Optimierung der Code-Generierung virtualisierender Ausführungsumgebungen zur Erzielung deterministischer Ausführungszeiten

Martin Däumler Matthias Werner

Lehrstuhl Betriebssysteme
Fakultät für Informatik
Technische Universität Chemnitz

22. November 2012

Inhalt

Motivation

Echtzeit 2010:

- Braun, S.; Obermeier, M.; Schmidt-Colinet, J.; Eben, K.; Kissel, M.: **Notwendigkeit von Metriken für neue Programmiermethoden automatisierungstechnischer Anlagen**. S. 11 – 20

- Schepeljanski, A.; Däumler, M., Werner, M.: **Entwicklung einer echtzeitfähigen CLI-Laufzeitumgebung für den Einsatz in der Automatisierungstechnik**. S. 21 – 30

Motivation

Merkmale moderner Allzweck-Programmiersprachen wie Java oder C#:

- Objektorientierte Programmierung
- Typensicherheit
- Ereignisbasierte Programmierung
- Ausnahmebehandlung
- Dynamisches Laden von Klassen
- **Verteilung über plattformunabhängigen Zwischencode**

Plattformunabhängigkeit durch virtualisierende Ausführungsumgebung

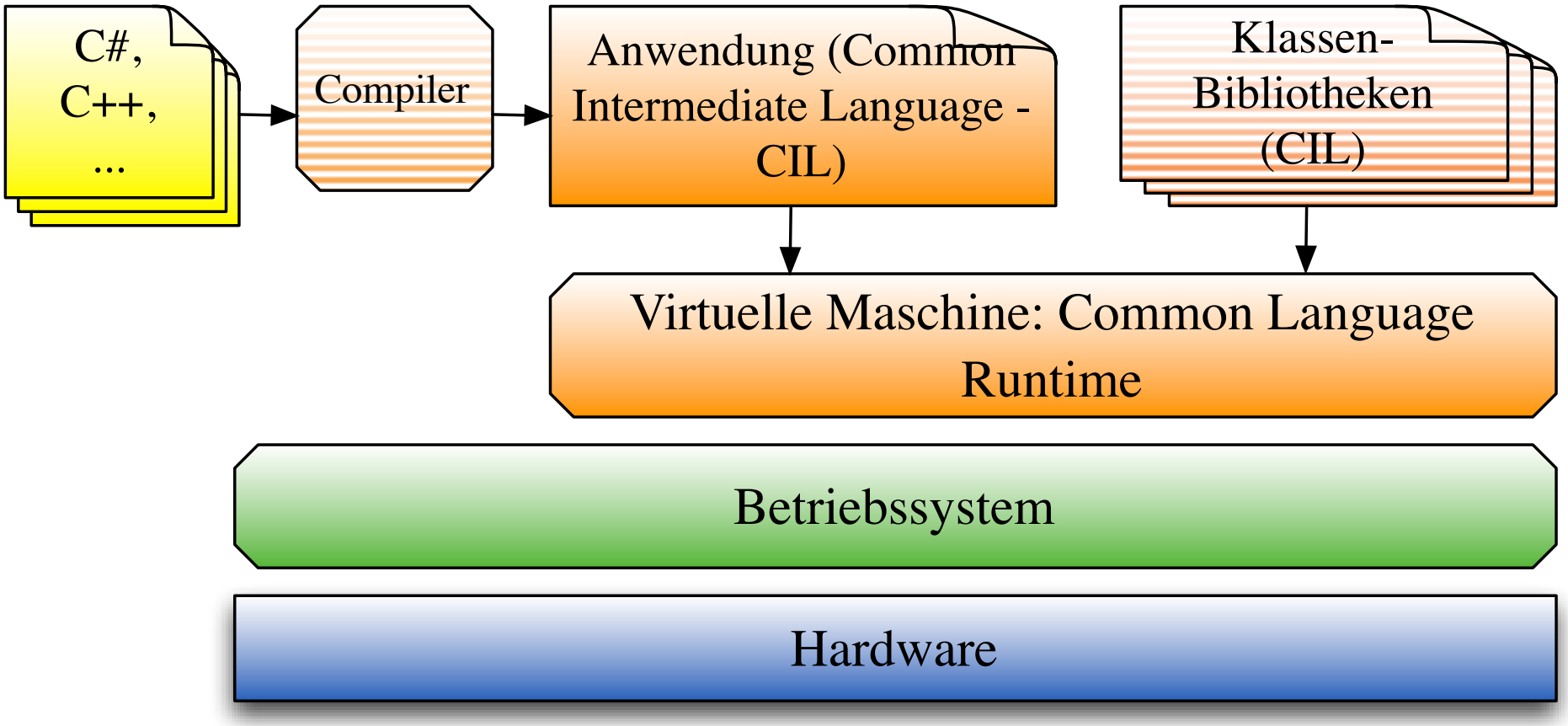

Abbildung 1 : Übersicht Arbeitsweise einer virtualisierenden Ausführungsumgebung

Mikro-Benchmark

Pseudo-Code:

```
main () {
    // Initialisierungen
    method0();
    ...
    method999();          } Messung 1

    for (i = 0; i < 3; i++) {
        method0();
        ...
        method999();      } Messungen 2, 3, 4
    }
}
```

Testsystem:

- CPU: Intel Atom Z510@1.1 GHz
- 1.0 GiB Hauptspeicher
- Betriebssystem: Linux-Kernel mit RT-Preempt-Patches, Version 2.6.33.5-rt23-v1

Messergebnisse mit „Just-in-Time“ (JIT)-Compiler

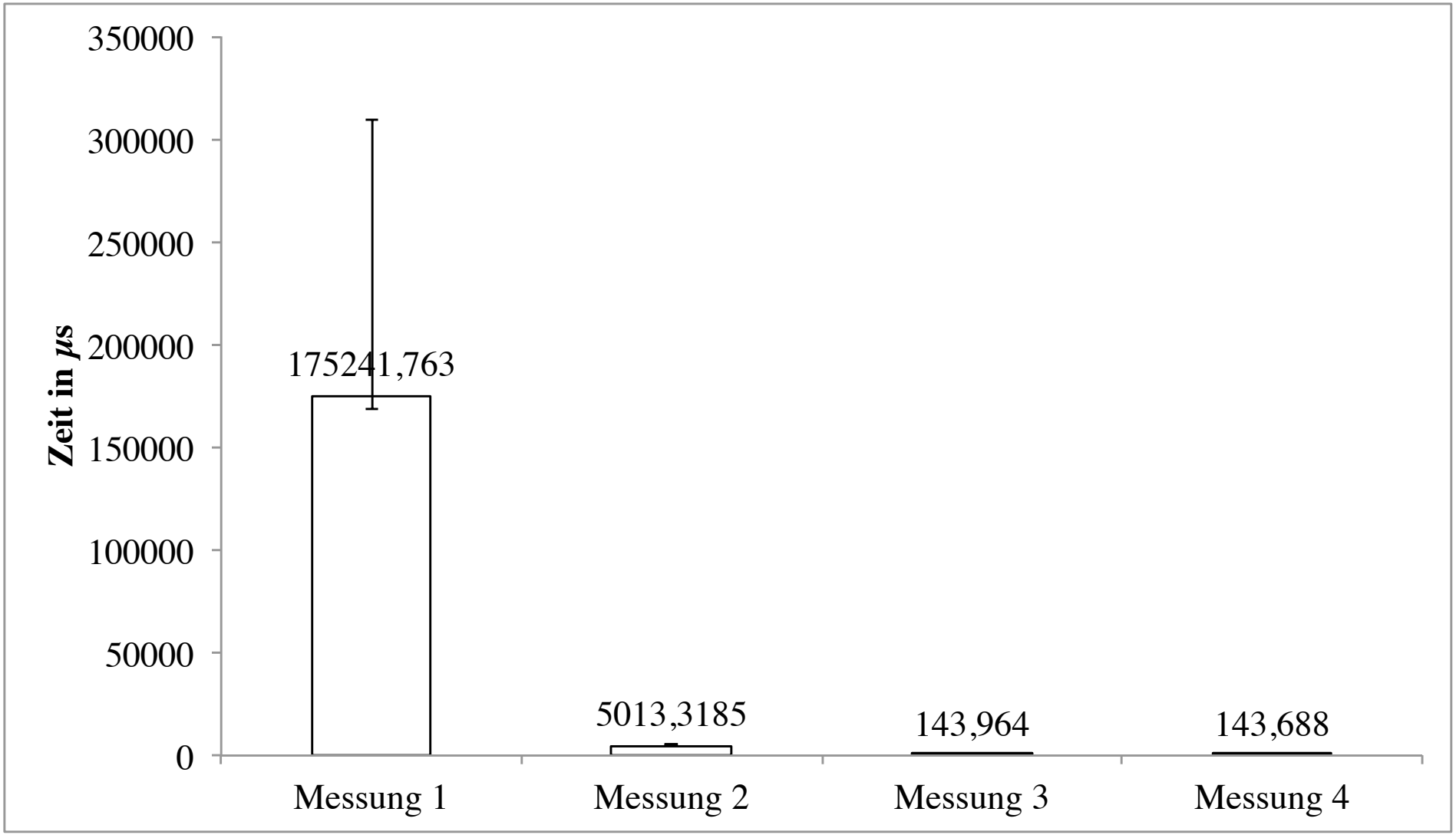

Abbildung 2 : Ausführungszeiten von 1000 Instanz-Methoden, Mono 2.6.1, JIT-Modus

Messergebnisse mit „Ahead-of-Time“ (AOT)-Compiler

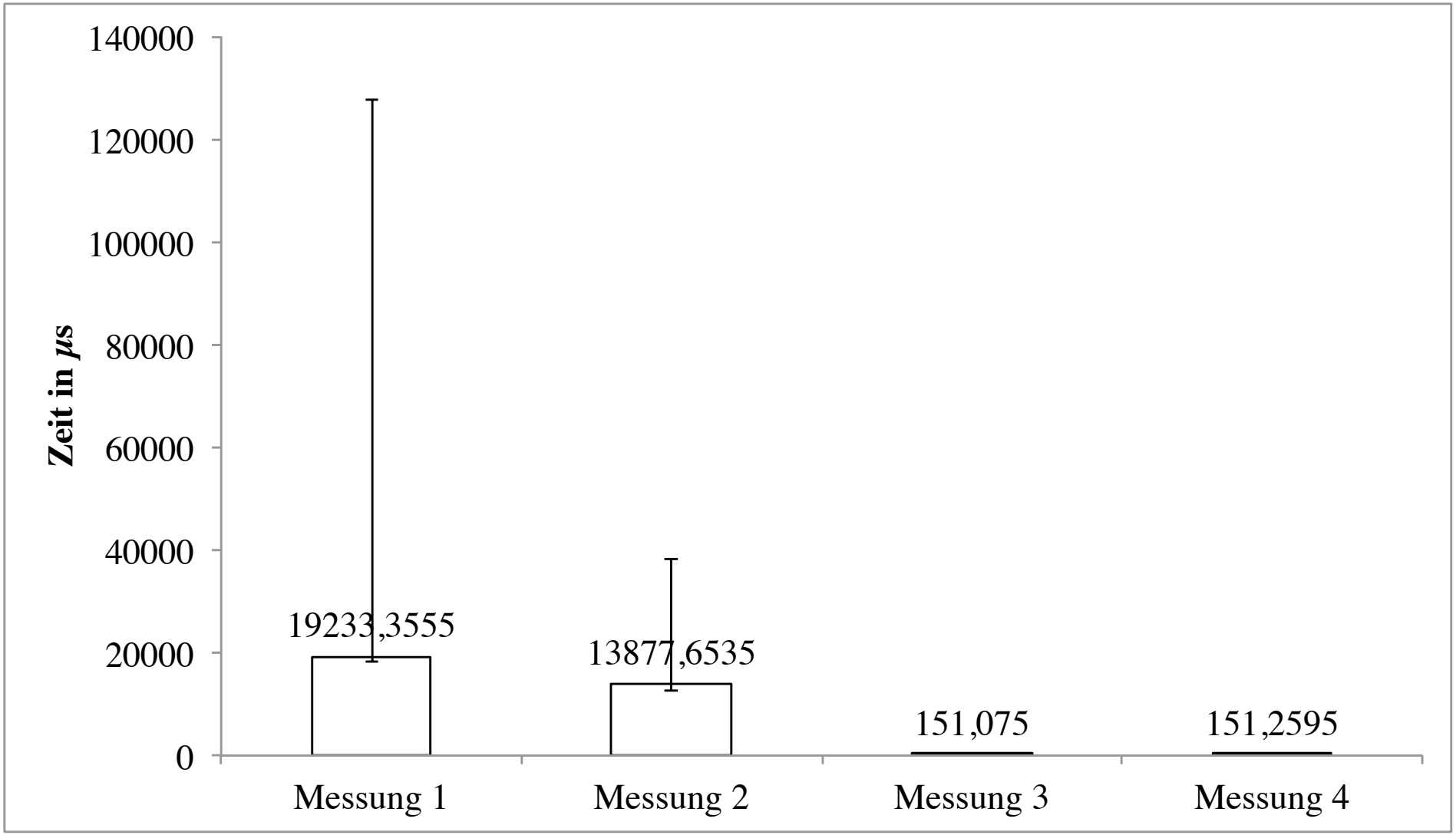

Abbildung 3 : Ausführungszeiten von 1000 Instanz-Methoden, Mono 2.6.1, AOT-Modus

Vor-Compilierung

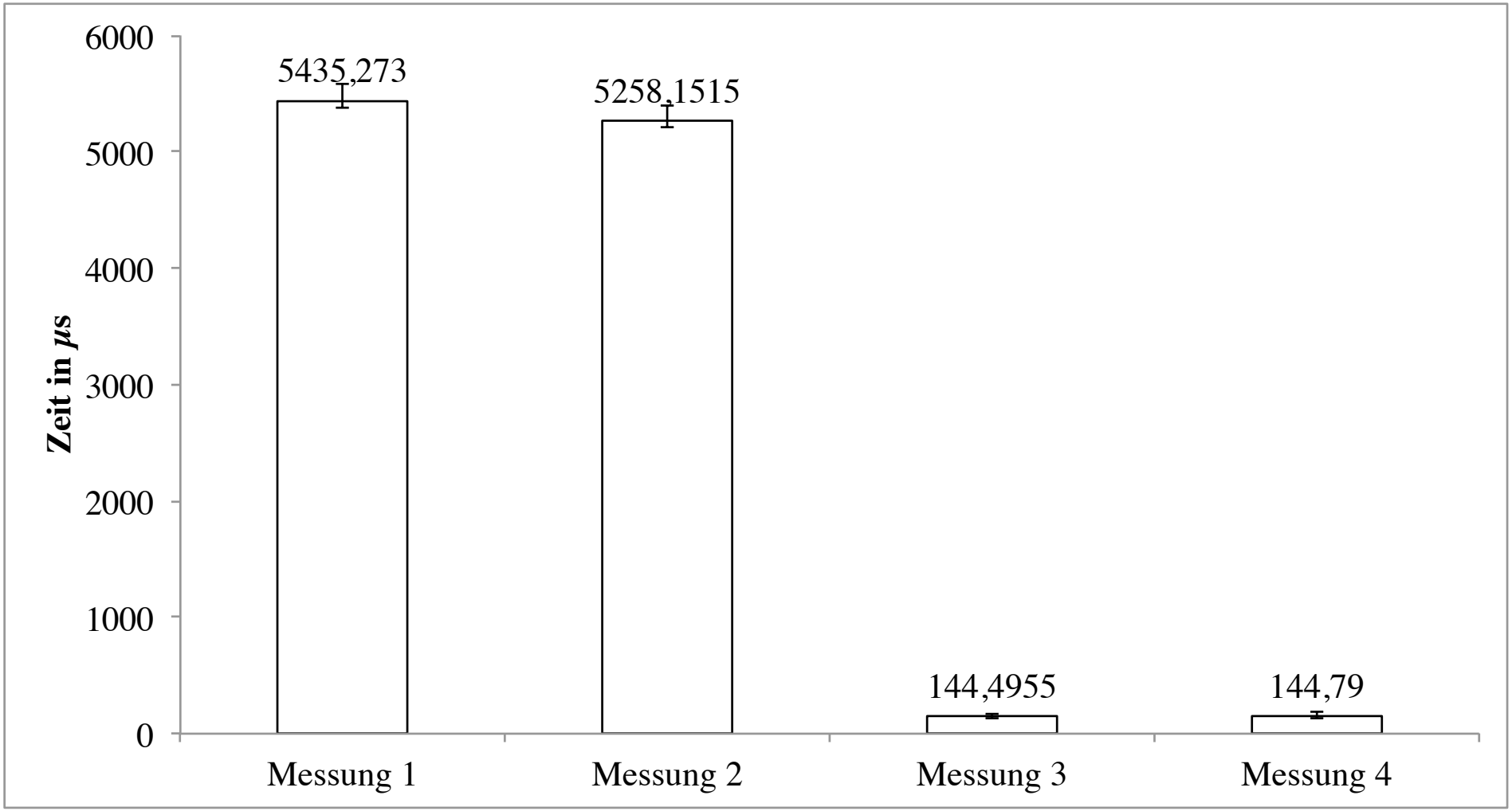

Abbildung 4 : Ausführungszeiten von 1000 Instanz-Methoden, Mono 2.6.1, JIT-basierte Vor-Compilierung

Indirekte Referenzen

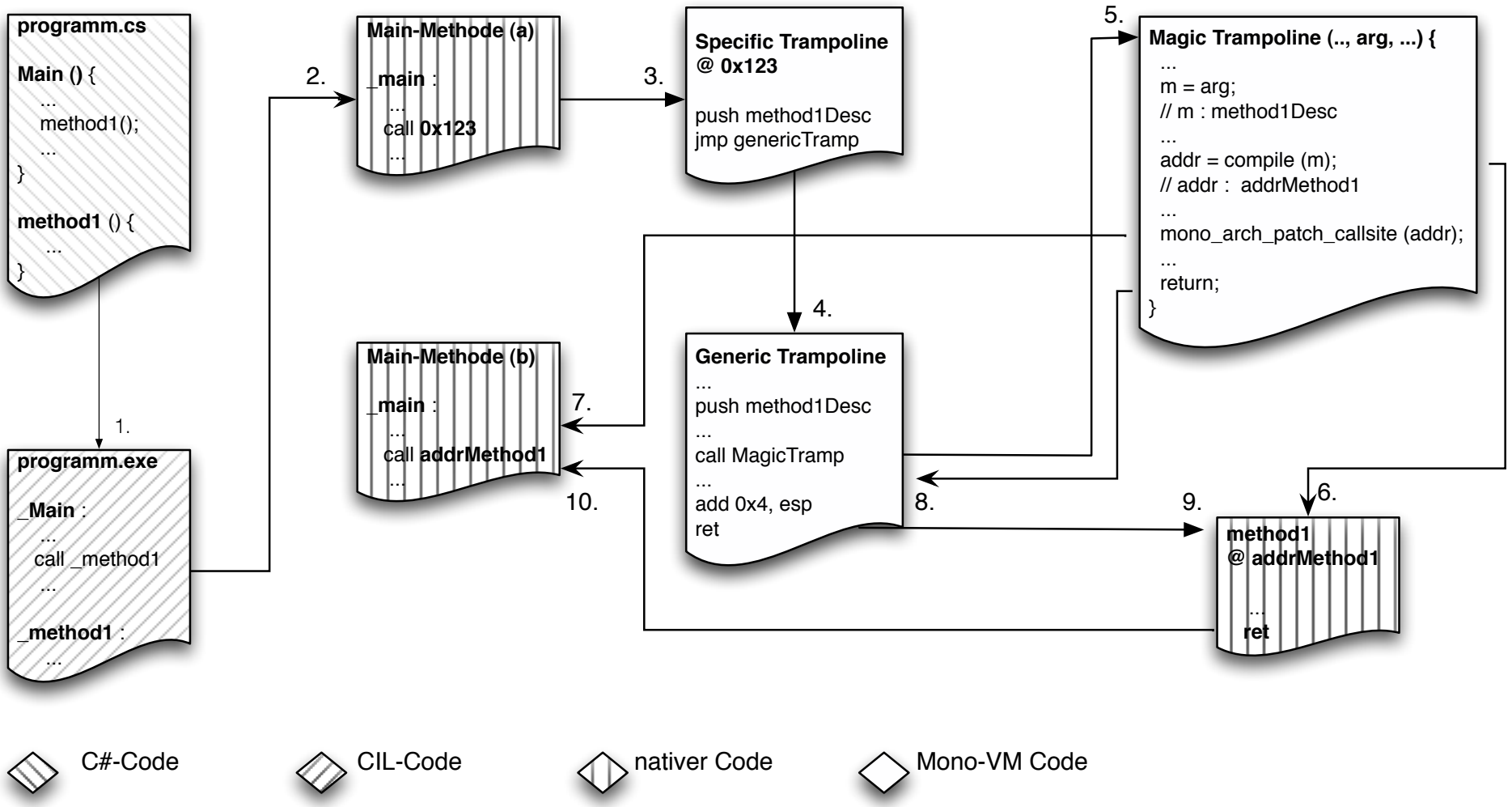

Abbildung 5 : Erste Ausführung eines Methodenaufrufs in JIT-compiliertem Code, Mono-VM

Indirekte Referenzen

1. Compilerung des Hochsprachen-Codes in plattformunabhängigen Zwischencode
2. JIT-Compilierung der Einsprungmethode bei Programmstart
3. Ausführung eines Methoden-spezifischen Trampolines
4. Ausführung eines Trampolines, das generisch Methodenaufrufe behandelt
5. Ausführung von VM-internen C-Funktionen (Trampolin höchster Ebene)
6. JIT-Compilierung der aufgerufenen Methode
7. Manipulation des Methodenaufrufs im aufrufenden Code der Einsprungmethode, so dass er direkt auf die aufgerufene Methode zeigt
8. Abstieg in der Trampoline-Ebene
9. Ausführung der aufgerufenen Funktion
10. Rückkehr in den aufrufenden Code

Pre-Patch

- Aufzeichnung jeder emittierten indirekten Referenz („Patch“) während der Vor-Compilierung
- Abarbeitung der Trampoline vor Programmausführung:
  - Modifiziertes Trampoline: Rückkehr zu Aufrufenden bei Schritt 9 in Abbildung 5
  - Umlenken eines Patches in modifiziertes Trampoline passenden Typs
  - Manuelles Auslösen der indirekten Referenz (vgl. Schritt 3 in Abbildung 5)
  - Automatische Abarbeitung des Trampoline-Mechanismus (Schritt 4 bis 8 in Abbildung 5)
- $\Longrightarrow$ Nutzung vorhandener Funktionalität zur Transformation des nativen Codes, so dass er keine indirekten Referenzen enthält

Messergebnisse mit Pre-Patch

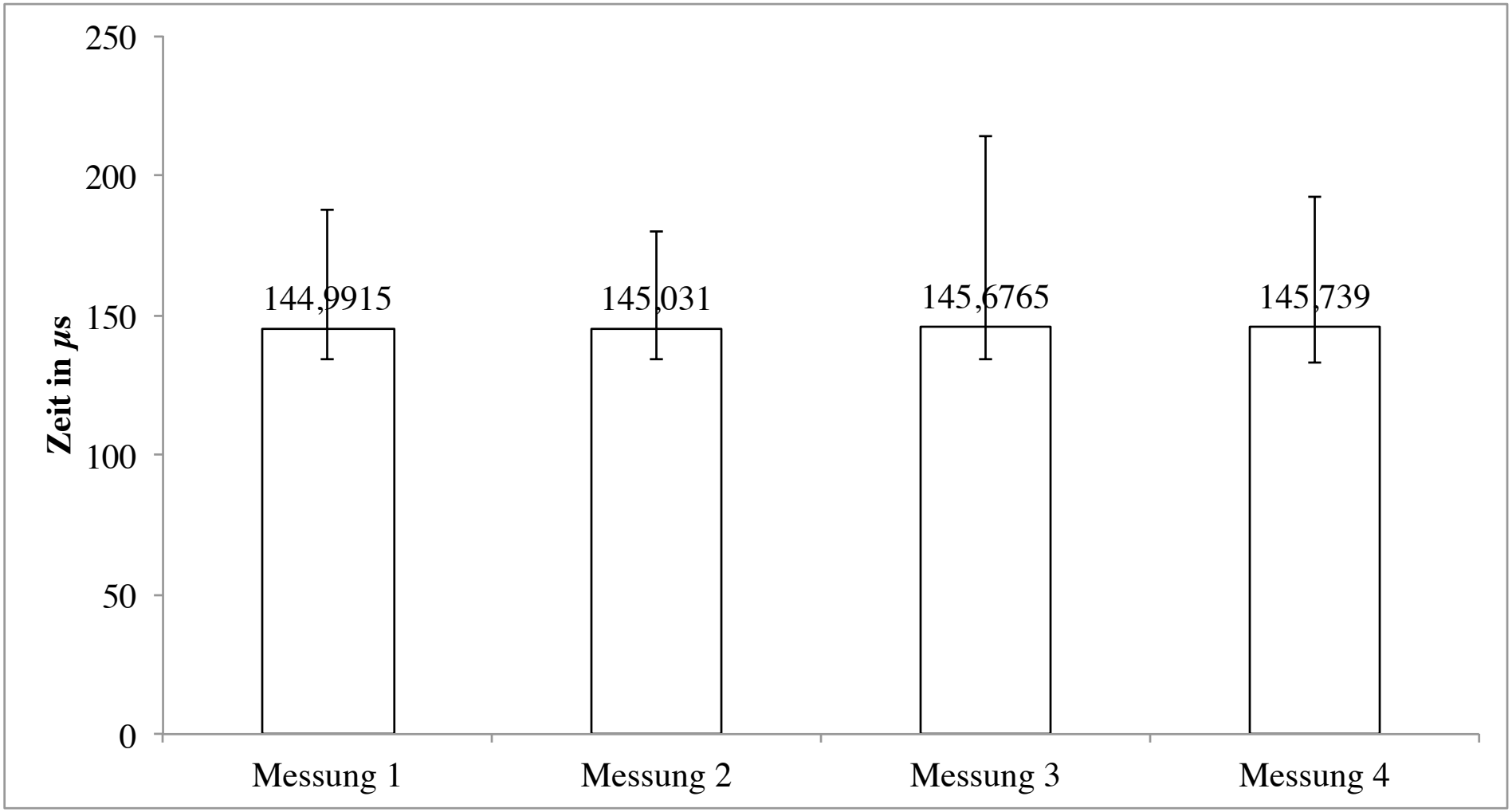

Abbildung 6 : Ausführungszeiten von 1000 Instanz-Methoden, Mono 2.6.1, Pre-Patch

Vergleich mit anderen Lösungen: Übersicht

- **Mono 2.10.8.1:**
  - CLI-VM
  - „Full-AOT“: AOT-Compilierung interner Hilfsfunktionen
  - keine Echtzeit-Lösung
- **IBM WebSphere Real Time V2 for RT Linux build 2.4:**
  - Java-VM
  - AOT-Compiler: lädt nativen Code zur Laufzeit
  - „Real-Time Specification for Java“ (RTSJ)-konform
- **JamaicaVM 6.0 Release 3 build 6928:**
  - Java-VM
  - AOT-Compiler: generiert ausführbares Binary
  - RTSJ-konform

Vergleich mit anderen Lösungen: Instanz-Methoden

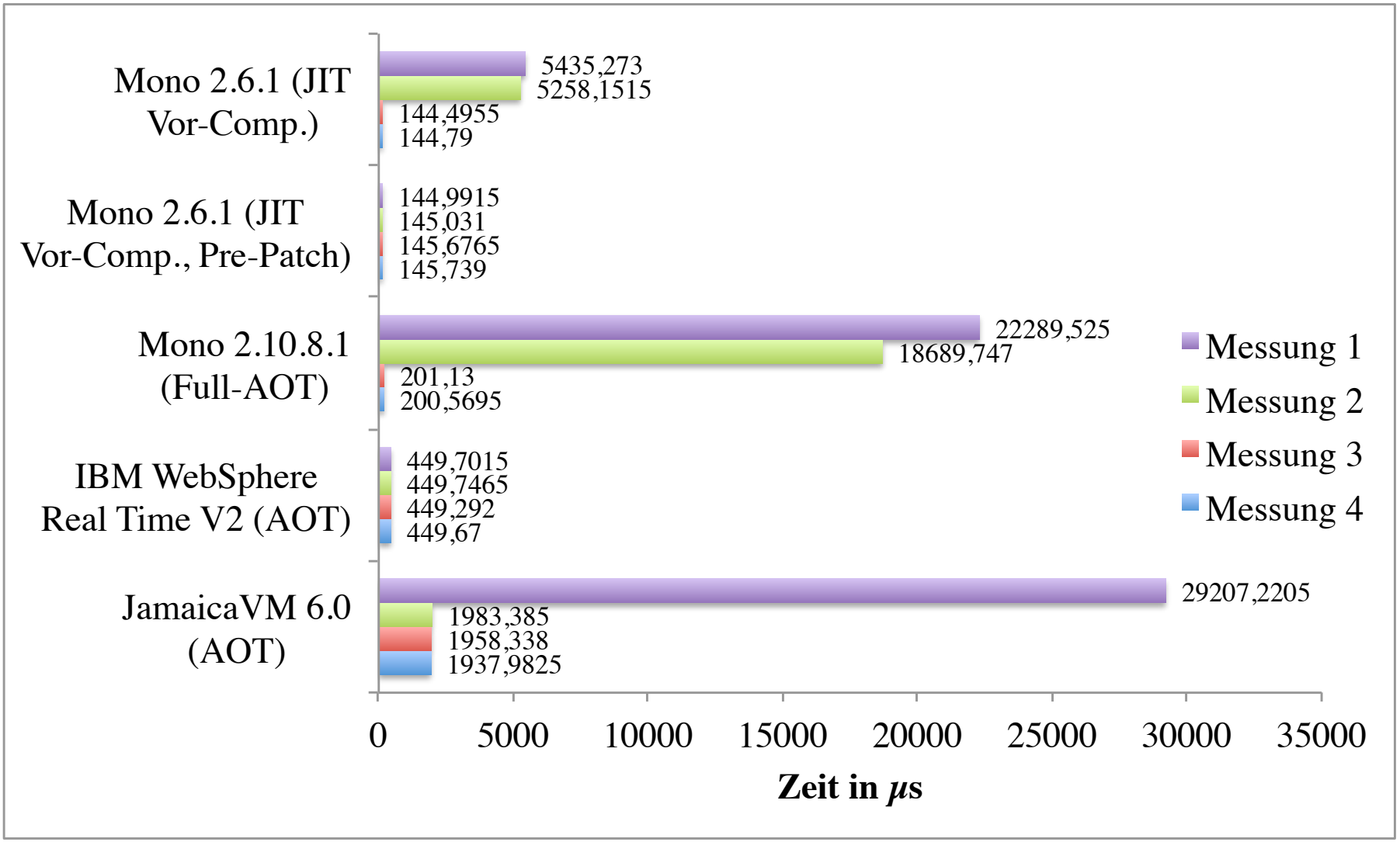

Abbildung 7 : Mittelwert der Ausführungszeiten von 1000 Instanz-Methoden, deterministischer Ausführungsmodus

Vergleich mit anderen Lösungen: Instanz-Methoden

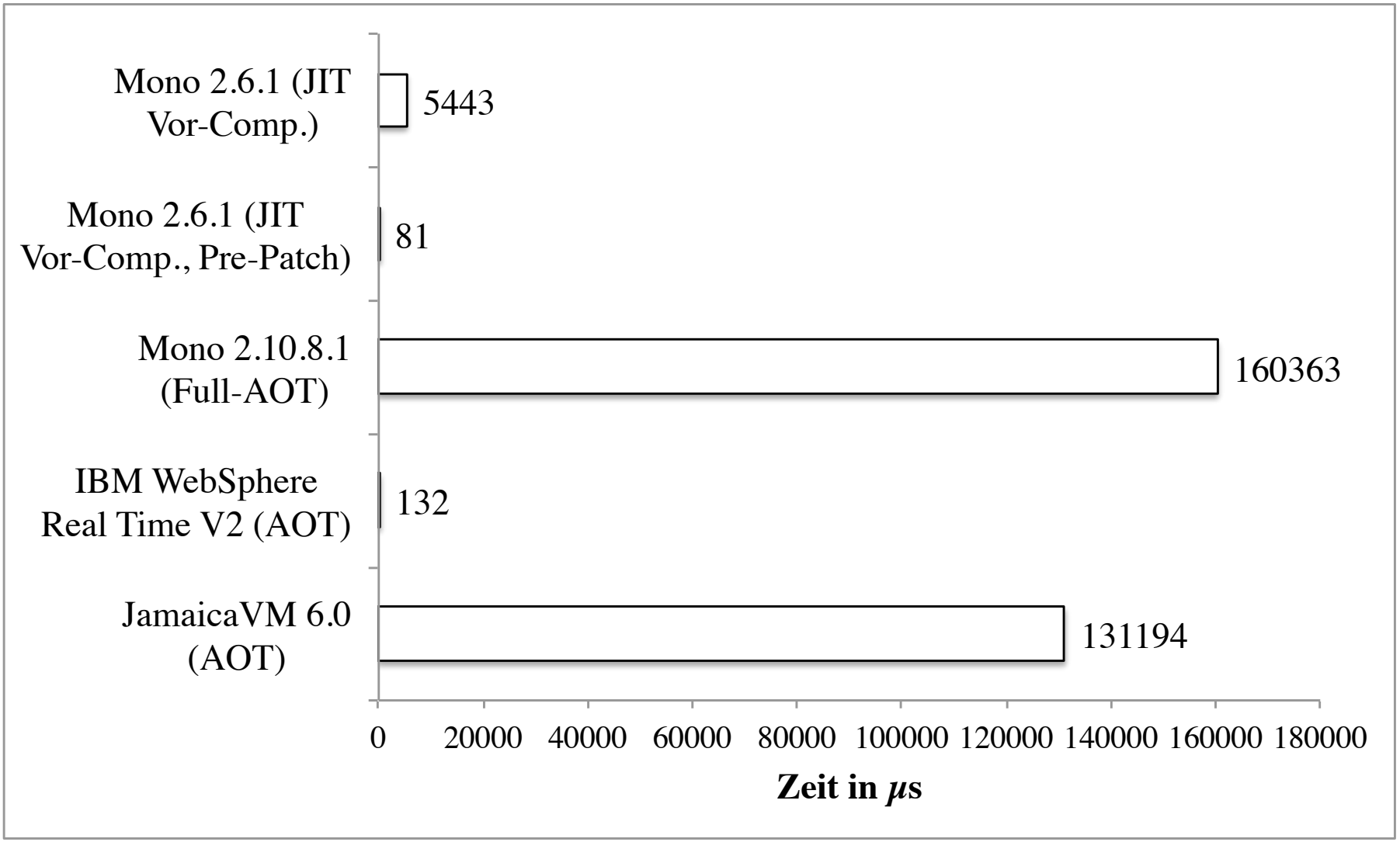

Abbildung 8 : Schwankungen der Ausführungszeiten von 1000 Instanz-Methoden, deterministischer Ausführungsmodus

Vergleich mit anderen Lösungen: Instanz-Methoden

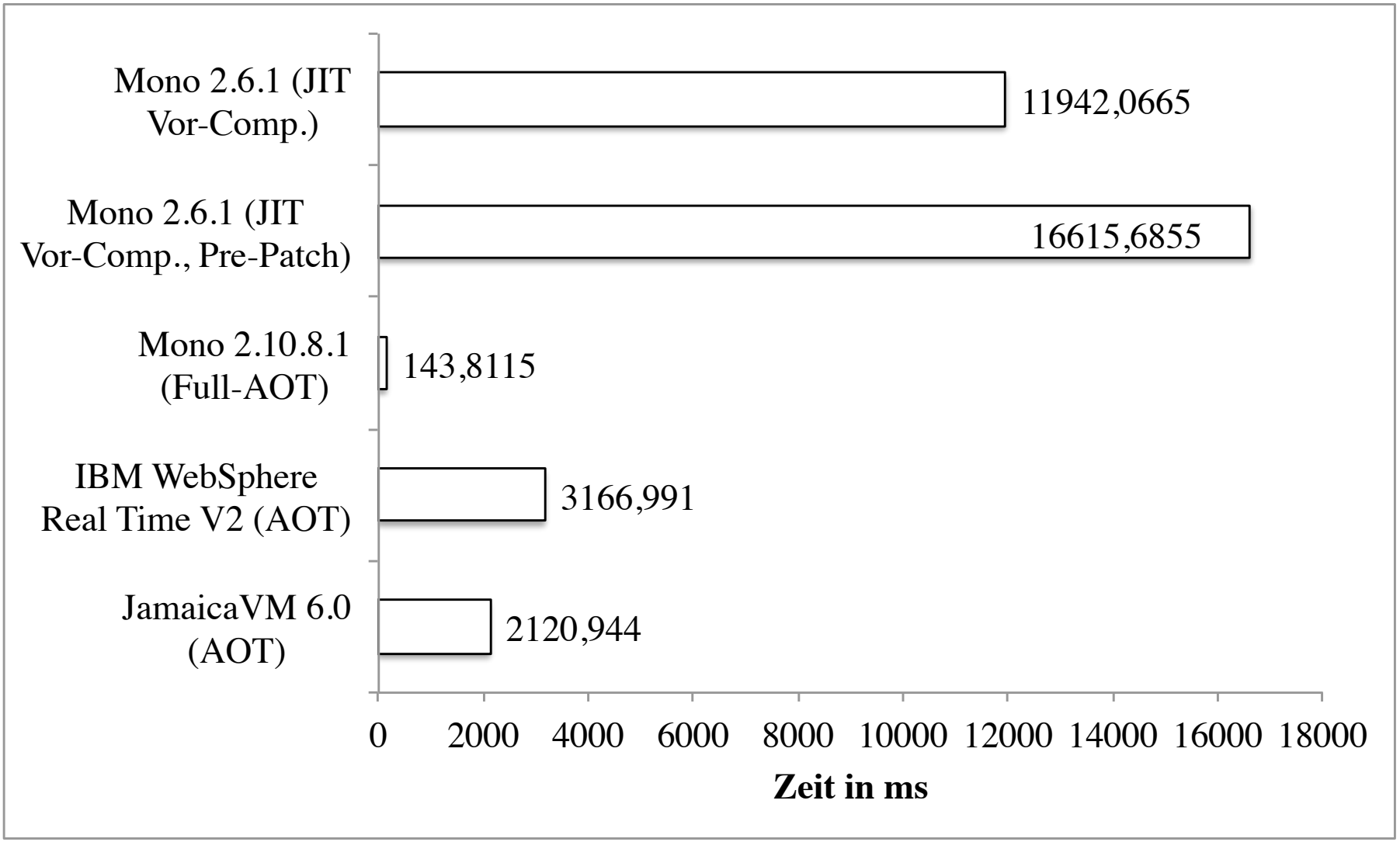

Abbildung 9 : Mittlere Startzeit des Mikro-Benchmarks mit 1000 Instanz-Methoden, deterministischer Ausführungsmodus

Vergleich mit anderen Lösungen: Interface-Methoden

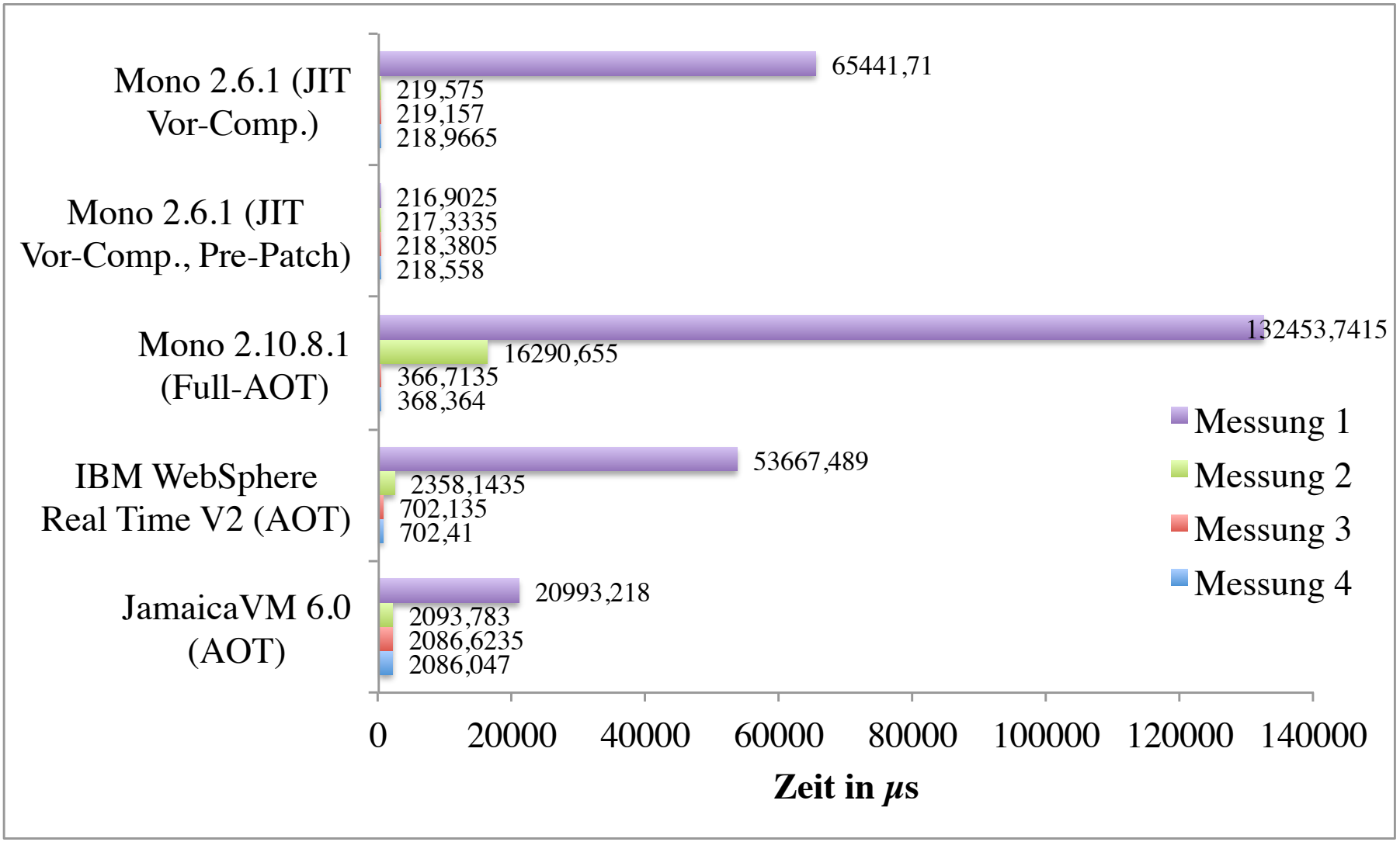

Abbildung 10 : Mittelwert der Ausführungszeiten von 1000 Interface-Methoden, deterministischer Ausführungsmodus

Vergleich mit anderen Lösungen: Interface-Methoden

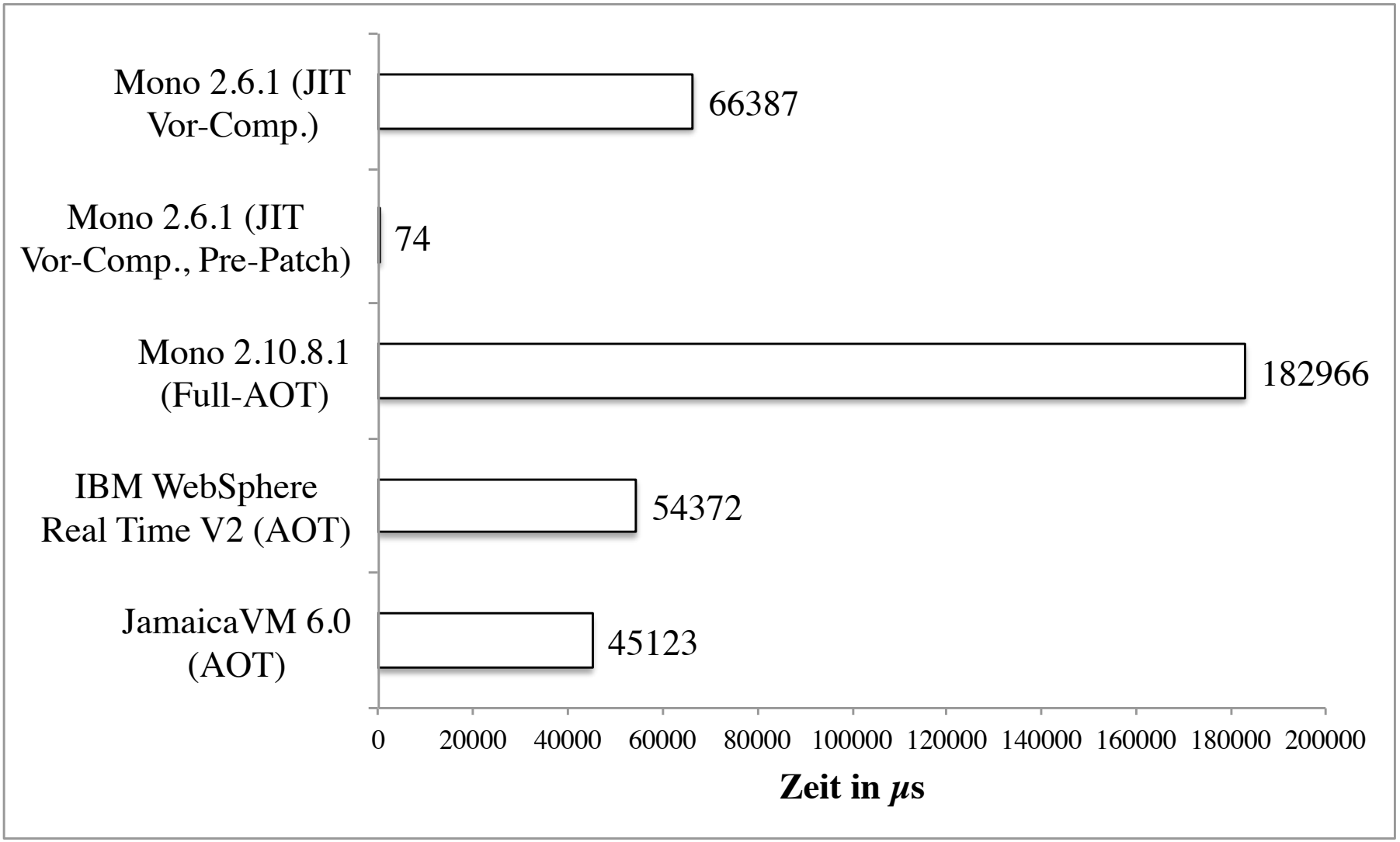

Abbildung 11 : Schwankungen der Ausführungszeiten von 1000 Interface-Methoden, deterministischer Ausführungsmodus

Vergleich mit anderen Lösungen: Interface-Methoden

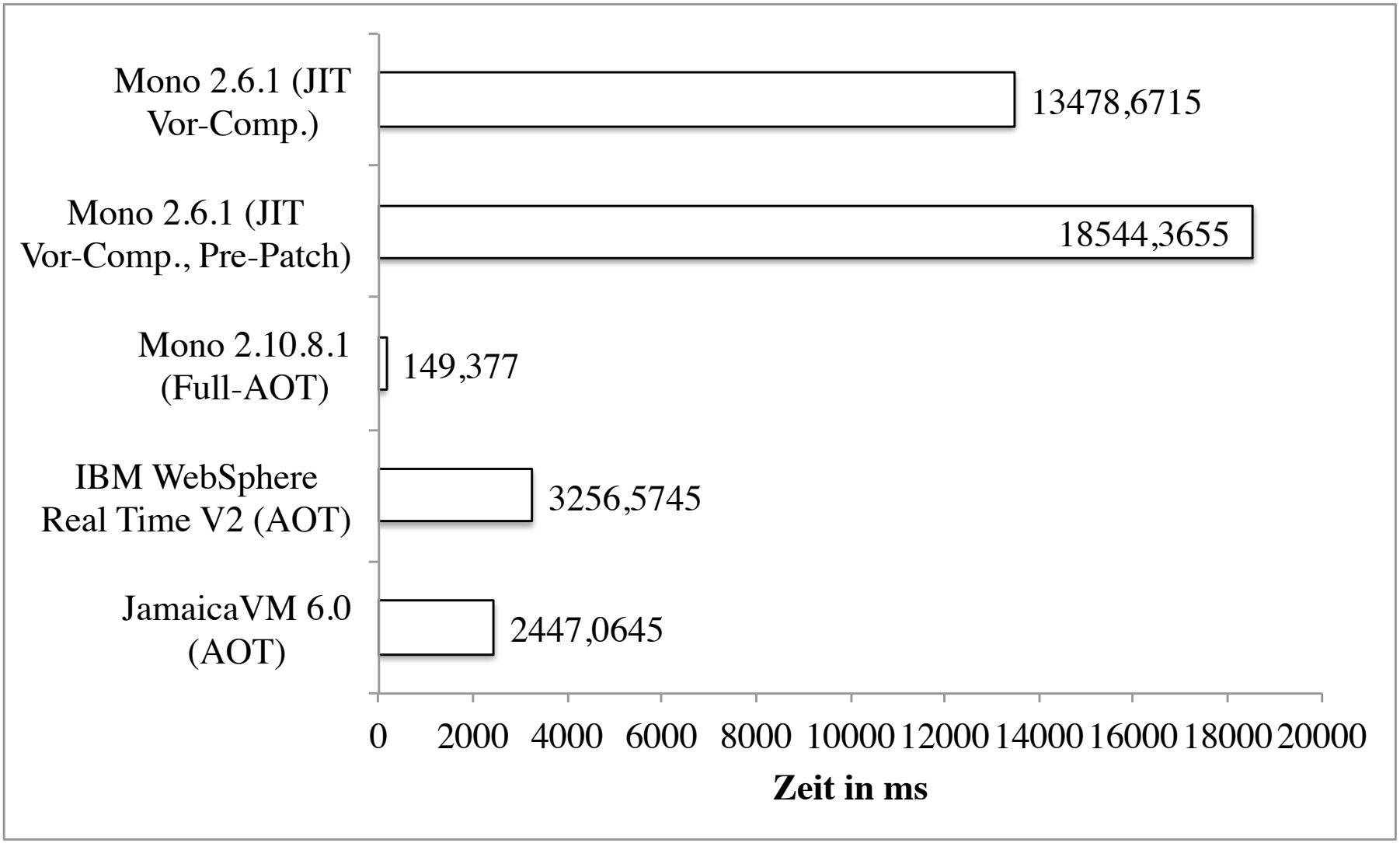

Abbildung 12 : Mittlere Startzeit des Mikro-Benchmarks mit 1000 Interface-Methoden, deterministischer Ausführungsmodus

Zusammenfassung

Es wurde eine Erweiterung der CLI-Implementierung „Mono“ vorgestellt.

- Vor-Compilierung und Pre-Patch erlauben eine die Ausführung von Anwendungscode ohne Eingriff der VM.
- Der Code-Generator und -Lader bleibt unverändert, so dass dynamische Sprachfeatures weiterhin nutzbar sind.
- Die Erweiterungen betreffen nur die VM. Anwendungen müssen nicht modifiziert werden.

Ausblick

- Reduzierung der Startzeit:
  - Eine Verringerung um ca. 20% ist durch Einsatz des AOT-Compilers zur Vor-Compilierung möglich.
  - Weitere Verbesserung durch Einsatz des Full-AOT-Compilers wird untersucht.
- Standard-Konformität: ECMA-335 §8.9.5 verbietet die Ausführung von Typ-Konstruktoren vor erster Referenzierung.

Ende des Vortrags

Vielen Dank für Ihre Aufmerksamkeit.
Haben Sie Fragen?