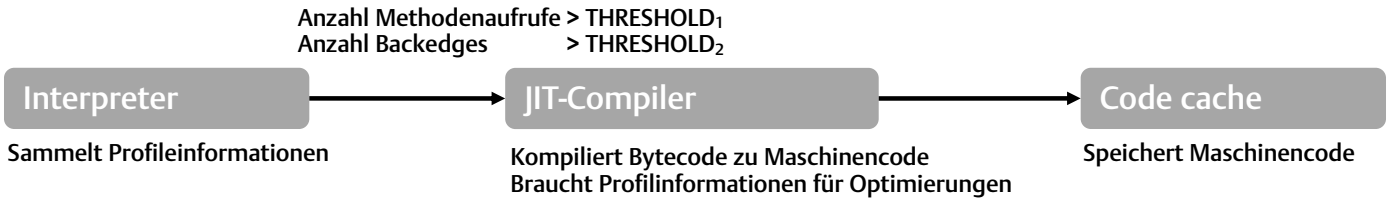
- Anzahl Methodenaufrufe
- Anzahl „Backedges“ (relevant auch für On-Stack Replacement)

Viel mehr vom„Verhalten“ einer Methode wird erfasst

- Profiling elementar für Kompileroptimierungen
- Mehr später

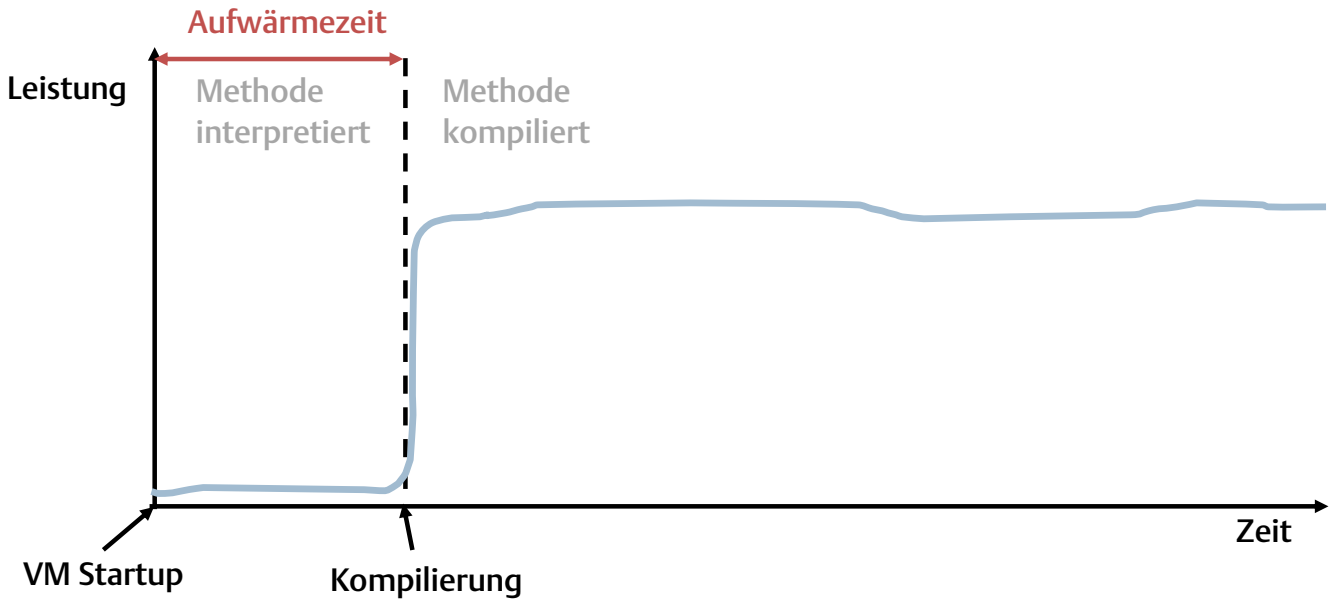
Das Leben einer Methode...

...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

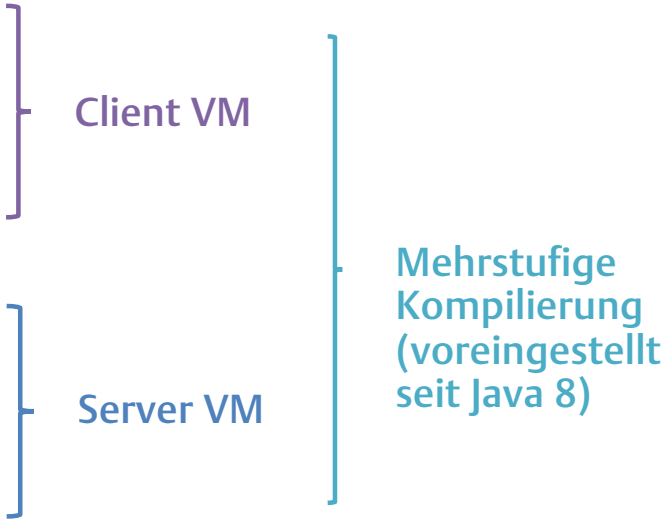
- Wenige Optimierungen
- Schnelle Kompilierung
- Niedriger Speichergebrauch

2. C2 Compiler

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

3. Graal Compiler

- Experimentell in JDK 10



Mehrstufige Kompilierung

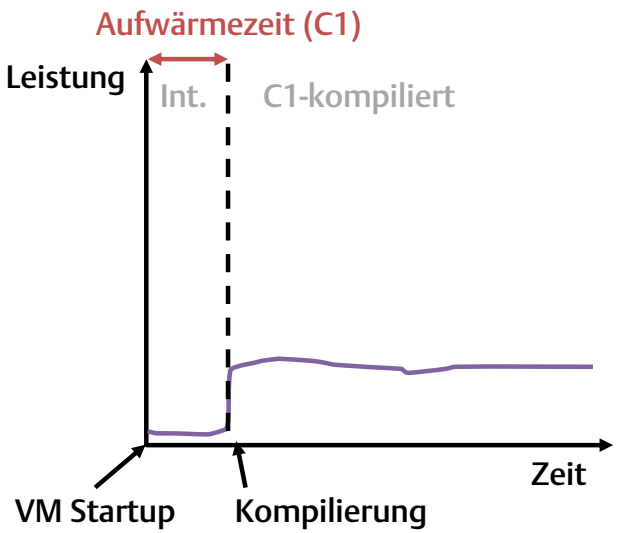
„Tiered Compilation“

Kombiniert das Nutzen vom

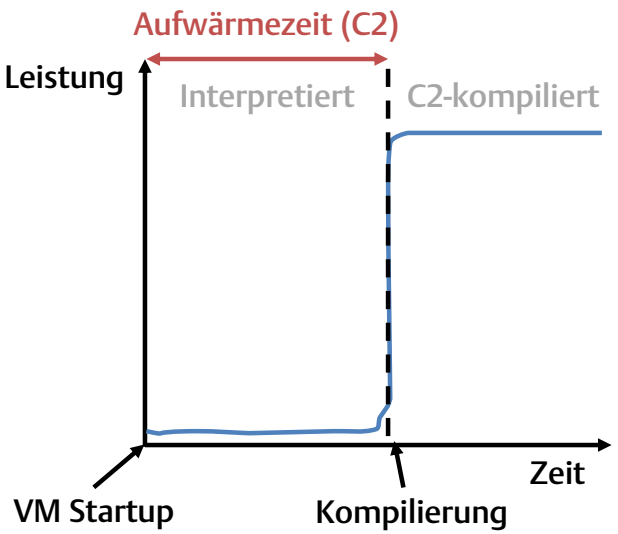
- Interpreter: schneller VM Start — **Tier 0**
- C1 Compiler: schnelle Kompilierung — **Tier 1**
- C2 Compiler: hohe maximale Leistung — **Tier 2**

Nutzen der mehrstufigen Kompilierung

Client VM (C1)

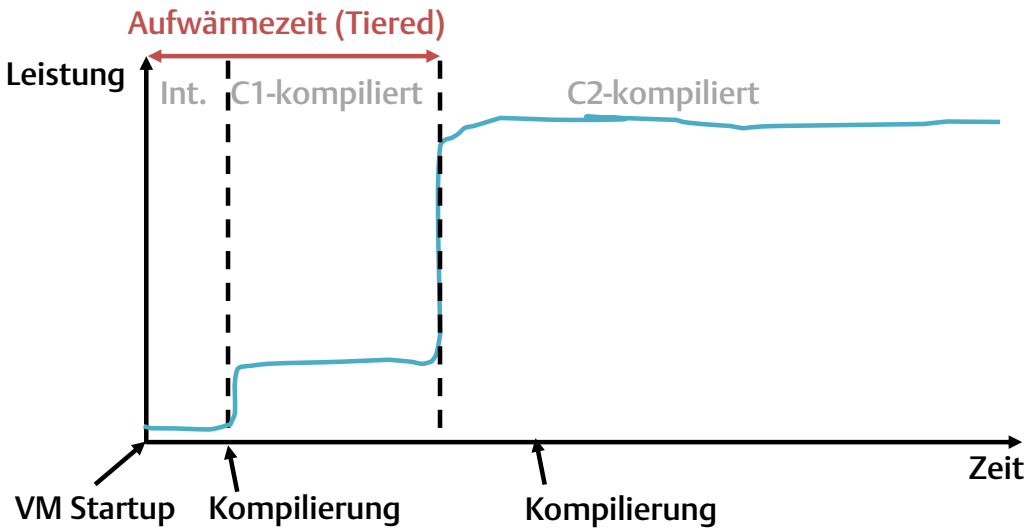


Server VM (C2)



Nutzen der mehrstufigen Kompilierung (Forts.)

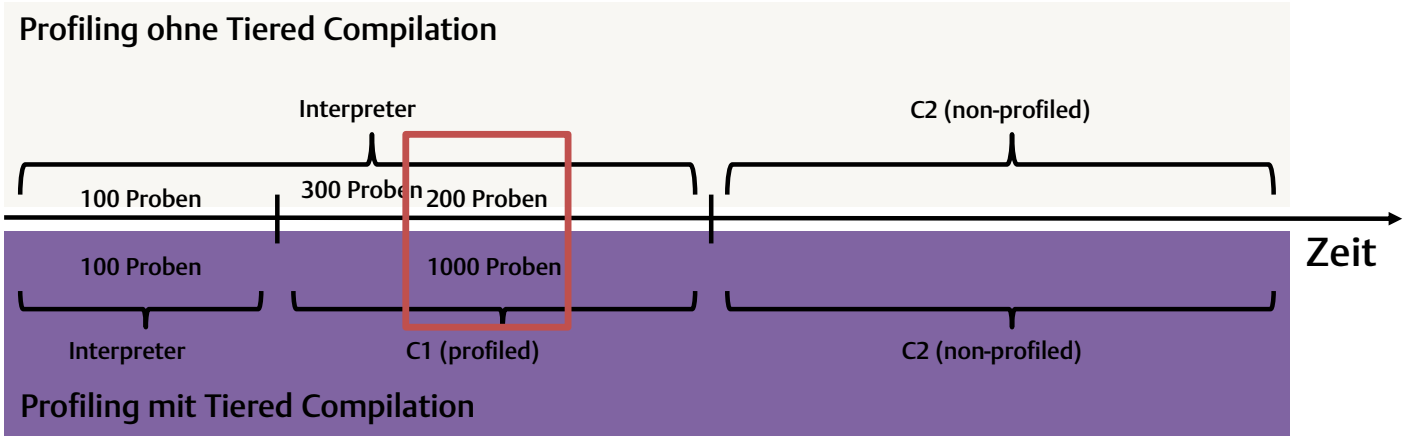
Mehrstufige Kompilierung



Bemerkung

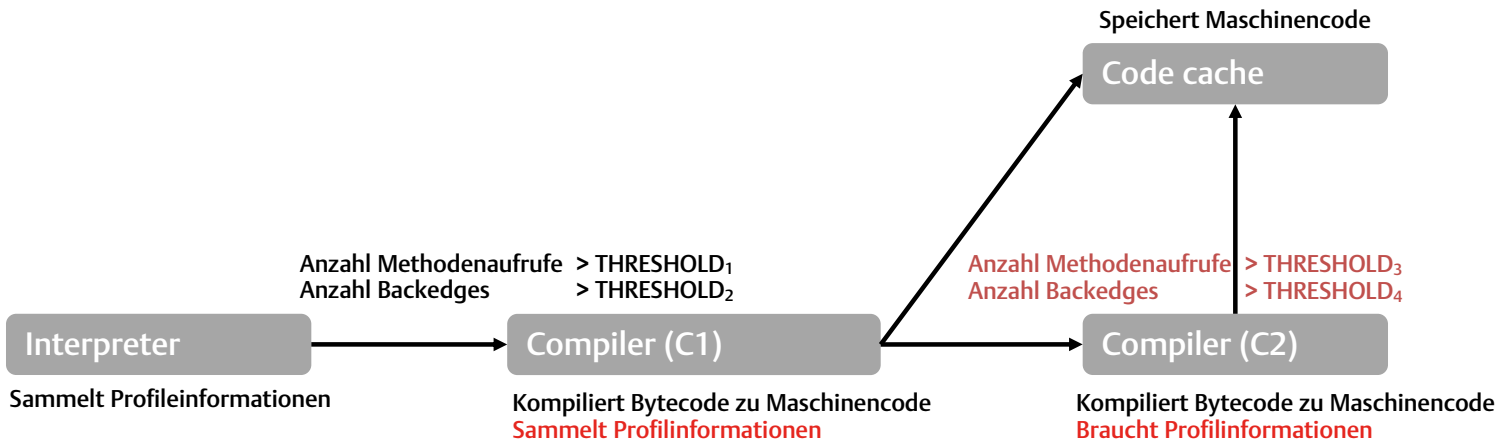
Aufwärmezeit (C1) < Aufwärmezeit (Tiered) ≤ Aufwärmezeit (C2)

Schnelleres Profiling

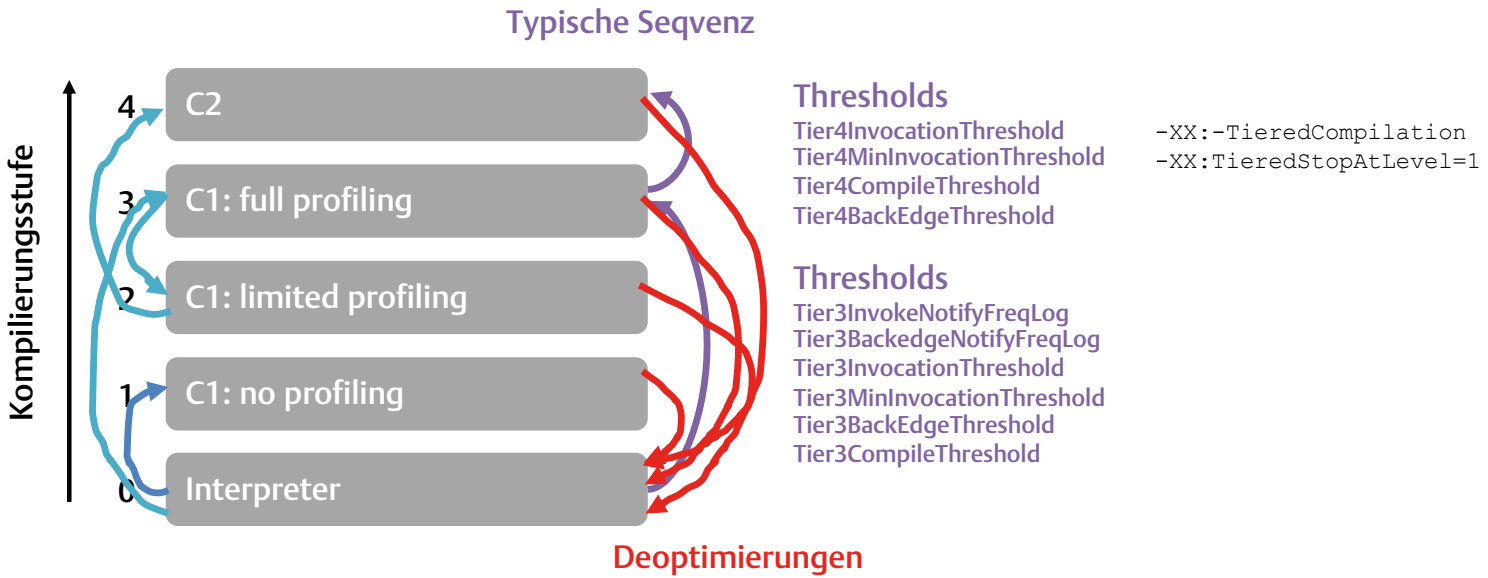


Das Leben einer Methode mit Tiered Compilation

(Geschichte von vorher ergänzt)



Kompilierungsstufen (detaillierte Ansicht)



Zusammenfassung: Tiered Compilation

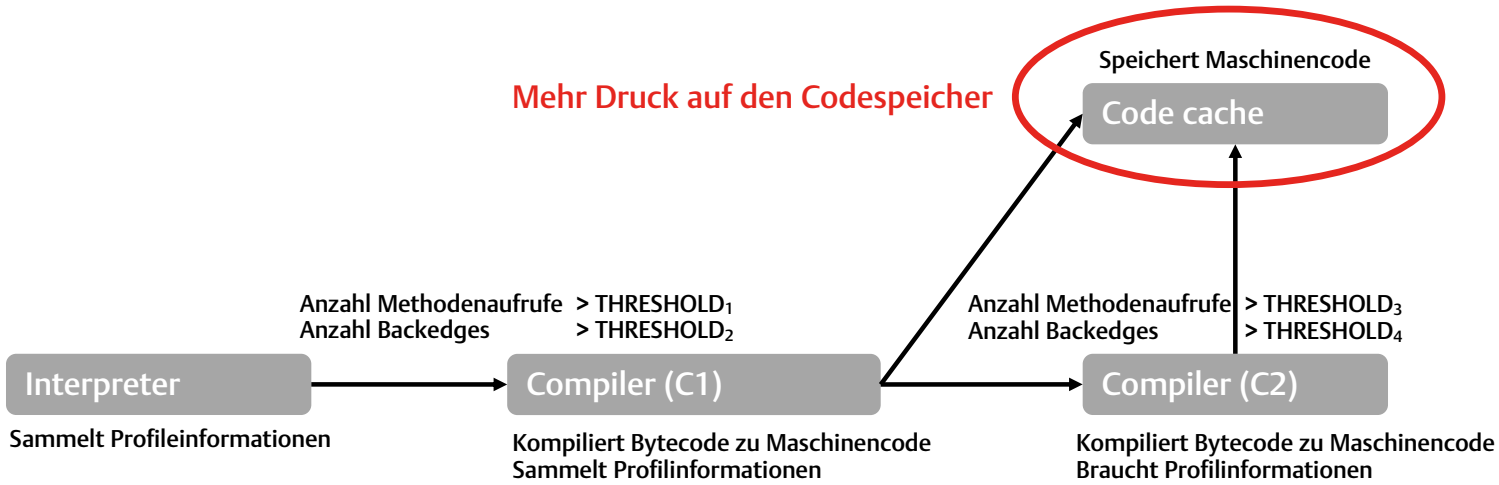
Kombiniert Nutzen vom Interpreter, C1 und C2

- Bessere Leistung der VM

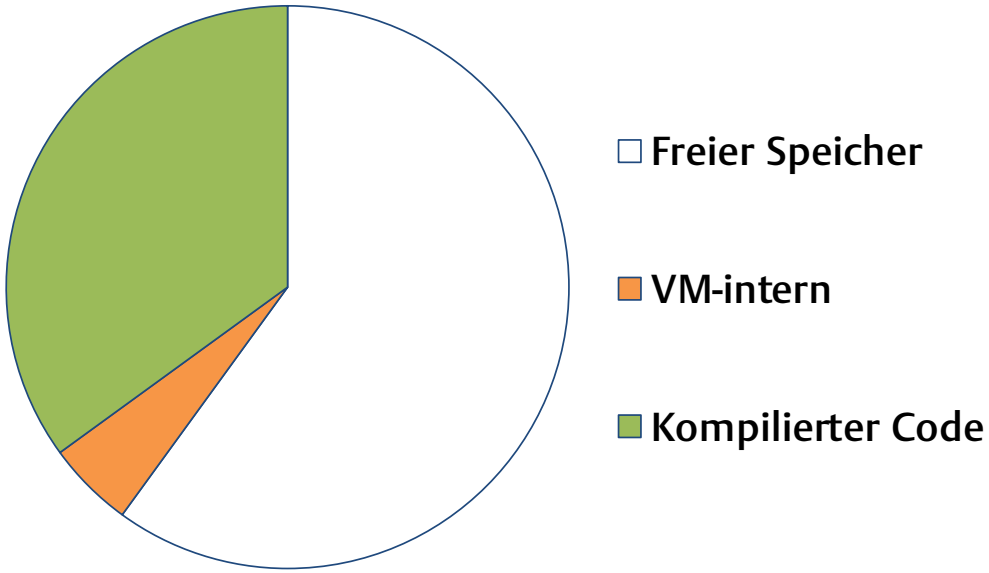
Nachteile

- Komplexe Implementierung und Konfigurierung
- Mehr Druck auf den Codespeicher

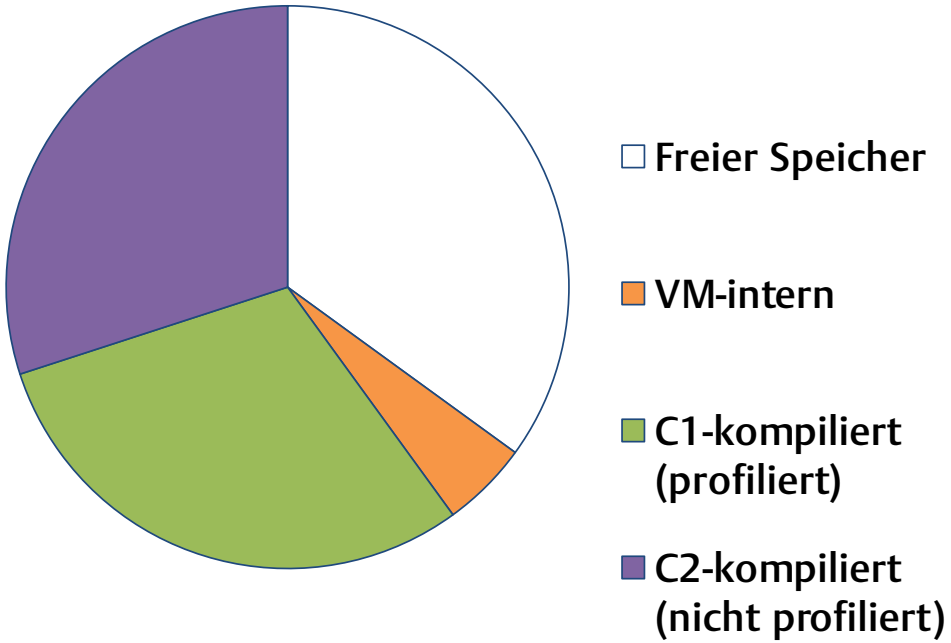
Das Leben einer Methode mit Tiered Compilation



Gebrauch des Code Cache (JDK 6 und 7)



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

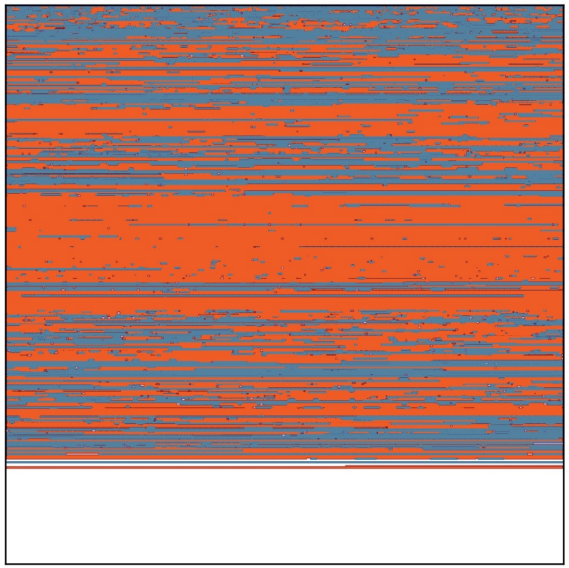


Herausforderungen

Mit Tiered Compilation VM generiert 4X mehr Code als ohne

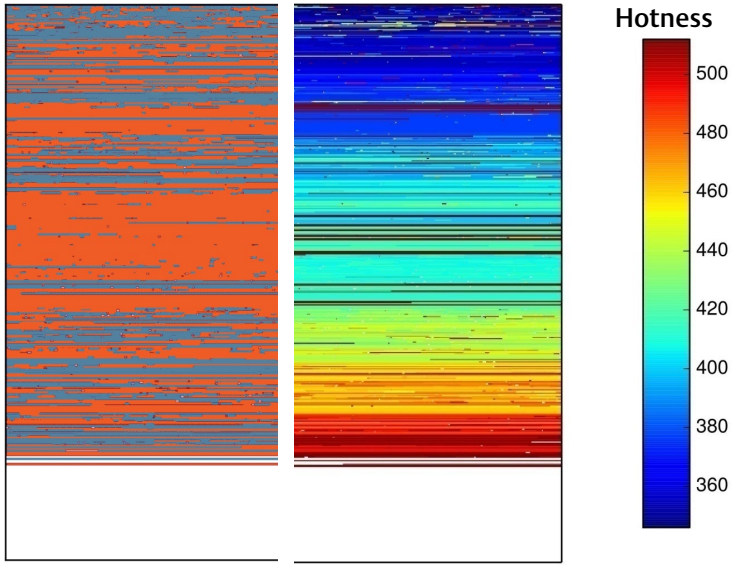
All Code ist in einem Code Cache gespeichert

- Hohe Fragmentierung
- Schlechte Lokalität



free space
profiled code
non-profiled code

Gebrauch des Code Cache



Gebrauch des Code Cache

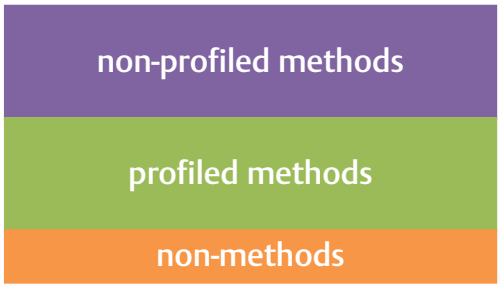
Design: Sorten kompilierter Code

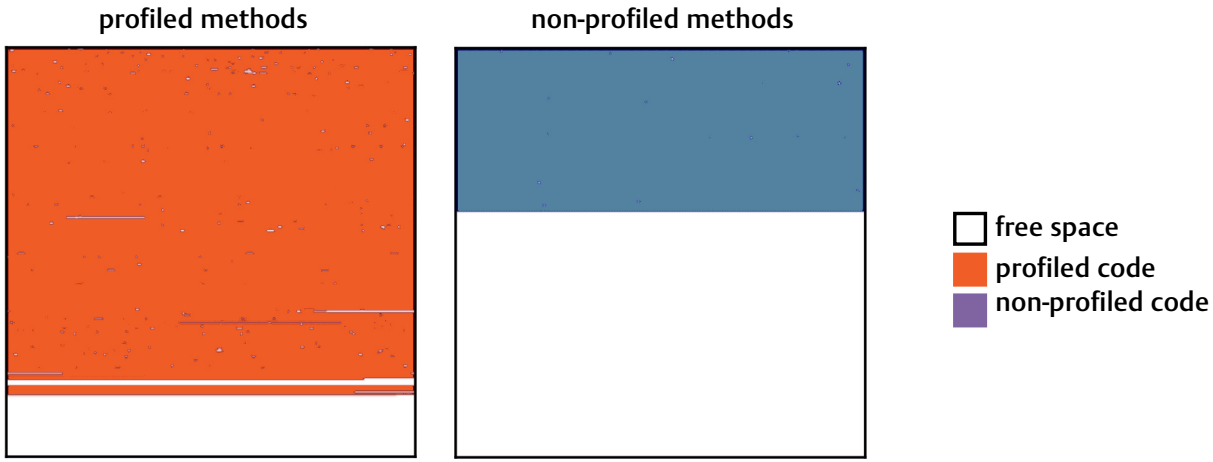
	Optimierungsgrad	Grösse	Kosten	Lebensdauer
Non-method code	optimiert	klein	niedrig	ewig
Profiled code (C1)	instrumentiert	mittel	niedrig	limitiert
Non-profiled code (C2)	hochoptimiert	gross	hoch	lang

Design

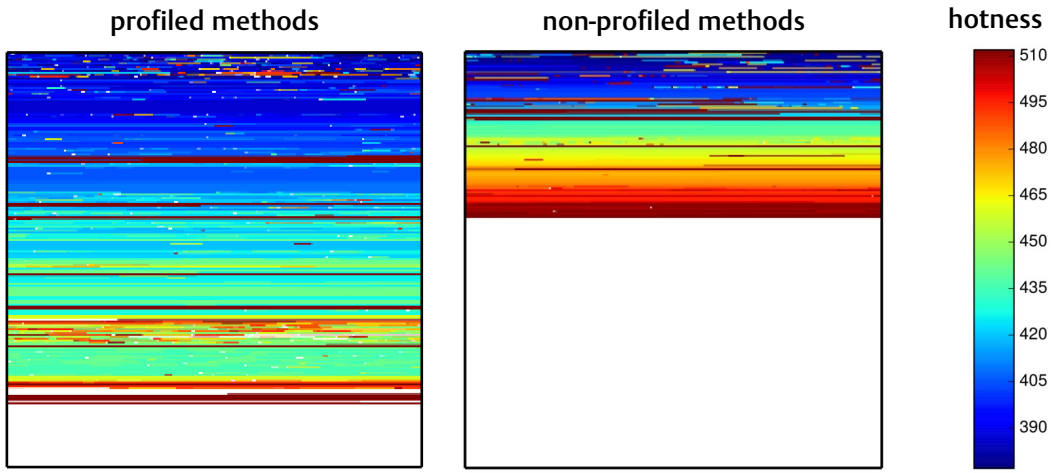
Ohne Segmented Code Cache

Mit Segmented Code Cache





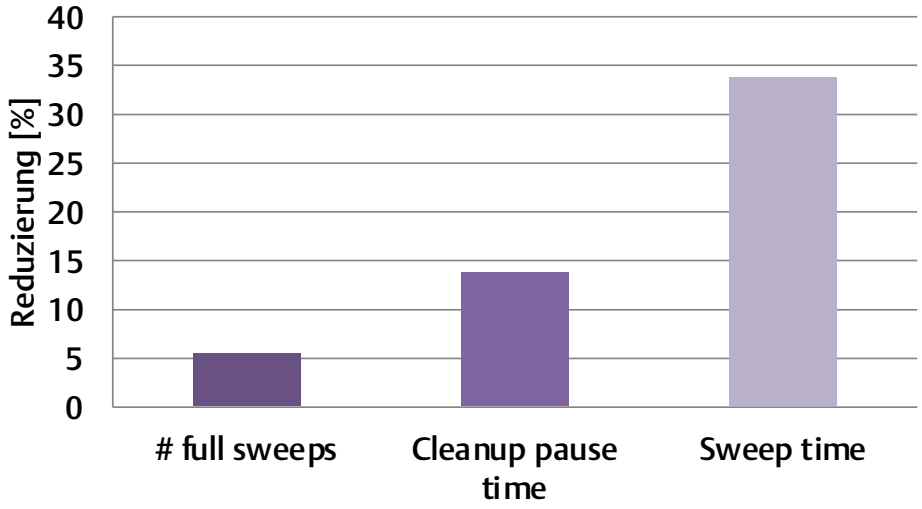
Segmented Code Cache: Resultate



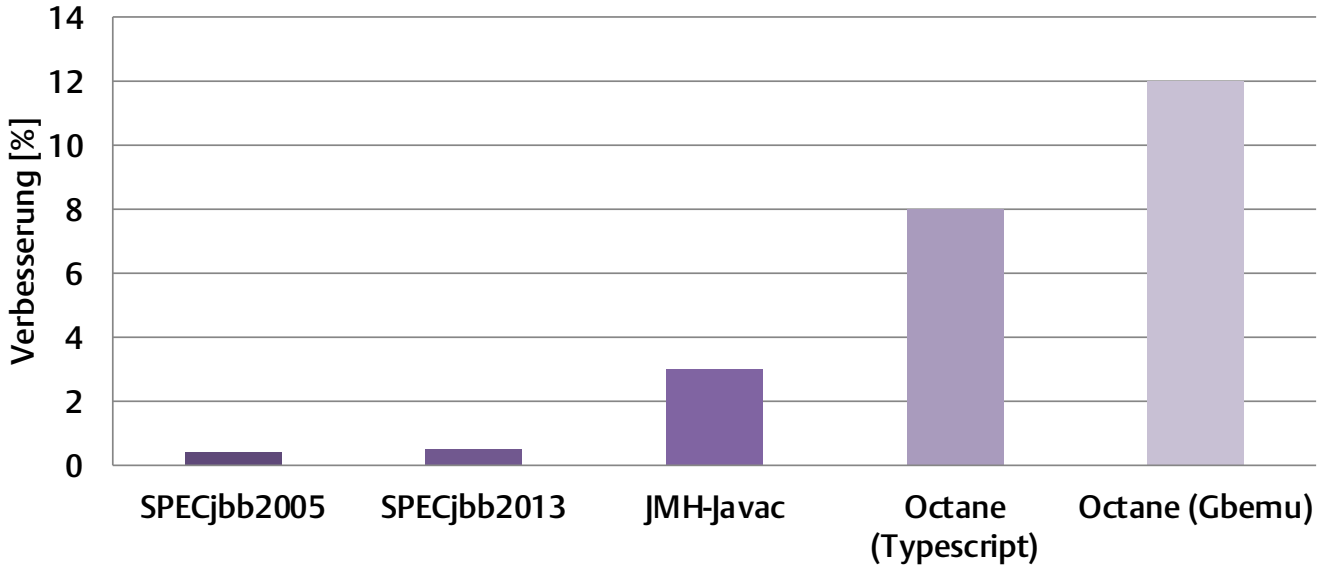
Segmented Code Cache: Resultate

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
```

Kurzfassung

Im Fokus heute: Kompilierung in der VM

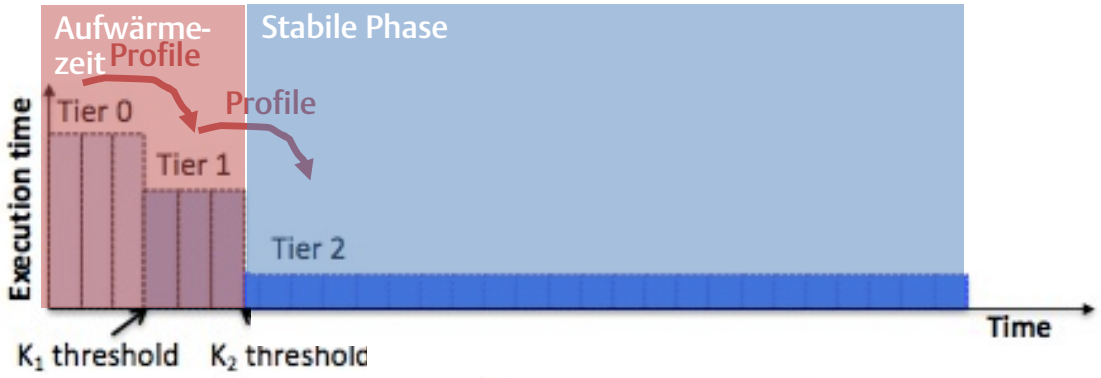
- Tiered Compilation
- Segmented Code Cache
- Profile Caching

Erwartete Leistung mit Tiered Compilation

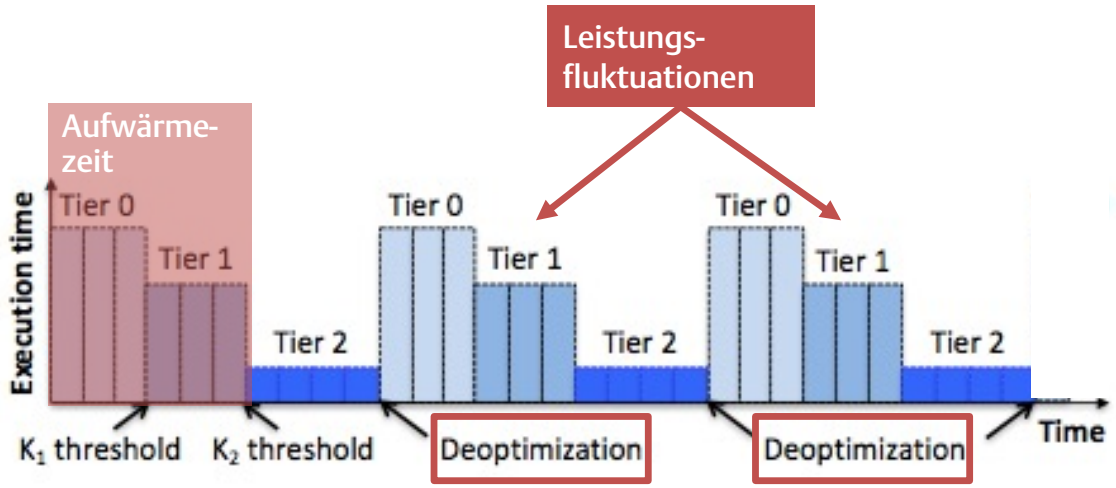
Aus der Perspektive der Ausführung einer Methode

- Methode wird mit dem selben Zustand und Parameter aufgerufen

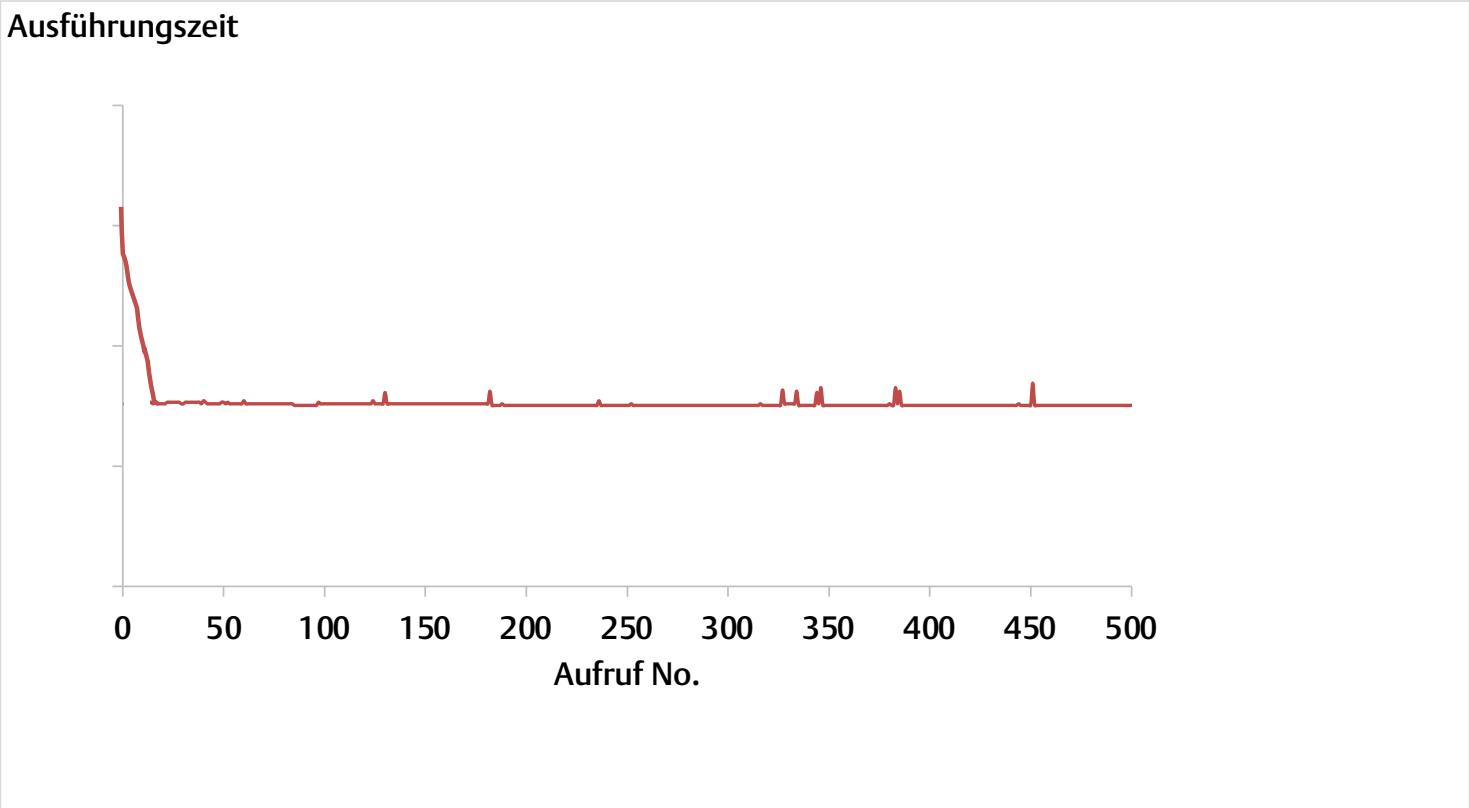
Erwartete Leistung



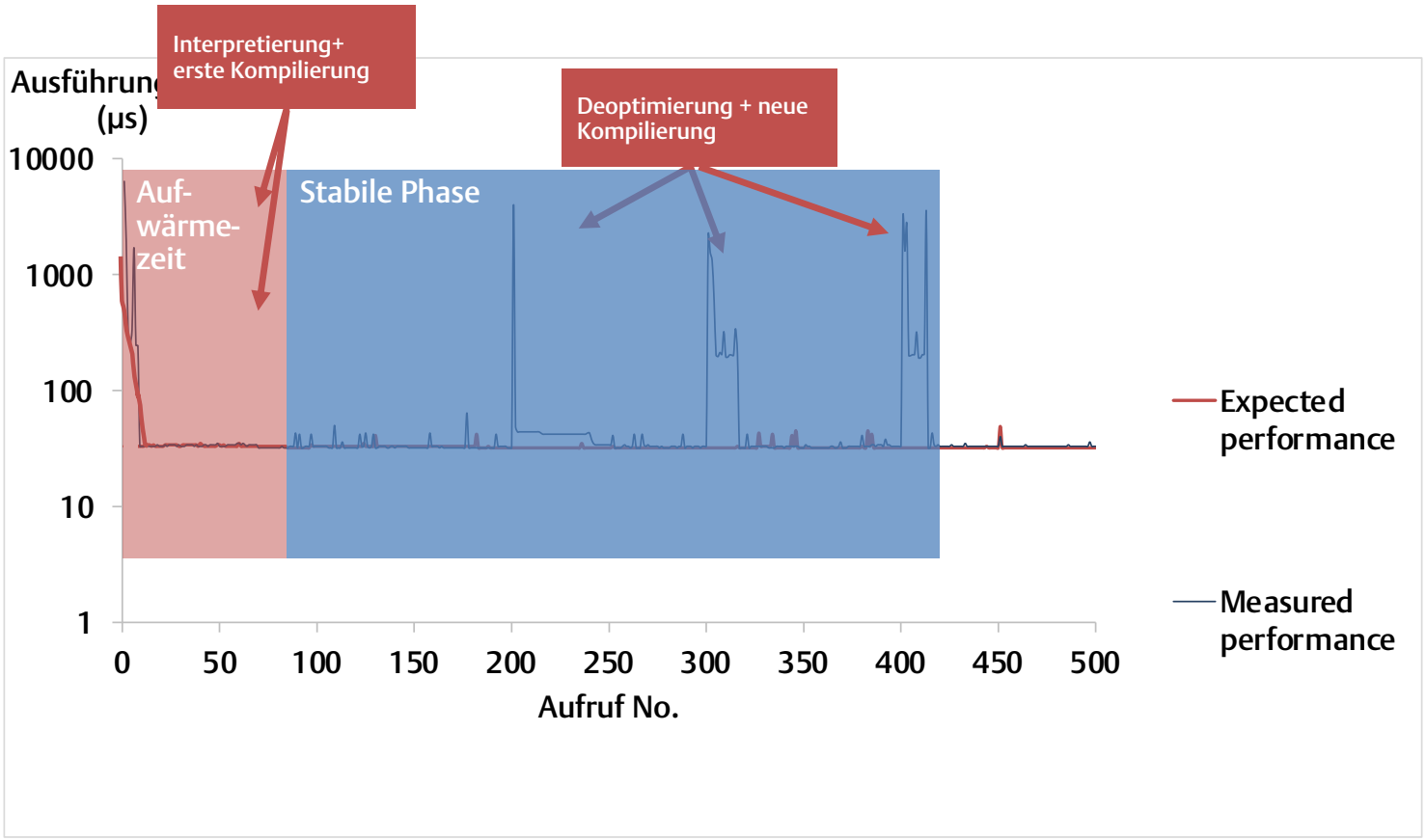
Eigentliche Leistung mit Tiered Compilation



Erwartete Leistung



Gemessene Leistung



Deoptimierungen

Compileroptimierungen sind basiert auf optimistische Annahmen

Annahmen basiert auf Profilinformationen

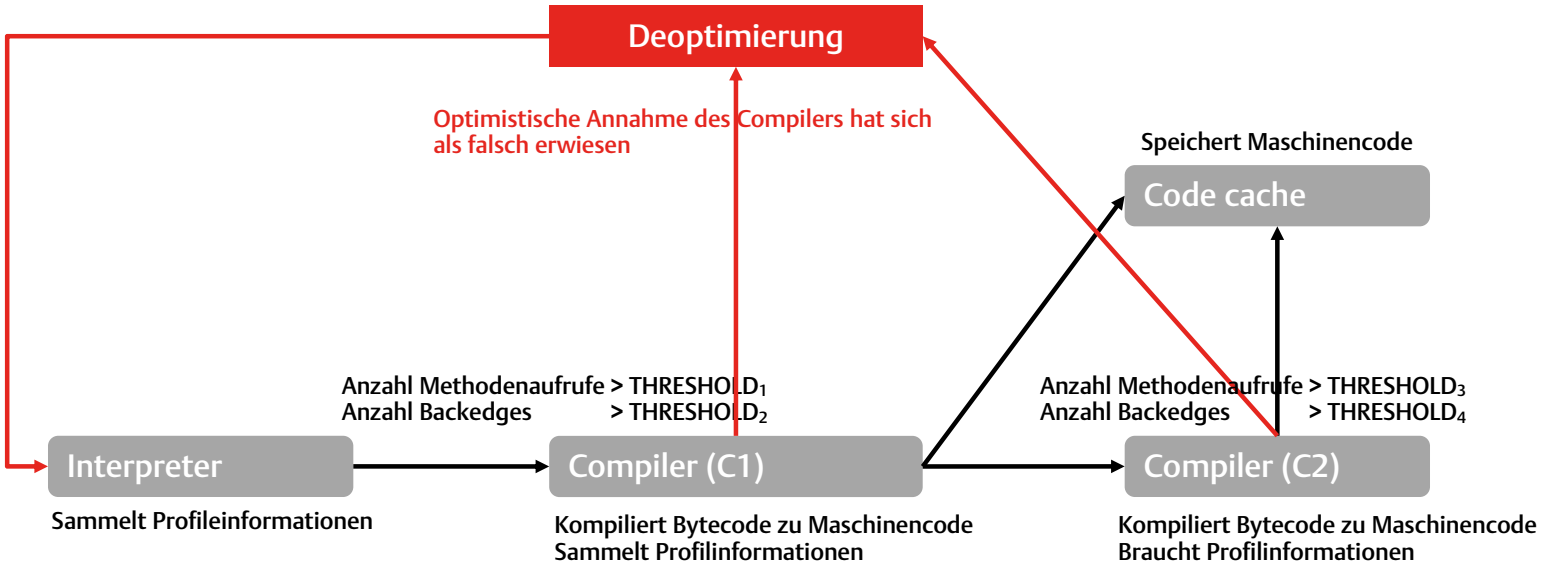
- Anzahl Methodenaufrufe und „Backedges“
- Ausgeführte Pfade in der Methode
- Typen bei Methodenaufrufe
- Typen der Methodenparameter
- Klassenhierarchie
- Und noch mehr anderes

Grundlegendes Prinzip: Vergangenheit = Zukunft

Falls optimistische Annahme des Compilers nicht mehr gilt: Deoptimierung

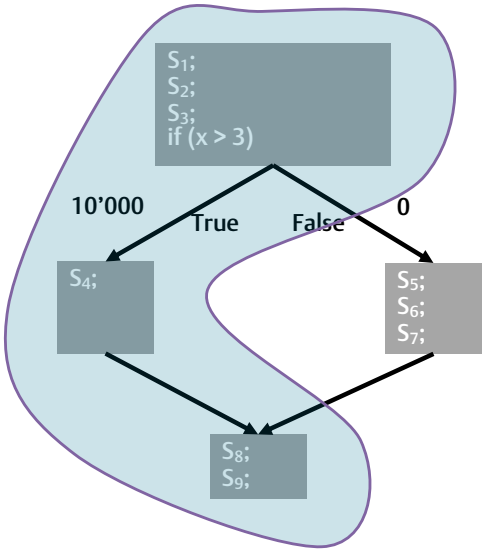
- Kompilierter Code weggeworfen
- Methode wird neu profiliert und erneut kompiliert (mit weniger optimistischen Annahmen)

Das Leben einer Methode: Komplette Geschichte



Beispieloptimierung: “Hot path”-Kompilierung

Kontrollflussgraph



Generierter Code

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?

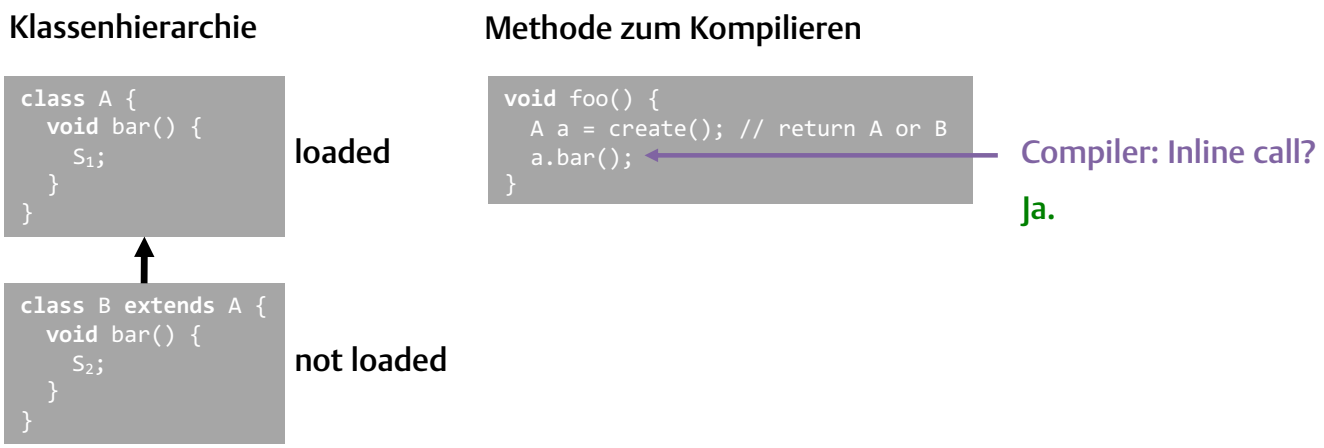
Producer

```
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;
}
```

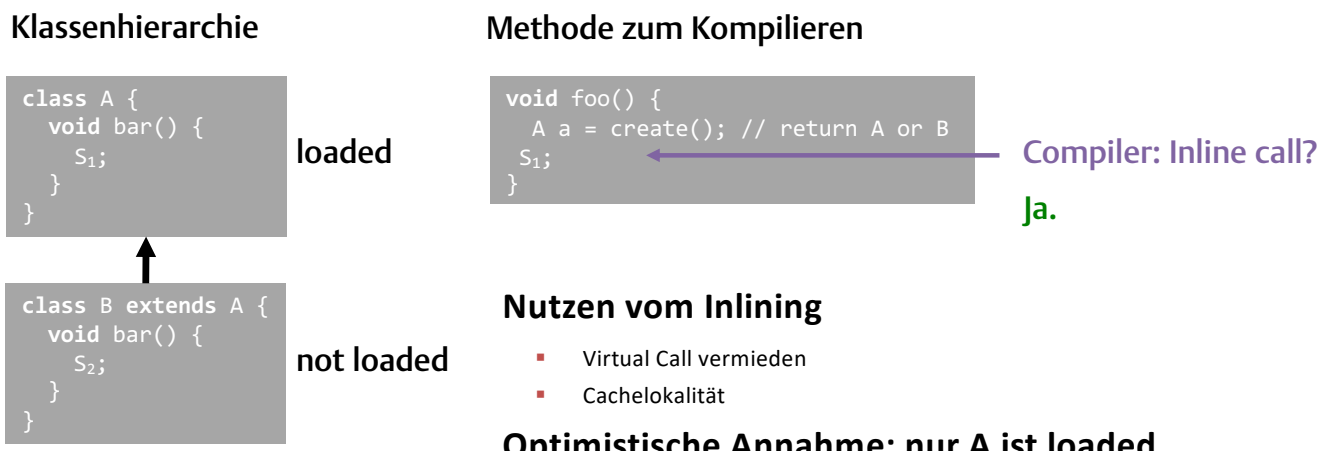
Abgedeckte Pfade

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

Beispieloptimierung 2: Virtual Call Inlining



Beispieloptimierung 2: Virtual Call Inlining



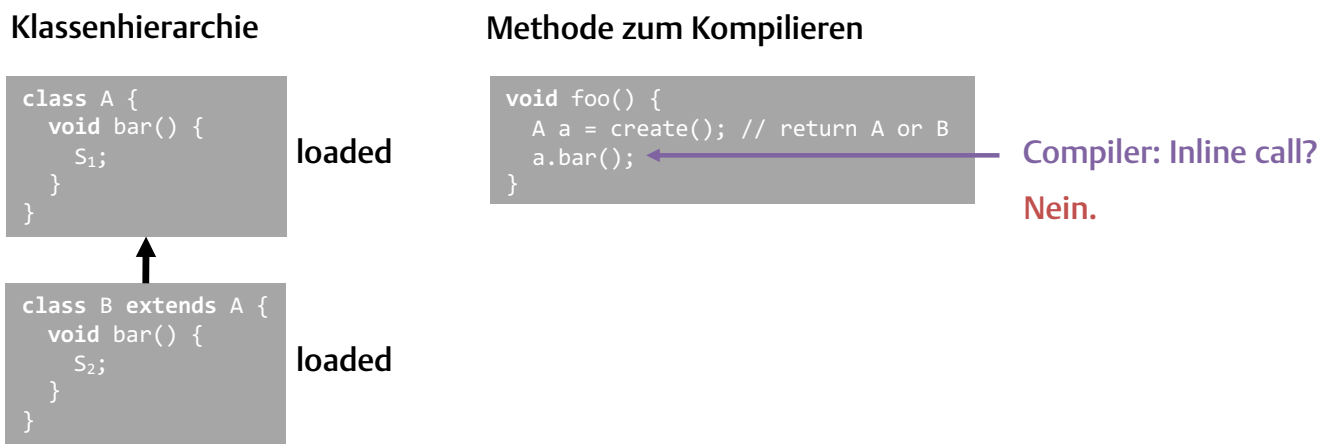
Nutzen vom Inlining

- Virtual Call vermieden
- Cachelokalität

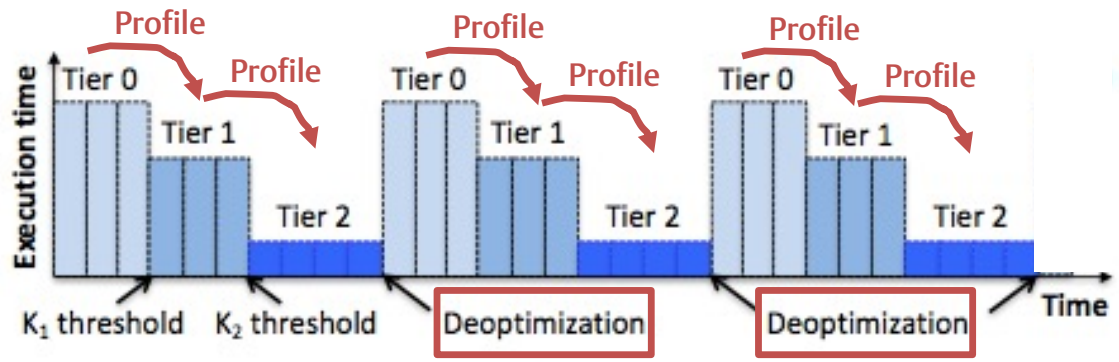
Optimistische Annahme: nur A ist loaded

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

Beispieloptimierung 2: Virtual Call Inlining



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?

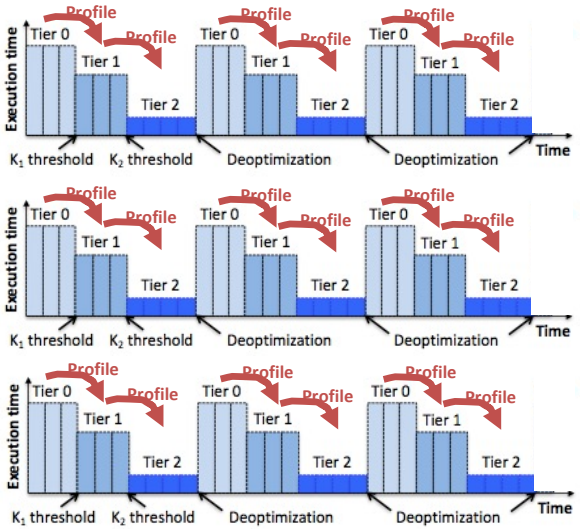
3. Training der VM vor dem produktiven Einsatz

   Kritische Methoden werden mit entsprechenden Parameter und Zustand aufgerufen

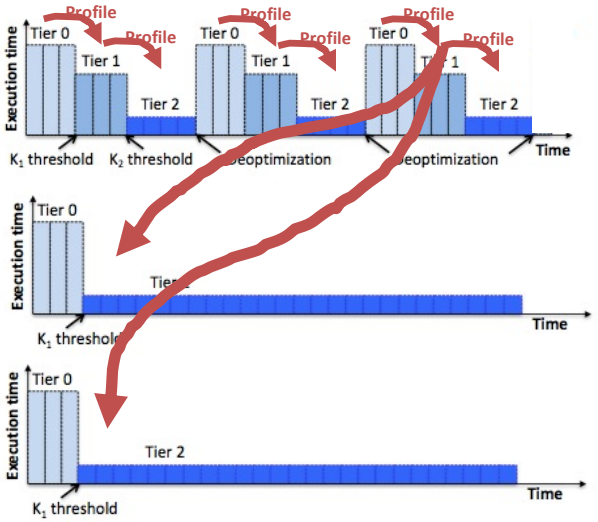
   Kompliziert: Es geht nicht nur um Programmpfade, sondern auch um Typen, Inlining, etc.

Bemerkung: Profile sind weggeworfen

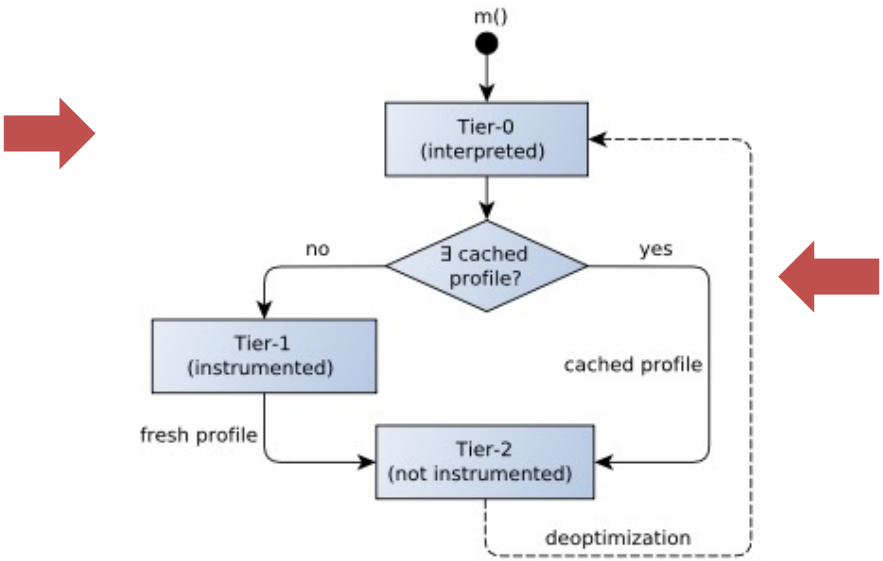
Baseline HotSpot



Profile Caching



Zustandsübergänge



Implementierung

Tiered Compilation erweitert mit neuen Zuständen

Speichern/Laden von Profilinformationen

- Existierendes Format
- In-memory Repräsentierung

Open source

- Patch zum JDK 9 build 29

Evaluierung

8-core Xeon E5520, Ubuntu 14.04

Zwei Konfigurationen

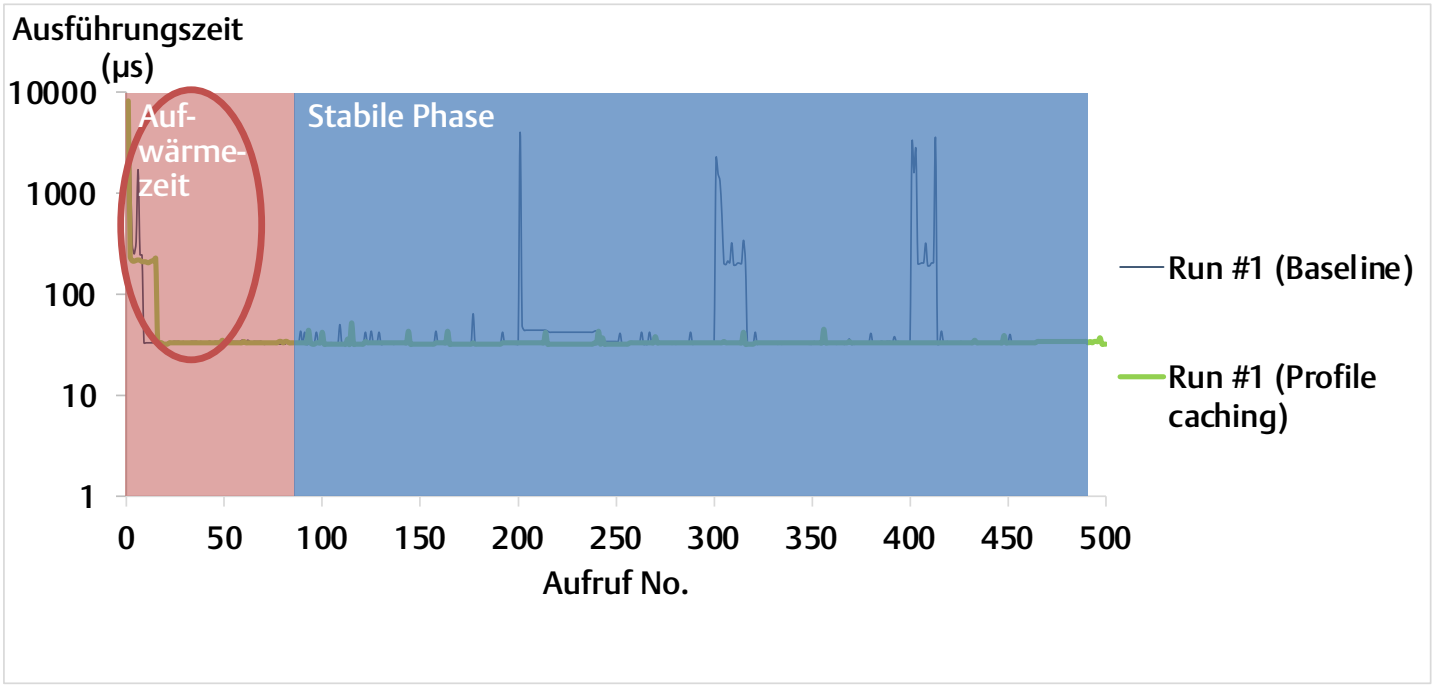
- *Baseline* = out-of-the-box VM
- *Profile caching*
  - Profile generiert mit Programm X und Input Y
  - Profile wird wiederverwendet mit (X, Y)

Producer-Consumer

SPECjvm2008

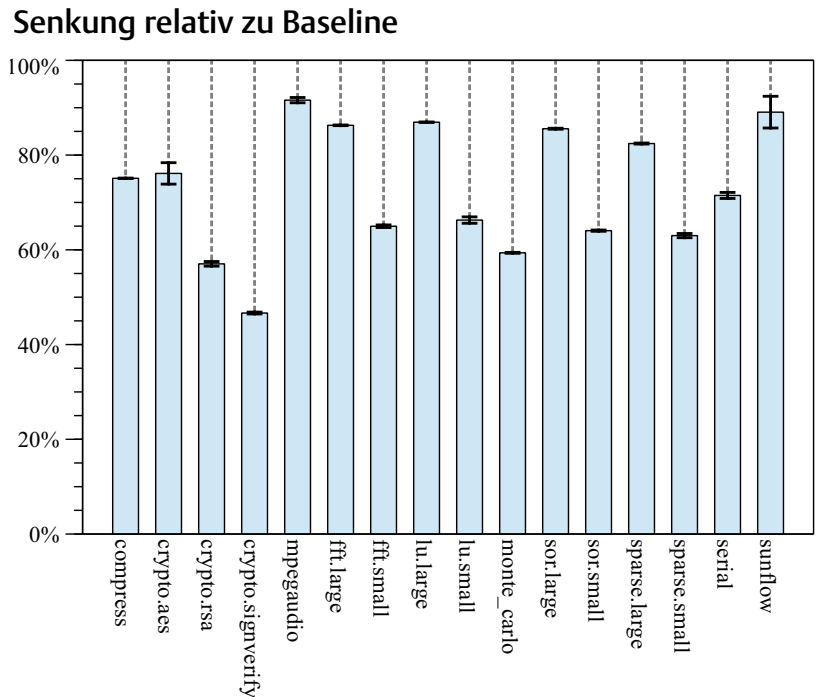
- 16/21 Programme

Producer-Consumer

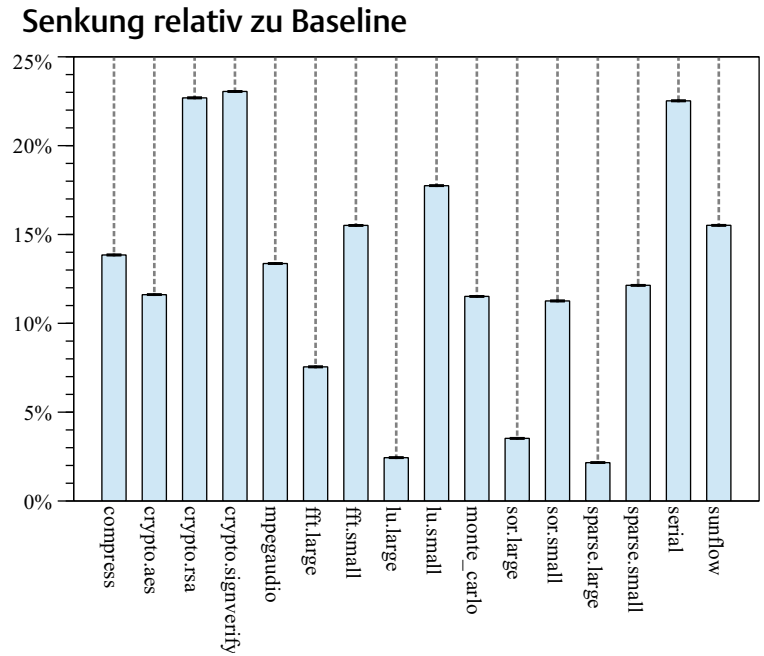


ergon

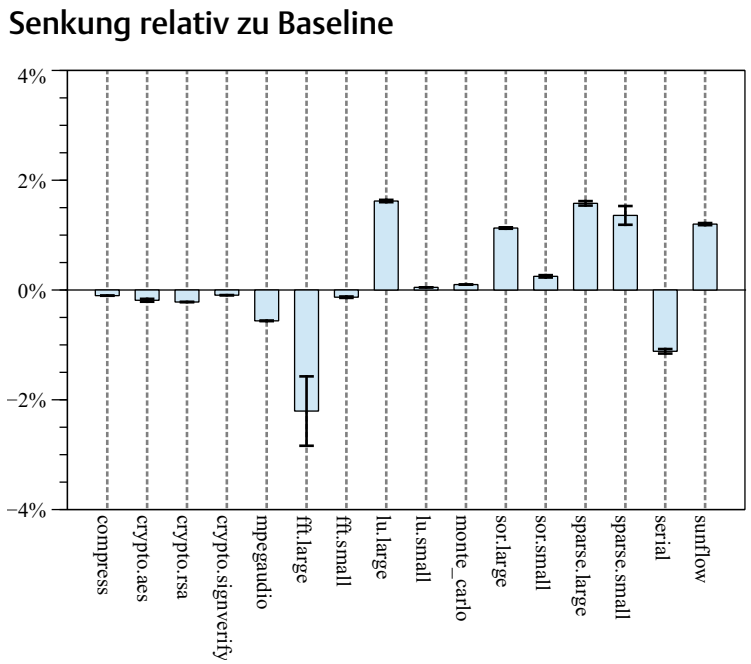
Deoptimierungen



Anzahl Kompilierungen



End-to-end Leistung



Diskussion

Deoptimierungen immer noch verfügbar

- Falls gecachter Profil mit dem Verhalten zur Laufzeit nicht übereinstimmt

Profile Caching (Zusammenfassung)

Reduziert Leistungsfluktuationen

- Weniger Deoptimierungen
- Weniger Kompilierungen

End-to-end Laufzeit ist nicht deutlich beeinträchtigt

(Mit)wirkende

Prof. Thomas Gross (ETH Zürich)

Tobias Hartmann (Oracle)

Dr. Albert Noll (Oracle/Swisscom)

Marcel Mohler (ETH Zürich/Vontobel)

Zusatzmaterial

Zum Thema JIT-Kompilierung

- Fachartikel über Profile Caching: `w11/manlang_2017.pdf`
- Detaillierte Erklärung von Deoptimierungen und des Producer-Consumer Beispiels

Zum Thema Object Lifecycle (und JIT-Kompilierung)

- Was passiert im `Phantom.java`?

Zusammenfassung

Kompilierung in der VM

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