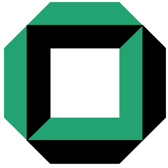
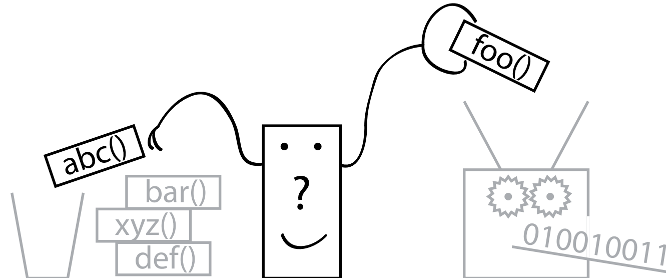


Universität Karlsruhe (TH)
Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation
Lehrstuhl Programmierparadigmen • Prof. Snelting

# Evaluierung heuristischer Steuerungsverfahren für Laufzeitübersetzer auf eingebetteten Systemen

Studienarbeit

Julian Oppermann




4. September 2009

Verantwortlicher Betreuer
Prof. Dr. Gregor Snelting

Betreuer
Dipl.-Inform. Matthias Braun
Dr. Florian Liekweg (aicas GmbH)

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben.

---

Ort, Datum Unterschrift

# Inhaltsverzeichnis

**1 Einführung** 3

**2 Grundlagen** 4
2.1 Grundlagen . . . 4
2.1.1 Java . . . 4
2.1.2 Eingebettete Systeme . . . 4
2.2 Laufzeitübersetzung . . . 4
2.2.1 Laufzeitübersetzung für Java . . . 4
2.2.2 Herausforderungen . . . 5
2.2.3 Heuristiken . . . 6
2.2.4 Laufzeitprofile . . . 7
2.3 Verwandte Arbeiten . . . 8

**3 Entwurf der Laufzeitübersetzersteuerung** 10
3.1 JamaicaVM . . . 10
3.2 Laufzeitdatenerfassung . . . 11
3.3 Profilerstellungsverfahren . . . 11
3.3.1 Bewertungsfunktionen . . . 11
3.3.2 Alterung . . . 14
3.4 Übersetzungsstrategie . . . 14

**4 Softwarearchitektur** 16
4.1 Zusammenfassung der Parameter . . . 16

**5 Analyseverfahren** 18
5.1 Testsysteme . . . 18
5.2 Beispielprogramme . . . 18
5.3 Versuchsaufbau . . . 19
5.3.1 Messungen . . . 19
5.3.2 Gesammelte Daten . . . 20
5.4 Analyse . . . 20
5.4.1 Was macht eine gute Laufzeitübersetzersteuerung aus? . . . 20
5.4.2 Referenzprofil . . . 20
5.4.3 Auswertung der Messprotokolle . . . 22

**6 Feinabstimmung der Parameter** 24
6.1 Erste Evaluierung . . . 24
6.1.1 Grenzen . . . 24
6.1.2 Messung auf pc-x86 . . . 24
6.1.3 Übertragung der Messung auf es-xscale . . . 28
6.2 Alterung . . . 29
6.2.1 Konfigurationen . . . 29

6.2.2 Auswertung . . . . . . . . . . 30
6.3 Kombination von Bewertungsfunktionen . . . . . . . . . . 30
6.3.1 Konfigurationen . . . . . . . . . . 31
6.3.2 Auswertung . . . . . . . . . . 31

**7 Schlussfolgerung und Ausblick** **33**
7.1 Zusammenfassung . . . . . . . . . . 33
7.2 Schlussfolgerungen . . . . . . . . . . 33

**8 Referenzen** **35**

# 1 Einführung

Mit immer komplexer werdenden eingebetteten Systemen erhält die Programmiersprache Java Einzug auf einer Vielzahl von Geräten abseits von Arbeitsplatzrechnern. Die Portabilität als Kernmerkmal der Sprache und ausgereifte Entwicklungswerkzeuge machen den Einsatz von Java auf eingebetteten Systemen attraktiv.

Problematisch ist jedoch, dass Java-Programme zur Laufzeit von einer virtuellen Maschine interpretiert werden müssen. Im Vergleich zur sonst häufig auf eingebetteten Systemen eingesetzten, systemnahen Programmierung ist die Interpretierung langsam. Ein Ansatz zur Verbesserung der Ausführungsgeschwindigkeit ist die Laufzeitübersetzung.

Ein Laufzeitübersetzer muss jedoch gezielt eingesetzt werden. Wir untersuchen in dieser Arbeit, welches Verbesserungspotential durch den Einsatz von einfachen, aber "schlauen" Heuristiken zur Steuerung des Laufzeitübersetzers erzielbar ist.

In Abschnitt 2 diskutieren wir, welche Herausforderungen für einen gewinnbringenden Einsatz eines Laufzeitübersetzers beachtet werden müssen. Außerdem zeigen wir einige Lösungsansätze aus Literatur und Praxis auf.

In Abschnitt 3 entwerfen wir unsere Laufzeitübersetzersteuerung. Die Entscheidung, welche Methoden zur Laufzeit übersetzt werden sollen, wird mit Hilfe von Bewertungsfunktionen getroffen, die dazu bestimmte Eigenschaften des Programms und seines Verhaltens zu Rate ziehen. Die implementierte Softwarearchitektur erläutern wir in Abschnitt 4.

Der Versuchsaufbau zur Evaluierung und Gütekriterien für die Übersetzersteuerung werden in Abschnitt 5 vorgestellt.

In Abschnitt 6 führen wir schließlich Messungen mit verschiedenen Konfigurationen der Übersetzersteuerung durch. Eine erste Messung untersucht die grundlegenden Eigenschaften der vorgestellten Lösung und ist der Ausgangspunkt für weitere Verbesserungsversuche.

Die gesammelten Erkenntnisse fassen wir in Abschnitt 7 zusammen und bewerten sie im Hinblick auf die Zielsetzung.

# 2 Grundlagen

Im folgenden Abschnitt erläutern wir die Grundlagen der Sprache Java, eingebetteter Systeme und der Laufzeitübersetzung. Anschließend stellen wir verwandte Arbeiten aus dem Gebiet der Laufzeitübersetzersteuerung vor.

## 2.1 Grundlagen

### 2.1.1 Java

Java ist eine objektorientierte Programmiersprache, die von Sun Microsystems 1996 vorgestellt wurde. [8] Aus Gründen der Portabilität werden Java-Programme vom Java-Übersetzer in *Bytecode* übersetzt, dessen Zielarchitektur keine konkrete Prozessorarchitektur, sondern eine abstrakte Kellermaschine ist. Daher wird eine Laufzeitumgebung benötigt, die den Bytecode interpretiert, die sogenannte *virtuelle Maschine*.

### 2.1.2 Eingebettete Systeme

Ein *eingebettetes System* ist ein Rechnersystem, das nicht Bestandteil anderer Rechnersysteme ist, sondern in seine Umwelt eingebunden ist und mit ihr interagiert. Unter diese Definition fällt ein großes Spektrum an Geräten, das von handelsüblichen Standardrechnern bis zu kleinsten Mikrosteuerbausteinen reicht. [7]

In dieser Arbeit liegt der Fokus auf Geräten, die zur Maschinensteuerung verwendet werden. Bezüglich ihrer Prozessorgeschwindigkeit und ihrem Speicherausbau sind sie leistungsschwächer als Standardrechner, aber leistungsstark genug, um ein vollständiges Betriebssystem und eine virtuelle Maschine für Java auszuführen.

## 2.2 Laufzeitübersetzung

Unter *Laufzeitübersetzung* versteht man die Transformation von einer Quell- in eine Zielsprache zur Laufzeit des Programms, wobei im Allgemeinen das Programm in der Zielsprache schneller als die Repräsentation in der Quellsprache auf der zugrunde liegenden Maschine ausgeführt werden kann. Das erste System dieser Art stammt aus dem Jahr 1960 für LISP. Der englische Begriff *Just-in-Time compilation* wurde von Gosling im Zusammenhang mit Java geprägt. [3]

### 2.2.1 Laufzeitübersetzung für Java

Bezogen auf ein Java-System bedeutet Laufzeitübersetzung, dass man aus dem Java-Bytecode, der von der virtuellen Maschine interpretiert werden muss, Instruktionen in der Maschinensprache für den zugrunde liegenden Prozessor erzeugt, welche dann direkt von diesem ausgeführt werden können.

Die ersten virtuellen Maschinen waren langsam, so dass relativ bald die ersten Laufzeitübersetzer für Java entwickelt wurden, um die Performanz-Probleme der Interpretierung

in den Griff zu bekommen. Verursacht werden sie hauptsächlich durch das explizite Laden und Dekodieren der Bytecode-Instruktionen bei der Interpretierung. [3, 13]

Allerdings sind Interpretierer klein und einfach zu implementieren, wohingegen ein Laufzeitübersetzer ein umfangreiches Softwaresystem ist, das zur Laufzeit einen Mehrbedarf an Rechenleistung und Arbeitsspeicher hat. [4]

Als Alternative zur Laufzeitübersetzung ist eine statische Übersetzung möglich. Damit verliert man aber die Portabilität der Java-Programme, eines der Kernmerkmale der Sprache. Zudem erlaubt Java es, dynamisch neue Klassen in das Laufzeitsystem zu laden, ein statischer Übersetzer kann aber prinzipbedingt nur die Teile des Programms übersetzen und beschleunigen, die zur Übersetzungszeit bekannt sind.

Die Sprache der virtuellen Maschine stellt mächtigere Instruktionen als konkrete Prozessoren zur Verfügung, so dass der Bytecode die gleiche Programmsemantik in kompakterer Form darstellen kann. Abbildung 1 zeigt: Interpretierte Programme sind kleiner, aber langsamer. Programme in Maschinensprache sind schneller, aber größer. [3] Ist bei einem System der verfügbare Speicher stark begrenzt, muss man unabhängig von statischer oder Laufzeitübersetzung einen Kompromiss zwischen Geschwindigkeit und Speicherverbrauch treffen.

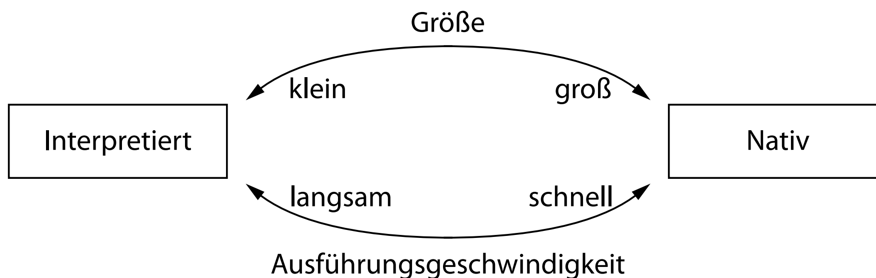



**Abbildung 1:** Unterschiede bei der Programmrepräsentation

Objektorientierte Entwürfe tendieren dazu, aus vielen, eher kleinen Methoden zu bestehen. [15] Zum Beispiel ist es gängige Praxis, Zugriffsmethoden für Instanzvariablen einzuführen, und jede Klasse hat mindestens den impliziten Standardkonstruktor. [13]

Die in dieser Arbeit entworfene Laufzeitübersetzersteuerung arbeitet daher auf Methodenebene.

### 2.2.2 Herausforderungen

Das Hauptproblem bei der Laufzeitübersetzung und ihrer Steuerung ist jedoch, dass der Gewinn an Geschwindigkeit den Mehraufwand zur Laufzeit amortisieren muss.

Wir definieren zunächst die *Laufzeit* $T$ als die Zeit, die für die Abarbeitung der Instruktionen eines Programms oder eines Programmteils benötigt wird. Die Laufzeitübersetzung ist sinnvoll, wenn man eine *Laufzeitverbesserung* erreicht. Dies ist dann der Fall,

wenn der Nutzen der Laufzeitübersetzung größer ist als ihre Kosten. Die Laufzeit hat sich verbessert, wenn die Differenz der Laufzeiten vor und nach dem Einsatz des Laufzeitübersetzers $\Delta T < 0$ ist.

Die Instruktionen des Programms liegen anfangs in Bytecode, nach der Laufzeitübersetzung auch in der Maschinensprache des Prozessors vor. Letztere nennt man auch *native* Instruktionen. Somit lässt sich die Laufzeit als Summe der *Interpretierungszeiten* $t_{interpret}(m)$ und der *Ausführungszeiten* $t_{native}(m)$ aller Methoden $m \in M$ auffassen. Durch sie wird beschreiben, wie lange die Interpretierung der Bytecode-Instruktionen bzw. die Ausführung der nativen Instruktionen einer Methode m dauert. Die Differenz $t_{interpret}(m) - t_{native}(m)$ drückt die Laufzeitverbesserung einer Methode aus. Wurde m nicht übersetzt, ist $t_{native}(m) = 0$. $i_{native}(m)$ bezeichnet die Anzahl der Aufrufe der Methode m nach ihrer Übersetzung. Nun kann man den Nutzen der Laufzeitübersetzung als

$$\sum_{m \in M} i_{native}(m) \cdot (t_{interpret}(m) - t_{native}(m))$$

angeben.

Die *Übersetzungszeit* $t_{compile}(m)$ beschreibt die Dauer der Transformation des Bytecodes einer Methode in native Instruktionen. Es gilt $t_{compile}(m) = 0$, wenn m nicht übersetzt wurde. Für die Erkennung, Verwaltung und Auswahl der Methoden zur Übersetzung wird der *Verwaltungsmehraufwand* $T_{measure}$ veranschlagt. Die Kosten der Laufzeitübersetzung ergeben sich zu

$$T_{measure} + \sum_{m \in M} t_{compile}(m)\,.$$

Der vollständige Zusammenhang für die Laufzeitverbesserung lautet also

$$\Delta T_{overall} \approx (T_{measure} + \sum_{m \in M} t_{compile}(m)) - (\sum_{m \in M} i_{native}(m) \cdot (t_{interpret}(m) - t_{native}(m)))\,.$$

[16]
Idealerweise wählt man, basierend auf oben genannter Formel, zu einem gegebenen Zeitpunkt die Methoden aus, die die Laufzeitverbesserung des Programms maximieren.
Im Allgemeinen kann man nicht davon ausgehen, über die benötigten Informationen zu verfügen, weil sie größtenteils dem *zukünftigen* dynamischen Verhalten des Programms unterliegen. [11]

### 2.2.3 Heuristiken

Das zukünftige Verhalten des Programms kann nur geschätzt, aber nicht sicher vorhergesagt werden. Trotzdem ist eine gute Auswahl der Übersetzungskandidaten erforderlich, weil die Kosten der Übersetzung leicht ihren Nutzen aufheben können. Aufgrund dieser Problematik stützen sich Laufzeitübersetzersteuerungen auf Heuristiken, die versuchen,

aus Eigenschaften eines Programms und dessen dynamischem Verhalten in der Vergangenheit Rückschlüsse auf das zukünftige Verhalten zu ziehen.

Eine grundlegende Annahme in den Heuristiken ist, dass das Programm sein Verhalten in der Zukunft nicht ändert. Man beschränkt sich auf die Betrachtung des aktuellen und vergangenen Verhaltens.

Die Ausführungszeit der übersetzten Methode ist von vielen architekturspezifischen Eigenschaften, wie beispielsweise Befehlssatz und Cacheverhalten, abhängig und lässt sich daher kaum schätzen.

Die Übersetzungszeit kann man zwar grob als linear in der Anzahl der Bytecode-Instruktionen abschätzen [11], jedoch führen z.B. aufwändige Optimierungen dazu, dass dieser Zusammenhang aufgeweicht wird.

Diese beiden Faktoren sind also wenig praktikabel für eine Laufzeitübersetzersteuerung. Abstrahiert man sie aus der Betrachtung, verbleibt die Interpretierungszeit als Kriterium. Eine bekannte empirische Erkenntnis von Knuth sagt aus, dass ein kleiner Teil der Anweisungen eines Programms für einen Großteil seiner Laufzeit verantwortlich ist. [5] Auch die Autoren von [12] stellen fest, das typischerweise 80 % und mehr der Methoden kaum Einfluss auf die Laufzeit haben. Es reicht also aus, die verbleibenden Methoden zu übersetzen. Laufzeit-relevante Methoden nennen wir *wichtig*, in der englischen Literatur werden sie als “hot spots” bezeichnet.

Nun ist die Frage, wie man bestimmt, welche Methoden wichtig sind. In [10] ist der in der Literatur vorherrschende Konsens zusammengefasst: “Eine lange laufende Methode ist vermutlich wichtig”. Der genaueste Weg, lange laufende Methoden zu finden, ist das Messen der während der Interpretierung verstrichenen Zeit. Da in typischen Java-Programmen sehr viele Methoden aufgerufen werden, macht der zu erwartende Mehraufwand für die Zeitmessung dieses Vorgehen untauglich für eine Laufzeitübersetzersteuerung. Man muss also auf anderem Wege eine Approximation für die erwartete Interpretierungszeit finden. In dieser Arbeit untersuchen wir dazu verschiedene, algorithmisch einfache Bewertungsfunktionen.

### 2.2.4 Laufzeitprofile

Wir fassen die von diesen Funktionen gefundenen Schätzungen zu einem *Laufzeitprofil* zusammen. An diesem Profil können wir ablesen, welche Methoden relevant für die Laufzeit des Programms sind, und nutzen es daher als Entscheidungsgrundlage für unsere Laufzeitübersetzersteuerung.

Allgemein ordnet ein Laufzeitprofil jeder Methode ein Maß für ihren Anteil an der Laufzeit des Programms zu. Das von uns erstellte Profil basiert auf *Bewertungspunkten*, die durch einen ganzzahligen Wert repräsentiert werden. Als Maß kommen aber auch andere Eigenschaften in Frage, beispielsweise die Anzahl der Aufrufe einer Methode, die Anzahl ihrer interpretierten Bytecode-Instruktionen oder auch direkt ihre Inter-

pretierungszeit.

Ein Laufzeitprofil kann *flach* oder *bergig* sein. [14] In flachen Profilen haben viele Methoden einen ähnlichen Anteil an der Laufzeit des Programms. Im Gegensatz dazu gibt es in einem bergigen Profil wenige, sehr wichtige Methoden und folglich viele unwichtige Methoden.

## 2.3 Verwandte Arbeiten

Die bekannten virtuellen Maschinen von Sun Microsystems und IBM zielen auf den Einsatz auf leistungsstarken Servern und Arbeitsplatzrechnern ab. Das Forschungsinteresse der Firmen liegt daher auf Optimierungen, die zu aufwändig für eingebettete Systeme sind. Die Laufzeitübersetzer unterstützen mehrere Stufen der Optimierung und benötigen somit auch eine komplexe Übersetzersteuerung. Für die Erkennung der wichtigen Programmteile verfolgen sie unterschiedliche Ansätze.

Die virtuelle Maschine des "IBM Development Kit for Java 1.1.8" speichert für jede Methode einen Zähler, der beim Aufruf der Methode dekrementiert wird. Enthält die Methode Schleifen, wird versucht, die Anzahl der Schleifeniterationen durch einen Mustervergleich zu erraten. Der Zähler wird um diese Anzahl dekrementiert. In einer neueren Version 1.4 der virtuellen Maschine wird zusätzlich pro Methode ein Zähler eingeführt, der inkrementiert wird, wenn eine Methode durch eine Stichprobennahme als gerade laufend identifiziert wird. Erreicht dieser zweite Zähler bestimmte Schwellwerte, wird die Methode mit einer höheren Optimierungsstufe neu übersetzt. [13, 12]

In der Version 6 wird die "HotSpot" virtuelle Maschine von Sun mit zwei verschiedenen Laufzeitübersetzern ausgeliefert. Der "Server"-Übersetzer setzt aufwändige Optimierungen ein und zielt auf maximale Ausführungsgeschwindigkeit. Der "Client"-Übersetzer stellt einen Kompromiss zwischen Übersetzungszeit und Ausführungsgeschwindigkeit dar. Wir betrachten den "Client"-Übersetzer. Auch er zählt pro Methode, wie oft sie aufgerufen wird. Zudem werden die Rücksprünge während der Interpretierung einer Methode gezählt. Erreichen diese einen gewissen Schwellwert, wird die Methode übersetzt und noch während sie interpretiert wird, auf die native Version umgeschaltet. [9]

Der Stand der Technik in der Laufzeitprofilerstellung wird unter anderem in [6, 16, 17] beschrieben. Dort werden sogenannte Knoten- oder Pfadprofile erstellt. Dabei handelt es sich um Laufzeitprofile, aus denen man ablesen kann, welche Pfade im Aufrufgraph häufig genutzt werden. Solche Informationen benötigt der Laufzeitübersetzer für weitergehende Optimierungen, wie z.B. den offenen Einbau von Methoden. Da der in dieser Arbeit verwendete Laufzeitübersetzer solche Optimierungen nicht durchführt und wir die Erstellung der Profile als zu aufwändig für ein eingebettetes System einschätzen, betrachten wir diese Ansätze nicht weiter.

In [11] wird eine Übersetzersteuerung beschrieben, die zwar keinen direkten Bezug auf den Einsatz auf eingebetteten Systemen hat, aber auch einfache Heuristiken zur Auswahl der Übersetzungskandidaten als Alternative zur unbedingten Übersetzung nutzt.

Die Autoren in [10] untersuchen eine ähnliche Problemstellung wie diese Arbeit. Sie entwickeln eine Heuristik, die während der Interpretierung mit geringem Aufwand die Interpretierungszeit approximiert. Dieser Ansatz erfordert aber eine Integration in die virtuelle Maschine und lässt sich somit nicht lose an eine bestehende virtuelle Maschine koppeln.

# 3 Entwurf der Laufzeitübersetzersteuerung

In diesem Abschnitt stellen wir den Aufbau der Laufzeitübersetzersteuerung vor und beschreiben die Implementierung als Erweiterung für die JamaicaVM. Sie gliedert sich in Laufzeitdatenerfassung, Profilerstellung und Übersetzungsstrategie. Die Zusammenarbeit der Komponenten illustriert Abbildung 2.

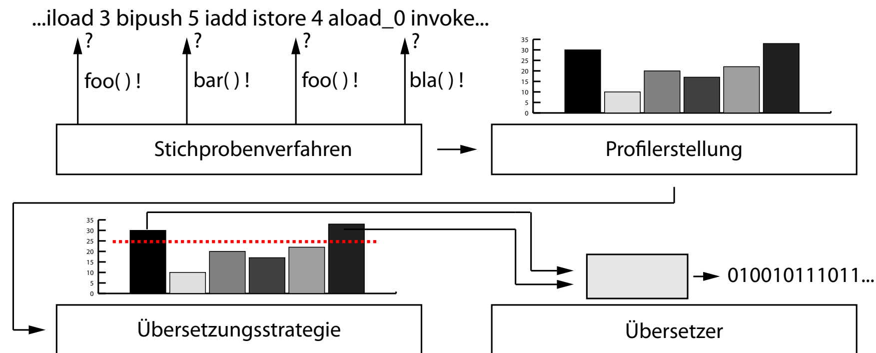



**Abbildung 2:** Aufbau der Laufzeitübersetzersteuerung

## 3.1 JamaicaVM

Eine Implementierung einer virtuellen Maschine für Java ist die *JamaicaVM*, ein Produkt der aicas GmbH aus Karlsruhe. Diese virtuelle Maschine ist auf vielen eingebetteten Plattformen lauffähig.
Zur Verbesserung der Ausführungsgeschwindigkeit besteht bei JamaicaVM traditionell die Möglichkeit, mit einem Werkzeug Java-Programme in C-Quelltexte zu übersetzen, die dann von dem C-Übersetzer der jeweiligen Plattform in die passende Maschinensprache übersetzt werden.

Mit JamaicaVM ist es möglich, in einem speziellen Lauf des Programms ein Laufzeitprofil zu erstellen, anhand dessen der statische Übersetzer entscheidet, welche Teile des Programms er übersetzt. Die Erstellung des Profils ist jedoch ein zusätzlicher Arbeitsschritt für den Anwender. Zudem ist dieses Vorgehen nur möglich, wenn das Programm, seine Laufzeitparameter und repräsentative Eingabedaten zur Übersetzungszeit bekannt sind.
Diese Problematik hat die Entwicklung eines Laufzeitübersetzers motiviert. Zur Zeit steht er für die Prozessorarchitekturen IA-32, PowerPC und ARM zur Verfügung. Der Übersetzer selbst ist vollständig in Java implementiert, greift aber über das Java Native Interface auf Methoden der virtuellen Maschine zu, beispielsweise um dort eine übersetzte Methode zu registrieren. Allerdings führt der Laufzeitübersetzer kaum Optimierungen durch und nutzt auch die Prozessorregister nicht effektiv, weil er mit der Zwischendarstellung des statischen Übersetzers arbeitet. In dieser Zwischendarstellung ist keine Unterstützung für diese Aufgaben vorgesehen, weil sie im Falle des statischen Übersetzers vom C-Übersetzer durchgeführt werden.

## 3.2 Laufzeitdatenerfassung

Die Laufzeitdatenerfassung beobachtet das dynamische Verhalten des Programms. Es gibt zwei Ansätze, die benötigten Laufzeitdaten zu sammeln. Das *Instrumentierungsverfahren* erweitert den Programmfluss um die Datenerfassung, etwa durch das Einfügen von Bytecode-Instruktionen oder durch explizite Verwaltungsmethodenaufrufe im Interpretierer. Instrumentierung liefert im Allgemeinen sehr genaue Daten. Problematisch ist jedoch, dass die Menge der anfallenden Daten vom Programmfluss abhängig und somit kaum zu kontrollieren ist. [16]

Das *Stichprobenverfahren* unterbricht periodisch die Ausführung auf der virtuellen Maschine und ermittelt die aktuell laufende Methode. Von Vorteil ist, dass sich durch Variation der Stichprobenfrequenz das Verhältnis von Genauigkeit und Mehraufwand der Datenerfassung gut abstimmen lässt. Zudem lässt es sich lose an die virtuelle Maschine angekoppelt implementieren. Wir verwenden daher in dieser Arbeit das Stichprobenverfahren.

Im Folgenden bezeichnen wir die Tatsache, dass eine Methode m durch die Stichprobennahme als gerade laufend identifiziert wurde, als ein *Stichprobenvorkommen von m*. Die Anzahl der Stichprobenvorkommen von m wird mit $p(m)$ bezeichnet.

## 3.3 Profilerstellungsverfahren

Mit den Daten aus der Laufzeitdatenerfassung erstellt das Profilerstellungsverfahren das für die Übersetzersteuerung erforderliche Laufzeitprofil. Wir verwalten die Methoden, die mindestens einmal in der Stichprobe auftreten, als *Kandidaten*. Für jeden Kandidaten wird eine Anzahl an *Bewertungspunkten* $\beta$ gespeichert. Die Verteilung der Bewertungspunkte interpretieren wir als das Laufzeitprofil.

Die Bewertungspunkte werden folgendermaßen vergeben. Das Profilerstellungsverfahren ist mit einer Menge von *Bewertungsfunktionen* konfiguriert. Diese Bewertungsfunktionen betrachten jeweils einen speziellen Aspekt eines Kandidaten und vergeben dafür eine Teilbewertung. Jedes Mal, wenn ein Kandidat in der Stichprobe vorkommt, werden die Bewertungsfunktionen abgefragt. Die Teilbewertungen können noch mit einem Faktor $\omega_X$ multipliziert werden, um sie auf einen ähnlichen Wertebereich zu normieren oder eine Bewertungsfunktion im Vergleich zu einer anderen stärker zu gewichten. Die gewichteten Teilbewertungen werden summiert und dann dem Kandidaten gutgeschrieben. Algorithmus 1 verdeutlicht das Verfahren.

### 3.3.1 Bewertungsfunktionen

Wir suchen ein Profilerstellungsverfahren, das ein Laufzeitprofil ohne explizite Messung der Interpretierungszeit approximiert. Wir betrachten einige, dafür in Frage kommende Eigenschaften und erläutern die Motivation, sie in eine Bewertungsfunktion umzusetzen. Die Bewertungsfunktionen formulieren wir immer so, dass die als Ergebnis der Bewertungsfunktionen angegebenen Werte dem betroffenen Kandidaten bei jeder Stichprobennahme gutgeschrieben werden.

**Algorithmus 1** Profilerstellung

```
procedure updateProfile(method)
  candidate = lookupOrCreate(method)
  scoreDelta := 0
  foreach RatingStrategy X do
    scoreDelta := scoreDelta + w_X * X.rate(candidate)
  end
  candidate.adjustScore(scoreDelta);
end
```

**Konstante Bewertungsfunktion (C)** Das Stichprobenverfahren findet Methoden, die lange laufen, häufig aufgerufen werden, oder auf die beides zutrifft, mit hoher Wahrscheinlichkeit. Da solche Methoden vermutlich wichtig sind, nutzen wir diese Eigenschaft ohne weitere Gewichtung, indem wir alle Methoden mit der gleichen Konstante bewerten.

**Algorithmus 2** Konstante Bewertungsfunktion

$C(m) = 1$

**Akkumulierende Bewertungsfunktion (A)** Wir übernehmen den Grundgedanken von C, verlangen aber, dass eine Methode mit jedem Stichprobenvorkommen ein größeres Gewicht bekommt, um die wichtigsten Methoden stärker von normalen Methoden abzugrenzen. Wir bewerten sie daher mit der Anzahl ihrer bisherigen Stichprobenvorkommen $p(m)$.

**Algorithmus 3** Akkumulierende Bewertungsfunktion

$A(m) = p(m)$

**Größe der Methode (B)** Wenn eine Methode aus vielen Bytecode-Instruktionen besteht, nehmen wir an, dass ihre Interpretierung länger dauert als die einer kleinen Methode. Daraus ziehen wir zwei Schlüsse. Erstens kann die Methode leichter einen signifikanten Anteil zur Laufzeit des Programms beitragen.

Zweitens ist das Laufzeitverbesserungspotential bei der gegebenen Methode größer als bei einer Methode mit weniger Instruktionen. Durch die Übersetzung kann für mehr Bytecode-Instruktionen das Laden und Dekodieren im Interpretierer eingespart werden. Zudem spart man den Mehraufwand für das Aufrufen des Laufzeitübersetzers, wenn man eine bestimmte Anzahl Bytecode-Instruktionen in einer großen, statt in vielen kleinen Methoden, übersetzen lässt.

Wir bewerten die Methode also mit ihrer Größe in Byte.

**Algorithmus 4** Größe der Methode

$B(m) = |m.bytecode|$

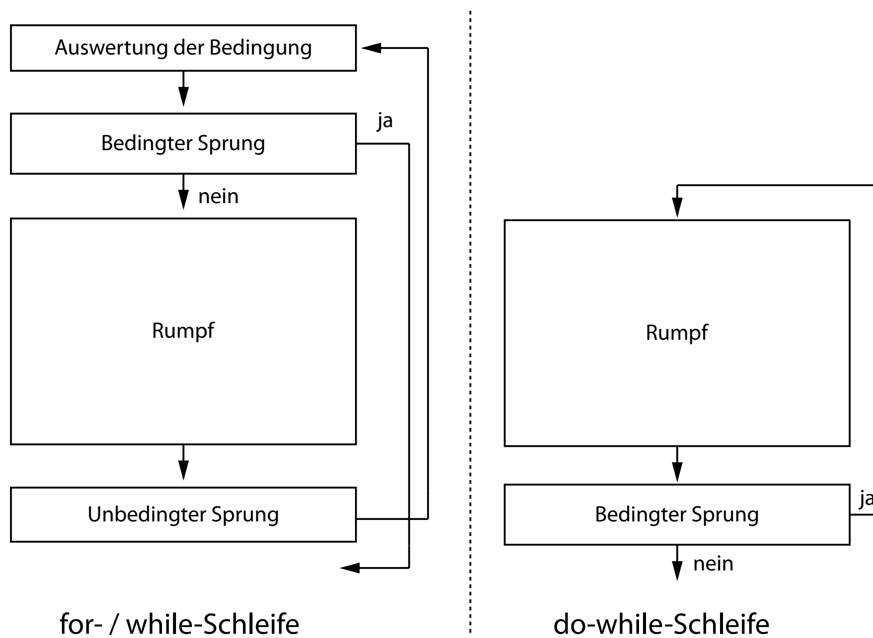



**Abbildung 3:** Muster für die Schleifenerkennung

**Erkennung von Schleifen (SA)** Enthält eine Methode Schleifen, kann sie für eine lange Interpretierungszeit verantwortlich sein, weil sie dynamisch mehr Instruktionen ausführt als in ihrem Bytecode vorhanden sind. Darum wollen wir Instruktionen, die sich in einem Schleifenrumpf befinden, ein größeres Gewicht zuordnen als Instruktionen außerhalb. Wir implementieren die Schleifenerkennung als einfache Mustererkennung. Abbildung 3 zeigt die Muster für typische for-, while- und do-while-Schleifen. Wir schätzen die Anzahl Schleifendurchläufe grob mit der Konstante $c_{loop} = 10$ ab.

**Algorithmus 5** Schleifenerkennung

```
procedure analyze(method, startIndex, endIndex)
  rating := 0
  foreach Instruction i in method between [startIndex, endIndex] do
     if i belongs to loop pattern [a,b]
       rating := bonus + c_loop * (b-a) + analyze(method, a+1, b-1)
     end
  end
return rating
```

$SA(m) = analyze(m, 0, |m.bytecode|)$

**Aufrufkellerinspektion (SW)** Wir inspizieren den Aufrufkeller mit beschränkter Tiefe, um die Aufrufer der aktuellen Methode festzustellen. Die Aufrufkellerinspektion ist aus zwei Gründen sinnvoll.

Wenn eine interpretierte Methode eine native Methode aufruft, oder in den nativen Instruktionen ein Aufruf einer nur in Bytecode vorliegenden Methode ausgeführt werden soll, muss die virtuelle Maschine zusätzliche Anweisungen ausführen, um die Aufrufkonventionen bezüglich Parameterübergabe und Rückgabewert herzustellen. Um diese Anweisungen zu vermeiden, erhalten Methoden, die noch interpretiert werden, aber von einer nativen Methode aufgerufen werden, einen Bonus $c_{context} = 10$.

Zusätzlich wird für den Aufrufer der aktuellen Methoden ein Bonus vorgemerkt, weil Methodenaufrufe bedingt durch die dynamische Bindung bei Java nicht vernachlässigbar sind. Bei einer Methode, die viele andere Methoden aufruft, ist die Wahrscheinlichkeit groß, dass sie von Bedeutung für die Laufzeit des Programms ist.

Der Bonus wird gutgeschrieben, wenn der Aufrufer das nächste Mal selbst an der Spitze des Aufrufkellers ist.

**Algorithmus 6** Aufrufkellerinspektion

$a_m$ bezeichnet die Vorkommen von $m$ auf dem Aufrufkeller anderer Methoden.

$$SW(m) = a_m + \begin{cases} c_{context} & \text{Aufrufer ist nativ} \\ 0 & \text{sonst} \end{cases}$$

### 3.3.2 Alterung

Die *Alterung* des Laufzeitprofils sorgt dafür, dass neuere Bewertungen stärker gewichtet werden als ältere. Intuitiv lässt sich die Alterung implementieren, in dem man die aktuelle Bewertung vor dem Aufaddieren der neuen Bewertungspunkte mit dem Alterungsfaktor multipliziert. Dabei muss man aber die zeitliche Differenz zum letzten Stichprobenvorkommen berechnen und einbeziehen.

In diesem Laufzeitprofilersteller wählen wir einen etwas anderen Ansatz. Die Alterung ist als eigener Ausführungsstrang implementiert, der periodisch die Bewertungspunkte jedes Kandidaten auf einen konfigurierbaren, prozentualen Anteil reduziert. Dies hat den Vorteil, dass alle Kandidaten automatisch gleichmäßig und unter Berücksichtigung der tatsächlich verstrichenen Zeit gealtert werden. Wir begrenzen den Mehraufwand, der durch das Iterieren über alle Kandidaten beim Aktualisieren der Bewertungspunkte entsteht, in dem wir Kandidaten ohne Bewertungspunkte löschen.

## 3.4 Übersetzungsstrategie

Die Übersetzungsstrategie nimmt die Klassifikation der Kandidaten in wichtige oder unwichtige Methoden durch einen Vergleich mit einem *Schwellwert* $\tau$ vor. Für diese Laufzeitübersetzersteuerung sind derzeit zwei solcher Strategien implementiert.

**So bald wie möglich mit festem Schwellwert (asap)** Während der gesamten Laufzeit gilt ein konstanter Schwellwert $\tau = c$. Erreicht ein Kandidat eine Bewertung, die größer als dieser Wert ist, wird er als wichtig markiert und in die Übersetzungswarteschlange

eingereiht. Die Warteschlange ist nach der Anzahl der Bewertungspunkte sortiert. Der Übersetzer wird aktiv, sobald sich Kandidaten in der Warteschlange befinden. Da er in einem Ausführungsstrang mit der höchsten Priorität läuft, verdrängt er das Programm und die Kandidaten in der Warteschlange werden sofort übersetzt.

**So bald wie möglich mit variablem Schwellwert (asap-p)** Die asap-p-Strategie erweitert asap um eine Anpassung des Schwellwerts zur Laufzeit. Der Schwellwert ist initial $\tau = \infty$ und wird periodisch nach einer konfigurierbaren Anzahl Stichprobennahmen als prozentualer Anteil aller bis zum aktuellen Zeitpunkt vergebenen Bewertungspunkte $\beta_{total}$ neu berechnet. Ist die Alterung aktiviert, so unterliegt auch $\beta_{total}$ dem Alterungsmechanismus. Ohne die Alterung wächst der Schwellwert mit der Zeit unbeschränkt, was zur Folge hat, dass die Laufzeitübersetzersteuerung nicht mehr auf neue Ausführungsphasen des Programms reagieren kann.

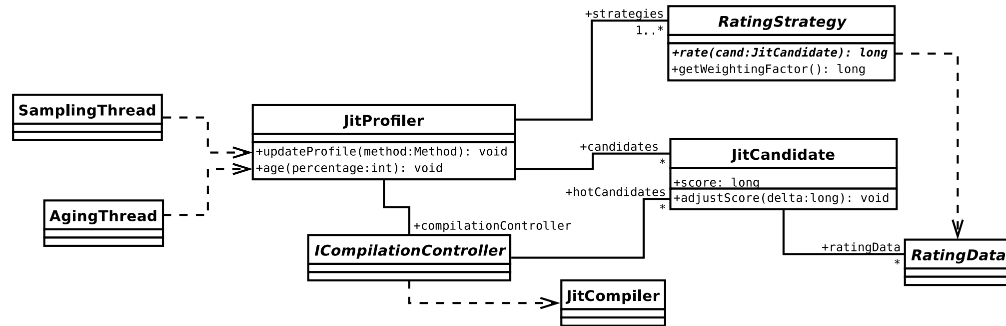



**Abbildung 4:** Vereinfachte Softwarearchitektur

# 4 Softwarearchitektur

Die implementierte Softwarearchitektur ist schematisch in Abbildung 4 dargestellt.

Kandidaten werden durch eine Klasse `JitCandidate` modelliert, die Bewertungspunkte und die Daten zur weiteren Analyse speichert. Die Bewertungsfunktionen sind als `RatingStrategy` gemäß des Strategiemusters abstrahiert. Jede Bewertungsfunktion kann in den Kandidatenobjekten ein eigenes Datenobjekt anlegen, wo sie ihre strategiespezifischen Daten ablegt.

Auf die implementierten Übersetzungsstrategien wird über die Schnittstelle `ICompilationController` zugegriffen. Die zentrale Klasse `JitProfiler` speichert die Kandidaten und verbindet alle Komponenten der Steuerung miteinander. Die Stichprobennahme und die Alterung sind als eigene Ausführungsstränge implementiert und rufen periodisch die Methoden zur Datenerfassung und zum Ausführen der Alterung in `JitProfiler` auf.

In `JitProfiler` sind alle Kandidaten in einer Streuwerttabelle gespeichert. Die Datenstruktur verfügt intern noch über eine doppelt-verkettete Liste der Elemente, was ein effizientes Iterieren über die Kandidaten ermöglicht.

Die beiden Übersetzungsstrategien nutzen eine verkettete Liste, die bei Bedarf sortiert wird, um die Übersetzungswarteschlange zu modellieren. Prinzipbedingt ist diese Warteschlange jedoch meistens leer und dient mehr der Entkopplung der Ausführungsstränge von Stichprobennahme und Übersetzungsstrategie.

## 4.1 Zusammenfassung der Parameter

Tabelle 1 gibt einen Überblick über die Parameter der vorgestellten Laufzeitübersetzersteuerung.

| Parameter | Einheit | Beschreibung |
|---|---|---|
| Stichprobenintervall | ms | Legt die Stichprobenfrequenz fest. |
| Schwellwert | Bewertungspunkte bzw. % BP | Bestimmt, ab wann ein Kandidat als wichtig klassifiziert wird. |
| Übersetzungsstrategie | "asap" oder "asap-p" | Wählt die Übersetzungsstrategie aus. |
| Neuberechnung des Schwellwerts | Stichprobennahmen | Legt bei asap-p fest, nach wie vielen Stichprobennahmen der Schwellwert aktualisiert wird. |
| Bewertungsfunktionen | Liste von "Gewichtung * Bewertungsfunktion" | Legt fest, welche Bewertungsfunktionen mit welcher Gewichtung zur Profilerstellung benutzt werden. |
| Alterungsintervall | ms | Legt die Alterungsfrequenz fest. |
| Alterungsfaktor | % | Bestimmt die Stärke der Alterung. |
| Aufrufkellerinspektionstiefe | 1 | Bestimmt, wie weit die Aufrufkette ausgehend von der gerade laufenden Methoden verfolgt wird. |

**Tabelle 1:** Parameter der Laufzeitübersetzersteuerung

# 5 Analyseverfahren

In diesem Abschnitt stellen wir den verwendeten Versuchsaufbau vor. Wir beschreiben die Testsysteme und verwendeten Beispielprogramme und erläutern die Analysemethoden, die später bei der Feinabstimmung angewendet werden.

## 5.1 Testsysteme

Die Messungen zur Analyse des Laufzeitprofilerstellers führen wir auf einem tragbaren Rechner ("pc-x86") und einer Steuereinheit für Gasanalysesysteme ("es-xscale") durch. Da auf dem Gerät ein Linux-System läuft, lassen sich die Messungen mit wenig Aufwand und sehr ähnlich wie auf pc-x86 durchführen. Tabelle 2 zeigt die technischen Daten der Systeme.

| Bezeichnung: | pc-x86 | es-xscale |
|---|---|---|
| Prozessor | Intel Core 2 Duo T7500, 2.20GHz | XScale-PXA255, 400 MHz |
| Cache | 4096 KB | je 32 KB Daten / Instruktionen |
| Arbeitsspeicher | 2048 MB | 128 MB |
| Betriebssystem | Linux 2.6.27 | Linux 2.4.19 |
| Linux BogoMIPS (pro CPU) | 4388 | 397 |
| Embedded CaffeineMark-Punkte | 660 | 68 |
| Besonderheiten | - | keine Gleitkommaeinheit |

**Tabelle 2:** Technische Daten der Testsysteme

## 5.2 Beispielprogramme

Als Beispielprogramme kommen Arbeitslasten aus der Benchmark-Sammlung SPECjvm-2008 zum Einsatz. [2] Die hier ermittelten Daten sind nicht mit anderen Messungen aus SPECjvm2008-Sammlung vergleichbar, weil der Umfang Eingabedaten der Arbeitslasten gegenüber den von der SPEC ausgelieferten Konfiguration verkleinert wurde. Die Charakteristika der Programme werden im folgenden erläutert.

**compress** Diese Arbeitslast liest ein Archiv im TAR-Format ein, komprimiert es mittels des Lempel-Ziv-Welch-Algorithmus und dekomprimiert die Daten anschließend wieder. Die Berechnungen werden in Ganzzahlarithmetik durchgeführt. Als Eingabedaten kommen auf pc-x86 ein 620 KB großes TAR-Archiv zum Einsatz, auf es-xscale aufgrund der geringeren Rechenleistung ein 90 KB großes TAR-Archiv. Beide Archive enthalten Java-Quelltexte und -Klassendateien.

**scimark.fft** In dieser Arbeitslast wird eine schnelle Fouriertransformation durchgeführt und anschließend ihr Inverses berechnet. Diese Berechnungen werden mit Fliesskommaarithmetik durchgeführt. Zum Einsatz kommt der kleine Eingabedatensatz mit drei Wiederholungen auf pc-x86 und mit nur einer Wiederholung auf es-xscale.

**compiler.compiler** Die Arbeitslast besteht aus einer Vorversion des Sun Java Übersetzers javac 1.7, welche Java-Quelltexte aus einem ZIP-Archiv entpackt und anschließend übersetzt. Als Eingabearchive dienen auf pc-x86 19 und auf es-xscale drei Java-Beispielprogramme.

Innerhalb der compiler-Arbeitslast werden ein große Anzahl verschiedener Methoden ausgeführt. Bedingt ist diese Tatsache unter anderem durch die intensive Nutzung des Besucher-Musters im Übersetzer, welche viele Methoden einführt.

## 5.3 Versuchsaufbau

Wie eingangs erwähnt ist der Haupteinsatzzweck für den Laufzeitübersetzer, Programme zu beschleunigen, die sich nicht mit dem statischen Übersetzer der JamaicaVM übersetzen lassen, weil sie vorab nicht vollständig bekannt sind. Wir simulieren diese Situation, indem wir das Knopflerfish-OSGi-Rahmenwerk einsetzen.[1] Es ermöglicht das einfache Laden von neuen Programmkomponenten zur Laufzeit. Die Arbeitslasten werden in solche Komponenten verpackt. Zusätzlich zu den Arbeitslastkomponenten haben wir eine Komponente erstellt, die die Beispielprogramme startet und ihre Ausführungszeit misst. Zudem kann sie das aktuelle Profil vom Laufzeitprofilersteller abrufen. Wir protokollieren während des Laufs die ermittelten Daten.

### 5.3.1 Messungen

Wir setzen eine JamaicaVM in der Version 3.4 ein, bei der die Pakete des Laufzeitübersetzers, der hier entworfenen Steuerung und der wichtigsten Datenstrukturen statisch nativ übersetzt wurden. Der verbleibende Teil der Java-Klassenbibliothek liegt nur in Bytecode vor, ebenso wie die Arbeitslasten. Der Haldenspeicher, der für Java-Objekte zur Verfügung steht, ist auf 64 MByte begrenzt.

Zuallererst messen wir die Zeit, die das Knopflerfish-Rahmenwerk zum Initialisieren braucht. Das aktuelle Profil wird über eine Schnittstelle aus den Datenbeständen der Laufzeitübersetzersteuerung ausgelesen. Dann beginnt die eigentliche Messung mit der ausgewählten Arbeitslast. Pro Messung wird diese dreimal hintereinander ausgeführt. Nach jeder Iteration wird wiederum das aktuelle Profil ausgelesen.

Mit diesen Messungen kann die Leistung der Laufzeitübersetzersteuerung gut beurteilt werden. In der Initialisierungsphase sollten möglichst keine Methoden übersetzt werden, weil die Rahmenwerkinitialisierung nur ein einziges Mal ausgeführt wird. In der ersten Iteration sollte die Steuerung versuchen, genug, aber nicht zu viele Methoden zu übersetzen, um nicht eine Laufzeitverschlechterung herbeizuführen. Spätestens nach der zweiten Iteration sollte ein Großteil der wichtigen Methoden erkannt sein, so dass in der dritten

Iteration eine Laufzeit nahe der Referenzzeit erreicht wird. Die Betrachtung der Gesamtlaufzeit für die Initialisierung und der drei Arbeitslastiterationen zeigt, ob sich der Einsatz des Übersetzers auch im Mittel gelohnt hat.

### 5.3.2 Gesammelte Daten

Der hier vorgestellte Laufzeitprofilersteller erfasst zusätzlich zu den für die Profilerstellung benötigten Informationen noch weitere Daten, die für die spätere Analyse benötigt werden. Insbesondere werden bereits übersetzte Methoden weiterhin als Kandidaten behandelt, um Aussagen über die Qualität des erstellten Profils machen zu können. Für jeden Kandidaten wird bei seinem ersten Stichprobenvorkommen ein Zeitstempel gespeichert. Weitere Zeitstempel werden gesetzt, wenn der Kandidat als wichtig klassifiziert wird ($t_{hot}$) und wenn er übersetzt wird ($t_{compiled}$). Der Status eines Kandidaten kann sich nur zu den Zeitpunkten der Stichprobe ändern, daher wählen wir als Einheit der Zeitstempel die Anzahl der bisher durchgeführten Stichproben.

Jeder Kandidat hat zudem einen Zähler $p$, der die Vorkommen des Kandidaten in der Stichprobe speichert. Der Stand dieses Zählers wird für die beiden Statusänderungen gespeichert ($p_{hot}$und $p_{compiled}$).

## 5.4 Analyse

### 5.4.1 Was macht eine gute Laufzeitübersetzersteuerung aus?

Die Qualität eines Laufzeitprofilerstellers kann an den folgenden drei Metriken festgemacht werden. [10]

- *Genauigkeit*: Es sollen möglichst alle wichtigen Methoden als solche erkannt und keine unwichtigen Methoden fälschlicherweise als wichtig klassifiziert werden. Dadurch soll sich der Mehraufwand schnell und auch bei kurz-laufenden Programmen amortisieren.
- *Erkennungs*zeit: Wichtige Methoden sollen so bald wie möglich erkannt und übersetzt werden. Dadurch wird das Laufzeitverbesserungspotential durch das Übersetzen voll ausgeschöpft.
- *Erkennungsmehraufwand*: Der Laufzeitprofilersteller soll möglich wenig Mehraufwand verursachen. Die Kosten für die Laufzeitprofilerstellung sollen nicht den Nutzen der Übersetzung überwiegen.

Für den praktischen Einsatz ist natürlich die Laufzeitverbesserung durch den Einsatz des Laufzeitübersetzers am interessantesten.

### 5.4.2 Referenzprofil

Neben quantitativen Maßzahlen, die sich beispielsweise gut für die Betrachtung der Laufzeitverbesserung eignen, wollen wir auch den Verlauf der erzeugten Profile betracht-

en. Diese sind inhärent ungenau, sollten aber trotzdem grob das Ausführungsverhalten des Programms widerspiegeln.

Als Referenzprofil dient ein vom statischen Profilersteller der JamaicaVM erzeugtes Laufzeitprofil. Dieses Profil nutzt als Maß die Anzahl der interpretierten Bytecode-Instruktionen, was das Analogon zum Prozessorzyklenzähler eines konkreten Prozessors ist. Auch wenn die Interpretierungszeit der verschiedenen Bytecode-Instruktionen variiert[1], ist das statisch erzeugte Profil hier als Vergleichsprofil gut geeignet. Aufgrund der großen Anzahl an interpretierten Instruktionen fallen die individuellen Unterschiede in der benötigten Zeit kaum ins Gewicht. Zudem hat sich diese Art der Profilerstellung im praktischen Einsatz bewährt.

Abbildung 5 zeigt die Referenzprofile der betrachteten Arbeitslasten. Man sieht, dass compress und scimark bergige Profile besitzen, und compiler ein Beispiel für ein flaches Laufzeitprofil ist. Aus dem Verlauf der Profile leiten wir ab, dass Methoden, die für mehr als $0,1\%$ der interpretierten Bytecode-Instruktionen verantwortlich waren, als wichtig gelten sollen.

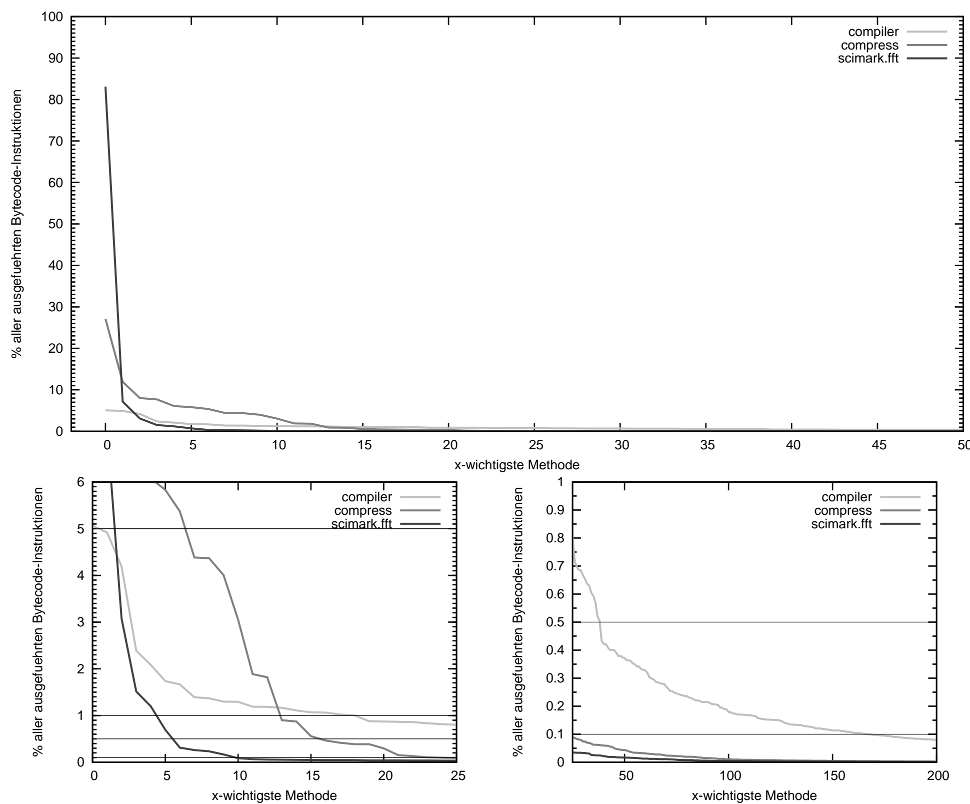



**Abbildung 5:** Referenzprofile der SPECjvm2008-Arbeitslasten

[1] beispielsweise dauert ein Methodenaufruf länger als eine arithmetische Operation

### 5.4.3 Auswertung der Messprotokolle

Aus den während der Messungen erzeugten Protokollen kann man die oben genannten Eigenschaften noch nicht direkt ablesen. Deshalb haben wir ein Analysewerkzeug entwickelt, das auf den Protokollen einfache statistische Auswertungen durchführt. Es werden immer mehrere Protokolle zu einer Laufzeitprofilerstellerkonfiguration erstellt und deren Daten gemittelt, um Ungenauigkeiten in den Messungen zu erkennen und auszugleichen.

**Vorverarbeitung** Direkt aus dem Messprotokoll lassen sich die Laufzeit der Initialisierungsphase $T_{startup}$ und die Laufzeiten der Arbeitslast-Iterationen $T_1, T_2, T_3$ auslesen und die Gesamtlaufzeit $T_{all}$ als Summe dieser Zeiten berechnen.

Für jeden Kandidaten wird der prozentuale Anteil seiner Bewertungspunkte bezüglich aller im Profil vergebenen Bewertungspunkte berechnet als $\beta_{rel} = \frac{\beta}{\beta_{total}}$. So kann man verschiedene Laufzeitprofile besser miteinander vergleichen.

Durch einfaches Zählen bestimmen wir die Anzahl der als wichtig klassifizierten Methoden $n_{hot}$ und der übersetzten Methoden $n_{compiled}$. Der Vergleich der Menge der wichtigen Kandidaten mit der Menge der wichtigen Methoden aus dem Referenzprofil liefert die Anzahl der Übereinstimmungen $n_{ref}$.

Die verstrichene Zeit $t_{queue}$ und Stichprobenvorkommen $p_{queue}$ während sich der Kandidat in der Übersetzungswarteschlange befindet, berechnen sich als $t_{queue} = t_{compiled} - t_{hot}$ und $p_{queue} = p_{compiled} - p_{hot}$. Sollten diese Werte groß sein, bedeutet dies, dass wichtige Kandidaten nicht schnell genug und somit zu spät übersetzt werden.

Interessant ist auch die Anzahl der Stichprobenvorkommen nach der Übersetzung $p_{after}$, die zur Definition des *Nutzungsverhältnisses* $\eta$ benötigt wird. Es beschreibt, wie verhältnismäßig häufig der Kandidat nach dem Übersetzen in der Stichprobe war. Es sei $\eta = \frac{p_{after}}{p}$.

Für die genannten Kennzahlen werden für jede Iteration die minimalen, maximalen und durchschnittlichen Werte berechnet.

**Genauigkeit** Der Indikator für ein genaues Laufzeitprofil ist ein großer Wert $n_{ref}$, idealerweise ist gleichzeitig $n_{compiled} \approx n_{ref}$, d.h. der Laufzeitprofilersteller hat die selben Methoden als wichtig klassifiziert, die auch der statische Profilersteller als wichtig identifiziert hat.

**Erkennungszeit** Die Erkennungszeit lässt sich am durchschnittlichen Wert von $p_{hot}$ festmachen. Ist der Wert groß, spricht dies für eine schlechte Erkennungszeit. Im Gegenzug ist $\varnothing p_{hot} \approx 1$ ideal, es besteht aber die Gefahr, dass die Übersetzersteuerung zu unselektiv gearbeitet hat, nämlich dann, wenn $n_{compiled} \gg n_{ref}$ ist.

Das durchschnittliche Nutzungsverhältnis sollte idealerweise nahe 1 sein, was bedeutet, dass viele Kandidaten korrekt als wichtig und früh- und damit rechtzeitig erkannt wurden. Das Nutzungsverhältnis ist also gleichermaßen ein Kriterium für die Genauigkeit wie für die Erkennungszeit.

**Erkennungsmehraufwand** Den Erkennungsmehraufwand kann man nur indirekt aus dem Vergleich zweier Messungen errechnen. Wir führen eine Messung ohne Laufzeitübersetzersteuerung ("no-jit") und eine Messung mit der gewählten Profilerstellerkonfiguration, aber ohne Übersetzer durch ("dont-compile"). Der Mehraufwand ergibt sich dann durch die Differenzen $\Delta T_X = T_{X_{dont-compile}} - T_{X_{no-jit}},\ X = startup$, 1, 2, 3, *all*.

**Laufzeitverbesserung** Analog zum Erkennungsmehraufwand vergleichen wir zur Bestimmung der Laufzeitverbesserung eine Messung der zu untersuchenden Konfiguration ("jit") mit einer Messung ohne Laufzeitübersetzersteuerung ("no-jit"). $\Delta T_X = T_{X_{jit}} - T_{X_{no-jit}},\ X = startup$, 1, 2, 3, *all*. Die Laufzeitverbesserung ist idealerweise immer negativ.

# 6 Feinabstimmung der Parameter

In diesem Kapitel stellen wir die Messungen vor, die wir auf der Suche nach praktikablen Parametern für den Laufzeitprofilersteller durchgeführt haben.

## 6.1 Erste Evaluierung

Die erste Evaluierung hat zum Ziel, die Grenzen des eingesetzten Laufzeitübersetzers zu ermitteln und brauchbare Startparameter für die Bewertungsfunktionen zu finden.

### 6.1.1 Grenzen

Zur Bestimmung des Bereichs, in dem die Laufzeiten der Arbeitslasten zu erwarten sind, wird eine Messung ohne Laufzeitübersetzersteuerung mit einer Messung, bei der die vom statischen Profilersteller als wichtig erachteten Methoden vorab vom Laufzeitübersetzer übersetzt wurden, verglichen. Die Ergebnisse zeigt Tabelle 3.

| Arbeitslast: | | compress | scimark.fft | compiler |
|---|---|---|---|---|
| wichtige / alle Methoden | | 24 / 2019 | 10 / 2008 | 168 / 4560 |
| pc-x86 | ohne | 11,1 s | 12,9 s | 12,3 s |
| | vorab | 0,9 s | 1,1 s | 5,2 s |
| | $\sum t_{compile}$ | 1,5 s | 0,9 s | 21,4 s |
| es-xscale | ohne | 19,7 s | 46,8 s | 47,9 s |
| | vorab | 1,9 s | 8,8 s | 23,8 s |
| | $\sum t_{compile}$ | 28,0 s | 18,6 s | 396,7 s |

**Tabelle 3:** Laufzeiten der Arbeitslasten mit und ohne Laufzeitübersetzer

### 6.1.2 Messung auf pc-x86

Nun vergleichen wir alle Bewertungsfunktionen separat. Wir wollen einen Überblick erlangen und überprüfen, welche Auswirkungen die Stichprobenfrequenz und die Wahl der Bewertungsfunktion auf den Mehraufwand hat, welche Bewertungsfunktionen gute Ergebnisse liefern und wie praxisnah die im vorherigen Abschnitt diskutierten Metriken sind.

**Konfiguration** Die Stichprobenfrequenz wird auf pc-x86 von 10 Hz bis 200 Hz variiert.[2] Als Richtwert dient die Frequenz der Zeitgebers auf dem verwendeten Linux-System, die 100 Hz beträgt.[3] Zum Einsatz kommt die Übersetzersteuerung mit festem Schwellwert. Der Schwellwert ist $\tau = 1000$. Dieser Wert ist willkürlich gewählt und dient als

[2] mit den Zwischenschritten 20, 50, 100 Hz

[3] JamaicaVM enthält einen eigenen Planer für die Ausführungsstränge (engl. “Scheduler”), der die Stränge auch mit kürzeren Intervallen als das Betriebssystem umschalten kann.

Bezugspunkt für die anderen Parameter. Die Alterung ist abgeschaltet. Für die Bewertungsfunktionen werden folgende Gewichtungsfaktoren verwendet:

- Konstante Bewertungsfunktion: $\omega_C = 100$, so dass Kandidaten nach dem zehnten Stichprobenvorkommen übersetzt werden.
- Methodengröße: $\omega_B = 5$. Eine Analyse der Java-Klassenbibliothek, als Repräsentant für typische Java-Programme, hat ergeben, dass 50% der Methoden kleiner als 18 Byte und 95% der Methoden kleiner als 220 Byte sind. Mit dieser Gewichtung werden Methoden über 200 Byte sofort übersetzt, ein Großteil der Methoden nach etwa dem zehnten Stichprobenvorkommen.
- Schleifenerkennung. $\omega_{SA} = 10$. Für Schleifen wird angenommen, dass sie 10 mal durchlaufen werden. Methoden, die eine Schleife mit mindestens 10 Bytecodes oder eine geschachtelte Schleife enthalten, werden sofort übersetzt.
- Aufrufkellerinspektion: $\omega_{SW} = 100$. Der Bonus zur Vermeidung von Kontextwechseln ist auf 10 festgelegt. Im Falle eines nativen Aufrufers wird die Methode sofort übersetzt. Die Vorkommen auf dem Aufrufkeller anderer Methoden werden analog der konstanten Bewertungsfunktion einfach gezählt. Eine Methode, die 10 mal als Aufrufer erkannt wurde, wird also übersetzt. Die Tiefe der Inspektion auf dem Keller ist 2, wir betrachteten also die verdrängte Methode und ihren Aufrufer.
- Akkumulierende Bewertungsfunktion: $\omega_A = 100$. Wenn die akkumulierende Bewertungsfunktion alleine und mit festem Schwellwert eingesetzt wird, verhält sie sich wie die konstante Bewertungsfunktion. Allerdings werden Kandidaten schon nach fünf Stichprobenvorkommen übersetzt.

Zu Vergleichszwecken wird zu jeder der oben genannten Konfigurationen noch eine Messung durchgeführt, bei der keine Methode übersetzt wird ("dont-compile"). Auf pc-x86 wird zudem eine unbedingte Übersetzung beim ersten Stichprobenvorkommen ausprobiert ("compile-imm"). Die Messung ohne Laufzeitübersetzersteuerung ist mit "no-jit" gekennzeichnet.

**Auswirkung der Stichprobenfrequenz** Zuallererst betrachten wir in den Abbildungen 6 bis 8 die Auswirkung der Laufzeitprofilerstellung ohne den Einsatz des Übersetzers.

Im Vergleich mit einer Messung ganz ohne die Laufzeitübersetzersteuerung wird der Mehraufwand für die Laufzeitprofilerstellung, in Abhängigkeit von der Stichprobenfrequenz, sichtbar. Die Laufzeiten der Arbeitslasten sind fast konstant, die Abweichungen führen wir auf die Messungenauigkeit zurück. Auf pc-x86 kann man die Stichprobenfrequenz so wählen, wie sie hilfreich für die anderen Gütekriterien ist.

Ansonsten gilt: Eine hohe Stichprobenfrequenz sorgt dafür, dass mehr Methoden übersetzt werden. Dafür gibt es zwei Gründe. Zum einen erhöht eine hohe Stichprobenfrequenz die Wahrscheinlichkeit, dass eine bestimmte Methode zum Zeitpunkt der Stichprobennahme gerade interpretiert wird. [14] Man findet also insbesondere auch mehr

verschiedene Methoden. Zum anderen gibt es eine größere Anzahl Stichproben, so dass auch eine größere Menge an Bewertungspunkten vergeben wird. Zusammen mit dem festen Schwellwert führt das dazu, dass mehr Methoden den Schwellwert überschreiten.

Aus den gleichen Gründen führt eine hohe Stichprobenfrequenz indirekt auch zu einer Laufzeitverbesserung, weil durch sie die Erkennung der wichtigen Methoden früher geschieht und schneller konvergiert, d.h. idealerweise in der zweiten Iteration der Arbeitslast keine neuen Methoden übersetzt werden müssen.

**compress** (Abb. 6) Bei der compress-Arbeitslast bringt der Einsatz des Laufzeitübersetzers auf pc-x86 in allen Messungen eine Laufzeitverbesserung. Die Bewertungsfunktionen A und SW sorgen für gute Laufzeiten in den einzelnen Iterationen und auch in der Gesamtlaufzeit. Je geringer die Stichprobenfrequenz ist, desto besser wird SW. SA liefert die schlechtesten Ergebnisse, da sie einige Methoden, die zwar sehr häufig aufgerufen werden, aber keine Schleifen enthalten, nicht als wichtig klassifiziert. Die unbedingte Übersetzung ist in der ersten Iteration und bei langsamer Stichprobenfrequenz schneller als die Bewertungsfunktionen, zeigt aber das schlechteste Nutzungsverhältnis. Man kann häufig gute Laufzeiten in den Iterationen beobachten, wenn ein gutes Nutzungsverhältnis und eine große Übereinstimmung mit dem Referenzprofil vorliegen. Die Gesamtlaufzeit ist dann kurz, wenn nicht wesentlich mehr andere Methoden übersetzt werden.

**scimark.fft** (Abb. 7) Auch bei der scimark.fft-Arbeitslast schadet die Laufzeitübersetzung nicht auf pc-x86. Bei hoher Stichprobenfrequenz liefern A, C und SW die besseren Gesamtlaufzeiten, sonst ist die unbedingte Übersetzung die beste Konfiguration. In den einzelnen Iterationen gibt es keine klaren Favoriten, und auch die Zusammenhänge der Laufzeit mit dem Nutzungsverhältnis und der Genauigkeit sind weniger signifikant als bei compress. Die Arbeitslast besteht aus nur wenigen Methoden, so dass die Gefahr, zu viel zu übersetzen, gering ist. Die Bewertungsfunktion sollte jedoch alle Methoden, die der statische Profilersteller identifiziert, auch erkennen, sonst leidet die Laufzeit in den Iterationen.

**compiler** (Abb. 8) Bei dieser Arbeitslast mit ihrem flachen Laufzeitprofil schadet der Laufzeitübersetzer selbst auf pc-x86. Die Konfigurationen, die viele Methoden übersetzen, schaffen in der dritten Iteration Laufzeiten, die etwas besser als die Interpretierungszeit sind. Die Kosten für das Übersetzen wirken sich aber dann negativ auf die Gesamtlaufzeit aus. Gut in der Gesamtlaufzeit sind hingegen die Konfigurationen, die kaum übersetzen - im Verhältnis zur Interpretierungszeit in einer Iteration ist der Laufzeitgewinn jedoch gering. Davon abgesehen liefern A und SW unter den Bewertungsfunktionen die beste Laufzeit. Die Bewertungsfunktion SW zeigt gute Übereinstimmungen mit dem statischen Laufzeitprofil, B ist noch etwas besser, wählt aber auch die meisten Methoden im Vergleich mit den anderen Bewertungsfunktionen aus. Die unbedingte Übersetzung erreicht auch in der dritten Iteration, in der über 900 Methoden übersetzt wurden, nicht die Referenzzeit mit vorab übersetzten Methoden. Das Nutzungsverhältnis zeigt keine deutliche Korrelation zur Laufzeit, so tritt das beste Nutzungsverhältnis beispielsweise zusammen

mit der schlechtesten Laufzeit der dritten Iteration auf.

**Gemeinsamkeiten bei den Arbeitslasten** Während der Initialisierungsphase des Knopflerfish-Rahmenwerks sind möglichst keine Methoden zu übersetzen, da diese später sehr wahrscheinlich nicht mehr gebraucht werden. Dies spiegelt sich auch bei den Messungen wieder - die beste Laufzeit für die Initialisierungsphase wird immer von einer Konfiguration erreicht, die in dieser kurzen Phase keine Zeit für das Übersetzen von Methoden verschwendet.

Auch ohne den Einsatz des Laufzeitübersetzers kommt es zu einer Laufzeitverbesserung nach der ersten Iteration. Dies kann man darauf zurückführen, dass zu diesem Zeitpunkt alle benötigten Klassen schon geladen sind.

Es ist zu beobachten, dass bei geringer Stichprobenfrequenz die Laufzeiten weniger nah beieinander liegen, als bei hoher Frequenz. Auf pc-x86 liefern die Stichprobenfrequenzen 50 Hz und 100 Hz im Durchschnitt die besten Gesamtlaufzeiten.

**Qualitative Analyse** Nach der Analyse der Kennzahlen der Messungen betrachten wir nun den Verlauf der erstellten Laufzeitprofile im Vergleich mit den Referenzprofilen.

Bemerkenswert ist, dass die beste Näherung von der Bewertungsfunktion C erreicht wird. Bei niedriger Stichprobenfrequenz wird deren Profil zwar treppenartig, d.h. es klassifiziert viele Methoden gleich, der Verlauf folgt aber immer dem Referenzprofil.

Bewertungsfunktion B erreicht bei der compiler-Arbeitslast eine gute Näherung über alle Frequenzen, bewertet aber bei compress und scimark die wichtigsten Methoden zu stark und das Mittelfeld bezüglich der Wichtigkeit zu gering. Dieses Verhalten ist generell auch bei A, SW und verstärkt bei SA und bei allen Arbeitslasten zu beobachten. A profitiert von hoher Stichprobenfrequenz.

**Fazit** Die Messungen haben gezeigt, dass die Laufzeitübersetzersteuerung gut für die Arbeitslasten mit bergigem Profil funktioniert. Die Gefahr, zu viel zu übersetzen ist gering und der Gewinn durch die wichtigen Methoden ist groß. Für Programme, die ein flaches Profil besitzen, ist die Steuerung noch zu unselektiv. In Tabelle 3 sieht man, dass die Übersetzungszeiten für die Methoden der compiler-Arbeitslast nicht in einer Iteration durch die erzielbare Laufzeitverbesserung amortisiert werden können. Die Laufzeitübersetzersteuerung sollte die Übersetzung also über mehrere Iterationen verteilen.

In den Messungen treten wie erwartet gute Genauigkeiten häufiger zusammen mit guten Laufzeiten auf. Eine Äquivalenz lässt sich aber nicht ableiten. Auf pc-x86 können auch eine größere Anzahl von Methoden ausreichend schnell übersetzt werden, so dass eine einseitige Ungenauigkeit[4] nicht weiter schadet. Anderseits ist auch das alleinige Übersetzen vieler Methoden keine Garantie für eine gute Laufzeit. Das Nutzungsverhältnis erlaubt keine Rückschlüsse auf die zu erwartende Laufzeit.

Die Bewertungsfunktionen A und SW haben in einer Vielzahl von Messungen gute Ergebnisse geliefert. Die Schleifenerkennung ist alleine eingesetzt nicht praktikabel, und

[4] D.h. viele Übereinstimmungen, aber auch viele unwichtige Methoden wurden erkannt.

auch ihre Gewichtung war zu hoch gewählt. Als Ergänzung zu anderen Bewertungsfunktionen ist sie allerdings denkbar.

Die Qualität der Bewertungsfunktion wird wichtiger, je geringer die Stichprobenfrequenz ist, weil bei vielen Stichproben Ungenauigkeiten in der Bewertung durch die große Anzahl von Stichprobennahmen ausgeglichen wird. Für die nachfolgenden Messungen wählen wir 50 Hz als Stichprobenfrequenz, weil sie ähnlich gut wie 100 Hz ist, aber prinzipiell weniger Mehraufwand verursacht.[5]

### 6.1.3 Übertragung der Messung auf es-xscale

Das XScale-System ist grob geschätzt um einen Faktor 10 langsamer als pc-x86.[6] Daher sollten sich vergleichbare Laufzeitprofile zur Messung mit 100 Hz auf pc-x86 ergeben, wenn als Stichprobenfrequenz 10 Hz eingestellt ist. Zum Vergleich testen wir außerdem 20 Hz und 5 Hz. Bei den Bewertungsfunktionen beschränken wir uns auf C und SW, die sich auf pc-x86 bewährt haben. Wir verzichten auf A, da A und C bei festem Schwellwert gleichermaßen Kandidaten nach einer konstanten Anzahl von Stichprobenvorkommen als wichtig markieren. Da die Konfiguration von C, nach zehn Stichprobenvorkommen zu übersetzen, zu hoch erscheint, messen wir nun mit $\omega_C = 200$. Für SW ist die Konfiguration weiterhin $\omega_{SW} = 100$, ebenso wieder der Schwellwert $\tau = 1000$.

**Stichprobenfrequenz** Im Gegensatz zu pc-x86 ist der Mehraufwand durch die Stichprobennahme auf es-xscale in den Abbildungen 9 bis 11 deutlich sichtbar. Bei 20 Hz beträgt er bei allen Arbeitslasten etwa 40%, bei 10 Hz rund 20 % und bei 5 Hz immerhin noch etwa 10% beim Einsatz von SW. Der Mehraufwand bei Einsatz von C ist etwas geringer.

Wir betrachten die Anzahl der erfassten Methoden und versuchen, eine Korrespondenz zu den Messungen auf pc-x86 herzustellen. Es stellt sich aber heraus, dass bei compress und scimark.fft auf es-xscale mehr Methoden gefunden werden als auf pc-x86 mit der zehnfachen Stichprobenfrequenz. Bei der compiler-Arbeitslast ist das Verhältnis umgekehrt. Ein Zusammenhang lässt sich nicht ableiten. Wir führen dies darauf zurück, dass sich die Eingabedaten der Arbeitslasten auf den Geräten unterscheiden und auch die reinen Interpretierungszeiten der Arbeitslasten auf es-xscale mehr variieren als auf pc-x86.

**compress** (Abb. 9) Nur bei 5 Hz schaffen C und SW eine Verbesserung der Gesamtlaufzeit. SW ist minimal besser als C und übersetzt 23 Methoden, von denen 17 in Übereinstimmung mit dem Referenzprofil sind. Bei schnellerer Stichprobennahme werden von beiden getesteten Bewertungsfunktionen zu viele, nicht benötigte Methoden ausgewählt.

[5] Auch wenn der gemessene Mehraufwand im Verhältnis zur Laufzeit der Messungen nicht ins Gewicht fällt, ist nicht auszuschließen, dass beispielsweise interaktive Programme von einer geringeren Anzahl an Unterbrechungen profitieren.

[6] vgl. Tabelle 3

**scimark.fft** (Abb. 10) Im Gegensatz zu compress führt die Laufzeitübersetzung hier in allen Konfigurationen zu einer Laufzeitverbesserung. Am besten schneiden die Bewertungsfunktionen wieder bei 5 Hz Stichprobenfrequenz ab. Bei SW ist allerdings der Unterschied zwischen 10 Hz und 5 Hz deutlicher zu sehen als bei C, weil SW bei 10 Hz viermal so viele Methoden übersetzt, aber nur eine weitere Übereinstimmung mit dem Referenzprofil erreicht wird.

**compiler** (Abb. 11) Bei der compiler-Arbeitslast zeigt die Laufzeitübersetzung weder in den Iterationen noch in der Gesamtlaufzeit einen Nutzen. Werden genug Methoden übersetzt, um die Arbeitslast zu beschleunigen, dominieren die Übersetzungskosten bei weitem die mögliche Laufzeitverbesserung um den Faktor 2. Andererseits lässt sich compiler durch das Übersetzten weniger Methoden nicht wesentlich beschleunigen.

**Fazit** Auf dem eingebetteten System ist die Genauigkeit der Laufzeitübersetzersteuerung noch wichtiger als auf pc-x86. Ebenso verursacht die Stichprobennahme einen signifikanten Mehraufwand, der sich bei compress kaum und bei compiler gar nicht amortisieren ließ. Die Erkennungszeit ist ähnlich ausgeglichen wie auf pc-x86, tritt von der Bedeutung her gegenüber den anderen Metriken in den Hintergrund.

## 6.2 Alterung

Wir haben beobachtet, dass die Erkennungsleistung der Bewertungsfunktionen in den meisten Fällen zu einseitig war. Es wurden ausreichend wichtige Methoden gefunden, gleichzeitig sind aber auch viele unwichtige Methoden übersetzt worden. pc-x86 konnte dies aufgrund seiner Rechenleistung gut kompensieren, auf es-xscale waren die negativen Auswirkungen jedoch deutlich messbar.

Um weniger Methoden zu übersetzen, könnte man einen höheren Schwellwert wählen. Allerdings verschlechtert sich dann die Erkennungszeit für die tatsächlich wichtigen Methoden in gleichem Maße. Ein weiteres Problem offenbart sich bei lange laufenden Programmen. Je mehr Stichprobennahmen insgesamt durchgeführt wurden, desto höher steigt die Wahrscheinlichkeit, dass auch unwichtige Methoden den Schwellwert erreichen.

Die Alterung der Bewertungspunkte bringt eine zeitliche Komponente in das erstellte Laufzeitprofil, so dass neuere Bewertungen eine größere Bedeutung haben als ältere. Durch den Einsatz der Alterung kann es zwar länger dauern bis die maximale Laufzeitverbesserung erreicht wird, die Gefahr, dass der Laufzeitübersetzereinsatz bis zu diesem Zeitpunkt schadet, ist aber geringer.

### 6.2.1 Konfigurationen

Wir evaluieren, ob die Alterung der Bewertungspunkte zu einer Verbesserung der Übersetzersteuerung führt, indem wir folgende Konfigurationen betrachten.

- Häufige, aber schwache Alterung. Periodisch mit $\frac{1}{10}$ der Stichprobenfrequenz werden die Kandidaten auf 90 % ihrer Bewertungspunkte reduziert. So wird eine kontinuierliche Alterung erreicht.

- Langsame, aber starke Alterung. Die Bewertungspunkte der Kandidaten werden nach 50 Stichprobennahmen halbiert. Methoden, die schnell den Schwellwert erreichen, werden bevorzugt.
- Intervallbildung. Nach 50 Stichprobennahmen werden die Bewertungspunkte aller Kandidaten gelöscht, so dass nur Kandidaten übersetzt werden, die in diesen diskreten Zeitabschnitten den Schwellwert erreichen.

Wir führen die Messungen mit einer Stichprobenfrequenz von 50 Hz auf pc-x86 und 5 Hz auf es-xscale durch. Bezüglich der Profilersteller-Konfiguration wählen wir die Favoriten aus der quantitativen und der qualitativen Analyse. Konkret vergleichen wir also $50 \cdot A + 50 \cdot SW$ mit $200 \cdot C$. Der Schwellwert ist wie bisher $\tau = 1000$.

### 6.2.2 Auswertung

**pc-x86** In Abbildung 13 sieht man, dass die Alterung die Anzahl der übersetzten Methoden effektiv beschränkt hat. Die Auswertung ergibt auch, dass die Erkennungszeit nicht beeinträchtigt wurde. Abbildung 12 zeigt, dass der Laufzeitübersetzer in keiner Messung geschadet hat. Die langsame, aber starke Alterung schneidet im Durchschnitt am besten ab, die Intervallbildung ist etwas schlechter als die anderen Ansätze. Bei compress und scimark ist die Kombination von A und SW besser, bei compiler liefert C die beste Laufzeit. Positiv fällt auf, dass bei den Arbeitslasten mit bergigen Profil bereits nach Iteration 1 eine gute Genauigkeit erreicht wurde und in den nachfolgenden Iteration kaum noch weitere Methoden übersetzt wurden, was zu sehr guten Iterationslaufzeiten führt.

**es-xscale** Auf dem eingebetteten System zeigen sich für compress und scimark die gleichen positiven Resultate wie auf pc-x86. Der Laufzeitübersetzer schafft bei der compiler-Arbeitslast zwar immer noch keine Laufzeitverbesserung, der Schaden ist aber im Gegensatz zu den Messungen ohne Alterung auf circa 25 % begrenzt. Abbildung 14 zeigt die Messergebnisse, die Daten zur Betrachtung der Genauigkeit sind in Abbildung 16 dargestellt.

In Abbildung 15 betrachten wir nun den durch die Alterung verursachten Mehraufwand. Bei compress und scimark liegt er bei etwa 10 %, er fällt hierbei also gegenüber der Stichprobennahme nicht ins Gewicht. Bei der compiler-Arbeitslast können wir einen kleinen Mehraufwand beobachten, der durch die Iteration über eine größere Anzahl verwalteter Kandidaten verursacht wird.

## 6.3 Kombination von Bewertungsfunktionen

Durch die Alterung wurde die Anzahl der übersetzten Methode so beschränkt, dass fast ausschließlich Kandidaten in Übereinstimmung mit dem Referenzprofil übersetzt wurden. Allerdings sind insgesamt nur etwas über 60 % der Methoden aus dem Referenzprofil bei compress und scimark als wichtig klassifiziert worden. Bei compiler beträgt der Anteil sogar durchschnittlich nur 10 %. Wir fragen uns, ob wir die Genauigkeit und dadurch

die Laufzeit weiter verbessern können.

Dazu führen wir Messungen durch, die auf den Ergebnissen der vorherigen Untersuchungen aufbauen, und zusätzlich eine Kombination von mehreren Bewertungsfunktionen einsetzen. Während der qualitativen Analyse der ersten Messreihen haben wir gesehen, dass die nicht-konstanten Bewertungsfunktionen die Bewertungen der wichtigsten Methoden verstärken und weniger wichtige Methoden abgeschwächt werden. Wir erhoffen uns eine bessere Genauigkeit durch die gezielte Verfälschung des Laufzeitprofils, durch die auch flache Profile einigermaßen bergig gemacht werden.

### 6.3.1 Konfigurationen

Da die zu erwartende Menge an vergebenen Bewertungspunkten schwer abzuschätzen ist, setzen wir die asap-p-Übersetzungsstrategie ein und untersuchen variable Schwellwerte mit 0,1 % bis 10 % der Bewertungspunkte. Der Schwellwert wird nach jeweils 100 Stichprobennahmen aktualisiert. Die Stichprobenfrequenz beträgt wie bisher 50 Hz und die Bewertungen der Kandidaten werden alle 50 Stichprobennahmen halbiert.

Die Laufzeitprofilerstellung konfigurieren wir folgt:

- $100 \cdot SW + 100 \cdot A =: Base$: Die Bewertungsfunktionen SW und A mit gleichen Gewichtungsfaktoren haben sich in den vorhergegangen Messungen bewährt. Diese Kombination ist auch Bestandteil der weiteren Konfigurationen.
- $Base + 5 \cdot B$: Die Bewertungsfunktion B war separat betrachtet nur mittelmäßig. Wir betrachten sie hier, weil das von ihr erstellte Laufzeitprofil im Vergleich zu C nur leicht verstärkt ist.
- $Base + 3 \cdot SA$: Die Schleifenerkennung war in den Einzelmessungen unbrauchbar, da sie Methoden mit Schleifen extrem über- und schleifenlose Methoden unterbewertet hat. Diese Eigenschaft wollen wir uns nun als Erweiterung der anderen Bewertungsfunktionen zu Nutze machen. Der Gewichtungsfaktor wird gegenüber der ursprünglichen Messung reduziert, weil er sich als zu hoch herausgestellt hat.
- $Base + 5 \cdot B + 3 \cdot SA$: Da die von B und SA bewerteten Aspekte weitestgehend orthogonal sind, betrachten wir sie in dieser Konfiguration zusammen.
- $100 \cdot C$: Diese Konfiguration repräsentiert eine neutrales Laufzeitprofil.

### 6.3.2 Auswertung

Wir betrachten die Laufzeiten der Messungen in den Abbildungen 17, 19 und 21 und stellen fest, dass sich keine Verbesserung gegenüber den vorangegangenen Messungen eingestellt hat.

Die Kennzahlen für die Genauigkeit lesen wir in der Abbildungen 18, 20 und 22 ab. Dabei ist bemerkenswert, dass die konstante Bewertungsfunktion in allen drei Arbeitslasten die meisten Methoden in Übereinstimmung mit dem Referenzprofil erkennt, aber

auch die meisten unnötigen Methoden übersetzt. Diese Bewertungsfunktion arbeitet offensichtlich nicht gut mit dem variablen Schwellwert zusammen.

Die Konfiguration Base liefert die genausten Ergebnisse. Bei compress und scimark.fft schneiden auch die Konfigurationen mit Beteiligung der Bewertungsfunktion B gut ab. Eine größere Übereinstimmung mit dem Referenzprofil wird nur bei compiler erreicht, führt aber wie oben erwähnt nicht zu einer Verbesserung der Laufzeit. Es fällt auf, dass in keiner Konfiguration eine Übereinstimmung von 100 % erreicht wird. Wir erklären dies damit, dass die Referenzprofile mit nur einer Arbeitslastiteration erstellt wurden. So werden, bedingt durch die 0,1 %-Klassifikation der Profile, Methoden als wichtig angesehen, die bei weiteren Arbeitslastiterationen im Verhältnis an Bedeutung verlieren würden.

Wir verzichten auf die Durchführung dieser Messung auf es-xscale, da zu erwarten ist, dass die getesteten Konfigurationen auch auf dem eingebetteten System zu keiner Verbesserung führen.

# 7 Schlussfolgerung und Ausblick

In diesem letzten Abschnitt fassen wir die Arbeit zusammen und bewerten die Ergebnisse kritisch.

## 7.1 Zusammenfassung

In Abschnitt 2 haben wir die Einsatzmöglichkeiten und Herausforderungen der Laufzeitübersetzung besprochen. Die wichtigste Erkenntnis ist, dass Laufzeitübersetzung sich lohnen muss, d.h. dass die gewonnene Laufzeitverbesserung die durch die Übersetzung verursachten Kosten amortisieren muss. Es ist allgemein bekannt, dass nicht alle Methoden einen Einfluss auf die Laufzeit des Programms haben. Der gängige Ansatz ist daher, selektiv nur die Methoden zu übersetzen, die einen signifikanten Anteil an der Laufzeit des Programms haben.

Die Bestimmung dieser relevanten Methoden ist Ziel der in Abschnitt 3 vorgestellten Laufzeitübersetzersteuerung. Ein Stichprobenverfahren ermittelt periodisch die aktuell laufende Methode, verschiedene Bewertungsfunktionen erstellen daraus ein Laufzeitprofil und eine einfache Übersetzungsstrategie entscheidet durch Vergleich mit einem Schwellwert, ob die Methode als wichtig gelten soll oder nicht. Die Implementierung haben wir in Abschnitt 4 beschrieben.

In Abschnitt 5 haben wir einen Versuchsaufbau entwickelt, um die vorgeschlagene Laufzeitübersetzersteuerung zu evaluieren. Als Testsysteme dienen ein tragbarer Rechner und das Steuergerät eines Gasanalysegeräts. Beispielprogramme aus der SPECjvm2008-Sammlung werden als Stellvertreter für unterschiedliche Programmtypen verwendet. Für die Analyse der Messergebnisse stellen wir die Metriken Genauigkeit, Erkennungszeit, Erkennungsmehraufwand und Laufzeitverbesserung auf und zeigen, wie man ihre Güte aus den Messprotokollen bestimmt.

Nach dieser Vorbereitung führten wir in Abschnitt 6 drei Messungen durch. Eine erste Evaluierung klärte die Eigenschaften des Stichprobenverfahrens und der Bewertungsfunktionen. Es zeigte sich, dass die Laufzeitübersetzersteuerung mit den getesteten Konfiguration noch zu unselektiv war. Das eingebettete System profitierte von niedriger Stichprobenfrequenz und großer Genauigkeit der Laufzeitprofilerstellung. Wir wählten die qualitativ beste Bewertungsfunktion und eine Kombination der schnellsten Bewertungsfunktionen aus und aktivierten die Alterung, was zu einer Verbesserung der Genauigkeit und dadurch zu besseren Laufzeiten auf beiden Testsystemen führte. Als dritten Ansatz versuchten wir, durch eine Kombination der Bewertungsfunktionen und einem variablen Schwellwert eine weitere Laufzeitverbesserung zu erzielen, was uns aber nicht gelangt.

## 7.2 Schlussfolgerungen

Die Messungen haben gezeigt, dass man einen Laufzeitübersetzer auch bei eingebetteten Systemen mit begrenzten Ressourcen gewinnbringend einsetzen kann. Allerdings liefert schon die einfachste, nämlich die konstante Bewertungsfunktion, sehr gute Ergebnisse. Die Verwendung der anderen Bewertungsfunktionen sorgt zwar prinzipiell für eine

schnellere Erkennung der wichtigen Methoden, die tatsächliche Laufzeitverbesserung ist aber gering. Der Einsatz der Alterung hat sich als sinnvoll erwiesen, die Kombination von mehreren Bewertungsfunktionen und ein variabler Schwellwert haben hingegen in den durchgeführten Messungen keinen weiteren Vorteil gebracht.

Eine Ressourcen schonende und genaue Laufzeitübersetzersteuerung ist notwendig, aber nicht hinreichend für die größtmögliche Laufzeitverbesserung. Das Potential, ein Programm nur durch die geschickte Auswahl der Übersetzungskandidaten zu beschleunigen, ist begrenzt. Hauptauschlaggebend ist die Qualität der vom Übersetzer erzeugten nativen Instruktionen. Eine weitere Untersuchung der Laufzeitübersetzersteuerung scheint erst wieder sinnvoll zu sein, wenn der Übersetzer weitergehende Optimierungen unterstützt. Eventuell benötigen diese Optimierungen dann weitergehende Informtionen über das dynamische Verhalten des Programms.

# 8 Referenzen

## Literatur

[1] Knopflerfish OSGi Framework. http://knopflerfish.org/.

[2] SPEC jvm 2008. http://www.spec.org/jvm2008/.

[3] Aycock, J. A brief history of just-in-time. *ACM Comput. Surv. 35*, 2 (2003), 97–113.

[4] Badea, C., Nicolau, A., and Veidenbaum, A. V. A simplified java bytecode compilation system for resource-constrained embedded processors. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems* (New York, NY, USA, 2007), ACM, pp. 218–228.

[5] Bentley, J. Programmimg pearls. *Commun. ACM 28*, 9 (1985), 896–901.

[6] Bond, M. D., and McKinley, K. S. Continuous Path and Edge Profiling. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 130–140.

[7] Feldbusch, F. Vorlesungsfolien zu Optimierung und Synthese eingebetteter Systeme (ES1), Wintersemester 2008/2009.

[8] Gosling, J., and McGilton, H. The Java Language Environment. http://java.sun.com/docs/white/langenv/.

[9] Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., and Cox, D. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Archit. Code Optim. 5*, 1 (2008), 1–32.

[10] Lee, S.-W., Moon, S.-M., and Kim, S.-M. Enhanced hot spot detection heuristics for embedded java just-in-time compilers. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems* (New York, NY, USA, 2008), ACM, pp. 13–22.

[11] Schilling, J. L. The simplest heuristics may be the best in Java JIT compilers. *SIGPLAN Not. 38*, 2 (2003), 36–46.

[12] Suganuma, T., Ogasawara, T., Kawachiya, K., Takeuchi, M., Ishizaki, K., Koseki, A., Inagaki, T., Yasue, T., Kawahito, M., Onodera, T., Komatsu, H., and Nakatani, T. Evolution of a java just-in-time compiler for IA-32 platforms. *IBM J. Res. Dev. 48*, 5/6 (2004), 767–795.

[13] Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T. Overview of the IBM Java just-in-time compiler. *IBM Syst. J. 39*, 1 (2000), 175–193.

[14] Suganuma, T., Yasue, T., and Nakatani, T. An Empirical Study of Method In-lining for a Java Just-in-Time Compiler. In *Proceedings of the 2nd Java™ Virtual Machine Research and Technology Symposium* (Berkeley, CA, USA, 2002), USENIX Association, pp. 91–104.

[15] Trapp, M. *Optimierung objektorientierter Programme. Übersetzungstechniken, Analysen und Transformationen.* PhD thesis, University of Karlsruhe, Faculty of Informatik, Oct. 2001.

[16] Whaley, J. A portable sampling-based profiler for Java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande* (New York, NY, USA, 2000), ACM, pp. 78–87.

[17] Yasue, T., Suganuma, T., Komatsu, H., and Nakatani, T. An Efficient Online Path Profiling Framework for Java Just-In-Time Compilers. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques* (Washington, DC, USA, 2003), IEEE Computer Society, p. 148.

## Abbildungsverzeichnis

1 Unterschiede bei der Programmrepräsentation . . . . . . . . . . . . . . . . . 5
2 Aufbau der Laufzeitübersetzersteuerung . . . . . . . . . . . . . . . . . . . . 10
3 Muster für die Schleifenerkennung . . . . . . . . . . . . . . . . . . . . . . . 13
4 Vereinfachte Softwarearchitektur . . . . . . . . . . . . . . . . . . . . . . . 16
5 Referenzprofile der SPECjvm2008-Arbeitslasten . . . . . . . . . . . . . . . . 21
6 Erste Evaluierung der compress-Arbeitslast auf pc-x86 . . . . . . . . . . . 38
7 Erste Evaluierung der scimark.fft-Arbeitslast auf pc-x86 . . . . . . . . . . 39
8 Erste Evaluierung der compiler-Arbeitslast auf pc-x86 . . . . . . . . . . . 40
9 Erste Evaluierung der compress-Arbeitslast auf es-xscale . . . . . . . . . . 41
10 Erste Evaluierung der scimark.fft-Arbeitslast auf es-xscale . . . . . . . . . 42
11 Erste Evaluierung der compiler-Arbeitslast auf es-xscale . . . . . . . . . . 43
12 Laufzeiten der Messungen mit Alterung auf pc-x86 . . . . . . . . . . . . . 44
13 Genauigkeit der Messungen mit Alterung auf pc-x86 . . . . . . . . . . . . 45
14 Laufzeiten der Messungen mit Alterung auf es-xscale . . . . . . . . . . . . 46
15 Mehraufwand durch aktivierte Alterung auf es-xscale . . . . . . . . . . . . 47
16 Genauigkeit der Messungen mit Alterung auf es-xscale . . . . . . . . . . . 48
17 Laufzeiten mit Kombination der Bewertungsfunktionen (compress) . . . . 49
18 Genauigkeit mit Kombination der Bewertungsfunktionen (compress) . . . 49
19 Laufzeiten mit Kombination der Bewertungsfunktionen (scimark.fft) . . . 50
20 Genauigkeit mit Kombination der Bewertungsfunktionen (scimark.fft) . . 50
21 Laufzeiten mit Kombination der Bewertungsfunktionen (compiler) . . . . 51
22 Genauigkeit mit Kombination der Bewertungsfunktionen (compiler) . . . . 51

## Tabellenverzeichnis

1 Parameter der Laufzeitübersetzersteuerung . . . . . . . . . . . . . . . . . . 17
2 Technische Daten der Testsysteme . . . . . . . . . . . . . . . . . . . . . . . . 18
3 Laufzeiten der Arbeitslasten mit und ohne Laufzeitübersetzer . . . . . . . 24

## Algorithmenverzeichnis

1 Profilerstellung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2 Konstante Bewertungsfunktion . . . . . . . . . . . . . . . . . . . . . . . . . 12
3 Akkumulierende Bewertungsfunktion . . . . . . . . . . . . . . . . . . . . . . 12
4 Größe der Methode . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
5 Schleifenerkennung . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
6 Aufrufkellerinspektion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

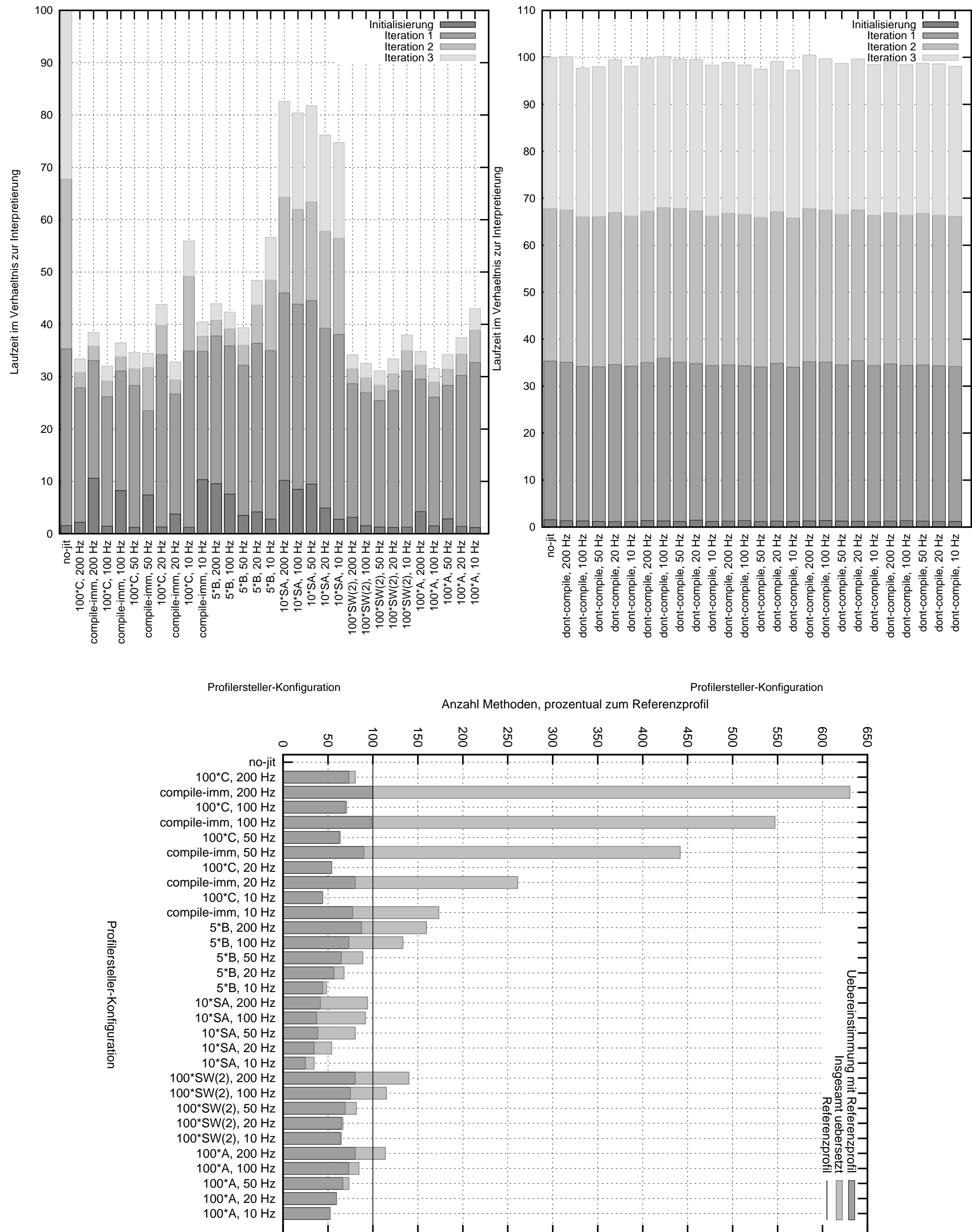



**Abbildung 6:** Erste Evaluierung der compress-Arbeitslast auf pc-x86. Links: Messung mit aktiviertem Übersetzer. Rechts: Vergleich des Mehraufwands durch Stichprobennahme und Profilerstellung. Unten: Betrachtung der Genauigkeit durch den Vergleich der Übereinstimmungen mit dem Referenzprofil mit den insgesamt übersetzten Methoden.

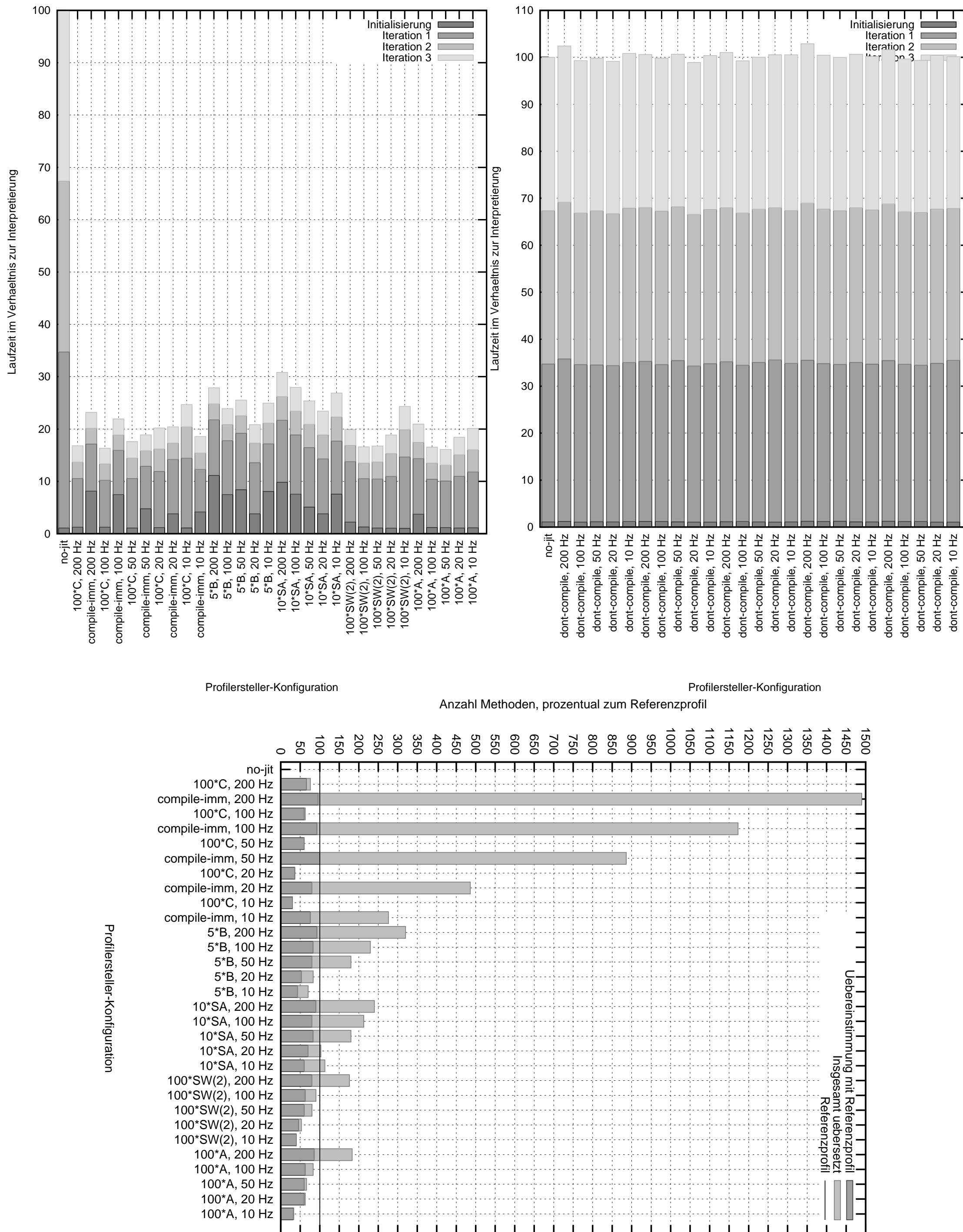



**Abbildung 7:** Erste Evaluierung der scimark.fft-Arbeitslast auf pc-x86. Links: Messung mit aktiviertem Übersetzer. Rechts: Vergleich des Mehraufwands durch Stichprobennahme und Profilerstellung. Unten: Betrachtung der Genauigkeit durch den Vergleich der Übereinstimmungen mit dem Referenzprofil mit den insgesamt übersetzten Methoden.

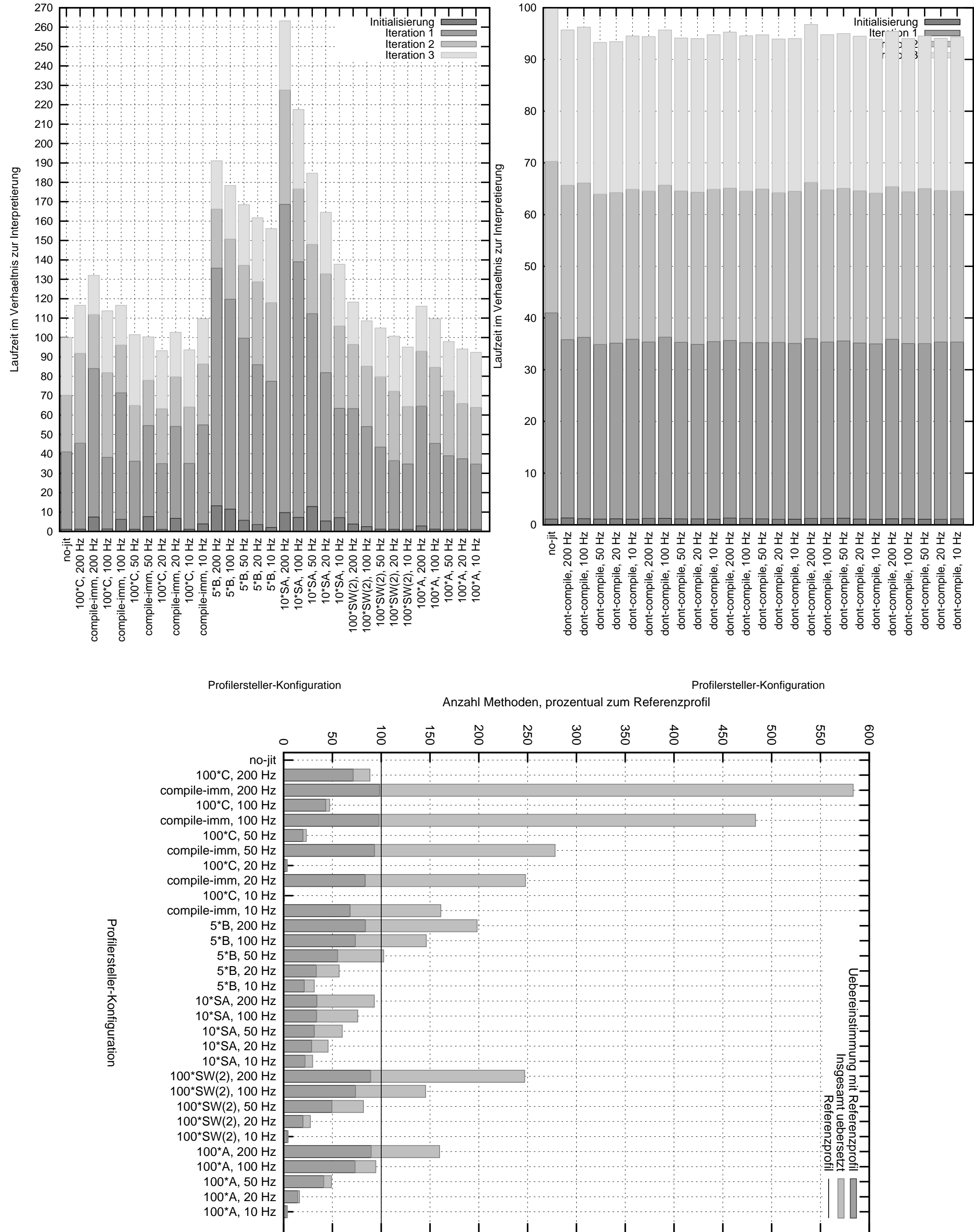


**Abbildung 8:** Erste Evaluierung der compiler-Arbeitslast auf pc-x86. Links: Messung mit aktiviertem Übersetzer. Rechts: Vergleich des Mehraufwands durch Stichprobennahme und Profilerstellung. Unten: Betrachtung der Genauigkeit durch den Vergleich der Übereinstimmungen mit dem Referenzprofil mit den insgesamt übersetzten Methoden.

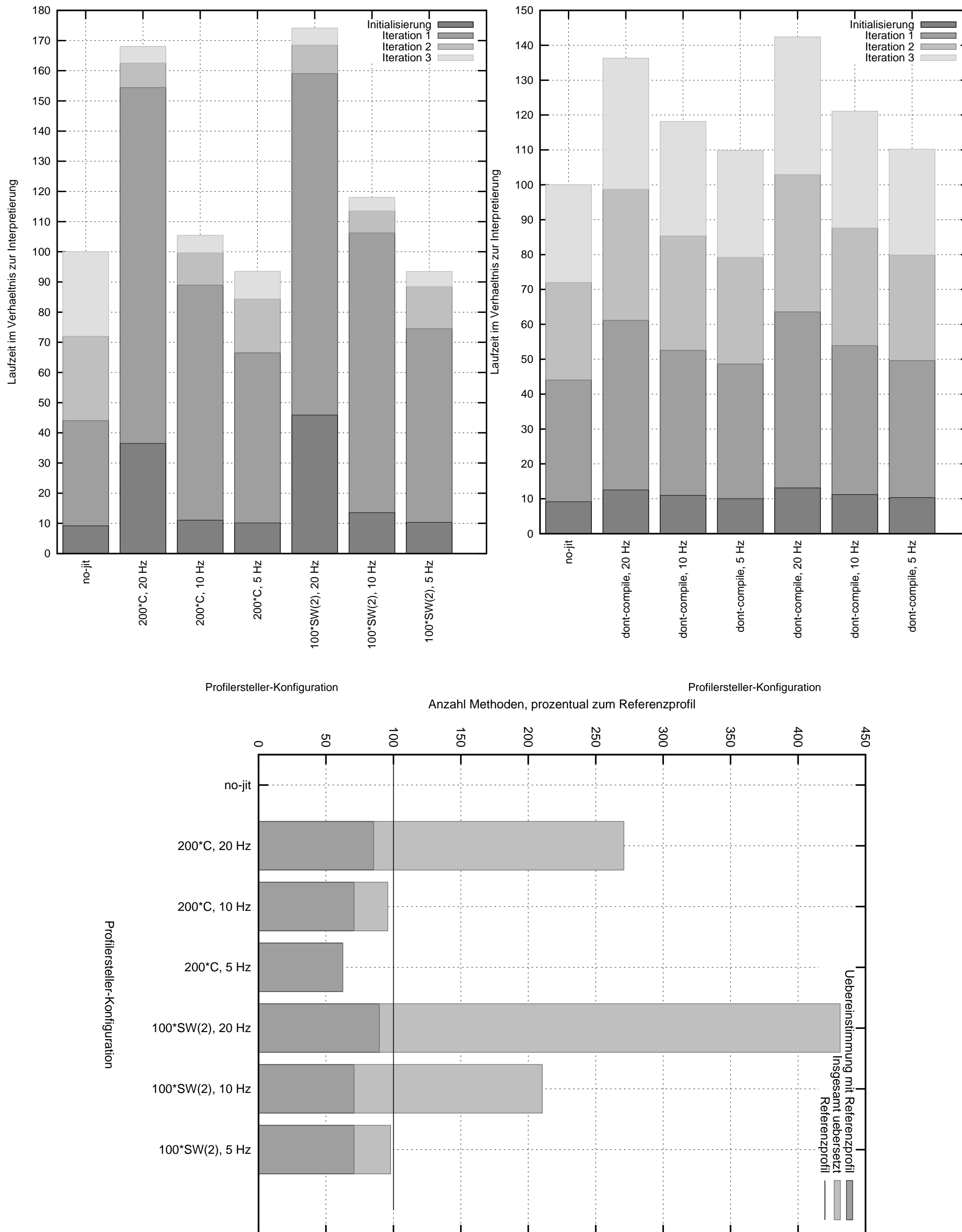



**Abbildung 9:** Erste Evaluierung der compress-Arbeitslast auf es-xscale. Links: Messung mit aktiviertem Übersetzer. Rechts: Vergleich des Mehraufwands durch Stichprobennahme und Profilerstellung. Unten: Betrachtung der Genauigkeit durch den Vergleich der Übereinstimmungen mit dem Referenzprofil mit den insgesamt übersetzten Methoden.

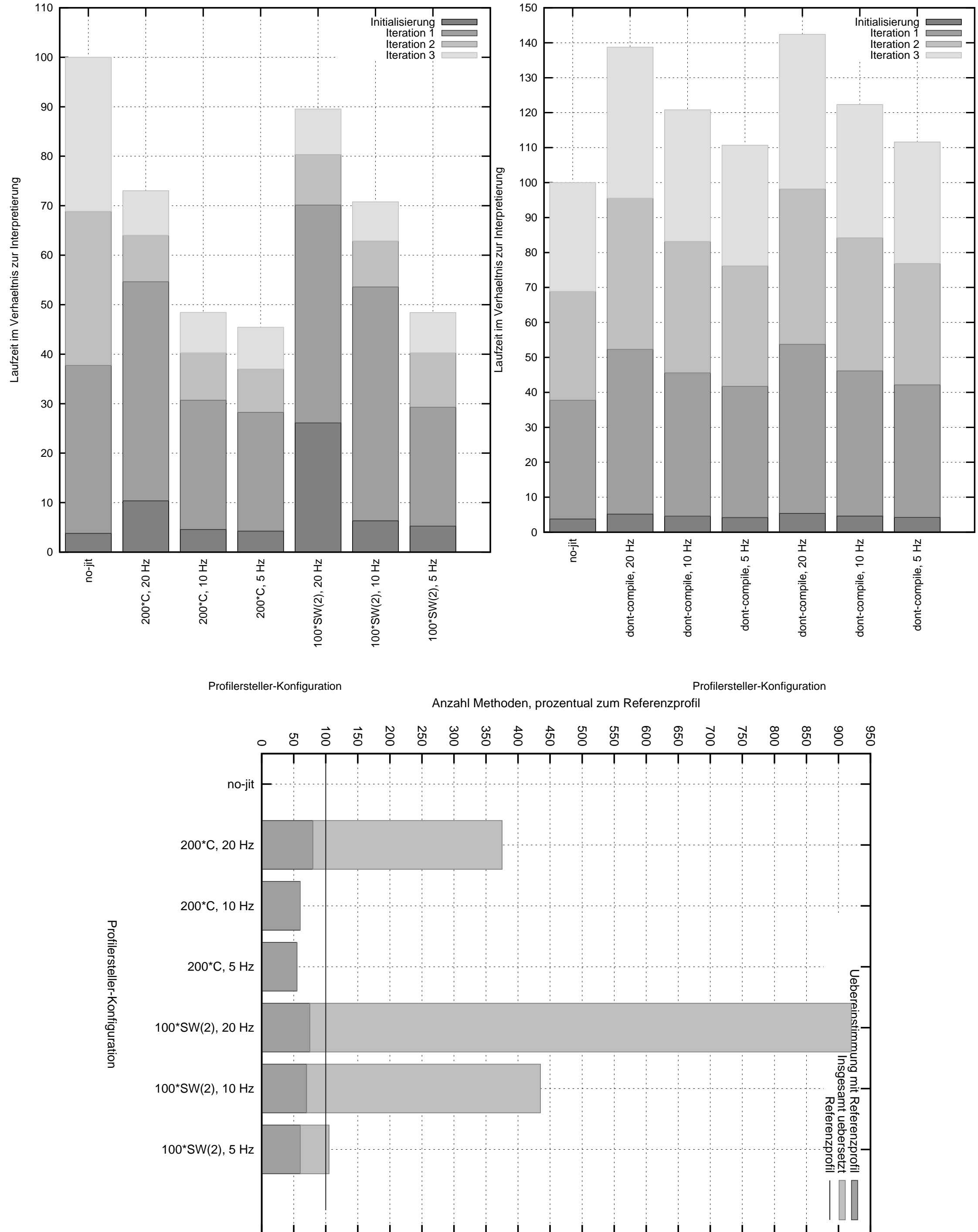



**Abbildung 10:** Erste Evaluierung der scimark.fft-Arbeitslast auf es-xscale. Links: Messung mit aktiviertem Übersetzer. Rechts: Vergleich des Mehraufwands durch Stichprobennahme und Profilerstellung. Unten: Betrachtung der Genauigkeit durch den Vergleich der Übereinstimmungen mit dem Referenzprofil mit den insgesamt übersetzten Methoden.

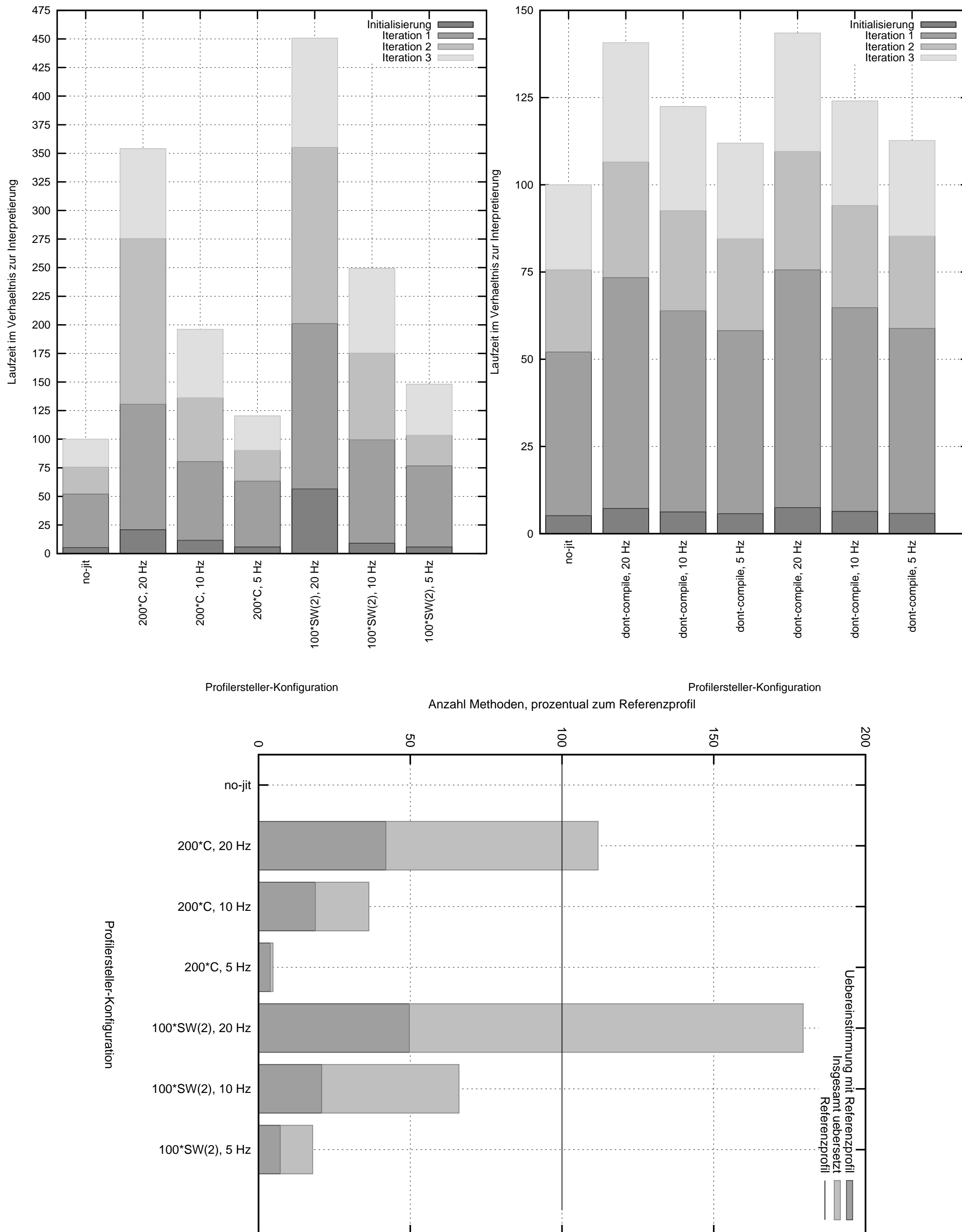



**Abbildung 11:** Erste Evaluierung der compiler-Arbeitslast auf es-xscale. Links: Messung mit aktiviertem Übersetzer. Rechts: Vergleich des Mehraufwands durch Stichprobennahme und Profilerstellung. Unten: Betrachtung der Genauigkeit durch den Vergleich der Übereinstimmungen mit dem Referenzprofil mit den insgesamt übersetzten Methoden.

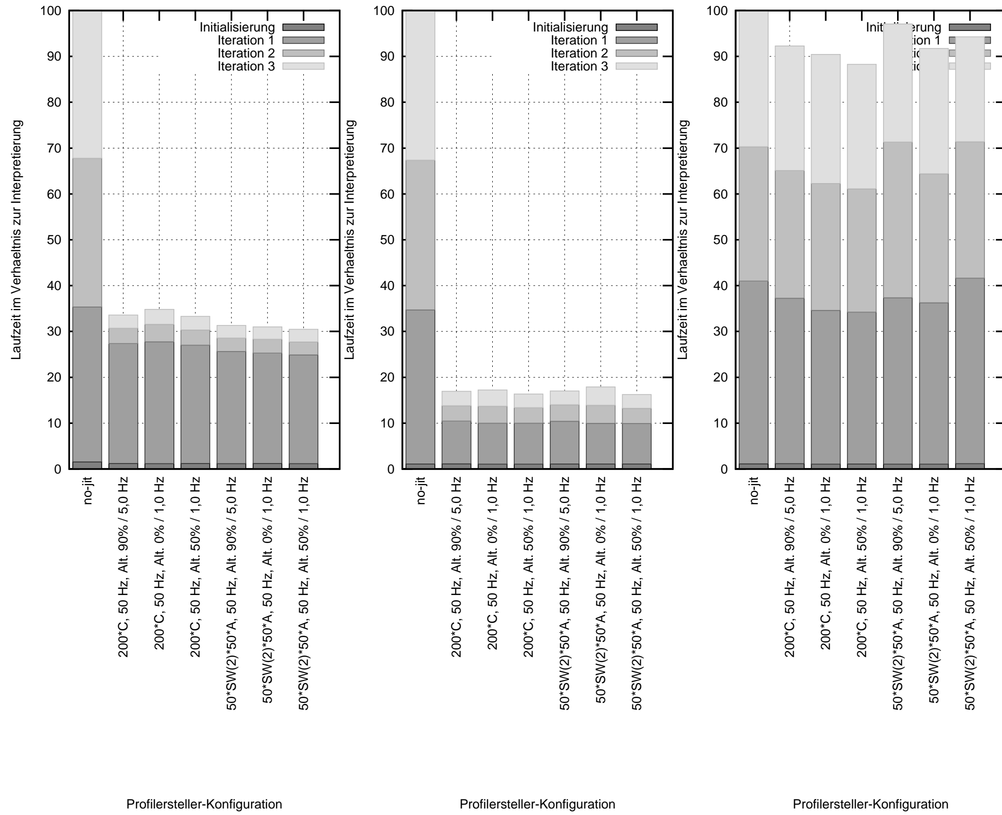



**Abbildung 12:** Laufzeiten der Messung mit eingeschalteter Alterung auf pc-x86. v.l.n.r.: compress, scimark.fft, compiler.

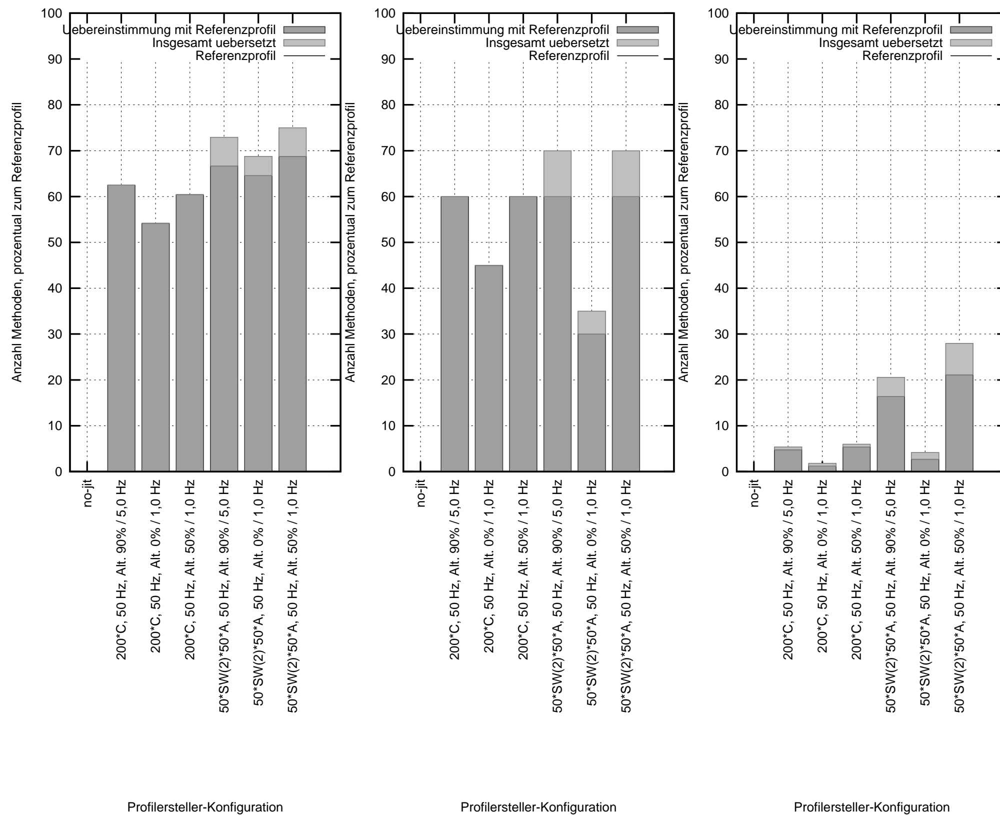



**Abbildung 13:** Betrachtung der Übereinstimmung mit dem Referenzprofil bei eingeschalteter Alterung auf pc-x86.
v.l.n.r.: compress, scimark.fft, compiler.

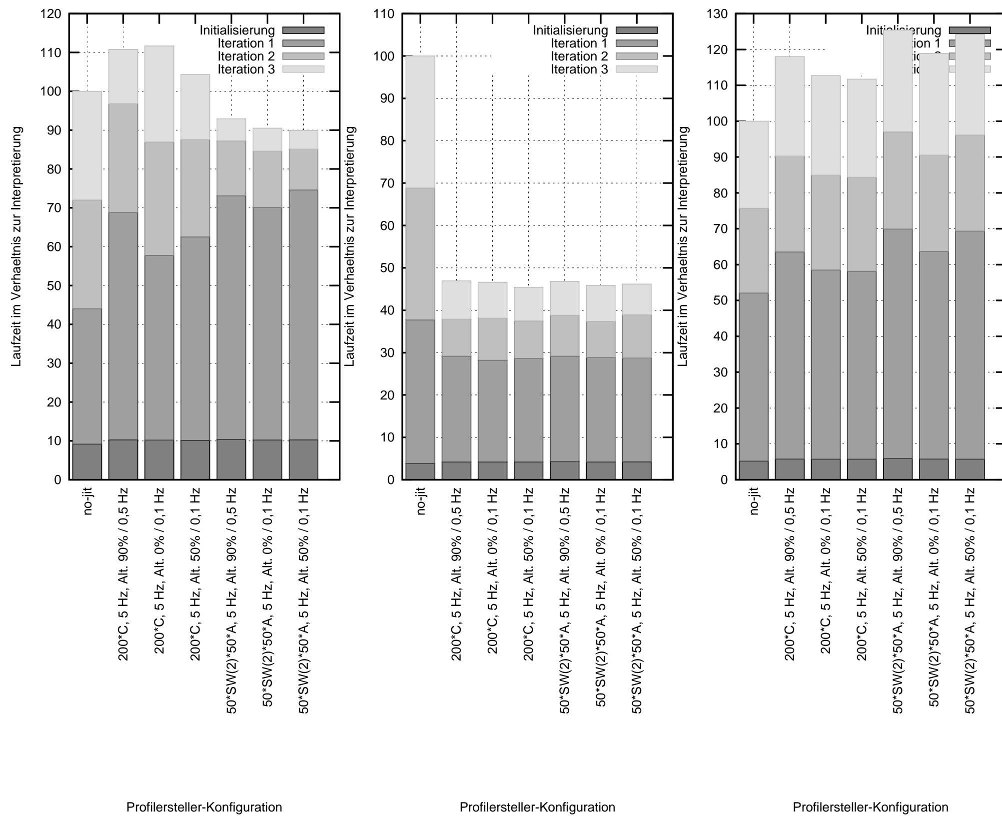



**Abbildung 14:** Ergebnisse der Messung mit eingeschalteter Alterung auf es-xscale. v.l.n.r.: compress, scimark.fft, compiler.

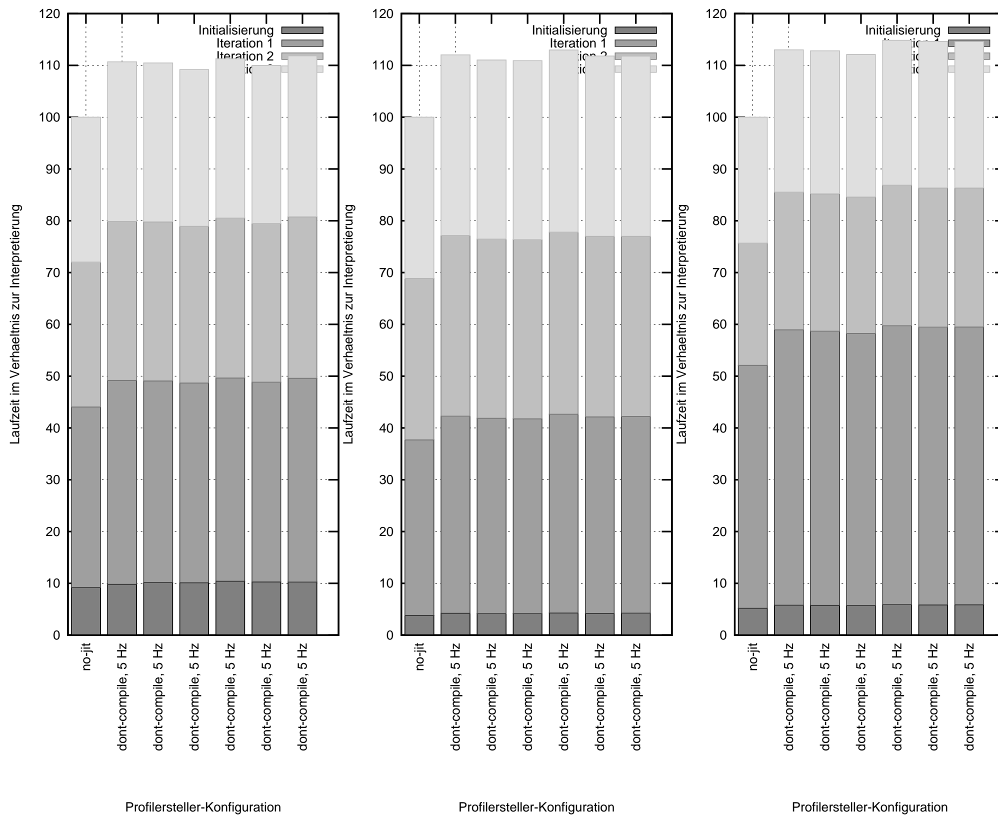



**Abbildung 15:** Mehraufwand für die Alterung auf es-xscale.
v.l.n.r.: compress, scimark.fft, compiler.

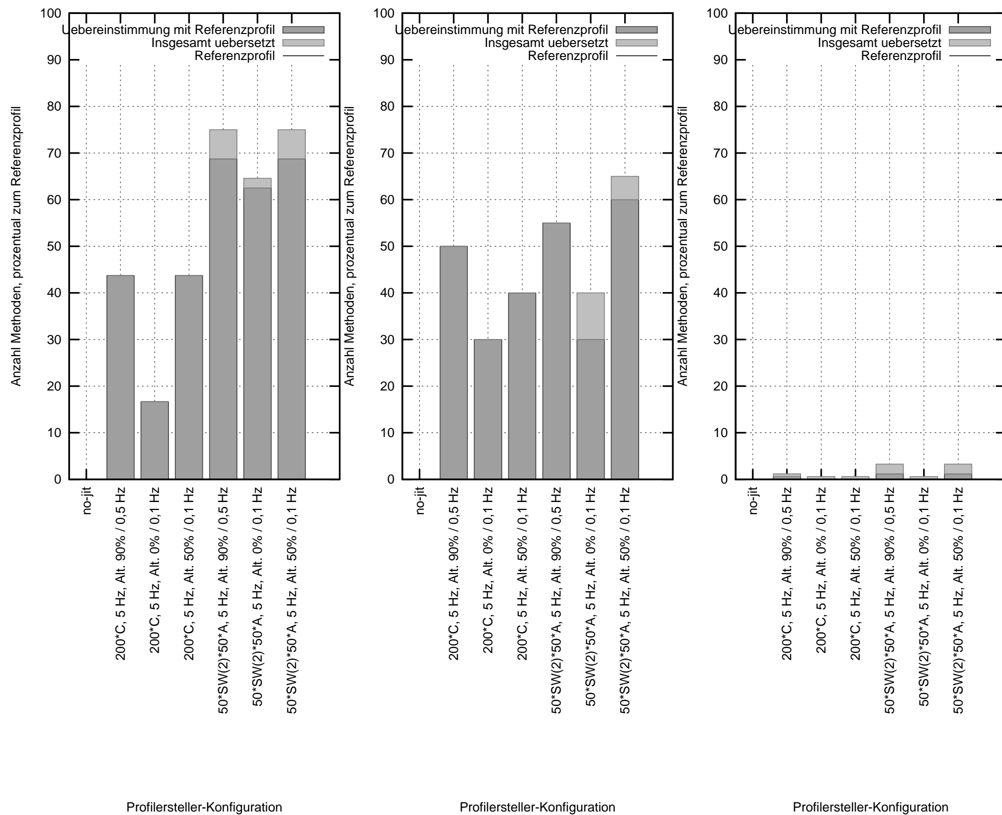



**Abbildung 16:** Betrachtung der Übereinstimmung mit dem Referenzprofil bei eingeschalteter Alterung auf es-xscale.
v.l.n.r.: compress, scimark.fft, compiler.

**Abbildung 17:** Ergebnisse der Messungen mit Kombination der Bewertungsfunktionen bei der compress-Arbeitslast auf pc-x86.

**Abbildung 18:** Betrachtung der Übereinstimmung mit dem Referenzprofil bei Kombination der Bewertungsfunktionen bei der compress-Arbeitslast auf pc-x86.

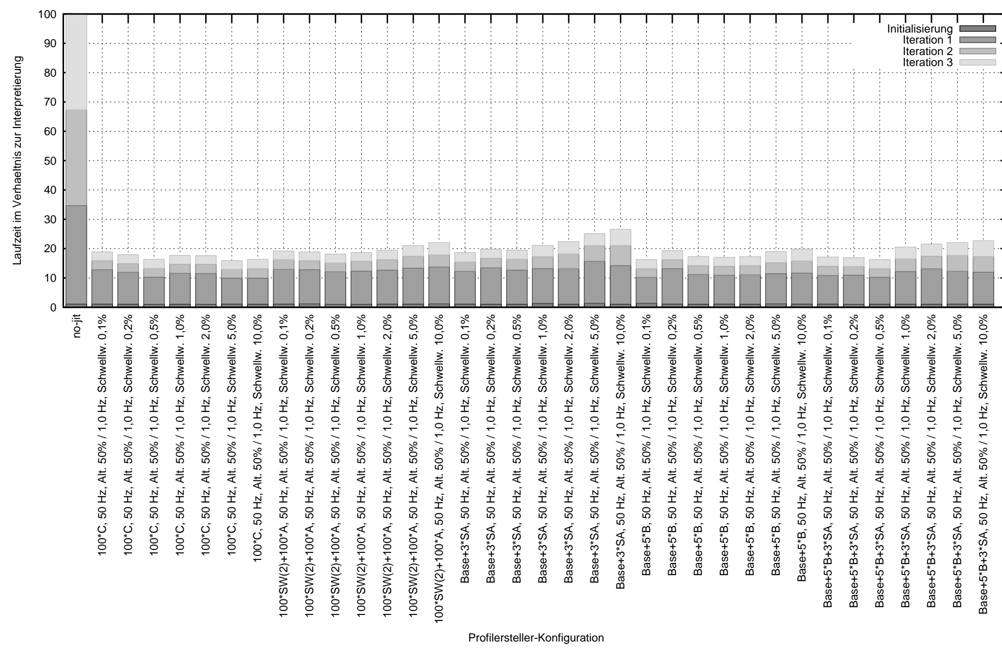



**Abbildung 19:** Ergebnisse der Messungen mit Kombination der Bewertungsfunktionen bei der scimark.fft-Arbeitslast auf pc-x86.

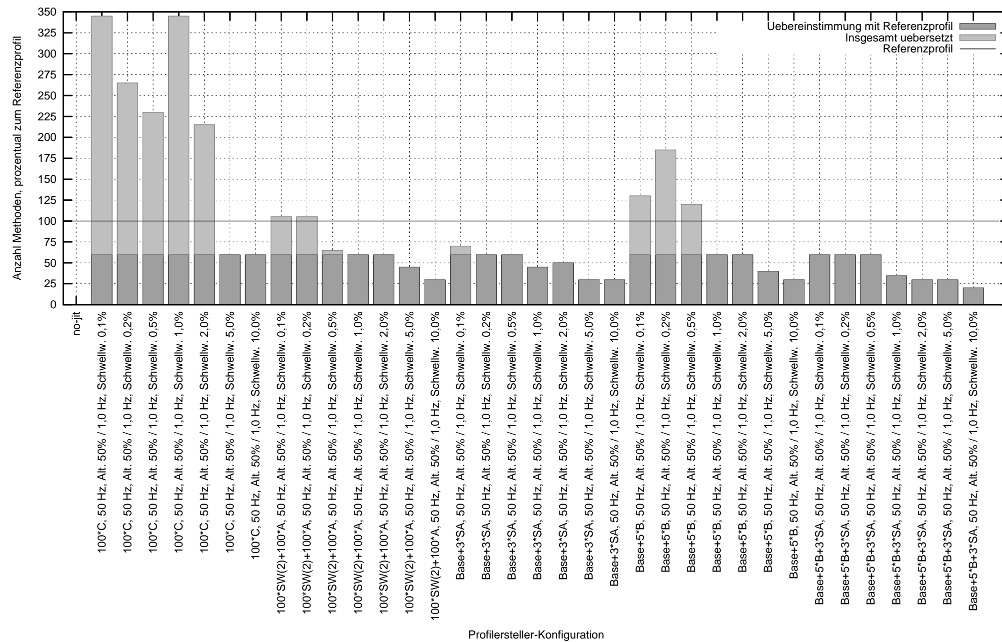



**Abbildung 20:** Betrachtung der Übereinstimmung mit dem Referenzprofil bei Kombination der Bewertungsfunktionen bei der scimark.fft-Arbeitslast auf pc-x86.

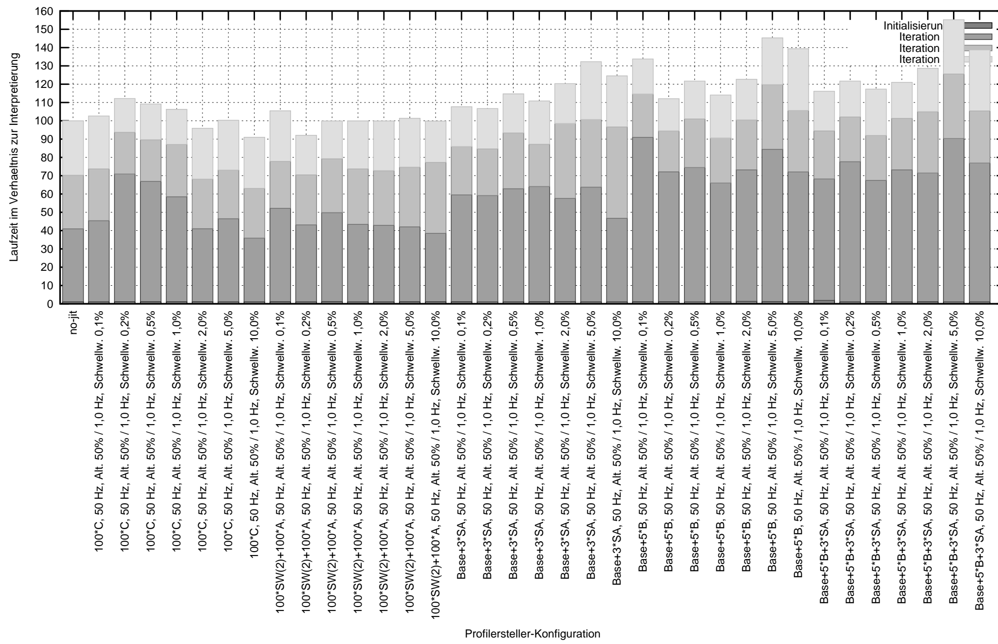



**Abbildung 21:** Ergebnisse der Messungen mit Kombination der Bewertungsfunktionen bei der compiler-Arbeitslast auf pc-x86.

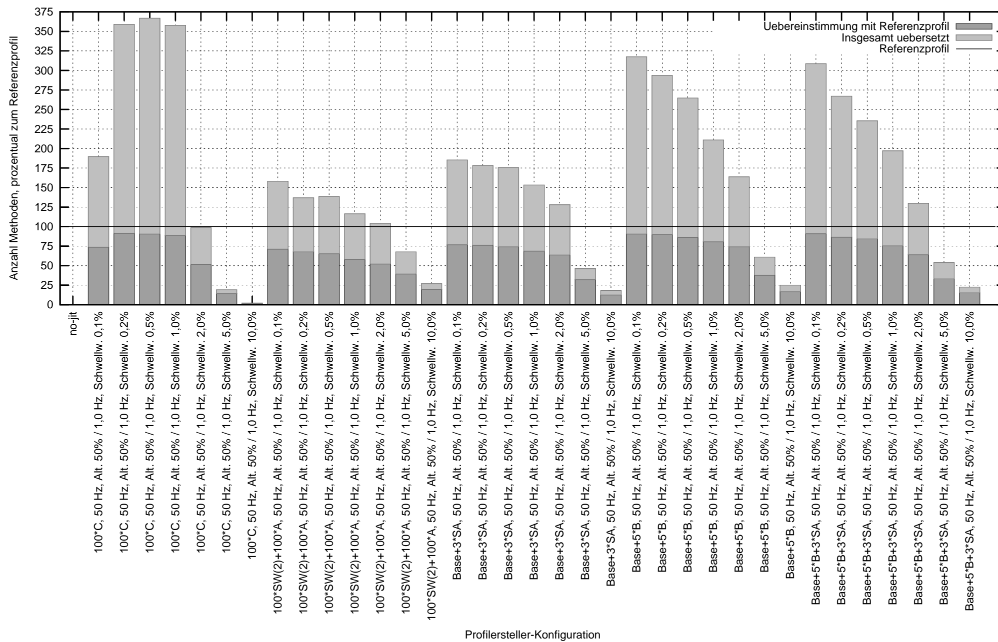



**Abbildung 22:** Betrachtung der Übereinstimmung mit dem Referenzprofil bei Kombination der Bewertungsfunktionen bei der compiler-Arbeitslast auf pc-x86.