Karlsruher Institut für Technologie

Institut für Programmstrukturen
und Datenorganisation (IPD)

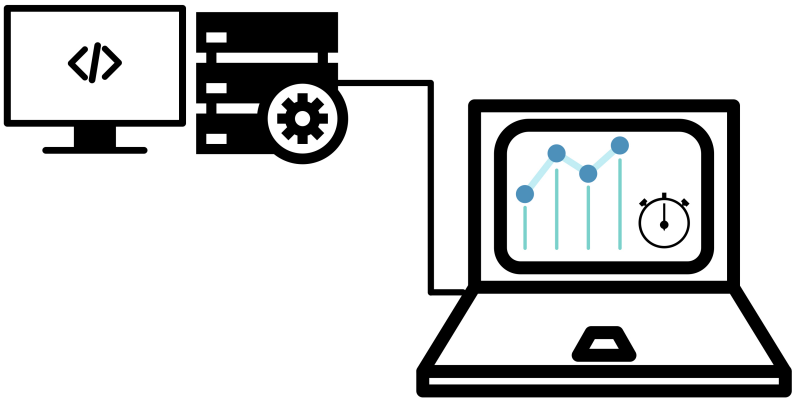
Lehrstuhl Prof. Dr.-Ing. Snelting

Automatische Systemkonfiguration für reproduzierbare Benchmarks

Bachelorarbeit von

Jacques Marco Jung

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Johannes Bechberger
Abgabedatum:	20. März 2020

Zusammenfassung

In der Softwareentwicklung wird die Laufzeit von Software gemessen und verglichen. Ein Problem dabei ist die Schwankung der Laufzeit, die bei mehreren Messungen auftritt. Um einen aussagekräftigen Vergleich zweier Softwareversionen anstellen und den Unterschied statistisch nachweisen zu können, muss diese Schwankung beachtet werden. Die Standardabweichung stellt hierfür eine Metrik dar. Je höher die Standardabweichung relativ zur Laufzeit einer Software ist, desto schwieriger ist der Vergleich. Daher ist es sinnvoll die Standardabweichung der Laufzeit zu reduzieren.

Es gibt Betriebssystem- und Hardwareparameter, die die Laufzeit der Software beeinflussen und Schwankungen verursachen. Das Ändern dieser Parameter kann die Standardabweichung reduzieren. Dazu gehören die Deaktivierung von Simultaneous Multithreading, die Deaktivierung des Schedulers auf bestimmten Prozessorkernen und das Verändern der CPU-Frequenz. Um das Ändern der richtigen Parameter zu automatisieren wurde im Rahmen dieser Arbeit das Tool *Autotune* entwickelt.

Autotune verwendet dabei zwei Algorithmen, die die Parameter auf einem Computer setzen, die Standardabweichung der Laufzeit der gewünschten Software messen und mit dem vorherigen Wert vergleichen, um zu entscheiden, ob der Parameter beibehalten oder verworfen wird. Bei den Algorithmen handelt es sich zum einen um einen auf Simulated Annealing basierenden Algorithmus, der eine sehr niedrige Standardabweichung erreicht. Zum anderen gibt es einen optimierten Algorithmus, der auf Erfahrungswerten basiert und dadurch schneller ausgeführt wird. Durch das Verwenden von Erfahrungswerten wird jedoch nicht immer die minimale Standardabweichung erreicht.

Aus Benchmarks ergibt sich, dass die entscheidenden Parameter, die die Standardabweichung reduzieren, für verschiedene Computer und Software sehr unterschiedlich sein können. Die gängigen Techniken zur Reduzierung der Standardabweichung arbeiten mit einer festen Auswahl an Parametern, die unabhängig von der verwendeten Hardware und Software gesetzt werden. *Autotune* erreicht durch seine an die Hardware und Software angepasste Auswahl an Parametern eine durchschnittlich viermal niedrigere Standardabweichung im Vergleich zu den gängigen Techniken.

Den größten Effekt haben die Parameter auf CPU-bound Software. Memory-bound Software hingegen hat ohnehin schon eine geringe Standardabweichung. Parallele Software kann auf einem Prozessorkern sequenziell ausgeführt eine niedrigere Standardabweichung haben als parallel ausgeführt.

Inhaltsverzeichnis

1. Einführung

In der Softwareentwicklung wird Software immer weiter optimiert, sodass sie schneller und effizienter läuft. Dabei muss geprüft werden, welche Auswirkungen Änderungen am Code auf die Laufzeit der Software haben. Bedingt durch Schwankungen in der gemessenen Laufzeit ist es problematisch einen geeigneten und aussagekräftigen Vergleich zwischen Softwareversionen anzustellen. Oft sind die Unterschiede der Laufzeiten so gering, dass auch kleine Schwankungen einen Vergleich unmöglich machen. Diese Arbeit und das in deren Rahmen entwickelte Tool sind vor allem an Softwareentwickler und Akademiker gerichtet.

Angenommen es wurden für die Versionen 1 und 2 einer Software die Laufzeiten in der Abbildung 1.1 gemessen. Die einzelnen Messungen schwanken so sehr, dass man nicht feststellen kann, ob sich die Version 2 im Vergleich zu Version 1 verbessert hat. Legt man die beiden Graphen übereinander, sieht man, dass die Verteilung von Softwareversion 2 zwar im Gesamten etwas weiter links liegt, sich jedoch auch einige Werte überschneiden. So ist es nicht möglich eindeutig festzustellen, welche der Versionen schneller ist.

Daher ist es notwendig die Schwankungen so weit wie möglich zu minimieren. Diese hängen nicht nur von der Software selbst ab, sondern auch von äußeren Einflüssen wie Hardware und Betriebssystem. Es gibt bereits einige Empfehlungen, unter anderem von *LLVM* [1], und auch Software, wie *Pyperf* [2], die die Schwankungen reduzieren sollen. Die Empfehlungen und Programme beachten jedoch nur eine kleine Teilmenge der möglichen Einflüsse und reduzieren die Schwankungen daher nur leicht. Das Ziel dieser Arbeit ist es, aus einer großen Menge an Parametern die auszuwählen, welche die Schwankungen minimieren. Dazu wurde im Rahmen dieser Arbeit das Tool *Autotune* entwickelt. *Autotune* testet den Einfluss von Hardware- und Softwareparametern auf die Schwankung der Laufzeit, wählt diejenigen aus, die die Standardabweichung am weitesten reduzieren und wendet diese auf dem Computer an. Auf diese Weise werden für jeden Computer automatisch die optimalen Parameter gefunden und die Schwankungen minimiert.

Mindert man nun die äußeren Einflüsse, so ergeben sich beim Messen die Werte in Abbildung 1.2. Hier ist es einfacher festzustellen, welche Verbesserungen Version 2 im Gegensatz zu Version 1 mit sich bringt. Alle Werte sind näher am Mittelwert und die Werte von Version 2 liegen immer unter denen von Version 1. Legt man hier

die beiden Graphen übereinander, so überschneiden sich nur wenige Werte und die Verteilung der Version 2 liegt deutlich weiter links von der Verteilung der Version 1.

In Kapitel 2 werden zunächst einige Grundlagen der Statistik und des Benchmarkens eingeführt. Dann werden in Kapitel 3 die Aspekte der Hardware und des Betriebssystems behandelt, welche die Schwankungen verursachen. Des Weiteren wird in Kapitel 4 erklärt, wie das entwickelte Tool *Autotune* arbeitet um diese Schwankungen zu reduzieren. Die damit erzielten Verbesserungen der Schwankungen und die Effekte auf die Laufzeit der Software, die das Reduzieren der Einflüsse mit sich bringt, werden in der Evaluation in Kapitel 5 aufgezeigt. Zum Schluss wird im Fazit noch einmal das Wichtigste zusammengefasst und ein Ausblick gegeben.

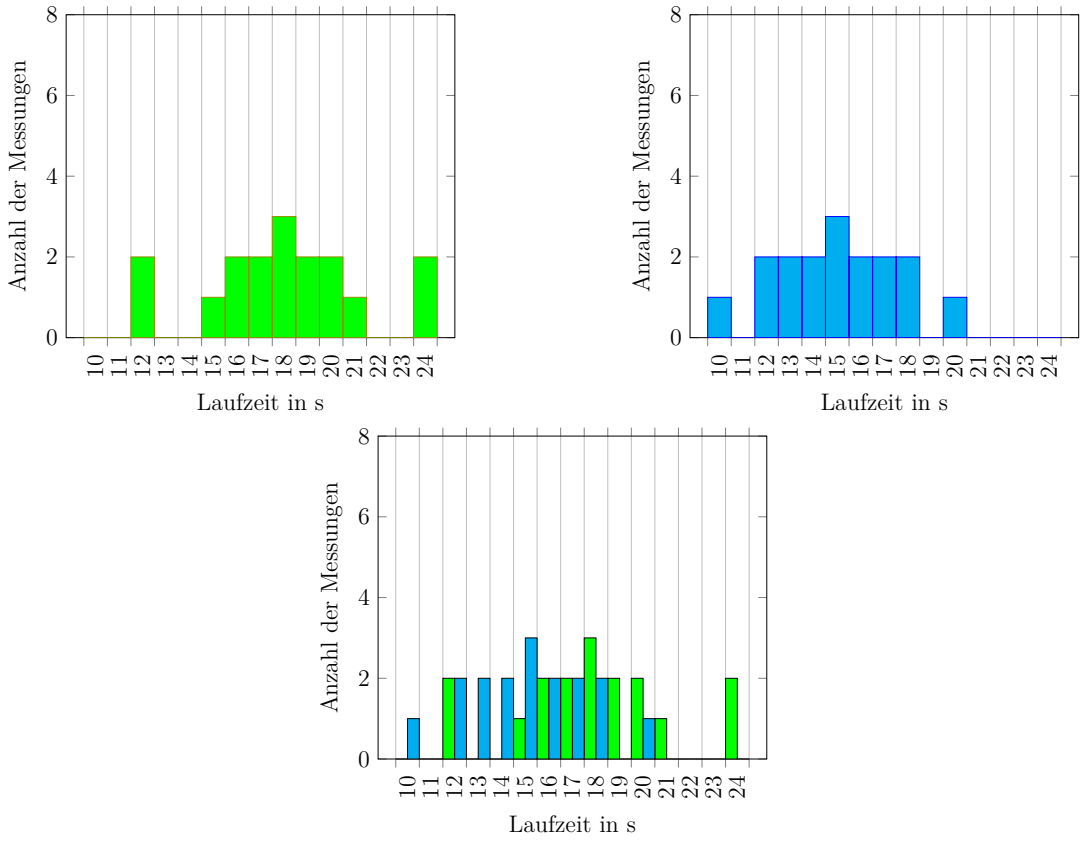


Abbildung 1.1.: Messung der Laufzeit von Softwareversion 1 (links) und Softwareversion 2 (rechts) mit äußeren Einflüssen und Mittelwerten von $\overline{x_1} = 18$ und $\overline{x_2} = 15$. Der untere Graph zeigt die Überlagerung der beiden oberen Graphen.

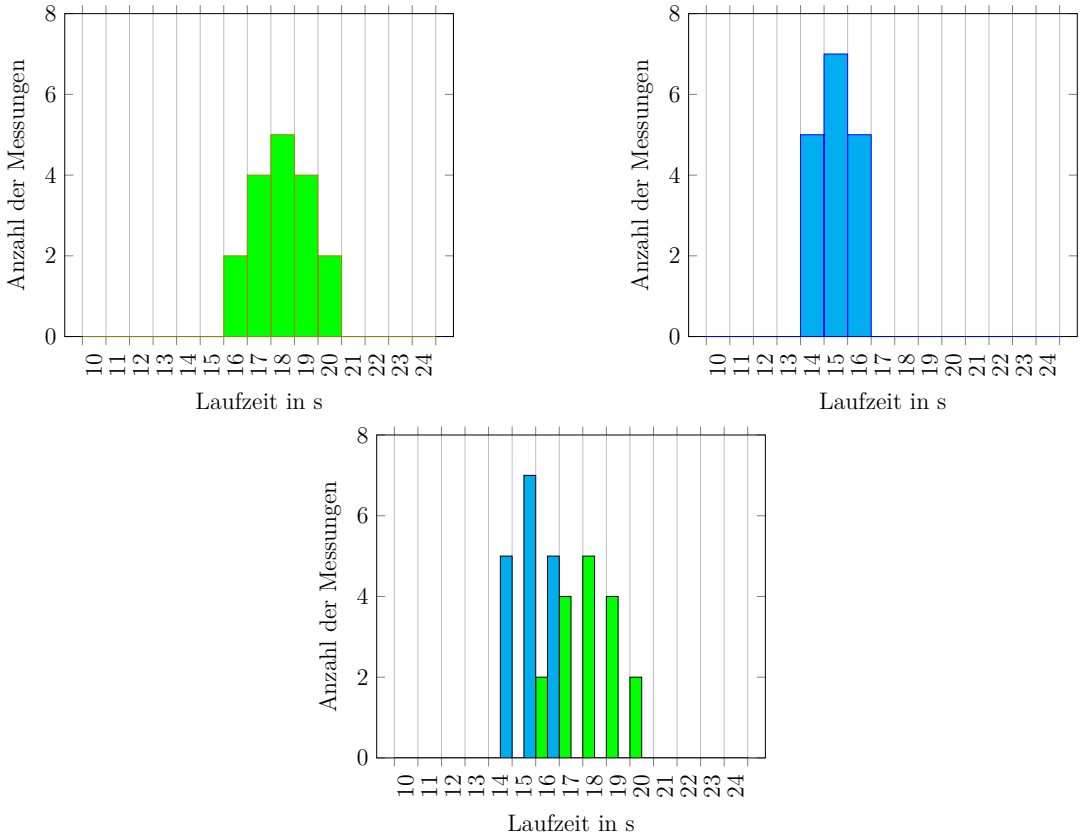


Abbildung 1.2.: Messung der Laufzeit von Softwareversion 1 (links) und Softwareversion 2 (rechts) mit abgeschwächten Einflüssen und Mittelwerten von $\overline{x_1} = 18$ und $\overline{x_2} = 15$. Der untere Graph zeigt die Überlagerung der beiden oberen Graphen.

2. Grundlagen und Verwandte Arbeiten

In den folgenden Abschnitten werden einige Grundlagen und deren Bezug zum Thema der Arbeit behandelt. Dabei wird zuerst der Begriff „Systemkonfiguration“ erläutert. Dann wird auf mathematische Grundlagen eingegangen und weshalb diese essentiell für den Vergleich der Laufzeiten von Software sind. Zum Schluss werden Benchmarks erklärt und warum eine Systemkonfiguration einige Probleme beim Benchmarken beheben kann.

2.1. Begriff der Systemkonfiguration

Die Systemkonfiguration ist im Rahmen dieser Arbeit das Einstellen von Betriebssystemparametern und Hardwareparametern eines Systems beziehungsweise eines Computers, sodass die Streuung der Laufzeit von Software minimiert wird. Es gibt Parameter und Hardwareeigenschaften, die in heutigen Computern verwendet werden, um Abstürze zu vermeiden und schnelle Ausführungszeiten zu erreichen. Diese führen jedoch oftmals zu einer starken Streuung in den Ausführungszeiten. Unter anderem werden moderne Prozessoren zeitweise dynamisch übertaktet. Das führt zu einer kürzeren Ausführungszeit, jedoch auch zu größeren Schwankungen bei wiederholter Ausführung. Deshalb ist es wichtig, diese Einflüsse mit Hilfe der Systemkonfiguration einzudämmen und dadurch eine bessere Umgebung für den Vergleich von Software zu schaffen. Weitere Details zur Systemkonfiguration und den einstellbaren Parametern werden in Kapitel 3 behandelt.

2.2. Statistische Grundlagen

Um die richtigen Parameter für die Systemkonfiguration zu finden, werden einige Grundlagen aus der Statistik benötigt. Es geht in dieser Arbeit primär um das Reduzieren der Streuung der Laufzeit, die beim Ausführen von Software auftritt.

Mit Hilfe des arithmetischen Mittels (2.1) kann die mittlere Laufzeit angegeben und verglichen werden. Die Streuung wird mathematisch mit der Varianz (2.2) und der Standardabweichung (2.3) beschrieben. Die Systemkonfiguration soll nun die Varianz reduzieren. Mit dem in Abschnitt 2.2.2 beschriebenen F-Test ist es möglich diese Reduzierung nachzuweisen. Möchte man nun Softwareversionen in ihren Laufzeiten vergleichen, kann man bei reduzierter Varianz (unter Annahme der gleichen Differenz der arithmetischen Mittel) mit Hilfe des in Abschnitt 2.2.3 beschriebenen t-Tests nachweisen, dass eine Softwareversion schneller ist als eine andere. Voraussetzung ist dabei eine vorhandene Differenz zwischen den Laufzeiten.

2.2.1. Empirische Varianz und Standardabweichung

Die Standardabweichung (2.3) gibt die Streuung der Werte einer Verteilung an. Sie ist ein wichtiges Maß für die Aussagekraft des arithmetischen Mittels. Des Weiteren ist die Standardabweichung eine gute Approximation für die Signifikanz des Unterschieds zweier Messreihen, wie Abschnitt 2.2.3 zeigen wird. Da die Laufzeit von Software nur stichprobenartig gemessen wird, werden hier die empirische Varianz und die empirische Standardabweichung verwendet. Die empirische Standardabweichung ergibt sich aus der Quadratwurzel der empirischen Varianz [3]. Zur Berechnung benötigt man zudem das arithmetische Mittel.

Arithmetisches Mittel:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n} \tag{2.1}$$

Empirische Varianz:

$$s^2 = \frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1} \tag{2.2}$$

Empirische Standardabweichung:

$$s = \sqrt{s^2} = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \overline{x})^2}{n-1}} \tag{2.3}$$

In den Gleichungen (2.1), (2.2) und (2.3) sind die x_i die Werte der Messreihe, $\overline{x}$ das arithmetische Mittel, n die Anzahl der Werte, s^2 die empirische Varianz und s die empirische Standardabweichung.

2.2.2. F-Test

Mit dem F-Test [4] kann überprüft werden, ob sich die Varianzen zweier unterschiedlicher Messreihen signifikant unterscheiden. So kann nach der Systemkonfiguration geprüft werden, ob sich die Varianz der Laufzeit einer Software signifikant reduziert hat. Dazu geht man folgendermaßen vor:

1) Seien die Werte mit Index 1 (s_1^2, n_1) die Kennzahlen der Messreihe mit der größeren Varianz und sei zudem eine Normalverteilung vorausgesetzt. Es gibt weitere Tests zum Nachweis der Reduzierung der Varianz, die keine Normalverteilung voraussetzen. Dazu zählen der Levene-Test [5] und der Brown-Forsythe-Test [6].

2) Berechne F-Wert: $F = \frac{s_1^2}{s_2^2}$.

3) Berechne Freiheitsgrade $df_1 = n_1 - 1$ und $df_2 = n_2 - 1$.

4) Wähle Signifikanzniveau α (z.B. $\alpha = 5\%$).

5) Lese kritischen Wert in der F-Tabelle 2.1 ab.

6) Wenn $F > kritischerWert$, ist die Varianz signifikant unterschiedlich.

Dabei ist s_i^2 die Varianz und n_i die Anzahl der Messwerte zur Messreihe i. Zu jedem α gibt es eine zugehörige F-Tabelle, in der der kritische Wert abgelesen werden kann. Tabelle 2.1 zeigt einen Ausschnitt einer solchen F-Tabelle für $\alpha = 0.05$. Den kritischen Wert erhält man, indem man die Spalte zum Freiheitsgrad df_1 und die Zeile zum Freiheitsgrad df_2 sucht. Ist F größer als der abgelesene kritische Wert, so ist der Unterschied der Varianzen signifikant. Durch die Wahl der größeren Varianz in Schritt 1 ist somit s_2^2 signifikant kleiner als s_1^2. Das Signifikanzniveau α entscheidet dabei, wie signifikant der Unterschied sein muss.

In der Abbildung 2.1 sind Beispielwerte für die Messung der Laufzeit einer Software vor und nach einer Systemkonfiguration aufgeführt. Berechnet man hierfür den F-Wert, ergibt sich: $F = \frac{s_1^2}{s_1^2} = \frac{117{,}86}{1{,}57} = 75{,}07$. Für die Freiheitsgrade ergibt sich: $df_1 = df_2 = 14$. Das Signifikanzniveau sei $\alpha = 0.05$. Abgelesen an der F-Tabelle ergibt sich ein kritischer Wert von $2{,}48$. Nun sieht man: $F = 75{,}07 > 2{,}48 = kritischerWert$. Somit sind die Varianzen signifikant unterschiedlich. Da die Standardabweichung $s_2 = 1{,}25$ der Messungen nach der Systemkonfiguration unter der Standardabweichung $s_1 = 10{,}86$ liegt, hat die im Beispiel angewendete Systemkonfiguration eine signifikante Verbesserung der Varianz hervorgebracht.

df_2 \ df_1	3	4	...	12	13	14
3	9.28	9.12	...	8.74	8.73	8.71
4	6.59	6.39	...	5.91	5.89	5.87
⋮	⋮	⋮	...	⋮	⋮	⋮
12	3.49	3.26	...	2.69	2.66	2.64
13	3.41	3.18	...	2.60	2.58	2.55
14	3.34	3.11	...	2.53	2.51	2.48

Tabelle 2.1.: Ausschnitt einer F-Tabelle mit Signifikanzniveau $\alpha = 0.05$.

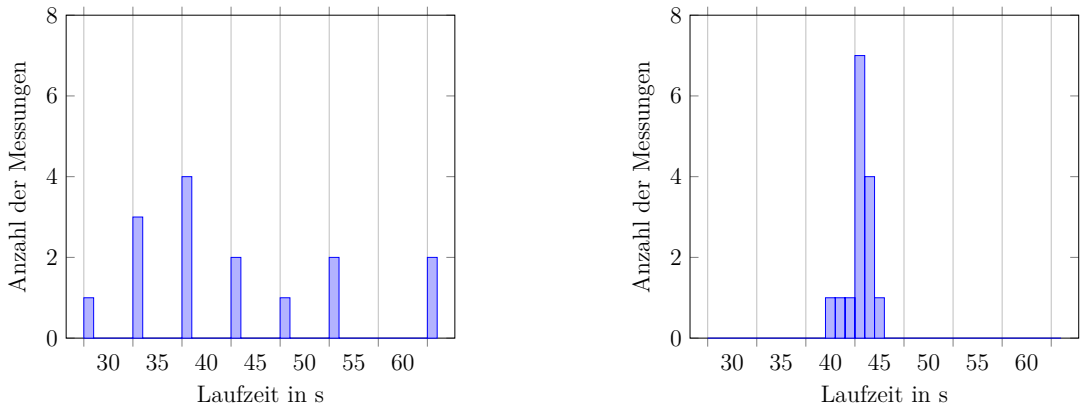


Abbildung 2.1.: Messung der Laufzeit einer Software vor der Systemkonfiguration (links) mit $s = 10,86$ und $s^2 = 117,86$ und nach der Systemkonfiguration (rechts) mit $s = 1,57$ und $s^2 = 1,25$.

2.2.3. Vergleich von Software durch Anwendung des t-Tests

Im folgenden Abschnitt wird verdeutlicht, warum die Minimierung der Varianz für den Vergleich von Software notwendig ist. In vielen Fällen wird Software nur mit Hilfe des arithmetischen Mittels verglichen. Dabei wird keine Angabe über die Varianz oder Standardabweichung gemacht. Dadurch hat der Leser der Ergebnisse keine Möglichkeit, die Aussagekraft der Werte zu prüfen. Das macht Gernot Heiser in seiner Arbeit über Verstöße bei Systembenchmarks deutlich: Unterschiedliche Werte von verschiedenen Computern ohne Angabe der Varianz haben keine Bedeutung [7]. Die Angabe der Standardabweichung oder Varianz ist jedoch auch nicht immer ausreichend [7]. Stattdessen verwendet man den Zweistichproben-t-Test [8]:

Während der F-Test angibt, ob sich zwei Varianzen signifikant unterscheiden, gibt der t-Test an, ob sich zwei arithmetische Mittel signifikant unterscheiden. Die Berechnung funktioniert wie folgt:

1) Seien die Werte mit Index 1 (s_1^2, n_1, $\overline{x}_1$) die Werte der Messreihe mit dem größeren arithmetischen Mittel und sei zudem eine Normalverteilung vorausgesetzt.

2) Berechne t-Wert: $t = \frac{\overline{x}_1 - \overline{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$.

3) Berechne Freiheitsgrad $df = n_1 + n_2 - 2$.

4) Wähle Signifikanzniveau α (z.B. $\alpha = 5\%$).

5) Lese kritischen Wert in einer t-Tabelle ab.

6) Wenn $t > kritischerWert$, dann ist das arithmetische Mittel signifikant unterschiedlich.

Die Berechnung zeigt: Je kleiner die Varianz, desto größer der t-Wert. Je größer der t-Wert, desto besser kann man zeigen, dass zwei arithmetische Mittel signifikant unterschiedlich sind. Wird die Varianz minimiert, so kann ein zuvor nicht erreichtes Signifikanzniveau erreicht werden und damit nachgewiesen werden, dass eine Softwareversion schneller als eine andere ist. Die Varianz zu reduzieren und damit einen besseren Vergleich zu ermöglichen, ist das Ziel dieser Arbeit.

Da aus der Berechnung klar wird, dass das Ergebnis des t-Tests stark von der Standardabweichung abhängt, werden häufig nur das arithmetische Mittel und die Standardabweichung verwendet, um einen Vergleich zu bewerten. Ist das arithmetische Mittel von Softwareversion A kleiner als das von Softwareversion B und die Standardabweichung beider Messungen in Relation zur Differenz beider arithmetischen Mittel sehr klein, kann man davon ausgehen, dass Softwareversion A schneller ist. Um sicherzugehen und die Signifikanz nachzuweisen, sollte jedoch nach Möglichkeit stets der t-Test oder ein vergleichbarer Test angewendet werden.

2.3. Optimierungsalgorithmen

Ein Optimierungsalgorithmus dient dazu die optimale Lösung für ein gegebenes Problem unter bestimmten Bedingungen zu finden [9]. Ein solches Problem ist zum Beispiel das Finden eines globalen Minimums einer Funktion. Das bedeutet, dass für eine mathematische Funktion $f(x)$ das x gesucht wird, für das $f(x)$ minimal wird. In dieser Arbeit werden Optimierungsalgorithmen verwendet, um aus einer Menge von Parametern für die Systemkonfiguration diejenigen auszusuchen, die die Varianz der Laufzeit von Software minimieren. Die im Rahmen dieser Arbeit entwickelte Software

verwendet Algorithmen, die auf dem Greedy-Algorithmus und Simulated Annealing-Algorithmus basieren. Diese werden in den folgenden Abschnitten erklärt.

2.3.1. Greedy Algorithmus

Ein Greedy-Algorithmus [10] entscheidet sich immer für die aus lokaler Sicht nächstbeste Teillösung. Er kann Entscheidungen nicht rückgängig machen und nur auf dem bereits Entschiedenen aufbauen. Dadurch findet er oft nur lokale Minima, was der Graph in Abbildung 2.2 veranschaulicht. Angenommen der Algorithmus startet an Punkt B, dann hat er die Möglichkeit im nächsten Schritt nach links oder nach rechts zu gehen. Der Greedy-Algorithmus entscheidet sich hier aus lokaler Sicht nach rechts zu gehen, da dort der Wert für $f(x)$ kleiner ist. Das führt dazu, dass der Algorithmus letztendlich bei Punkt A endet. Das eigentliche globale Minimum liegt aber bei Punkt D. Trotzdem werden Greedy-Algorithmen oft eingesetzt, da sie schnell konvergieren. Der Algorithmus 1 zeigt den Aufbau eines Greedy-Algorithmus.

Algorithmus 1 Greedy

```
1: x ← Startlösung
2: while x hat nicht getestete Nachbarn do
3:     y ← Nachbar von x
4:     if f(y) < f(x) then
5:         x ← y
```

Es gibt Algorithmen, die auf dem Greedy-Algorithmus basieren und zum Beispiel durch die Möglichkeit, Entscheidungen rückgängig zu machen, bessere Lösungen finden [10]. Ein Beispiel für diesen Ansatz ist Simulated Annealing.

2.3.2. Simulated Annealing

Der Grundgedanke für Simulated Annealing [11] stammt von Scott Kirkpatrick. Statt bei einem Minimierungsproblem immer nach unten zu gehen und dadurch in einem lokalen Minimum zu landen, sollte der Algorithmus manchmal auch kurzzeitig nach oben gehen, um danach wieder nach unten zu gehen und ein tieferliegendes Minimum zu finden:

„[...] [C]ontrolled uphill steps can also be incorporated in the search for a better solution.“ [12]

Simulated Annealing ist in Anlehnung an die Metallindustrie entstanden, in der man Metall erst hoch erhitzt und dann langsam abkühlen lässt. Im heißen Metall können

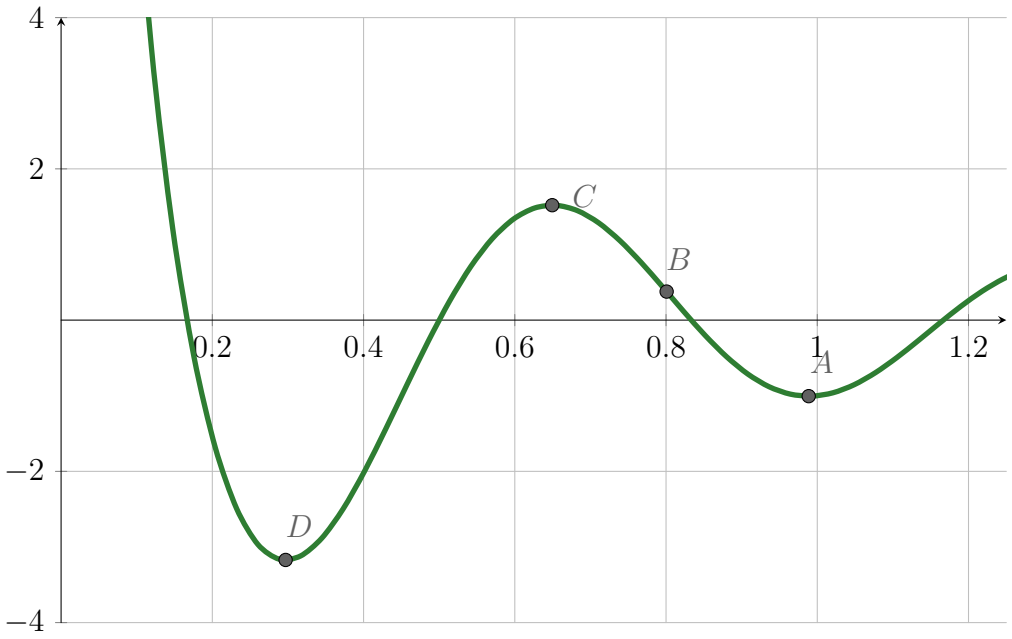


Abbildung 2.2.: Ausschnitt des Graphs von $f(x) = \frac{cos(3\pi x)}{x}$.

sich die Atome freier bewegen und ihre nicht optimalen Strukturen verlassen, um neue stabilere Strukturen mit anderen Atomen einzugehen. Wichtig ist dabei das Metall langsam abkühlen zu lassen, da die Atome Zeit benötigen, um die neuen Strukturen zu bilden. Diese Prozedur führt zu deutlich stabileren und flexibleren Metallen. Im Algorithmus von Simulated Annealing versucht man diese Idee auf die mathematische Optimierung zu übertragen. Befindet man sich in einem lokalen Minimum, so ist das im übertragenen Sinne eine nicht optimale Struktur. Im Algorithmus entspricht die Temperatur einer Wahrscheinlichkeit, die zu Beginn sehr hoch ist. Mit dieser Wahrscheinlichkeit werden auch schlechtere Teillösungen angenommen. So werden Maxima überwunden, um dahinterliegende Minima zu finden. Nach jeder Iteration wird die Wahrscheinlichkeit eine schlechtere Teillösung anzunehmen geringer. Dadurch wird das gegen Ende gefundene Minimum nicht wieder verlassen [13]. Im Algorithmus 2 ist dargestellt, wie ein Simulated Annealing-Algorithmus im Allgemeinen funktioniert.

Der Ablauf des Simulated Annealing-Algorithmus wird in Abbildung 2.2 veranschaulicht. Angenommen der Algorithmus startet bei Punkt B. Mit der anfänglich hohen Wahrscheinlichkeit auch schlechtere Teillösungen anzunehmen, kann der Algorithmus auch nach oben gehen. Anstatt direkt nach unten zu gehen und in Punkt A zu enden, kann der Simulated Annealing-Algorithmus das Maximum bei Punkt C überwinden und dann wieder nach unten gehen, um das Minimum bei Punkt D zu erreichen. Hier ist die Wahrscheinlichkeit eine schlechtere Teillösung anzunehmen dann so gering, dass der Algorithmus in Punkt D endet und nicht erneut nach oben geht.

Algorithmus 2 Simulated Annealing

1: $x \leftarrow$ *Startlösung*
2: *besteLösung* $\leftarrow x$
3: $t \leftarrow$ *Temperaturfunktion*
4: *temperatur* $\leftarrow$ *höchsteTemperatur*
5: **while** *temperatur* $>$ *Abbruchwert* **do**
6: $\quad y \leftarrow$ *Nachbar von* x
7: $\quad$ *zufallszahl* $\leftarrow$ *Zufallszahl aus* $[0.0, 1.0]$
8: $\quad$ **if** $f(y) < f(x)$ **then**
9: $\quad\quad x \leftarrow y$
10: $\quad$ **else if** $zufallszahl < exp(-\frac{f(y)-f(x)}{temperatur})$ **then**
11: $\quad\quad x \leftarrow y$
12: $\quad$ **if** $f(x) < f(besteLösung)$ **then**
13: $\quad\quad$ *besteLösung* $\leftarrow x$
14: $\quad$ *temperatur* $\leftarrow$ *t(temperatur)*

2.4. Benchmarks

Benchmarking dient dem Vergleich von Leistungen. In dieser Arbeit geht es um den Vergleich von Software-Performance. Dabei ist es essentiell, dass die Testumgebung stets gleich ist und keine äußeren Einflüsse die Ergebnisse beeinflussen können. Gernot Heiser beschreibt einen Verstoß dagegen als groben Fehler [7]. Es gibt fünf entscheidende Charakteristika, die Benchmarks und die Testumgebung erfüllen sollten [14]:

Relevanz
Der Benchmark sollte etwas Wichtiges aussagen.

Wiederholbarkeit
Der Benchmark sollte bei erneuter Ausführung dieselben Ergebnisse aufweisen.

Fairness
Jede Hardware oder Software, die verglichen wird, sollte gleichberechtigt am Test teilnehmen.

Überprüfbarkeit
Man sollte nachprüfen können, ob die dokumentierten Ergebnisse echt sind.

Ökonomie
Die Tester sollten sich das Ausführen des Benchmarks leisten können.

Nicht alle fünf Charakteristika sind immer zusammen erfüllbar. Laut Raghunath Nambiar und Meikel Poess schließen die vier letzten Merkmale oft das erste aus. Es lohnt sich nicht immer die Zeit für einen perfekten Benchmark zu investieren, da diese den eigentlichen Nutzen übersteigt. Stattdessen sollte man eine realistische Teilmenge der Charakteristika abdecken und der Zielgruppe verdeutlichen, in welchen Bereichen die Ergebnisse eingeschränkt sind und was bei der Interpretation zu beachten ist. Das genügt in den meisten Fällen, um der Zielgruppe die gewünschte Botschaft wissenschaftlich korrekt mitzuteilen [14]. Das Kapitel 5.8.3 der Evaluation zeigt, inwieweit die Charakteristika durch die Systemkonfiguration erfüllt werden.

Es gibt noch weitere Probleme bei der Auswahl und Vorbereitung von Benchmarks. Viele Benchmarks vergleichen Software nur in speziellen Aspekten, zum Beispiel ihrer Laufzeit. Die Systemkonfiguration dient in diesem Fall dazu, die Laufzeit der Programme zu stabilisieren. Dabei werden jedoch andere Aspekte der Software mit beeinflusst. Wird das automatische Übertakten des Prozessors, das in Kapitel 2.1 beschrieben wurde, deaktiviert, so verringert sich die Varianz und es steigert sich die Laufzeit des Programms. Viele Verbesserungen führen an anderen Stellen zu Verschlechterungen. Es gibt Benchmarks, die alle Aspekte abdecken, diese sind jedoch aufwendig und daher selten. Bei Benchmarks, die dies nicht tun, muss immer begründet werden, weshalb sie trotzdem realistisch sind, welche Annahmen man getroffen hat und welche Konfigurationen durchgeführt wurden [7].

2.5. Verwandte Arbeiten

Es gibt nur wenige Arbeiten, die sich mit dem Thema der Reduzierung der Standardabweichung der Laufzeit von Software beschäftigen. Die meisten Arbeiten gehen auf das Vorbereiten und das korrekte Ausführen von Benchmarks ein. Im Folgenden sind verwandte und ähnliche Arbeiten aufgeführt:

Das Tool *Pyperf* [15] von Victor Stinner verfolgt einen ähnlichen Ansatz wie *Autotune*. Beide Tools führen eine Systemkonfiguration durch. *Pyperf* hat dabei eine feste Auswahl an Betriebssystem- und Hardwareparametern, die mit einem Befehl auf dem eigenen Computer gesetzt werden können. *Autotune* verwendet im Vergleich zu *Pyperf* keine feste Auswahl an Parametern, sondern findet automatisiert die zur verwendeten Hardware und Software passenden Parameter. Die Evaluation zeigt nämlich, dass die Auswahl der Parameter, die die Standardabweichung reduzieren, von der Software und Hardware abhängig sind. Siehe dazu Kapitel 5.2. *Pyperf* kann des Weiteren Benchmarks für einen angegebenen Befehl ausführen und mit Hilfe von nicht gewerteten Vorläufen und mehreren Threads ein statistisch aussagekräftiges Ergebnis ausgeben.

In der Bachelorarbeit *Besser Benchmarken* [16] von Johannes Bechberger geht es auch um das Stabilisieren von Benchmarks. Das im Rahmen der genannten Arbeit entwickelte Tool *Temci* ermöglicht es wie *Pyperf*, verschiedene Betriebssystem- und Hardwareparameter zu setzen und Benchmarks so auszuführen, dass die Ergebnisse aussagekräftig sind. Auch hier gilt, dass die zur Verfügung stehenden Parameter im Vergleich zum Vorgehen von *Autotune* nicht an die Hardware und Software angepasst gesetzt werden. *Temci* bietet verschiedene Zeitmesswerkzeuge an und kann die Ergebnisse mit statistischen Werten und Graphen ausgeben. *Autotune* und *Temci* sind gut kombinierbar, indem man mit *Autotune* die Systemkonfiguration und dann mit *Temci* die Benchmarks durchführt.

In *Systems Benchmarking Crimes* [7] von Gernot Heiser werden häufige Fehler und Schwierigkeiten beim Benchmarken diskutiert. Dabei wird auf verschiedene Aspekte des Benchmarkens eingegangen und gezeigt, dass es unmöglich ist, einen perfekten Benchmark auszuführen. Des Weiteren werden generelle Tipps für die Systemkonfiguration gegeben.

Das Paper *Measuring and Understanding Variation in Benchmark Performance* [17] beschäftigt sich mit der Varianz der Laufzeit von Software. Dabei wird die Laufzeit zweier Programme auf drei verschiedenen Computern gemessen und untersucht, wieso die Ergebnisse schwanken und welche Faktoren Einfluss darauf haben. Die Messungen haben gezeigt, dass die größten Schwankungen durch Netzwerkschwankungen oder langsame Netzwerkinterfaces entstehen. Ebenfalls wird gezeigt, dass die Varianz auch vom verwendeten Computer und der Software abhängig ist. Dieser Aspekt wird von *Autotune* beachtet, indem die Systemkonfiguration individuell an die Software und Hardware angepasst wird, und im Rahmen dieser Arbeit vertieft.

Die Arbeit *Thread Tranquilizer: Dynamically Reducing Performance Variation* [18] behandelt den Einfluss von Speicherallokierung und Scheduling auf die Varianz der Laufzeit von Software. Dabei wird das Tool *Thread Tranquilizer* verwendet, welches Cache-Misses und Thread-Kontextwechsel misst und daraufhin die Speicherallokierung und den Scheduler so anpasst, dass diese reduziert werden. Dies führt wiederum zur Reduzierung der Varianz der Laufzeit der gebenchmarkten Software. *Autotune* reduziert Thread-Kontextwechsel durch das Setzen verschiedener Parameter.

3. Systemkonfiguration

Die Systemkonfiguration beschreibt das Einstellen von Hardware- und Softwareparametern, welche die Standardabweichung von Softwarelaufzeiten minimieren. In den folgenden Abschnitten werden die in *Autotune* verwendeten Parameter und ihre Auswirkungen vorgestellt. Für jeden Parameter wird dabei beispielhaft angegeben, wie man ihn auf einem Debian-basierten Betriebssystem konfigurieren kann. Einige Parameter müssen beim Start des Betriebssystems gesetzt werden. Unter Debian-basierten Betriebssystemen übernimmt der Bootloader *GRUB* den Start des Betriebssystems und kann, wie in den Beispielen beschrieben, so konfiguriert werden, dass er bestimmte Parameter beim Bootvorgang setzt.

3.1. CPU-Frequenz

Die CPU-Frequenz gibt die Geschwindigkeit des Prozessors an. Je höher die Frequenz, desto geringer die Laufzeit der Software, die auf dem Prozessor läuft. Wird die Frequenz dynamisch verändert, kommt es zu unterschiedlich langen Laufzeiten der gleichen Software. Um zu verhindern, dass eine softwareseitige Senkung der Frequenz vorgenommen wird, kann man für jeden Prozessorkern die minimale Frequenz einstellen und diese auf den maximalen Wert setzen. Trotzdem kann es vorkommen, dass der Prozessor selbst seine Frequenz senkt, wenn er zum Beispiel eine bestimmte Temperatur überschreitet, um eine Überhitzung zu verhindern.

Die minimale CPU-Frequenz kann in folgender Datei eingestellt werden:

`/sys/devices/system/cpu/cpuX/cpufreq/scaling_min_freq`

Hierbei steht X für den Prozessor, für den die Frequenz eingestellt werden soll. *Autotune* verwendet bei der Systemkonfiguration der minimalen Frequenz die maximale CPU-Frequenz aus der Datei `/sys/devices/system/cpu/cpuX/cpufreq/cpuinfo_max_freq`.

3.2. Governor

Der Governor [19] ist für das dynamische Setzen der CPU-Frequenz zuständig. Je nach Prozessormodell gibt es verschiedene einstellbare Stufen. Meistens kann zwischen „powersave“ für den energiesparenden Betrieb und „performance“ für volle Leistung gewählt werden. Dabei verändert der Governor die Frequenz, mit der der Prozessor arbeitet. Bei Computern mit mehreren Prozessorkernen kann der Governor für jeden Prozessorkern getrennt eingestellt werden. Durch das Wechseln der Frequenz während der Ausführung einer Software kann es zu einer erhöhten Standardabweichung der Laufzeit kommen. Das Einstellen des Governors auf „performance“ reduziert das ständige Wechseln der Frequenz und senkt damit die Standardabweichung. Des Weiteren wird damit auch eine kürzere Laufzeit erreicht.

Der Governor kann eingestellt werden indem einer der möglichen Werte aus Tabelle 3.1 in folgende Datei geschrieben wird:

`/sys/devices/system/cpu/cpuX/cpufreq/scaling_governor`

Hierbei steht X für den Prozessor, der eingestellt werden soll. *Autotune* verwendet den Parameter „performance“ für die Systemkonfiguration.

performance	Der Prozessor läuft meistens auf der maximalen Frequenz
powersave	Der Prozessor läuft meistens auf der minimalen Frequenz
userspace	Der Nutzer kann selbst eine Frequenz festlegen
ondemand	Die Frequenz passt sich an die Nutzung an
conservative	Ähnlich wie ondemand, jedoch mit feinerer Abstufung
schedutil	Der Scheduler bestimmt die CPU-Frequenz

Tabelle 3.1.: Mögliche Parameter für den Governor [19].

3.3. Turbo Boost

Die Turbo Boost-Technologie von Intel [20] ist für das dynamische Übertakten von Prozessoren zuständig. Prozessoren, die Turbo Boost unterstützen, haben eine minimale Frequenz, eine maximale Frequenz und eine Turbo Boost-Frequenz. Unter bestimmten Bedingungen wird der Prozessor kurzzeitig auf die Turbo Boost-Frequenz übertaktet. Das führt zu kurzzeitig schnelleren Ausführungszeiten und einer erhöhten Temperatur des Prozessors. Das kurzzeitig schnellere Ausführen der Software und die hohe Temperatur, die wiederum zu einer Senkung der Frequenz führt, haben beide eine höhere Standardabweichung der Laufzeit von Software zur Folge. Bei AMD-Prozessoren heißt die Turbo Boost-Technologie Turbo Core [21]. Turbo Boost und

Turbo Core können je nach Prozessor in einer der folgenden Dateien ausgeschaltet werden:

Für Intel-Prozessoren:

`/sys/devices/system/cpu/intel_pstate/no_turbo`

Die möglichen Parameter zum Einstellen des Turbo Boost bei Intel-Prozessoren sind in Tabelle 3.2 aufgeführt.

0	Turbo Boost ist eingeschaltet
1	Turbo Boost ist ausgeschaltet

Tabelle 3.2.: Mögliche Parameter für den Turbo Boost bei Intel-Prozessoren.

Für AMD-Prozessoren:

`/sys/devices/system/cpu/cpufreq/boost`

Die möglichen Parameter zum Einstellen des Turbo Core bei AMD-Prozessoren befinden sich in Tabelle 3.3.

0	Turbo Core ist ausgeschaltet
1	Turbo Core ist eingeschaltet

Tabelle 3.3.: Mögliche Parameter für den Turbo Boost bei AMD-Prozessoren.

3.4. Isolieren von Prozessorkernen

Die meisten modernen Prozessoren haben mehrere Prozessorkerne und können daher parallel verschiedene Software ausführen. Ein sogenannter „Scheduler“ [22] entscheidet dabei, welche Software auf welchem Prozessorkern wie lange läuft. Dabei wird Software während ihrer Ausführung unterbrochen und wechselt den Prozessorkern, auf dem sie ausgeführt wird. Das ständige Unterbrechen und Wechseln der Prozessorkerne führt zu einer hohen Standardabweichung in der Laufzeit, da die Anzahl und Länge der Unterbrechungen bei jeder Ausführung unterschiedlich sind. Durch das Isolieren von Prozessorkernen [15] kann man bestimmte Prozessorkerne von den übrigen trennen. Das bedeutet, dass Prozesse einer Software, die auf den isolierten Prozessorkernen laufen, nur zwischen den isolierten Prozessorkernen wechseln und nicht auf den übrigen Prozessorkernen ausgeführt werden können. Umgekehrt können Prozesse auf den übrigen Prozessorkernen nur zwischen diesen wechseln und nicht auf den isolierten Prozessorkernen ausgeführt werden. Wenn man nun auf den isolierten

Prozessorkernen nur eine Software mit einem Prozess ausführt, muss der Scheduler die Software nahezu nicht mehr unterbrechen. Das Reduzieren der Unterbrechungen führt zu einer deterministischeren Ausführung und damit zu einer niedrigeren Standardabweichung.

Im Folgenden werden beispielhaft die Prozessorkerne 0 und 1 isoliert. Diese Ziffern müssen durch die Nummern der zu isolierenden Prozessorkerne ersetzt werden. Um bestimmte Prozessorkerne zu isolieren und eine Software auf dem isolierten Bereich auszuführen, müssen folgende Schritte durchgeführt werden:

1) Kernelparameter `isolcpus=0,1` setzen. Dazu fügt man `isolcpus=0,1` zur Liste der Kernelparameter in der Datei `/etc/default/grub` hinzu.

2) `sudo update-grub` ausführen, um GRUB neu zu konfigurieren.

3) Computer neu starten.

4) Ein CPU-Set auf den isolierten Prozessorkernen erstellen und alle anderen Prozesse auf die übrigen Prozessorkerne verschieben.

5) Die gewünschte Software auf den isolierten Prozessorkernen starten.

Autotune isoliert immer den ersten Prozessorkern und alle dazugehörigen Simultaneous Multithreading-Kerne (3.5), um die Kontextwechsel zwischen den Prozessorkernen zu minimieren. Ist Simultaneous Multithreading eingeschaltet, gibt es noch Kontextwechsel zwischen den Simultaneous Multithreading-Kernen. Schaltet man SMT ab, finden diese Kontextwechsel auch nicht mehr statt.

3.5. Simultaneous Multithreading

Simultaneous Multithreading [23] ist eine Technologie, die es ermöglicht pro Prozessorkern mehrere Threads einer oder verschiedener Software parallel auszuführen. Benchmarks zeigen, dass Software mit mehreren Threads dadurch schneller ausgeführt wird [24]. Software mit nur einem Thread auf einem Prozessorkern ist jedoch langsamer, wenn Simultaneous Multithreading aktiviert ist. Siehe dazu Kapitel 5.4. Des Weiteren ist die Laufzeit durch das Teilen der Ressourcen der Prozessorkerne weniger deterministisch. Das Deaktivieren von Simultaneous Multithreading kann also je nach Software zu einer Reduzierung der Standardabweichung bezüglich der Laufzeit führen.

Simultaneous Multithreading kann in der folgenden Datei mit Hilfe der in Tabelle

3.4 gelisteten Parameter konfiguriert werden:

`/sys/devices/system/cpu/smt/control`

on	Simultaneous Multithreading ist eingeschaltet
off	Simultaneous Multithreading ist ausgeschaltet
notsupported	Simultaneous Multithreading wird nicht unterstützt
notimplemented	Simultaneous Multithreading ist nicht implementiert

Tabelle 3.4.: Mögliche Parameter für Simultaneous Multithreading.

Autotune prüft bei der Systemkonfiguration, ob Simultaneous Multithreading unterstützt wird und implementiert ist, und schaltet es dann mit dem Parameter `off` aus.

3.6. Perf

Perf [25] ist eine Software, die die Leistung der Hardware und Software eines Computers messen kann. Der Linux-Kernel benutzt Perf, um die Auslastung und Leistung des Computers in bestimmten Zeitintervallen zu überwachen, was auch für Softwareprofiling genutzt werden kann. Um die Messungen durchzuführen, wird der Prozessor vom Kernel in den festgelegten Zeitabständen unterbrochen. Auch wenn das Zeitintervall vom Kernel an die aktuelle Auslastung angepasst wird, um zu häufige Unterbrechungen zu vermeiden, kann das Unterbrechen zu einer höheren Standardabweichung der Laufzeit von Software führen. Um dies zu vermeiden, kann man eine maximale Anzahl an Messungen pro Sekunde festlegen. Der kleinste Wert, auf den man das Intervall setzen kann, ist $1\frac{sample}{second}$. Dadurch misst der Kernel nur noch einmal pro Sekunde die Auslastung und Leistung des Computers, was keine Nachteile mit sich bringt, solange man währenddessen kein Softwareprofiling vornehmen möchte.

Um das kleinstmögliche maximale Intervall einzustellen, muss man den Wert $1\frac{sample}{second}$ in folgende Datei schreiben:

`/proc/sys/kernel/perf_event_max_sample_rate`

Autotune verwendet für die Systemkonfiguration das kleinste Intervall von $1\frac{sample}{second}$. Der gesetzte Wert wird nach einem Neustart des Computers automatisch zurückgesetzt.

3.7. Address Space Layout Randomization

Die Address Space Layout Randomization [26], kurz ASLR, weist Software automatisch zufällige virtuelle Adressbereiche zu. Dies verhindert, dass die virtuellen Adressbereiche der Software direkt zugeordnet werden können und erschwert somit das Ausnutzen von Sicherheitslücken von Software. Durch das zufällige Zuordnen von Adressbereichen und den dadurch entstehenden Overhead folgt bei der Messung der Laufzeit eine höhere Standardabweichung, da die Adressbereiche und Zugriffszeiten nicht-deterministisch sind [27].

ASLR kann in drei Stufen eingestellt werden indem einer der Parameter aus Tabelle 3.5 in folgende Datei geschrieben wird:

`/proc/sys/kernel/randomize_va_space`

0	ASLR ist ausgeschaltet
1	ASLR randomisiert konservativ
2	ASLR randomisiert komplett

Tabelle 3.5.: Mögliche Parameter für ASLR [28].

Autotune schaltet bei der Systemkonfiguration ASLR mit dem Wert 0 komplett aus.

3.8. Software-Watchdogs

Software-Watchdogs [29] sind Teil des Betriebssystems und verhindern das Aufhängen des Systems durch zu hohe Auslastung der Hardware oder durch Deadlocks. Die Software-Watchdogs lösen in bestimmten Abständen eine Unterbrechung des Prozessors aus und können dadurch Fehlerzustände wie Deadlocks erkennen und beheben. Das Unterbrechen des Prozessors kann zu einer höheren Standardabweichung bei der Laufzeit von Software führen. Die Software-Watchdogs können wie folgt deaktiviert werden:

1) Kernelparameter `nowatchdog`, `nosoftlockup` und `nmi_watchdog=0` setzen. Dazu fügt man `nowatchdog nosoftlockup nmi_watchdog=0` zur Liste der Kernelparameter in der Datei `/etc/default/grub` hinzu.

2) `sudo update-grub` ausführen, um GRUB neu zu konfigurieren.

3) Computer neu starten.

Das Abschalten der Software-Watchdogs kann jedoch dazu führen, dass sich der Computer aufhängt oder abstürzt. *Autotune* deaktiviert die Watchdogs nur mit Zustimmung des Nutzers.

3.9. Read-Copy-Update

Read-Copy-Update (RCU) [30] ist ein Synchronisationsmechanismus des Linux-Kernels für paralleles Lesen und Schreiben von Software auf mehreren Prozessorkernen. Er stellt sicher, dass Daten, die von verschiedenen Prozessorkernen geschrieben und gelesen werden, immer aktuell und kohärent sind. Die RCU-Routinen zum Synchronisieren der Daten werden durch Unterbechnungen der Prozessorkerne aufgerufen. Diese Unterbrechungen können zu einer höheren Standardabweichung der Laufzeit der auf den Prozessorkernen laufenden Software führen. Die RCU-Routinen können für isolierte Prozessorkerne deaktiviert werden, damit diese nicht mehr unterbrochen werden. Dadurch werden die RCU-Routinen nur noch auf den übrigen Prozessorkernen ausgeführt.

Im Folgenden wird RCU beispielhaft für die Prozessorkerne 0 und 1 deaktiviert. Diese Werte müssen mit den Nummern der gewünschten Prozessorkerne ersetzt werden. Die Isolation dieser Prozessorkerne ist dabei vorausgesetzt.

1) Kernelparameter `rcu_nocbs=0,1` setzen. Dazu fügt man `rcu_nocbs=0,1` zur Liste der Kernelparameter in der Datei `/etc/default/grub` hinzu.

2) `sudo update-grub` ausführen, um GRUB neu zu konfigurieren.

3) Computer neu starten.

Autotune deaktiviert RCU immer auf genau den Prozessorkernen, die isoliert wurden.

3.10. Scheduling Clock Tick

Der Scheduling Clock Tick [31] gibt an, wie häufig der Scheduler den Prozessor unterbricht, um eine andere Software an die Reihe zu lassen. Dieses Unterbrechen führt je nach Auslastung des Prozessors zu einer hohen Standardabweichung der Laufzeit von Software. Läuft nur ein Thread einer Software auf einem Prozessorkern, ist es möglich, den Scheduler für diesen Prozessorkern abzuschalten. Durch die

Isolation von Prozessorkernen, die in Abschnitt 3.4 beschrieben wurde, kann man Software bis auf die zu messende Software auf die übrigen Prozessorkerne verschieben und dann den Scheduler deaktivieren. Dies funktioniert nur, wenn die Anzahl der Threads der zu messenden Software gleich der Anzahl der isolierten Prozessorkerne ist. Auf einem Prozessorkern darf nämlich nur ein Thread ausgeführt werden, um den Scheduler deaktivieren zu können. Die isolierten Prozessorkerne können trotz des deaktivierten Schedulers noch durch Hardware-Interrupts unterbrochen werden.

Im Folgenden wird der Scheduler beispielhaft für die Prozessorkerne 0 und 1 deaktiviert. Diese Werte müssen mit den Nummern der gewünschten Prozessorkerne ersetzt werden. Voraussetzung ist, dass die Prozessorkerne isoliert sind und nur eine Software auf ihnen läuft.

1) Kernelparameter `nohz_full=0,1` setzen. Dazu fügt man `nohz_full=0,1` zur Liste der Kernelparameter in der Datei `/etc/default/grub` hinzu.

2) `sudo update-grub` ausführen, um GRUB neu zu konfigurieren.

3) Computer neu starten.

Autotune deaktiviert den Scheduler immer auf genau den Prozessorkernen, die isoliert wurden.

3.11. Nice-Wert

Der Nice-Wert [32] gibt die Priorität einer Software an. Je niedriger der Nice-Wert einer Software, desto höher ist die Priorität der Software und desto mehr CPU-Zeit bekommt diese. Hat eine Software mehr CPU-Zeit, wird diese nicht so häufig durch den Scheduler unterbrochen. Weniger Unterbrechungen können zu einer niedrigeren Standardabweichung der Laufzeit der Software führen. Der Nice-Wert kann wie folgt angepasst werden:

```
sudo nice -n X "Befehl"
```

Dabei muss `"Befehl"` durch die gewünschte Software und `X` durch den gewünschten Nice-Wert ersetzt werden. Der Nice-Wert muss zwischen -20 und 19 liegen, wobei -20 für die höchste Priorität und 19 für die niedrigste Priorität steht. *Autotune* setzt den Nice-Wert immer auf -20.

3.12. IO-Nice-Wert

Der IO-Nice-Wert [33] gibt an, welche Priorität eine Software beim Zugriff auf den Hauptspeicher hat. Je höher die Priorität, desto kürzer sind die Wartezeiten der Software für den Zugriff auf den Hauptspeicher. Kürzere Wartezeiten können zu niedrigeren Standardabweichungen der Laufzeit von Software führen. Der IO-Nice-Wert kann wie folgt eingestellt werden:

```
sudo ionice -c X -n Y "Befehl"
```

IO-Nice unterscheidet zwischen Klassen, die hinter `-c` angegeben werden und Prioritäten, die hinter `-n` angegeben werden. Das heißt einer Software wird zuerst eine Klasse zugewiesen und dann eine Priorität innerhalb dieser Klasse. Mögliche Klassen und Prioritäten sind in Tabelle 3.6 gelistet. Beim oben gezeigten Befehl muss also `X` durch eine Klasse, `Y` durch eine Priorität und `"Befehl"` durch die auszuführende Software ersetzt werden. *Autotune* setzt den IO-Nice-Wert immer auf $X = 1$ und $Y = 0$.

Klasse	Priorität	Bedeutung
0	-	none (Software ist keiner Klasse zugeordnet).
1	0-7	real time (Software bekommt immer sofort Zugriff. Priorität gibt Größe des Zeitfensters an).
2	0-7	best-effort (Software in dieser Klasse mit gleichen Prioritäten bekommen Zugriff im Round-Robin-Verfahren).
3	-	idle (Software bekommt Zugriff nur, wenn keine andere Software den Zugriff anfordert).

Tabelle 3.6.: Mögliche Parameter für ionice.

3.13. ACPI CPU-Treiber

Intel Prozessoren werden unter Linux mit dem Intel P-State-Treiber [34] betrieben. Dieser Treiber kann den Prozessor in verschiedene Betriebsmodi versetzen, die entweder für Leistung oder für das Sparen von Strom ausgelegt sind. Wie auch beim Governor, der in Abschnitt 3.2 beschrieben wurde, führt ein Energiesparmodus zu hohen Standardabweichungen, bedingt durch das häufige Heruntertakten des Prozessors. Man kann den Intel P-State-Treiber ausschalten und stattdessen einen

Standardtreiber laden, der die Energiesparmodi nicht unterstützt und den Prozessor somit mit höchster Leistung betreibt. Der Intel P-State-Treiber kann wie folgt deaktiviert werden:

1) Kernelparameter `intel_pstate=disable` setzen. Dazu fügt man `intel_pstate=disable` zur Liste der Kernelparameter in der Datei `/etc/default/grub` hinzu.

2) `sudo update-grub` ausführen, um GRUB neu zu konfigurieren.

3) Computer neu starten.

3.14. Intel Power Save States deaktivieren

Prozessoren von Intel können vom Treiber in verschiedene Hardwarebetriebsmodi versetzt werden. Im Gegensatz zum Governor betreffen diese Betriebsmodi nicht nur die CPU-Frequenz, sondern auch die am Prozessor anliegende Spannung und das Flushen der Caches. Die Betriebsmodi sind in P-States [35] und C-States [36] unterteilt. Die P-States stehen für Leistung, wobei $P0$ die höchstmögliche Frequenz und Spannung bedeutet. Mit den C-States können verschiedene Sparstufen eingestellt werden. Das Wechseln dieser Betriebsmodi kann zu einer höheren Standardabweichung der Laufzeit der Software führen, die auf dem Prozessor ausgeführt wird. Um das Wechseln zu verhindern, kann man die P- und C-States deaktivieren, wodurch der Prozessor mit voller Spannung und Frequenz läuft:

1) Kernelparameter `idle=poll` setzen. Dazu fügt man `idle=poll` zur Liste der Kernelparameter in der Datei `/etc/default/grub` hinzu.

2) `sudo update-grub` ausführen, um GRUB neu zu konfigurieren.

3) Computer neu starten.

Das Deaktivieren der Power Save States ist vergleichbar mit der Nutzung des ACPI CPU-Treibers in Abschnitt 3.13. Während der ACPI CPU-Treiber den Prozessor vor allem auf die höchste Frequenz setzt, führt das Deaktivieren der Power Save States zusätzlich noch zum Erhöhen der Spannung. Dies wiederum führt dazu, dass der Prozessor beim Deaktivieren der Power Save States im Vergleich zur Nutzung des ACPI CPU-Treibers sehr heiß werden kann.

3.15. VM Stat Polling Interval

Das *VM Stat Polling Interval* [37] gibt an, wie häufig Statistiken über das Virtuelle Speicher-Management des Kernels abgefragt werden sollen. Je kleiner der Wert, desto häufiger wird laufende Software auf dem Prozessor unterbrochen, um die Statistiken abzufragen. Wird der Wert erhöht, kann die Standardabweichung der Laufzeit der Software, die auf dem Prozessor läuft, verringert werden.

Um das *VM Stat Polling Interval* zu reduzieren, muss der gewünschte Abstand in Sekunden, in dem die Statistiken abgefragt werden sollen, in folgende Datei geschrieben werden:

```
/proc/sys/vm/stat_interval
```

Autotune setzt den Wert auf 20 (Sekunden), um weniger Unterbrechungen des Prozessors und gleichzeitig ein funktionierendes Speichermanagement zu gewährleisten. Der verwendete Wert stammt aus dem Blog von Victor Stinner [38]. Die Evaluation in Kapitel A.1 zeigt jedoch, dass es schwierig ist nachzuweisen, welcher Wert den besten Effekt auf die Standardabweichung hat. In Kombination mit anderen Parametern hat der Wert von $20s$ jedoch einen Effekt.

3.16. Swap

Der Swap-Speicher ist ein Auslagerungsspeicher, der genutzt wird, um Daten aus dem Arbeitsspeicher auf den Hauptspeicher auszulagern, falls der Arbeitsspeicher voll ist. Wurden Daten ausgelagert, dauert es deutlich länger, die Daten wieder zu lesen, wenn diese von einer Software benötigt werden. Werden bei mehreren Ausführungen einer Software die Daten manchmal ausgelagert und manchmal nicht, kommt es zu einer höheren Standardabweichung der Laufzeit dieser Software. Der Swap-Speicher kann deaktiviert werden, wodurch keine Auslagerung von Daten mehr stattfinden kann. Dies kann jedoch zum Absturz des Computers führen, wenn der Arbeitsspeicher voll ist. Der Swap-Speicher lässt sich mit folgendem Befehl deaktivieren:

```
swapoff -a
```

3.17. Deaktivieren von Prozessorkernen

Bei Computern mit mehreren Prozessorkernen kann Software parallel ausgeführt werden. Dieses parallele Ausführen ist jedoch nicht so deterministisch wie das sequenzielle Ausführen auf nur einem Prozessorkern und kann zu einer höheren Standardabweichung der Laufzeit von Software führen. Um auf einem Mehrkernprozessor alle Prozesse auf nur einem Prozessorkern auszuführen, kann man die übrigen Prozessorkerne wie folgt abschalten:

Schreibe den Wert 0 in folgende Datei, wobei X für die Nummer des Prozessorkerns steht, der deaktiviert werden soll:

```
/sys/devices/system/cpu/cpuX/online
```

Der Prozessorkern 0 kann nicht abgeschaltet werden, es müssen daher alle Prozessorkerne > 0 deaktiviert werden.

4. Entwurf und Implementierung

Das im Rahmen dieser Arbeit in Swift entwickelte Tool *Autotune* findet für jeden Computer die Betriebssystemparameter und Hardwareparameter, die die Standardabweichung der Laufzeit von Software auf diesem Computer minimieren. Dazu verwendet *Autotune* verschiedene Algorithmen, die die Parameter testweise setzen, die Standardabweichung vergleichen und somit die besten Parameter finden. Der Aufbau von *Autotune* wird in Abschnitt 4.1 beschrieben. Die verwendeten Algorithmen werden in Abschnitt 4.2 beschrieben. Die Befehle und Optionen, mit denen man *Autotune* aufrufen kann, werden in Abschnitt 4.3 erläutert.

4.1. Architektur

Autotune ist in drei Teile untergliedert: Den Client, den Server und den Core. Der Client kann auf einem Computer gestartet werden, um entweder lokal auf diesem Computer nach den besten Parametern zu suchen, oder um auf einem anderen Computer den Server zu starten, um dort nach Parametern zu suchen. Im folgenden wird der Computer, auf dem der Client läuft, Kontrollrechner und der Computer, auf dem der Server läuft, Benchmarkrechner genannt. In den folgenden Abschnitten werden die drei Teile von *Autotune* näher beschrieben. Des Weiteren zeigt Abbildung 4.1 den Zusammenhang der einzelnen Teile.

4.1.1. Core

Der Core ist der Teil von *Autotune*, der dem Client und Server benötigte Funktionen und Informationen bereitstellt. Dazu gehört vor allem das Ausführen von Benchmarks und das Setzen und Zurücksetzen der Parameter. Außerdem stellt der Core auch die Schnittstelle zum Nutzer bereit und gibt die Ergebnisse zurück. Somit läuft der Core sowohl auf dem Kontrollrechner als auch auf dem Benchmarkrechner.

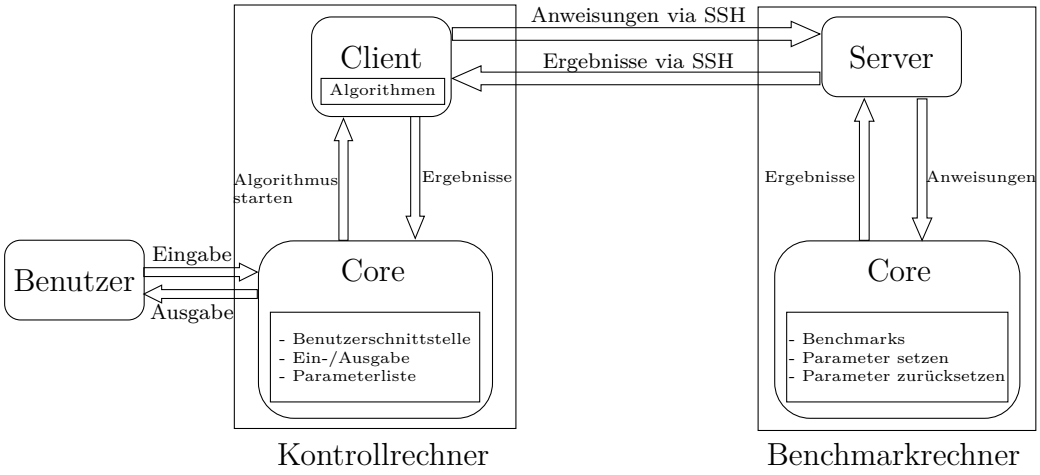


Abbildung 4.1.: Architektur von *Autotune*.

4.1.2. Client

Der Client wird auf dem Kontrollrechner gestartet, sobald *Autotune* gestartet wird. Der Client beinhaltet die beiden Algorithmen, die in Abschnitt 4.2 erklärt werden, und gibt je nach Modus dem Core auf dem Kontrollrechner oder dem Server auf dem Benchmarkrechner Anweisungen, um zum Beispiel die Benchmarks zu starten. Man kann den Client in zwei verschiedenen Modi starten, die im Folgenden beschrieben werden.

Lokaler Modus

Im lokalen Modus werden die von den Algorithmen gewählten Parameter und die Benchmarks auf dem Kontrollrechner gesetzt und ausgeführt. Bei diesem Modus ist zu beachten, dass kein Neustart des Computers während der Ausführung der Algorithmen möglich ist. Daher werden nur Parameter getestet, die keinen Neustart benötigen. Dies führt je nach Computer nicht zur minimalen Standardabweichung. Der Vorteil liegt darin, dass man nur einen Computer und keine Netzwerkverbindung benötigt. Der Benchmarkrechner ist in diesem Fall der gleiche Computer wie der Kontrollrechner.

Entfernter Modus

Im entfernten Modus werden die Parameter und Benchmarks auf dem Benchmarkrechner ausgeführt. Dazu muss auf dem Benchmarkrechner ein SSH-Server mit

Root-Login laufen und dieser muss vom Kontrollrechner aus erreichbar sein. Außerdem muss *Autotune* installiert sein. In diesem Modus gibt der Client auf dem Kontrollrechner dem Server auf dem Benchmarkrechner die Anweisungen. Der Server auf dem Benchmarkrechner dient dabei nur als Netzwerkschnittstelle und gibt die Anweisungen an den Core auf dem Benchmarkrechner weiter. Dies hat den Vorteil, dass der Benchmarkrechner neu gestartet werden kann, während der Algorithmus auf dem Kontrollrechner weiter ausgeführt wird. Diese Variante sollte angewandt werden, wenn alle Parameter in den Test miteinbezogen und die minimale Standardabweichung erreicht werden sollen.

4.1.3. Server

Der Server auf dem Benchmarkrechner wird durch den Client auf dem Kontrollrechner gestartet. Der Server ist dabei eine Schnittstelle, die Befehle vom Client entgegennimmt und an den Core weitergibt. Ergebnisse werden umgekehrt vom Core an den Server weitergegeben, der diese dann an den Client auf dem Kontrollrechner sendet. Der Server selbst hat außer der Möglichkeit, den Benchmarkrechner neu zu starten, keine eigene Funktionalität und gibt die Anweisungen, die er vom Kontrollrechner bekommt, an den Core weiter. Damit der Server vom Client gestartet werden kann, müssen *Autotune* und alle Abhängigkeiten von *Autotune* auf dem Benchmarkrechner installiert sein.

4.2. Algorithmen

Autotune bietet zwei verschiedene Algorithmen an, die zum Finden der besten Parameter für die minimale Standardabweichung der Laufzeit von Software verwendet werden können. Diese Algorithmen werden in den folgenden zwei Abschnitten erläutert.

Simulated Annealing

Der erste Algorithmus basiert auf dem Prinzip von Simulated Annealing, welches in Kapitel 2.3.2 beschrieben wurde. Dieser Algorithmus nimmt die Menge aller möglichen Betriebssystemparameter und Hardwareparameter und geht die im folgenden Codeabschnitt beschriebenen Schritte durch:

Algorithmus 3 Autotune Simulated Annealing

```
1:  x ← Startlösung ohne Parameter
2:  xStandardabweichung ← 100.0
3:  besteLösung ← x
4:  besteLösungStandardabweichung ← 100.0
5:  t ← 0.75
6:  temperatur ← 100.0
7:  parameterListe ← möglicheParameter.zufälligeReihenfolge()
8:  while parameterListe.nichtLeer() do
9:      parameter ← parameterListe.pop()
10:     y ← x + parameter
11:     y.setzen()
12:     yStandardabweichung ← benchmarkAusführen()
13:     zufallszahl ← Zufallszahl aus [0.0, 1.0]
14:     if yStandardabweichung < xStandardabweichung then
15:         x ← y
16:         xStandardabweichung ← yStandardabweichung
17:     else if zufallszahl < exp(−(yStandardabweichung−xStandardabweichung)/temperatur) then
18:         x ← y
19:         xStandardabweichung ← yStandardabweichung
20:     if xStandardabweichung < besteLösungStandardabweichung then
21:         besteLösung ← x
22:         besteLösungStandardabweichung ← xStandardabweichung
23:     temperatur ← t ∗ temperatur
    return besteLösung, besteLösungStandardabweichung
```

Optimierter Algorithmus

Der zweite Algorithmus basiert auf dem Greedy-Algorithmus, der in Kapitel 2.3.1 beschrieben wurde, und ist zusätzlich optimiert, um das Finden der besten Betriebssystemparameter und Hardwareparameter zu beschleunigen. Dazu setzt der Algorithmus zu Beginn vier essentielle Parameter, die *Autotune* in den meisten Fällen mit dem Simulated Annealing-Algorithmus auch setzt, siehe Kapitel 5.2.2:

- Den Governor, siehe Kapitel 3.2.
- Die CPU-Frequenz, siehe Kapitel 3.1.
- Den Nice-Wert, siehe Kapitel 3.11.
- Den IO-Nice-Wert, siehe Kapitel 3.12.

Zudem verlängern diese Parameter nicht die Laufzeit der Software und können so ohne negative Folgen zu Beginn gesetzt werden. Danach unterteilt der Algorithmus die restlichen Parameter in drei Gruppen:

Isolation Parameter, die von der Isolation der Prozessorkerne abhängig sind.
Dazu gehört: *Isolation von Prozessorkernen, Deaktivierung von RCU* und *Deaktivierung des Schedulers*.

Normal Parameter, die man ohne Abhängigkeiten setzen kann.
Dazu gehört: *VM Stat Polling Interval, Deaktivierung von SMT, Deaktivierung von Turbo Boost, Perf, ASLR, Deaktivierung des Intel P-State-Treibers* und Deaktivierung des Swaps.

Gefährlich Parameter, die die Systemleistung und -stabilität negativ beeinflussen können.
Dazu gehört: *Deaktivierung von Prozessorkernen, Deaktivierung von Watchdogs* und Deaktivierung der Intel Power Save States.

Der Algorithmus beachtet außerdem einen vom Nutzer vorher festgelegten Schwellenwert für die Standardabweichung. Wird der Schwellenwert unterschritten, bricht der Algorithmus ab und gibt die bis dahin gefundenen Parameter aus, mit denen dieser Wert erreicht wurde. Der optimierte Algorithmus geht die oben genannten Kategorien der Reihenfolge nach durch. Ist der Schwellenwert nach einer Kategorie unterschritten, wird abgebrochen. Andernfalls macht der Algorithmus mit der nächsten Kategorie weiter. Die gefährlichen Parameter werden nur getestet, falls der Nutzer dem zu Beginn ausdrücklich zugestimmt hat. Das Vorgehen ist im Codeausschnitt 4 nochmals genauer beschrieben.

In Abbildung 4.2 ist dargestellt, wie die Standardabweichung im Verlauf des optimierten Algorithmus reduziert wird. Da immer nur Parameter zur Lösung hinzugefügt werden, die die Standardabweichung im Vergleich zu den vorherigen Parametern reduzieren, fällt der Graph stetig.

Der Graph in Abbildung 4.3 zeigt den Verlauf des Simulated Annealing-Algorithmus. Hier sieht man, wie der Algorithmus nicht nur Parameter aussucht, die die Standardabweichung weiter reduzieren als die vorherigen. So wurde zum Beispiel Parameter B mitaufgenommen, obwohl er eine höhere Standardabweichung bewirkt. Parameter G kennzeichnet die beste Lösung. Diese beinhaltet alle blau markierten Parameter und Parameter G. Die rot gekennzeichneten Parameter wurden verworfen, da kein kleineres Minimum als Parameter G gefunden wurde.

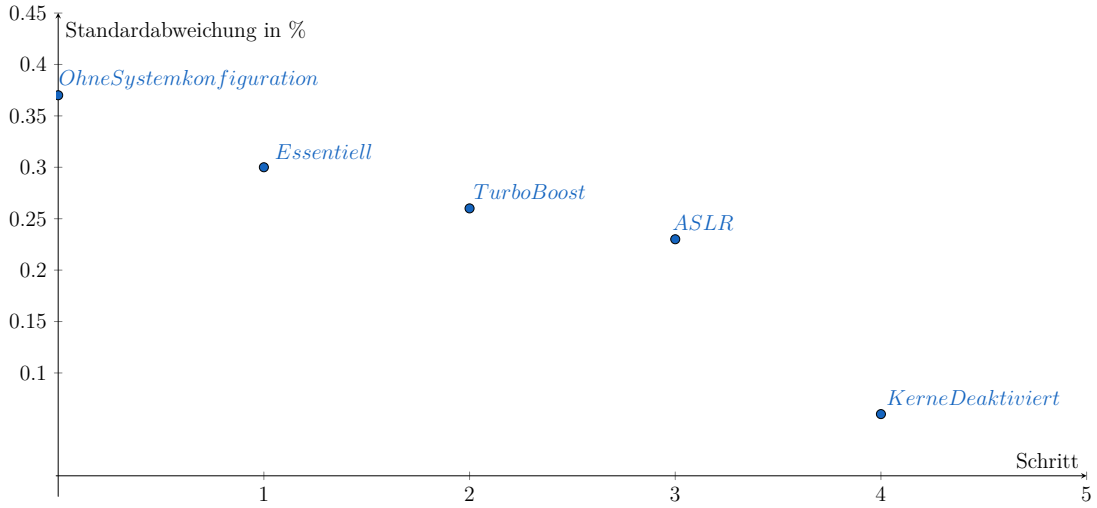


Abbildung 4.2.: Beispielverlauf des optimierten Algorithmus.

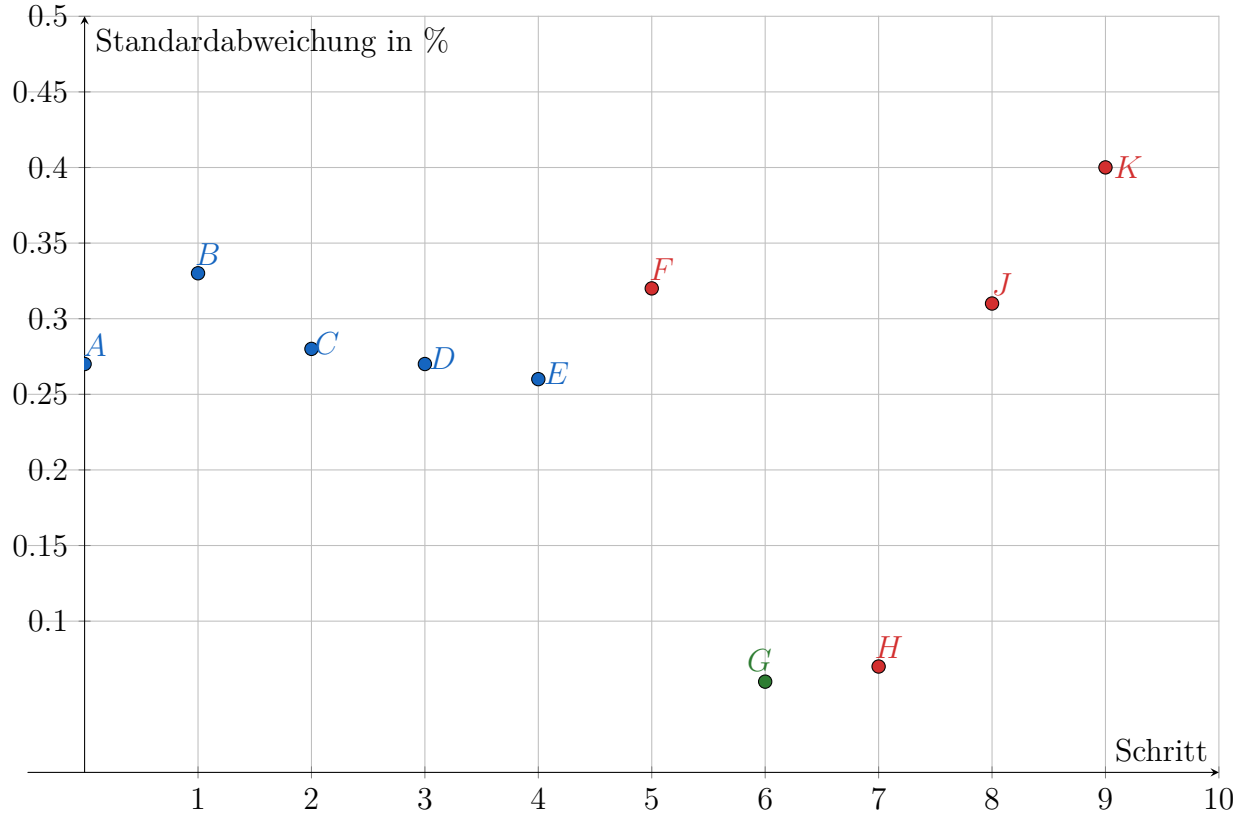


Abbildung 4.3.: Beispielverlauf des Simulated Annealing-Algorithmus.

4.3. Nutzung

Autotune wird über die Kommandozeile gestartet und kann entweder über eine Konfigurationsdatei oder mit Hilfe von Befehlszeilenargumenten konfiguriert werden. In Tabelle 4.1 sind die möglichen Befehlszeilenargumente gelistet.

`--runs`	Anzahl der Benchmarks pro Schleifendurchlauf (siehe `--loopCount`).
`--valuesPerRun`	Anzahl der Benchmarks, die jeder Pyperf-Thread durchführen soll (nur bei Benutzung von *Pyperf*).
`--optimizeForCommand`	Festlegen des Befehls, für den die Systemkonfiguration durchgeführt werden soll (nur bei Benutzung von *Pyperf*).
`--simulatedAnnealing`	Führt den Simulated Annealing-Algorithmus anstatt des optimierten Algorithmus aus.
`--preserveMulticore`	Deaktiviert alle Parameter, die die Anzahl der Kerne auf 1 beschränkt.
`--verbose`	Im Verbose-Modus gibt *Autotune* mehr Informationen aus.
`--reset`	Setzt die zuvor gesetzten Parameter auf den ursprünglichen Zustand zurück (kombinierbar mit `--local` oder `--remote`).
`--warmups`	Anzahl der Schleifendurchgänge, die nicht gewertet werden.
`--loopCount`	Anzahl der Schleifendurchgänge, die gewertet werden.
`--fast`	Der Fast-Modus beschleunigt den optimierten Algorithmus, indem `--valuesPerRun 1`, `--runs 5`, `--warmups 0` und `--loopCount 1` gesetzt werden.
`--benchmarkingTool`	Wahl zwischen *Phoronix* und *Pyperf* als Benchmarkingtool.
`--cpuBound`	Führt die Systemkonfiguration für CPU-bound Software durch (nur in Kombination mit Phoronix).
`--memoryBound`	Führt die Systemkonfiguration für Memory-bound Software durch (nur in Kombination mit Phoronix).
`--createConfig`	Erstellt eine Konfigurationsdatei mit Standardwerten im Homeverzeichnis des Nutzers.
`--remote`	Führt *Autotune* im entfernten Modus aus.
`--local`	Führt *Autotune* im lokalen Modus aus.
`--load`	Lädt eine zuvor generierte Ergebnisdatei.

Tabelle 4.1.: Befehlszeilenargumente für *Autotune*.

Algorithmus 4 Optimierter Algorithmus

```
1: lösung ← Startlösung ohne Parameter
2: lösungStandardabweichung ← benchmarkAusführen()
3: schwellenwert ← NutzerFragen()
4: sollGefährlicheParameterTesten ← NutzerFragen()
5: essentielleParameterListe ← essentielleParameter
6: isolationsParameterListe ← isolationsParameter.zufälligeReihenfolge()
7: normaleParameterListe ← normaleParameter.zufälligeReihenfolge()
8: gefährlicheParameterListe ← gefährlicheParameter.zufälligeReihenfolge()
9: for parameter in essentielleParameterListe do
10:     parameter.setzen()
11: isolation.setzen()
12: isolationStandardabweichung ← benchmarkAusführen()
13: if isolationStandardabweichung < xStandardabweichung then
14:     lösung ← lösung + isolation
15:     lösungStandardabweichung ← isolationStandardabweichung
16:     for parameter in isolationsParameterListe do
17:         parameter.setzen()
18:         parameterStandardabweichung ← benchmarkAusführen()
19:         if parameterStandardabweichung < lösungStandardabweichung then
20:             lösung ← lösung + parameter
21:             lösungStandardabweichung ← parameterStandardabweichung
22:         else
23:             parameter.zurücksetzen()
24: if lösungStandardabweichung <= schwellenwert then
25:     return lösung, lösungStandardabweichung
26: for parameter in normaleParameterListe do
27:     parameter.setzen()
28:     parameterStandardabweichung ← benchmarkAusführen()
29:     if parameterStandardabweichung < lösungStandardabweichung then
30:         lösung ← lösung + parameter
31:         lösungStandardabweichung ← parameterStandardabweichung
32:     else
33:         parameter.zurücksetzen()
34: if lösungStandardabweichung <= schwellenwert then
35:     return lösung, lösungStandardabweichung
36: if sollGefährlicheParameterTesten then
37:     for parameter in gefährlicheParameterListe do
38:         parameter.setzen()
39:         parameterStandardabweichung ← benchmarkAusführen()
40:         if parameterStandardabweichung < lösungStandardabweichung then
41:             lösung ← lösung + parameter
42:             lösungStandardabweichung ← parameterStandardabweichung
43:         else
44:             parameter.zurücksetzen()
    return lösung, lösungStandardabweichung
```

5. Evaluation

Dieses Kapitel zeigt, welchen Effekt die Systemkonfiguration auf die Standardabweichung der Laufzeit verschiedener Software hat. Dabei werden verschiedene Parameter auf verschiedenen Computern gesetzt und getestet. Des Weiteren wird verglichen, wie weit sich die Standardabweichung mit *Autotune* im Vergleich zu anderen Vorgehen zur Stabilisierung von Benchmarks reduzieren lässt. Einige Benchmarks wurden nur auf einem der zur Verfügung stehenden Computern ausgeführt. Abschnitt 5.8.2 zeigt jedoch, dass die Ergebnisse von einem Computer auf andere übertragbar und damit repräsentativ sind.

5.1. Verwendete Hardware und Software

In diesem Abschnitt wird beschrieben, mit welchen Computern und mit welcher Software die Benchmarks in den darauffolgenden Abschnitten durchgeführt wurden.

5.1.1. Verwendete Hardware

Um zu untersuchen, wie sich die Betriebssystem- und Hardwareparameter auf verschiedener Hardware auf die Standardabweichung der Laufzeit von Software auswirken, wurden verschiedene Computer für die Benchmarks verwendet. Unter den verwendeten Computern befinden sich sowohl Laptops als auch Tower-PCs mit verschiedenen Prozessoren und Festplatten. Dies ermöglicht es zu evaluieren, welchen Einfluss die Hardware auf die Standardabweichung der Laufzeit von Software hat und wie weit diese auf den verschiedenen Computern reduziert werden kann. Die verwendete Hardware ist in Abbildung 5.1 aufgeführt.

Computer A	
Hersteller	LENOVO
Modell	ThinkPad T420s
Art	Laptop
Prozessor	Intel Core i5-2520M
Min. Frequenz	0,80GHz
Max. Frequenz	2,50GHz
Turbo Boost	3,20GHz
Anzahl Kerne	2
SMT	Ja
Arbeitsspeicher	8GB
Hauptspeicher	150GB SSD

Computer B	
Hersteller	FUJITSU
Modell	ESPRIMO P910
Art	Tower
Prozessor	Intel Core i7-3770
Min. Frequenz	1,60GHz
Max. Frequenz	3,40GHz
Turbo Boost	3,90GHz
Anzahl Kerne	4
SMT	Ja
Arbeitsspeicher	32GB
Hauptspeicher	500GB HDD

Computer C	
Hersteller	HP
Modell	285 G3 MT
Art	Tower
Prozessor	AMD Ryzen 5 2400G
Min. Frequenz	1,60GHz
Max. Frequenz	3,60GHz
Turbo Boost	3,90GHz
Anzahl Kerne	4
SMT	Ja
Arbeitsspeicher	6GB
Hauptspeicher	240GB SSD

Computer D	
Hersteller	FUJITSU
Modell	ESPRIMO P956
Art	Tower
Prozessor	Intel Core i7-6700
Min. Frequenz	0,80GHz
Max. Frequenz	3,40GHz
Turbo Boost	4,00GHz
Anzahl Kerne	4
SMT	Ja
Arbeitsspeicher	64GB
Hauptspeicher	1TB SSD

Computer E	
Hersteller	Apple
Modell	MacBook Pro 8,2
Art	Laptop
Prozessor	Intel Core i7-2720QM
Min. Frequenz	0,80GHz
Max. Frequenz	2,20GHz
Turbo Boost	3,30GHz
Anzahl Kerne	4
SMT	Ja
Arbeitsspeicher	16GB
Hauptspeicher	1TB SSD

Computer F	
Hersteller	Dell
Modell	Vostro 1710
Art	Laptop
Prozessor	Intel Celeron 550
Min. Frequenz	2,00GHz
Max. Frequenz	2,00GHz
Turbo Boost	Nein
Anzahl Kerne	1
SMT	Nein
Arbeitsspeicher	3GB
Hauptspeicher	150GB HDD

Abbildung 5.1.: Für die Evaluation verwendete Hardware.

5.1.2. Verwendete Software

Auf den im vorherigen Abschnitt aufgeführten Computern war zum Zeitpunkt der Benchmarks und der Ausführung von *Autotune* Ubuntu 19.10 als Betriebssystem

installiert. Folgende Software wurde verwendet, um die Werte und Statistiken in den folgenden Abschnitten zu messen und zu berechnen:

Benchmarkingtools

- Pyperf [2]
- Phoronix-Test-Suite [39]

Gebenchmarkte Software

HMMER [40]: CPU-bound Software mit Unterstützung für Multithreading. Die Anzahl der Threads ist gleich der Anzahl der Kerne.

Cachebench [41]: Memory-bound Software mit einem Thread.

ImageMagick [42]: Kompiliert das Tools ImageMagick. Dieser Benchmark wird verwendet, um die Auswirkungen der Systemkonfiguration auf das Kompilieren von Software zu evaluieren.

Encode MP3 [43]: Enkodiert eine MP3-Datei. Das Enkodieren von Musik- und Videodateien ist CPU-bound und sehr zeitaufwändig. Deshalb ist es wichtig, Software, die mit Enkodierung arbeitet, effizienter und schneller zu machen.

Git [44]: Führt verschiedene Git-Befehle aus. Dieser Benchmark ist Teil der Phoronix-Test-Suite und durch die weite Verbreitung von Git interessant zu evaluieren.

5.2. Auswirkungen der Systemkonfiguration mit Autotune

In diesem Abschnitt werden Auswirkungen der Systemkonfiguration auf die Standardabweichung der Laufzeit von Software im Detail betrachtet. Es werden außerdem die beiden von *Autotune* verwendeten Algorithmen verglichen und untersucht, wie sich die Systemkonfiguration auf verschiedenen Computern im Vergleich auswirkt.

5.2.1. Auswirkungen der einzelnen Parameter

Dieser Abschnitt zeigt, welchen Effekt die einzelnen, nicht kombinierten Betriebssystemparameter und Hardwareparameter auf die Laufzeit und die Standardabweichung der Laufzeit von Software haben. Für die Messung wurden der Computer A und der Benchmark *HMMER* aus der *Phoronix-Test-Suite* verwendet. HMMER ist repräsentativ für CPU-bound Software, wie Abschnitt 5.2.4 zeigen wird. Es wurden zehnmal 15 Benchmarks pro Parameter durchgeführt. Aus den zehn resultierenden Werten für die Standardabweichung und Laufzeit wurden die mittlere Standardabweichung und mittlere Laufzeit berechnet, welche in Tabelle 5.1 für alle Parameter angegeben sind.

Parameter	Standardabweichung	Laufzeit
Ohne Parameter	5,76%	38,9s
Prozessorkerne isoliert	3,10%	46,7s
Scheduler deaktiviert	2,22%	49,3s
CPU-Frequenz	2,52%	46,6s
Perf	3,01%	45,0s
Governor	3,09%	43,8s
Swap deaktiviert	3,39%	43,2s
Turbo Boost deaktiviert	3,71%	42,9s
VM Stat Polling Interval	3,86%	42,6s
Nice-Wert	4,07%	42,3s
Intel Power Save States deaktiviert	3,73%	42,6s
RCU deaktiviert	3,46%	43,8s
SMT deaktiviert	3,25%	44,4s
Prozessorkerne deaktiviert	3,02%	48,8s
IO-Nice-Wert	3,23%	48,3s
Watchdogs deaktiviert	3,26%	47,9s
ASLR	3,34%	47,6s
Intel P-State Treiber deaktiviert	3,36%	47,4s

Tabelle 5.1.: Effekt der einzelnen Parameter auf die Standardabweichung der Laufzeit von HMMER auf dem Computer A.

Betrachtet man die Werte aus Tabelle 5.1, fällt auf, dass das Setzen einzelner Parameter nur eine kleine Reduzierung der Standardabweichung bewirkt. Die Parameter führen nur in Kombination zu einer starken Reduzierung der Standardabweichung, wie in den Abschnitten 5.2.2 und 5.2.3 zu sehen ist. Einige der Parameter haben neben dem positiven Effekt auf die Standardabweichung einen negativen Effekt auf die Systemleistung und damit auch auf die Laufzeit der Software. Wie in Tabelle 5.1 erkennbar ist, zählen dazu vor allem das Deaktivieren und Isolieren von Prozessorkernen. Wie weit die Standardabweichung ohne Verwendung dieser Parameter reduziert werden kann, wird in Kapitel 5.4 beschrieben.

5.2.2. Vergleich der Algorithmen

Das Tool *Autotune* findet automatisiert die Betriebssystem- und Hardwareparameter, die kombiniert gesetzt die Standardabweichung der Laufzeit einer Software auf ihr Minimum reduzieren. In diesem Abschnitt werden für die beiden verwendeten Algorithmen die gefundenen Kombinationen von Parametern und die daraus resultierenden Werte für die Standardabweichung verglichen.

	Simulated Annealing	Optimierter Algorithmus
Minimale Standardabw.	0,06%	0,07%
Maximale Standardabw.	0,12%	0,23%
Mittlere Standardabw.	0,08%	0,11%
rel. Standardabw. der Standardabw.	21%	52%
Durchlaufzeit Algorithmus	38h	28h
Anzahl Benchmarks pro Parameter	$3 * 15$	$3 * 15$
Nicht gewertete Probeläufe	$1 * 15$	$1 * 15$

Tabelle 5.2.: Vergleich der für HMMER erreichten Werte der beiden von *Autotune* verwendeten Algorithmen auf dem Computer A.

Die Tabelle 5.2 zeigt, dass der Simulated Annealing-Algorithmus die Standardabweichung weiter reduzieren kann als der optimierte Algorithmus. Er braucht jedoch durchschnittlich zehn Stunden länger. Die Abbildung 5.2 macht den Unterschied der durchschnittlich erreichten Standardabweichung deutlich. Die Betriebssystemparameter und Hardwareparameter, die zu der minimalen Standardabweichung führen, sind nicht immer dieselben. Mehrere Durchläufe haben gezeigt, dass unterschiedliche Kombinationen zu dem gleichen oder einem ähnlichen Ergebnis bezüglich der Standardabweichung führen, was Tabelle 5.3 zeigt. Die Werte dazu wurden auf Computer A gemessen. Die Ergebnisse auf den anderen Computern sind ähnlich, sie unterscheiden sich nur im minimal erreichbaren Wert für die Standardabweichung, was in Abschnitt 5.2.3 gezeigt wird.

Tabelle 5.4 listet das jeweils beste Ergebnis der beiden Algorithmen auf. Hier überschneiden sich einige Parameter vor allem deswegen, weil der optimierte Algorithmus auf Grundlage der Ergebnisse des Simulated Annealing-Algorithmus entwickelt wurde. In Tabelle 5.3 werden die Parameter *Nice-Wert, CPU-Frequenz, IO-Nice-Wert und Governor* in den meisten Durchläufen genutzt. Diese Parameter werden im optimierten Algorithmus ohne das vorherige Testen des Einflusses auf die Standardabweichung gesetzt. Weitere häufig genutzte Parameter werden nicht automatisch gesetzt, da diese zu einer schlechteren Systemleistung und -stabilität führen könnten. So zum Beispiel das Deaktivieren der Kerne. Siehe dazu Abschnitt 5.2.1.

Durchlauf / Parameter	1	2	3	4	5
Nice-Wert	x	x	x	x	x
CPU-Frequenz	x	x	x	x	
IO-Nice-Wert	x	x	x	x	
ASLR	x	x	x	x	
Kerne deaktiviert	x	x	x	x	x
Watchdogs deaktiviert	x	x	x	x	
P-State-Treiber deaktiviert	x			x	x
Swap deaktiviert		x	x	x	
Intel Power Save States deakt.		x	x		
Governor		x	x		x
Perf		x	x	x	x
VM Stat Polling Interval			x		
Standardabweichung	0,06%	0,07%	0,07%	0,07%	0,08%
Laufzeit	53,2s	82,9s	53,1s	53,0s	53,2s
durchschn. Standardabw.	0,07%				
rel. Standardabw. der Standardabw.	10%				

Tabelle 5.3.: Ergebnisse der fünf besten Durchläufe des Simulated Annealing-Algorithmus auf dem Computer A.

Algorithmus / Parameter	Simulated Annealing	Optimiert
Nice-Wert	x	x
CPU-Frequenz	x	x
IO-Nice-Wert	x	x
ASLR	x	
Kerne deaktiviert	x	x
Watchdogs deaktiviert	x	
P-State-Treiber deaktiviert	x	
Intel Power Save States deaktiviert		x
Governor		x
Turbo Boost deaktiviert		x
Standardabweichung	0,06%	0,07%

Tabelle 5.4.: Vergleich der besten Ergebnisse der beiden von *Autotune* verwendeten Algorithmen auf dem Computer A. Die Standardabweichung der gegebenen Kombination von Parametern wurde einmal gemessen.

Schnelle Ausführung des optimierten Algorithmus

Autotune bietet zusätzlich zu den beiden Algorithmen eine Option, den optimierten Algorithmus schnell auszuführen. Wird *Autotune* mit dem Befehlszeilenparameter `--fast` ausgeführt, werden nur einmal fünf statt dreimal fünfzehn Benchmarks pro Parameter und keine nicht gewerteten Probeläufe ausgeführt, um die Standardabweichung zu messen. Dies beschleunigt die Ausführung des Algorithmus deutlich, führt jedoch zu einem schlechteren Ergebnis, also einer höheren Standardabweichung im Vergleich zur normalen Ausführung des optimierten Algorithmus. Die gemessenen Werte sind in Tabelle 5.5 angegeben.

	Schneller Algorithmus	Optimierter Algorithmus
Minimale Standardabw.	0,10%	0,07%
Maximale Standardabw.	0,25%	0,23%
Mittlere Standardabw.	0,19%	0,11%
rel. Standardabw. der Standardabw.	26%	52%
Anzahl Messungen	10	10
Durchschn. Laufzeit HMMER	67,2s	110s
Durchlaufzeit Algorithmus	1h	28h

Tabelle 5.5.: Vergleich der mit dem schnellen und optimierten Algorithmus für HMMER erreichten Werte auf dem Computer A.

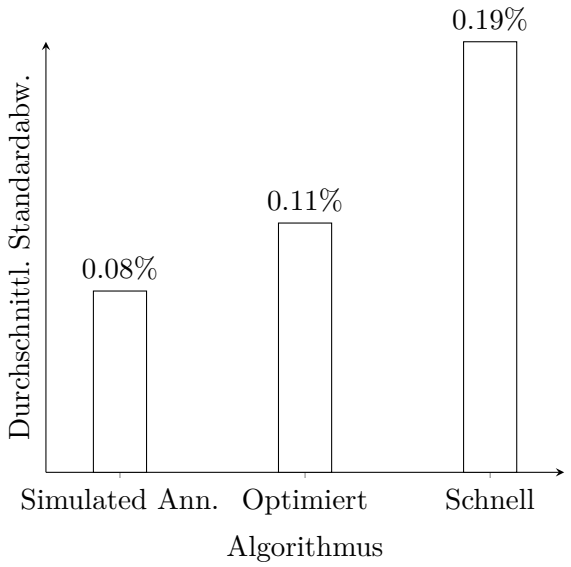


Abbildung 5.2.: Vergleich der mit dem Simulated Annealing-, dem optimierten und dem schnell ausgeführten optimierten Algorithmus durchschnittlich erreichten Standardabweichung.

Die lokale Ausführung

Wird *Autotune* nur auf dem Kontrollrechner ausgeführt, können nicht alle Betriebssystem- und Hardwareparameter gesetzt werden, da in diesem Fall kein Neustart des Benchmarkrechners möglich ist. *Autotune* führt im lokalen Modus den optimierten Algorithmus mit allen Parametern aus, die keinen Neustart des Computers benötigen. Dies führt dazu, dass die Standardabweichung der Laufzeit von Software nicht auf den minimal erreichbaren Wert reduziert werden kann. Die Werte in Tabelle 5.6 zeigen, wie sich die Ausführung von *Autotune* auf einem Computer von der Ausführung auf zwei Computern unterscheidet.

	Lokale Ausführung	Entfernte Ausführung
Minimale Standardabw.	0,07%	0,06%
Maximale Standardabw.	0,14%	0,11%
Mittlere Standardabw.	0,10%	0,09%
rel. Standardabw. der Standardabw.	24%	20%
Durchlaufzeit Algorithmus	4h	9h
Durchschn. Laufzeit HMMER	44,8s	81,9s
Anzahl Benchmarks pro Parameter	$3 * 15$	$3 * 15$
Nicht gewertete Probeläufe	$1 * 15$	$1 * 15$

Tabelle 5.6.: Vergleich der Ergebnisse von HMMER bei der Ausführung von *Autotune* auf dem Kontrollrechner mit den Ergebnissen der Ausführung auf einem separaten Benchmarkrechner (Computer B).

5.2.3. Die Auswirkungen der Systemkonfiguration auf verschiedenen Computern

Je nach Hardware des verwendeten Computers können die Ergebnisse von *Autotune* stark abweichen. Auf manchen Computern hat die Laufzeit von Software schon ohne Systemkonfiguration niedrige Standardabweichungen, welche durch *Autotune* nur unmerklich weiter reduziert werden können, so zum Beispiel auf Computer C. Auf anderen Computern hat die Laufzeit von Software anfänglich ohne Systemkonfiguration sehr hohe Standardabweichungen von über 2%, welche durch *Autotune* stark reduziert werden können, wie zum Beispiel auf Computer A. In den Tabellen 5.7 und 5.8 sind die jeweils besten Ergebnisse der beiden Algorithmen und die daraus resultierenden Standardabweichungen je Computer gelistet.

Computer / Parameter	A	B	C	D	E	F
Nice-Wert	x	x	x	x	x	x
CPU-Frequenz	x	x	x	x	x	
IO-Nice-Wert	x	x	x	x	x	x
ASLR		x				x
Kerne deaktiviert	x	x		x	x	
Watchdogs deaktiviert				x		
Governor	x	x	x	x		
Perf			x			
Isolation der Kerne			x			
Turbo Boost deaktiviert	x	x				
Scheduling Clock Tick			x			
Standardabw. ohne Param.	5,76%	0,47%	0,47%	0,41%	0,22%	0,25%
Standardabw. mit genannten Param.	0,07%	0,07%	0,22%	0,10%	0,11%	0,06%

Tabelle 5.7.: Vergleich der besten Ergebnisse des optimierten Algorithmus von *Autotune* für HMMER auf verschiedenen Computern. Für die Berechnung der Standardabweichung für die angegebenen Parameter wurde eine Messung mit 45 Benchmarks durchgeführt.

5.2.4. Ergebnisse für weitere Software

In Tabelle 5.9 ist dargestellt, welche Parameter der optimierte Algorithmus von *Autotune* für verschiedene Software findet und wie weit die Standardabweichung der Laufzeit dieser Software reduziert werden kann. *Autotune* reduziert die Standardabweichung dabei für drei der vier Software auf mindestens die Hälfte. Gebenchmarkt wurde HMMER, das Enkodieren einer MP3-Datei, das Kompilieren des Tools *ImageMagick* und das Ausführen verschiedener Git-Befehle. Außer den Git-Befehlen sind die Software CPU-bound. Die Git-Befehle basieren mehr auf dem Lesen und Schreiben von Daten und haben auch nach der Systemkonfiguration noch eine hohe Standardabweichung. Kann für eine CPU-bound Software die Standardabweichung reduziert werden, ist dies auch für die anderen möglich.

5.3. Vergleich von Autotune mit den LLVM-Tipps und Pyperf

Auf der Website des *LLVM*-Compilers werden Tipps [1] zur Stabilisierung von Benchmarks gegeben. Zudem hat Victor Stinner mit *Pyperf* [2] ein Tool entwickelt,

Computer / Parameter	A	B	C	D
Nice-Wert	x	x	x	x
CPU-Frequenz	x		x	x
IO-Nice-Wert	x	x	x	x
ASLR	x	x	x	x
Kerne deaktiviert	x	x	x	x
Watchdogs deaktiviert	x		x	x
Governor		x	x	x
Swap deaktiviert		x		x
Perf		x		x
Turbo Boost deaktiviert		x		x
VM Stat Polling Interval			x	x
Intel Power Save States deaktiviert			x	x
P-State-Treiber deaktiviert	x			
Standardabw. ohne Param.	5,76%	0,47%	0,47%	0,41%
Standardabw. mit Param.	0,06%	0,06%	0,08%	0,09%

Tabelle 5.8.: Vergleich der besten Ergebnisse des Simulated Annealing-Algorithmus von *Autotune* für HMMER auf verschiedenen Computern. Für die Berechnung der Standardabweichung für die angegebenen Parameter wurde eine Messung mit 45 Benchmarks durchgeführt.

das einige Parameter setzt, die die Standardabweichung der Laufzeit von Software reduzieren sollen. In diesem Abschnitt werden die Auswirkungen der *LLVM*-Tipps, von *Pyperf* und der Systemkonfiguration mit *Autotune* auf die Standardabweichung verschiedener Software verglichen.

Folgende sind die Tipps von *LLVM*:

- Randomisierung des Adressraums abschalten, siehe Kapitel 3.7.
- Setzen des Governors auf *performance*, siehe Kapitel 3.2.
- Isolieren von Prozessorkernen, siehe Kapitel 3.4.
- Turbo Boost deaktivieren, siehe Kapitel 3.3.
- SMT deaktivieren, siehe Kapitel 3.5.

Pyperf setzt folgende Parameter beim Ausführen von `pyperf system tune`:

- Setzen von Perf Event Sample Rate auf 1s, siehe Kapitel 3.6.

Parameter \ Software	HMMER	MP3	ImageMagick	Git
Nice-Wert	x	x	x	x
CPU-Frequenz	x	x	x	x
IO-Nice-Wert	x	x	x	x
Governor	x	x	x	x
Perf	x			
Isolation der Kerne	x			
Scheduling Clock Tick	x			
ASLR		x		
Intel Power Save States deakt.		x		
SMT deaktiviert		x		
Watchdogs deaktiviert		x		
VM Stat Polling Interval			x	
Kerne deaktiviert			x	
P-State-Treiber deaktiviert				x
Standardabw. ohne Systemkonf.	0,47%	1,02%	0,27%	1,41%
Standardabw. mit Systemkonf.	0,22%	0,23%	0,09%	1,19%

Tabelle 5.9.: Vergleich der Ergebnisse des optimierten Algorithmus von *Autotune* für verschiedene Software auf Computer C.

- Randomisierung des Adressraums abschalten, siehe Kapitel 3.7.
- Setzen der CPU-Frequenz auf die maximale Frequenz, siehe Kapitel 3.1.
- Setzen des Governors auf *performance*, siehe Kapitel 3.2.
- Turbo Boost deaktivieren, siehe Kapitel 3.3.
- Isolieren von Prozessorkernen, siehe Kapitel 3.4.
- Den Scheduler auf den isolierten Prozessoren deaktivieren, siehe Kapitel 3.10.
- RCU auf den isolierten Prozessoren deaktivieren, siehe Kapitel 3.9.
- Nice-Wert setzen, siehe Kapitel 3.11.

Tabelle 5.10 zeigt, dass *Autotune* deutlich niedrigere Werte für die Standardabweichung der Laufzeit von HMMER erzielt, als die von *LLVM* angegebenen Tipps und das Tool *Pyperf*. Vergleicht man die von *LLVM* und *Pyperf* vorgeschlagenen Parameter mit den von *Autotune* ausgesuchten Parametern, fällt auf, dass es einige Überschneidungen gibt. Jedoch wählt *Autotune* noch weitere Parameter, die in

Kombination mit den anderen zu einer geringeren Standardabweichung führen. Dazu gehört unter anderem das Deaktivieren von Prozessorkernen, das Deaktivieren von Watchdogs und das Deaktivieren des Intel P-State-Treibers. Die meisten dieser Parameter haben eigenständig keinen großen Effekt auf die Standardabweichung, wie in Abschnitt 5.2.1 zu sehen ist. In der richtigen Kombination ist jedoch eine Reduzierung der Standardabweichung um etwa das Vierfache im Vergleich zu den *LLVM*-Tipps und *Pyperf* möglich. Der Vergleich wird in Abbildung 5.3 verdeutlicht.

	LLVM-Tipps	Pyperf	Autotune
Minimale Standardabw.	0,30%	0,29%	0,06%
Maximale Standardabw.	0,51%	0,36%	0,15%
Mittlere Standardabw.	0,40%	0,32%	0,09%
rel. Standardabw. der Standardabw.	16%	8%	33%
Mittlere Laufzeit HMMER	23,9s	47,1s	47,5s
Anzahl Benchmarks pro Parameter	$10 * 45$	$10 * 45$	$10 * 45$

Tabelle 5.10.: Vergleich der durchschnittlichen Laufzeit und der Standardabweichung der Laufzeit von HMMER auf Computer B.

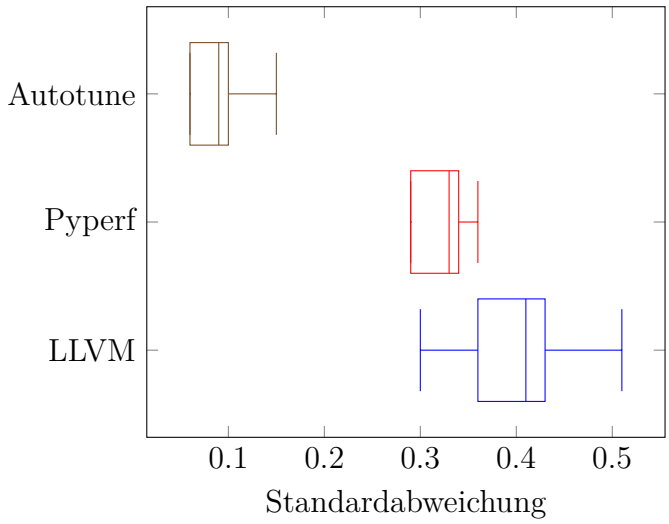


Abbildung 5.3.: Vergleich der mit den *LLVM*-Tipps, mit *Pyperf* und mit *Autotune* erreichten Standardabweichung für HMMER.

5.4. Software mit Multithreading

Software, die mehrere Threads verwendet, wird durch das Isolieren und Deaktivieren von Prozessorkernen in ihrer Laufzeit negativ beeinflusst. Des Weiteren müssen gegebenenfalls verschiedene Parallelisierungen einer Software in ihrer Laufzeit verglichen

werden. Daher bietet *Autotune* den Befehlszeilenparameter `--preserveMulticore`, mit dem die Isolation und Deaktivierung von Prozessorkernen verhindert und die parallele Ausführung ermöglicht werden kann. Das führt zu einer deutlich niedrigeren Laufzeit der Software, jedoch auch zu einer etwas höheren Standardabweichung im Vergleich zur Ausführung von *Autotune* mit den genannten Parametern. Die Vergleichswerte für Computer B sind in Tabelle 5.11 dargestellt und in Abbildung 5.4 veranschaulicht.

	Alle Parameter	Ohne Isol. / Deakt.
Minimale Standardabw.	0,06%	0,20%
Maximale Standardabw.	0,11%	0,27%
Mittlere Standardabw.	0,09%	0,24%
rel. Standardabw. der Standardabw.	20%	9%
Mittlere Laufzeit HMMER	81,9s	11,1s
Anzahl Benchmarks pro Parameter	$10 * 45$	$10 * 45$

Tabelle 5.11.: Standardabweichung und Laufzeit von HMMER mit und ohne Isolation und Deaktivierung von Prozessorkernen unter Verwendung des optimierten Algorithmus auf Computer B.

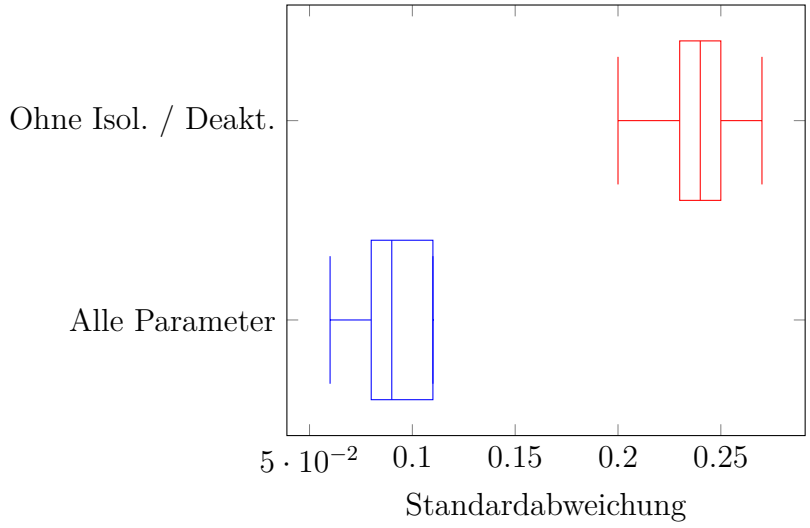


Abbildung 5.4.: Standardabweichung und Laufzeit von HMMER mit und ohne Isolation und Deaktivierung von Prozessorkernen unter Verwendung des optimierten Algorithmus auf Computer B.

Die Standardabweichung ist ohne die Isolation oder Deaktivierung von Prozessorkernen durchschnittlich etwa 2,7-mal größer als mit allen Parametern. Dafür ist die Laufzeit jedoch nur etwa ein Sechstel so lang wie mit allen Parametern. Wenn man großen Wert auf die Laufzeit legt und Software benchmarken möchte, die viele Threads nutzt, sollte man die Isolation und Deaktivierung von Prozessorkernen verhindern. Die Standardabweichung ist dennoch deutlich kleiner als ohne die Systemoptimierung.

5.5. Signifikanz der Reduzierung der Standardabweichung

Im Folgenden wird gezeigt, dass die Reduzierung der Standardabweichung der Laufzeit von Software mit *Autotune* signifikant ist. Dabei wird das Vorgehen aus Kapitel 2.2.2 verwendet.

1) Folgende Werte sind aus den Benchmarks mit HMMER gegeben: $s_1^2 = 0,22$, $n_1 = 41$, $s_2^2 = 0,06$, $n_2 = 41$ (optimierter Algorithmus auf Computer E).

2) Berechne F-Wert: $F = \frac{s_1^2}{s_2^2} = \frac{0,22}{0,06} = 3,67$.

3) Berechne Freiheitsgrade $df_1 = 40$ und $df_2 = 40$.

4) Wähle Signifikanzniveau α (z.B. $\alpha = 0,001 = 0,1\%$).

5) Kritischer Wert aus der F-Tabelle: $2,73$

6) Es gilt $F = 3,67 > 2,73 = kritischerWert$. Daraus folgt, dass die Varianz, beziehungsweise die Standardabweichung signifikant reduziert wurde.

5.6. Nicht-Determinismus des Ergebnisses von Autotune

In den vorigen Abschnitten ist zu erkennen, dass sich pro Durchlauf von *Autotune* auf demselben Computer für dieselbe Software andere Kombinationen von Parametern ergeben. Der erzielte Wert für die Standardabweichung ist in etwa derselbe. Die Parameter, die *Autotune* dafür verwendet hat, sind jedoch oftmals unterschiedlich. Es führen verschiedene Kombinationen von Parametern zu demselben Ergebnis bezüglich der Standardabweichung. *Autotune* testet die Parameter in zufälliger Reihenfolge durch. Es scheint für jeden Computer eine minimale Standardabweichung zu geben, die nicht mit den zur Verfügung stehenden Parametern unterschritten werden kann. Dieser Wert liegt bei etwa 0,06%, welcher auf keinem der Computer und mit keinem der Algorithmen unterschritten wurde. Diese Grenze gilt nur für CPU-bound Software, wie Abschnitt 5.7 zeigt. Die verschiedenen Ergebnisse folgen aus der zufälligen Reihenfolge. Wird bei Durchlauf 1 der Parameter A zuerst getestet und wird dadurch bereits die minimale Standardabweichung erreicht, so wird Parameter B nicht mit in die Lösung aufgenommen. Wird hingegen bei Durchlauf 2 Parameter B zuerst getestet und damit die minimale Standardabweichung erreicht, wird B und nicht

A in die Lösung mitaufgenommen. Durch diese verschiedenen Kombinationen von Parametern, die die Standardabweichung gleich weit reduzieren, ist das Ergebnis der Parameter, die *Autotune* zur Reduzierung verwendet, nicht-deterministisch.

5.7. Vergleich von CPU-bound und Memory-bound Software

In diesem Abschnitt wird untersucht, ob es bei der Reduzierung der Standardabweichung der Laufzeit einen Unterschied zwischen CPU-bound und Memory-bound Software gibt. Die vorangehenden Abschnitte zeigen, dass sich die Standardabweichung von CPU-bound Software, wie HMMER, sehr gut mit Hilfe der Systemkonfiguration reduzieren lässt. Memory-bound Software hingegen besteht hauptsächlich aus Speicherzugriffen und hängt daher nicht so stark von Faktoren ab, die den Prozessor betreffen. Tabelle 5.12 zeigt, dass die Standardabweichung von Memory-bound Software ohne Systemkonfiguration sehr niedrig ist und sich kaum weiter reduzieren lässt. Das liegt vor allem daran, dass die meisten Parameter auf den Prozessor bezogen sind und hier keine Rolle spielen. Dennoch liegt der Wert der minimal erreichbaren Standardabweichung für Memory-bound Software unter dem minimalen Wert von CPU-bound Software. Die in Abschnitt 5.6 genannte Untergrenze bezieht sich also nur auf CPU-bound Software.

Vor der Systemkonfiguration	
Standardabweichung	0,06%
Mittlere Laufzeit	2685s
Nach der Systemkonfiguration	
Minimale Standardabw.	0,02%
Maximale Standardabw.	0,03%
Mittlere Standardabw.	0,03%
rel. Standardabw. der Standardabw.	18%
Mittlere Laufzeit	2682s
Anzahl Benchmarks	$10 * 45$

Tabelle 5.12.: Vergleich der Standardabweichung der Laufzeit von Cachebench (Memory-bound) vor und nach der Systemkonfiguration mit *Autotune* auf Computer C.

5.8. Übertragbarkeit der Parameter

Autotune kann für jeden Computer und jede Software die Parameter finden, die die Standardabweichung minimieren. Dieser Abschnitt untersucht, ob es möglich ist, die Parameter, die für eine Software auf einem Computer gefunden wurden, auch für andere Software auf demselben Computer zu verwenden. Des Weiteren wird untersucht, ob die auf einem Computer für eine Software ausgewählten Parameter auf einem anderen Computer für die gleiche Software angewendet werden können und dort auch die Standardabweichung reduzieren.

5.8.1. Übertragbarkeit auf andere Software

Die Tabelle 5.13 zeigt, welchen Einfluss die Parameter, die für HMMER auf Computer C gefunden wurden, auf die Standardabweichung anderer Software auf demselben Computer hat. Die für HMMER auf dem Computer C gefundenen Parameter reduzieren auch die Standardabweichung der Laufzeit des Enkodierens einer MP3-Datei, jedoch weniger effektiv als mit anderen Parametern. Die Standardabweichung des Kompilierens von ImageMagick und des Ausführens von Git-Befehlen verschlechtert sich durch das Verwenden derselben Parameter im Vergleich zur Ausführung ganz ohne Parameter. Das bedeutet, man sollte die Parameter nur für die Software verwenden, für die sie von *Autotune* ausgewählt wurden. Für andere Software sollte die Systemkonfiguration erneut mit *Autotune* ausgeführt werden, um die zu dieser Software passenden Parameter zu finden.

Dadurch, dass Parameter nicht auf andere Software übertragbar sind, ist der Vergleich von Software gegebenenfalls schwierig. Da sich die Parameter auch auf die Laufzeit selbst auswirken, muss ein Vergleich mit den gleichen Parametern für jede zu vergleichende Software durchgeführt werden. Handelt es sich um Software, deren Standardabweichung der Laufzeit nicht mit den gleichen Parametern reduziert werden kann, ist ein Vergleich nicht möglich. Handelt es sich jedoch um zwei Versionen derselben Software, oder Software, die sich ähnlich ist, dann sind die Parameter übertragbar und ein aussagekräftiger Vergleich ist möglich.

5.8.2. Übertragbarkeit auf andere Computer

Die Tabelle 5.14 zeigt, dass die auf Computer A gefundenen Parameter auch auf andere Computer übertragbar sind und dort auch einen positiven Effekt auf die Standardabweichung der gleichen Software haben. Wie die Werte bestätigen, lässt

sich jedoch auch in diesem Fall die Standardabweichung weiter reduzieren, wenn man *Autotune* auf jedem Computer separat ausführt.

	HMMER	MP3	ImageMagick	Git
Standardabw. ohne Systemkonf.	0,47%	1,02%	0,27%	1,41%
Standardabw. mit HMMER-Param.	0,22%	0,63%	0,77%	37,84%
Standardabw. mit spezifischen Param.	0,22	0,23%	0,09%	1,19%
Anzahl Benchmarks	$10 * 45$	$10 * 45$	$10 * 45$	$10 * 45$

Tabelle 5.13.: Vergleich des Einflusses der für HMMER gefundenen Parameter (*CPU-Frequenz, Governor, IO-Nice-Wert, Nice-Wert, Perf, Prozessorkerne isoliert, Scheduler deaktiviert*) auf verschiedene Software auf Computer C mit dem Einfluss der für die Software spezifisch gefundenen Parametern.

Computer	A	B	C	E
Standardabw. mit Computer A-Param.	0,08%	0,08%	0,29%	0,14%
Standardabw. mit spezifischen Param.	0,08%	0,06%	0,22%	0,11%

Tabelle 5.14.: Vergleich des Einflusses der auf Computer A für HMMER gefundenen Parameter (*CPU-Frequenz, Governor, IO-Nice-Wert, Nice-Wert, Intel Power Save States deaktiviert, Prozessorkernen deaktiviert Scheduler deaktiviert*) mit dem Einfluss der auf den anderen Computern spezifisch für HMMER gefundenen Parametern.

5.8.3. Erfüllbarkeit der fünf Charakteristika für Benchmarks

In Kapitel 2.4 wurden die fünf Charakteristika für Benchmarks eingeführt. Die vorherigen Abschnitte haben gezeigt, dass durch die Systemkonfiguration die Standardabweichung verringert wird. Dadurch ist die *Wiederholbarkeit* von Benchmarks gewährleistet. Durch die Übertragbarkeit der Parameter auf andere Computer ist auch die *Überprüfbarkeit* der Benchmarkergebnisse sichergestellt. Da das Verwenden der gleichen Parameter für unterschiedliche Software jedoch nur bedingt möglich ist, ist die *Fairness* der Benchmarks nicht mit Autotune und der Systemkonfiguration erfüllbar. Auf die *Relevanz* und die *Ökonomie* eines Benchmarks haben Autotune und die Systemkonfiguration keinen Einfluss. Die Erfüllung dieser Charakteristika ist von der Wahl des Benchmarks abhängig.

6. Fazit und Ausblick

Die Evaluation hat gezeigt, dass eine Verringerung der Standardabweichung der Laufzeit von Software möglich ist. Einzelne Betriebssystem- und Hardwareparameter haben oft keinen oder sogar einen negativen Effekt auf die Standardabweichung der Laufzeit von Software. Mit Hilfe von *Autotune* und einer Kombination von Parametern kann die Standardabweichung mit einer Signifikanz von 99,9% reduziert werden. Das erzielbare Minimum der Standardabweichung liegt bei allen getesteten Computern zwischen 0,06% und 0,09%, wenn der Simulated Annealing-Algorithmus angewendet wird.

Das Finden der Parameter, die die Standardabweichung minimieren, ist jedoch zeitaufwändig und kann teilweise über 30 Stunden dauern. Aus diesem Grund gibt es den optimierten Algorithmus, der in der schnellen Variante nur etwa eine Stunde benötigt. Allerdings ist mit der schnellen Ausführung nicht garantiert, dass die minimale Standardabweichung erreicht wird. Die Standardabweichung wird dabei durchschnittlich auf einen Wert von 0,19% reduziert.

In jedem Fall ist mit den erreichten Werten für die Standardabweichung ein Vergleich zweier Softwareversionen ohne Probleme möglich. Es muss allerdings beachtet werden, dass die Parameter nicht auf andere Software übertragen werden können und daher ein Vergleich unterschiedlicher Software gegebenenfalls nicht möglich ist.

Die erreichten Werte für die Standardabweichung liegen unter den Werten, die mit den *LLVM*-Tipps und *Pyperf* erreichbar sind. Trotz der guten Ergebnisse für die Standardabweichung ist die Auswahl der Parameter, die für bestimmte Hardware und Software von *Autotune* gefunden werden, nicht-deterministisch. Verschiedene Kombinationen führen zum selben oder zu einem ähnlichen Ergebnis. Das Resultat ist jedoch nicht genau reproduzierbar und es wird auch nicht immer exakt der gleiche minimale Wert für die Standardabweichung erreicht.

Zukünftig könnte *Autotune* um mehr Erfahrungswerte ergänzt werden, wodurch die Ausführung beschleunigt werden würde. Des Weiteren wäre es interessant, Tools wie den *Thread Tranquilizer* [18] in *Autotune* zu integrieren, die Cache-Misses und Thread-Kontextwechsel messen und gezielt vermeiden können. Zudem könnte man einen Algorithmus entwickeln, der für unterschiedliche Software die Kombination an Parametern findet, die die Standardabweichung jeder Software minimiert. Da sich

diese Arbeit auf konfigurierbare Parameter beschränkt, könnte es noch andere Faktoren geben, die Schwankungen der Laufzeit von Software verursachen und bisher nicht behandelt wurden. So zum Beispiel die Hardware oder das Betriebssystem selbst, welche gegebenenfalls in zukünftigen Arbeiten behandelt werden könnten. Nicht getestet wurden der Einfluss von HDDs und SSDs und der Einfluss von unterschiedlichen Betriebssystemen auf die Standardabweichung derselben Software.

Dennoch kann man zusammenfassend sagen, dass *Autotune* ein gutes Werkzeug für die Vorbereitung von Benchmarks darstellt.

Literaturverzeichnis

[1] LLVM, "Benchmarking tips — LLVM 10 documentation." https://llvm.org/docs/Benchmarking.html.

[2] V. Stinner, "Run a benchmark — pyperf 1.6.1 documentation." https://pyperf.readthedocs.io/en/latest/run_benchmark.html.

[3] U. Kuckartz, *Statistik: eine verständliche Einführung*, vol. 91. 2013.

[4] H. Sahner, *Schließende Statistik: Eine Einführung für Sozialwissenschaftler*, ch. F-Test und, pp. 111–131. Wiesbaden: VS Verlag für Sozialwissenschaften, 2005.

[5] U. Kuckartz, S. Rädiker, T. Ebert, and J. Schehl, *Varianzanalyse: mehr als zwei Mittelwerte vergleichen*, pp. 185–206. Wiesbaden: VS Verlag für Sozialwissenschaften, 2013.

[6] M. B. Brown and A. B. Forsythe, "Robust Tests for the Equality of Variances," *Journal of the American Statistical Association*, vol. 69, no. 346, pp. 364–367, 1974.

[7] G. Heiser, "Gernot's List of Systems Benchmarking Crimes." http://gernot-heiser.org/benchmarking-crimes.html, 2015.

[8] I. Frost, *Statistische Testverfahren, Signifikanz und p-Werte: Allgemeine Prinzipien verstehen und Ergebnisse angemessen interpretieren*, ch. Beispiel:, pp. 13–15. Wiesbaden: Springer Fachmedien Wiesbaden, 2017.

[9] M. Pieper, *Mathematische Optimierung: Eine Einführung in die kontinuierliche Optimierung mit Beispielen*, ch. Einleitung, pp. 1–2. Wiesbaden: Springer Fachmedien Wiesbaden, 2017.

[10] D. Logofătu, *Algorithmen und Problemlösungen mit C++: Von der Diskreten Mathematik zum fertigen Programm*, ch. Greedy, pp. 287–304. Wiesbaden: Vieweg+Teubner, 2010.

[11] W.-M. Lippe, *Soft-Computing*. eXamen.press, Berlin/Heidelberg: Springer-Verlag, 2006.

[12] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[13] P. Rossmanith, *Simulated Annealing*, pp. 393–400. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[14] R. Nambiar and M. Poess, "Performance Evaluation and Benchmarking," Lecture Notes in Computer Science, (Lyon, France), Springer Berlin Heidelberg, 2009.

[15] V. Stinner, "My journey to stable benchmark, part 1 (system)." https://vstinner.github.io/journey-to-stable-benchmark-system.html, 2016.

[16] J. Bechberger, "Besser Benchmarken," apr 2016.

[17] N. J. Wright, S. Smallen, C. Olschanowsky, J. Hayes, and A. Snavely, "Measuring and Understanding Variation in Benchmark Performance," *HPCMP Users Group Conference*, vol. 0, pp. 438–443, 2009.

[18] K. K. Pusukuri, R. Gupta, and L. Bhuyan, "Thread Tranquilizer: Dynamically reducing performance variation.," *TACO*, vol. 8, p. 46, 2012.

[19] D. Brodowski, N. Golde, R. J. Wysocki, and V. Kumar, "CPU frequency and voltage scaling code in the Linux(TM) kernel." https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[20] Intel®, "Intel® Turbo-Boost-Max-Technik 3.0." https://www.intel.de/content/www/de/de/architecture-and-technology/turbo-boost/turbo-boost-max-technology.html.

[21] AMD, "Turbo CORE Technologie | AMD." https://www.amd.com/de/technologies/turbo-core.

[22] A. S. Tanenbaum and H. Bos, *Modern operating systems*. Boston: Prentice Hall, 4. ed. ed., 2015.

[23] M. Nemirovsky and D. M. Tullsen, "Multithreading Architecture," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 1, pp. 1–109, 2013.

[24] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.

[25] PerfWiki, "Perf Wiki." https://perf.wiki.kernel.org/index.php/Main_Page, 2015.

[26] A. K. Sood, R. Enbody, A. K. Sood, and R. Enbody, "System Exploitation," *Targeted Cyber Attacks*, pp. 37–75, jan 2014.

[27] J. C. Detter and R. Mutschlechner, "Performance and Entropy of Various ASLR Implementations," 2015.

[28] V. Stinner, "Benchmarks — Victor Stinner's Notes 1.0 documentation." https://vstinner.readthedocs.io/benchmark.html, 2014.

[29] Kernel.org, "Softlockup detector and hardlockup detector (aka nmi_watchdog)." https://www.kernel.org/doc/Documentation/lockup-watchdogs.txt.

[30] P. E. McKenney, "What is RCU? – Read, Copy, Update." https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt.

[31] Kernel.org, "NO_HZ: Reducing Scheduling-Clock Ticks." https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

[32] D. MacKenzie, "nice." https://manpages.debian.org/testing/coreutils/nice.1.en.html, 2019.

[33] J. Axboe and K. Zak, "ionice." https://manpages.debian.org/testing/util-linux/ionice.1.en.html, 2011.

[34] Kernel.org, "Intel P-State driver." https://www.kernel.org/doc/Documentation/cpu-freq/intel-pstate.txt.

[35] T. I. Kidd, "What exactly is a P-state? (Pt. 1) | Intel® Software." https://software.intel.com/en-us/blogs/2008/05/29/what-exactly-is-a-p-state-pt-1, 2015.

[36] T. I. Kidd, "C-states, C-states and even more C-states | Intel® Software." https://software.intel.com/en-us/blogs/2008/03/27/update-c-states-c-states-and-even-more-c-states/, 2008.

[37] R. van Riel and P. W. Morreale, "Documentation for the sysctl files in /proc/-sys/vm." https://www.kernel.org/doc/Documentation/sysctl/vm.txt.

[38] V. Stinner, "Tune the system for benchmarks." https://pyperf.readthedocs.io/en/latest/system.html, 2016.

[39] Phoronix, "Phoronix Test Suite - Linux Testing & Benchmarking Platform,

Automated Testing, Open-Source Benchmarking." https://www.phoronix-test-suite.com/.

[40] Hmmer.org, "HMMER." http://hmmer.org/.

[41] Phoronix, "OpenBenchmarking.org - CacheBench Test Profile." https://openbenchmarking.org/test/pts/cachebench.

[42] Openbenchmarking.org, "Timed ImageMagick Compilation." https://openbenchmarking.org/test/pts/build-imagemagick.

[43] Openbenchmarking.org, "LAME MP3 Encoding." https://openbenchmarking.org/test/pts/encode-mp3.

[44] Openbenchmarking.org, "Git Test Profile." https://openbenchmarking.org/test/pts/git.

Erklärung

Hiermit erkläre ich, Jacques Marco Jung, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Ort, Datum	Unterschrift

A. Anhang

A.1. Optimaler Wert für das VM Stat Polling Interval

In Kapitel 3.15 wurde das *VM Stat Polling Interval* beschrieben und wie das Ändern des Intervalls zu einer Reduzierung der Standardabweichung führen kann. Um herauszufinden welcher Wert für das Intervall die Standardabweichung am weitesten reduziert, wurden Benchmarks für verschiedene Werte vorgenommen, die in Tabelle A.1 angegeben und in Abbildung A.1 graphisch dargestellt sind. Die Werte zeigen, dass es nicht möglich ist, einen optimalen Wert für das Intervall zu finden. Das liegt vor allem daran, dass viele Parameter erst in Kombination mit anderen Parametern einen Effekt auf die Standardabweichung haben. Das im Blog von Victor Stinner [38] vorgeschlagene Intervall von 20s führt zu einer Standardabweichung von 0,44% und damit zu einem der niedrigsten Werte in der Tabelle.

Intervall	Standardabweichung
1s	1,33%
2s	0,42%
4s	0,72%
8s	0,53%
10s	0,75%
20s	0,44%
40s	0,49%
60s	0,81%
80s	0,62%
100s	0,42%
200s	0,54%
300s	0,68%
400s	0,53%
500s	0,50%
1000s	0,69%

Tabelle A.1.: Benchmarks von HMMER mit verschiedenen Werten für das *VM Stat Polling Interval* auf Computer C. Es wurden jeweils 30 Benchmarks pro Intervallwert durchgeführt um die Standardabweichung zu berechnen.

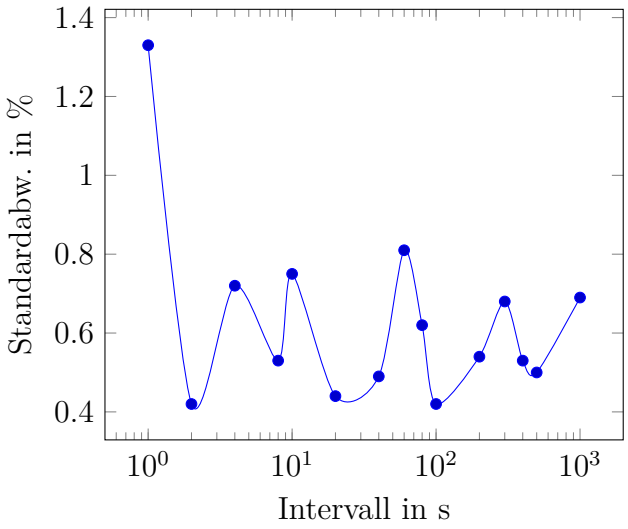


Abbildung A.1.: Verlauf der Standardabweichung der Laufzeit von HMMER bei Verwendung verschiedener Werte für das VM Stat Polling Interval auf Computer C.

A.2. Weitere Ergebnisse des Simulated Annealing-Algorithmus

A.2.1. Computer B

- Nice-Wert, IO-Nice-Wert, VM Stat Polling Interval, Intel Power Save States deaktiviert, Turbo Boost deaktiviert, ASLR, Governor, Scheduler deaktiviert, Watchdogs deaktiviert, Perf, Swap deaktiviert, Kerne deaktiviert.
  Standardabweichung: 0.07%

- Watchdogs deaktiviert, Perf, Intel P-State-Treiber deaktiviert, Kerne deaktiviert.
  Standardabweichung: 0.09%

- VM Stat Polling Interval, CPU-Frequenz, Governor, ASLR, Nice-Wert, Swap deaktiviert, Perf, Watchdogs deaktiviert, Scheduler deaktiviert, Kerne deaktiviert, Intel Power Save States deaktiviert, IO-Nice-Wert.
  Standardabweichung: 0.06%

- Turbo Boost deaktiviert, VM Stat Polling Interval, IO-Nice-Wert, ASLR, Perf, Nice-Wert, Swap deaktiviert, Governor, Intel Power Save States deaktiviert, Watchdogs deaktiviert, Kerne deaktiviert.

Standardabweichung: 0.07%

- Swap deaktiviert, Watchdogs deaktiviert, Intel P-State-Treiber deaktiviert, CPU-Frequenz, Governor, VM Stat Polling Interval, ASLR, Nice-Wert, Kerne deaktiviert.
  Standardabweichung: 0.06%

- Kerne deaktiviert, IO-Nice-Wert, Intel P-State-Treiber deaktiviert.
  Standardabweichung: 0.1%

- Nice-Wert, IO-Nice-Wert, Swap deaktiviert, Perf, ASLR, Kerne deaktiviert, Kerne deaktiviert, Turbo Boost deaktiviert, Governor.
  Standardabweichung: 0.06% Laufzeit: 11.5s

- Swap deaktiviert, VM Stat Polling Interval, Kerne deaktiviert, Turbo Boost deaktiviert.
  Standardabweichung: 0.1% Laufzeit: 47.26s

- Kerne deaktiviert.
  Standardabweichung: 0.15% Laufzeit: 41.39s

- Swap deaktiviert, Intel Power Save States deaktiviert, Turbo Boost deaktiviert, Governor, VM Stat Polling Interval, IO-Nice-Wert, Nice-Wert, Watchdogs deaktiviert, Kerne deaktiviert.
  Standardabweichung: 0.09% Laufzeit: 41.36s

A.2.2. Computer C

- Governor, Turbo Boost deaktiviert, IO-Nice-Wert, Perf, Intel Power Save States deaktiviert, Nice-Wert, ASLR, VM Stat Polling Interval, Kerne deaktiviert.
  Standardabweichung: 0.15%

- ASLR, IO-Nice-Wert, Nice-Wert, Perf, Watchdogs deaktiviert, CPU-Frequenz, Scheduler deaktiviert, VM Stat Polling Interval, Governor, Intel P-State-Treiber deaktiviert, Kerne deaktiviert, Swap deaktiviert.
  Standardabweichung: 0.1%

- CPU-Frequenz, ASLR, Governor, Intel P-State-Treiber deaktiviert, Perf, Watchdogs deaktiviert, Kerne deaktiviert.
  Standardabweichung: 0.16%

- Intel P-State-Treiber deaktiviert, VM Stat Polling Interval, ASLR, CPU-

Frequenz, Governor, Kerne deaktiviert.
Standardabweichung: 0.13%

- IO-Nice-Wert, Governor, ASLR, CPU-Frequenz, Nice-Wert, Perf, Intel Power Save States deaktiviert, Watchdogs deaktiviert, Kerne deaktiviert, VM Stat Polling Interval.
  Standardabweichung: 0.1%

- VM Stat Polling Interval, ASLR, Perf, IO-Nice-Wert, Turbo Boost deaktiviert, Intel Power Save States deaktiviert, Nice-Wert, Kerne deaktiviert, CPU-Frequenz, Swap deaktiviert.
  Standardabweichung: 0.14%

- IO-Nice-Wert, Perf, CPU-Frequenz, Kerne deaktiviert, VM Stat Polling Interval, ASLR, Intel P-State-Treiber deaktiviert, Nice-Wert, Governor.
  Standardabweichung: 0.12%

- ASLR, Nice-Wert, Watchdogs deaktiviert, Governor, Kerne deaktiviert, VM Stat Polling Interval, CPU-Frequenz, Intel Power Save States deaktiviert, IO-Nice-Wert.
  Standardabweichung: 0.08% Laufzeit: 38.24s

- Prozessorkerne isoliert, Intel P-State-Treiber deaktiviert, Swap deaktiviert.
  Standardabweichung: 0.25% Laufzeit: 41.5s

- Perf, IO-Nice-Wert, ASLR, Kerne deaktiviert, Watchdogs deaktiviert, VM Stat Polling Interval, Governor, Nice-Wert, Intel P-State-Treiber deaktiviert.
  Standardabweichung: 0.11% Laufzeit: 38.36s

A.2.3. Computer D

- Perf, Swap deaktiviert, ASLR, Watchdogs deaktiviert, Intel Power Save States deaktiviert, CPU-Frequenz, VM Stat Polling Interval, Nice-Wert, Scheduler deaktiviert, Turbo Boost deaktiviert, IO-Nice-Wert, Prozessorkerne isoliert, RCU deaktiviert.
  Standardabweichung: 0.16%

- Prozessorkerne isoliert, Scheduler deaktiviert, Turbo Boost deaktiviert, Watchdogs deaktiviert, Governor, Perf, VM Stat Polling Interval, Intel Power Save States deaktiviert, ASLR.
  Standardabweichung: 0.18%

- Perf, Swap deaktiviert, Prozessorkerne isoliert, RCU deaktiviert, Scheduler deaktiviert, Watchdogs deaktiviert, ASLR, Intel Power Save States deaktiviert. **Standardabweichung: 0.18%**

- VM Stat Polling Interval, ASLR, Perf, Kerne deaktiviert, Watchdogs deaktiviert, CPU-Frequenz, Turbo Boost deaktiviert, Intel Power Save States deaktiviert. **Standardabweichung: 0.09%**

- Perf, IO-Nice-Wert, Intel Power Save States deaktiviert, VM Stat Polling Interval, Prozessorkerne isoliert, RCU deaktiviert, ASLR, Scheduler deaktiviert. **Standardabweichung: 0.17% Laufzeit: 9.74s**

- IO-Nice-Wert, Nice-Wert, CPU-Frequenz, Turbo Boost deaktiviert, VM Stat Polling Interval, Watchdogs deaktiviert, Governor, Intel Power Save States deaktiviert, ASLR, Perf, Swap deaktiviert, Kerne deaktiviert. **Standardabweichung: 0.09% Laufzeit: 30.58s**

- Intel Power Save States deaktiviert, VM Stat Polling Interval. **Standardabweichung: 0.32% Laufzeit: 37.32s**

- Nice-Wert, Perf, Watchdogs deaktiviert, VM Stat Polling Interval, Swap deaktiviert, Intel Power Save States deaktiviert, IO-Nice-Wert, ASLR, Prozessorkerne isoliert, RCU deaktiviert, Governor, Scheduler deaktiviert, CPU-Frequenz. **Standardabweichung: 0.16% Laufzeit: 45.27s**

- Swap deaktiviert, ASLR, Watchdogs deaktiviert, IO-Nice-Wert, Perf, CPU-Frequenz, VM Stat Polling Interval, Turbo Boost deaktiviert, Governor, Kerne deaktiviert, Nice-Wert, Intel Power Save States deaktiviert. **Standardabweichung: 0.1% Laufzeit: 45.24s**

- Kerne deaktiviert, IO-Nice-Wert, Governor, Intel Power Save States deaktiviert. **Standardabweichung: 0.1% Laufzeit: 37.08s**

A.3. Weitere Ergebnisse des optimierten Algorithmus

A.3.1. Computer B

- Governor, CPU-Frequenz, IO-Nice-Wert, Nice-Wert, VM Stat Polling Interval, Swap deaktiviert, Prozessorkerne deaktiviert **Standardabweichung: 0.08% Laufzeit: 42.71s**

- Nice-Wert, CPU-Frequenz, IO-Nice-Wert, Governor, Swap deaktiviert, ASLR, Prozessorkerne deaktiviert
  Standardabweichung: 0.08% Laufzeit: 43.72s

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Swap deaktiviert, Perf, Turbo Boost deaktiviert, Prozessorkerne deaktiviert, Intel Power Save States deaktiviert
  Standardabweichung: 0.09% Laufzeit: 97.61s

- Nice-Wert, IO-Nice-Wert, Governor, CPU-Frequenz, Prozessorkerne deaktiviert, Intel Power Save States deaktiviert
  Standardabweichung: 0.08% Laufzeit: 85.41s

- Governor, Nice-Wert, IO-Nice-Wert, CPU-Frequenz, Perf, Prozessorkerne deaktiviert, Watchdogs deaktiviert
  Standardabweichung: 0.11% Laufzeit: 85.61s

- Governor, IO-Nice-Wert, Nice-Wert, CPU-Frequenz, Prozessorkerne deaktiviert, Watchdogs deaktiviert
  Standardabweichung: 0.107% Laufzeit: 85.47s

- IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Governor, Turbo Boost deaktiviert, Prozessorkerne deaktiviert, Intel Power Save States deaktiviert
  Standardabweichung: 0.09% Laufzeit: 97.61s

- Nice-Wert, IO-Nice-Wert, CPU-Frequenz, Governor, Swap deaktiviert, Prozessorkerne deaktiviert
  Standardabweichung: 0.11% Laufzeit: 85.50s

- Governor, Nice-Wert, CPU-Frequenz, IO-Nice-Wert, Turbo Boost deaktiviert, ASLR, Prozessorkerne deaktiviert
  Standardabweichung: 0.07% Laufzeit: 97.96s

- Governor, Nice-Wert, CPU-Frequenz, IO-Nice-Wert, Prozessorkerne deaktiviert
  Standardabweichung: 0.11% Laufzeit: 97.73s

A.3.2. Computer C

- Governor, IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.28% Laufzeit: 38.21s

- CPU-Frequenz, Governor, IO-Nice-Wert, Nice-Wert, Perf, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.22% Laufzeit: 38.72s

- IO-Nice-Wert, Governor, Nice-Wert, CPU-Frequenz, Prozessorkerne isoliert.
  Standardabweichung: 0.24% Laufzeit: 38.18s

- CPU-Frequenz, Nice-Wert, IO-Nice-Wert, Governor, Intel Power Save States deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.29% Laufzeit: 80.95s

- Nice-Wert, CPU-Frequenz, IO-Nice-Wert, Governor, Prozessorkerne isoliert.
  Standardabweichung: 0.34% Laufzeit: 38.3s

- CPU-Frequenz, Nice-Wert, IO-Nice-Wert, Governor, Prozessorkerne isoliert.
  Standardabweichung: 0.29% Laufzeit: 38.2s

- CPU-Frequenz, IO-Nice-Wert, Governor, Nice-Wert, Swap deaktiviert, Prozessorkerne isoliert.
  Standardabweichung: 0.3% Laufzeit: 38.28s

- CPU-Frequenz, Nice-Wert, IO-Nice-Wert, Governor, Prozessorkerne isoliert.
  Standardabweichung: 0.28% Laufzeit: 38.25s

- Nice-Wert, Governor, IO-Nice-Wert, CPU-Frequenz, Prozessorkerne deaktiviert.
  Standardabweichung: 0.34% Laufzeit: 80.9s

- IO-Nice-Wert, Governor, Nice-Wert, CPU-Frequenz, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.38% Laufzeit: 38.2s

A.3.3. Computer D

- CPU-Frequenz, IO-Nice-Wert, Intel Power Save States deaktiviert.
  Standardabweichung: 0.29%

- Governor, ASLR.
  Standardabweichung: 0.31%

- CPU-Frequenz, Intel Power Save States deaktiviert.
  Standardabweichung: 0.30%

- Governor.
  Standardabweichung: 0.37%

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Intel Power Save States deaktiviert.
  Standardabweichung: 0.29% Laufzeit: 7.41s

- CPU-Frequenz, Nice-Wert, Governor, IO-Nice-Wert, Swap deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.13% Laufzeit: 60.77s

- CPU-Frequenz, Governor, Nice-Wert, IO-Nice-Wert, Prozessorkerne deaktiviert, Watchdogs deaktiviert.
  Standardabweichung: 0.097% Laufzeit: 60.67s

- Governor, IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.137% Laufzeit: 60.67s

- CPU-Frequenz, Nice-Wert, IO-Nice-Wert, Governor, Swap deaktiviert, Intel Power Save States deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.21% Laufzeit: 60.73s

- Governor, IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Watchdogs deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.16% Laufzeit: 60.73s

A.3.4. Computer E

- IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Governor, Prozessorkerne deaktiviert.
  Standardabweichung: 0.14% Laufzeit: 103.23s

- IO-Nice-Wert, Governor, Nice-Wert, CPU-Frequenz, Prozessorkerne deaktiviert, Intel P-State-Treiber deaktiviert.
  Standardabweichung: 0.19% Laufzeit: 103.2s

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Prozessorkerne deaktiviert.
  Standardabweichung: 0.11% Laufzeit: 103.24s

- Nice-Wert, IO-Nice-Wert, Governor, CPU-Frequenz, Swap deaktiviert.
  Standardabweichung: 0.32% Laufzeit: 17.61s

- Nice-Wert, Governor, CPU-Frequenz, IO-Nice-Wert, Turbo Boost deaktiviert,

Prozessorkerne deaktiviert.
Standardabweichung: 0.13% Laufzeit: 153.42s

- Governor, CPU-Frequenz, IO-Nice-Wert, Nice-Wert, Turbo Boost deaktiviert, Prozessorkerne isoliert, RCU deaktiviert, Intel Power Save States deaktiviert.
  Standardabweichung: 0.18% Laufzeit: 93.48s

- IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Governor, Turbo Boost deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.13% Laufzeit: 154.89s

- Governor, IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Swap deaktiviert, Prozessorkerne deaktiviert, Intel Power Save States deaktiviert.
  Standardabweichung: 0.14% Laufzeit: 103.68s

- Nice-Wert, CPU-Frequenz, IO-Nice-Wert, Governor, Turbo Boost deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.15% Laufzeit: 154.94s

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Turbo Boost deaktiviert, Prozessorkerne deaktiviert.
  Standardabweichung: 0.14% Laufzeit: 154,91s

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Prozessorkerne deaktiviert.
  Standardabweichung: 0.12%

A.4. Weitere Ergebnisse der schnellen Ausführung des optimierten Algorithmus

A.4.1. Computer B

- CPU-Frequenz, Governor, IO-Nice-Wert, Nice-Wert, Turbo Boost deaktiviert, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.25% Laufzeit: 47.19s

- CPU-Frequenz, Governor, Nice-Wert, IO-Nice-Wert, VM Stat Polling Interval, Turbo Boost deaktiviert, Prozessorkerne isoliert, RCU deaktiviert, Scheduler deaktiviert.
  Standardabweichung: 0.21% Laufzeit: 47.14s

- CPU-Frequenz, Governor, IO-Nice-Wert, Nice-Wert, Perf, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.27% Laufzeit: 41.67s

- CPU-Frequenz, IO-Nice-Wert, Governor, Nice-Wert, ASLR, Swap deaktiviert, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.15% Laufzeit: 41.59s

- CPU-Frequenz, Governor, Nice-Wert, IO-Nice-Wert, Prozessorkerne isoliert, RCU deaktiviert, VM Stat Polling Interval, Perf, Swap deaktiviert, ASLR.
  Standardabweichung: 0.05% Laufzeit: 41.55s

- IO-Nice-Wert, CPU-Frequenz, Governor, Nice-Wert, Swap deaktiviert, ASLR, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.35% Laufzeit: 41.66s

- Governor, CPU-Frequenz, Nice-Wert, IO-Nice-Wert, Perf, Swap deaktiviert, VM Stat Polling Interval, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.31% Laufzeit: 41.53s

- Nice-Wert, Governor, CPU-Frequenz, IO-Nice-Wert, Perf, Intel Power Save States deaktiviert, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.38% Laufzeit: 54.69s

- Governor, IO-Nice-Wert, CPU-Frequenz, Nice-Wert, Turbo Boost deaktiviert, Swap deaktiviert, Prozessorkerne isoliert.
  Standardabweichung: 0.23% Laufzeit: 47.05s

- CPU-Frequenz, Nice-Wert, Governor, IO-Nice-Wert, Prozessorkerne isoliert, Intel P-State-Treiber deaktiviert.
  Standardabweichung: 0.21% Laufzeit: 41.75s

A.4.2. Computer E

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, ASLR, VM Stat Polling Interval, Turbo Boost deaktiviert, Prozessorkerne isoliert.
  Standardabweichung: 0.16% Laufzeit: 76.26s

- Governor, IO-Nice-Wert, CPU-Frequenz, Nice-Wert, VM Stat Polling Interval, Prozessorkerne isoliert, RCU deaktiviert.
  Standardabweichung: 0.14% Laufzeit: 53.93s

- CPU-Frequenz, Governor, Nice-Wert, IO-Nice-Wert, Turbo Boost deaktiviert, ASLR, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.14% Laufzeit: 76.04s

- Governor, IO-Nice-Wert, Nice-Wert, CPU-Frequenz, Swap deaktiviert, Prozessorkerne isoliert.
  Standardabweichung: 0.26% Laufzeit: 53.81s

- IO-Nice-Wert, Nice-Wert, CPU-Frequenz, Governor, Swap deaktiviert, VM Stat Polling Interval, Turbo Boost deaktiviert, ASLR, Prozessorkerne isoliert.
  Standardabweichung: 0.17% Laufzeit: 76.11s

- IO-Nice-Wert, Nice-Wert, CPU-Frequenz, Governor, Swap deaktiviert, Prozessorkerne isoliert, RCU deaktiviert, Scheduler deaktiviert.
  Standardabweichung: 0.27% Laufzeit: 53.89s

- Nice-Wert, CPU-Frequenz, Governor, IO-Nice-Wert, ASLR, Turbo Boost deaktiviert, VM Stat Polling Interval, Prozessorkerne isoliert, RCU deaktiviert.
  Standardabweichung: 0.2% Laufzeit: 76.18s

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Swap deaktiviert, Prozessorkerne isoliert, RCU deaktiviert, Scheduler deaktiviert, Intel P-State-Treiber deaktiviert.
  Standardabweichung: 0.33% Laufzeit: 54.28s

- IO-Nice-Wert, Nice-Wert, Governor, CPU-Frequenz, Turbo Boost deaktiviert, ASLR, Prozessorkerne isoliert, Scheduler deaktiviert.
  Standardabweichung: 0.21% Laufzeit: 76.19s

- Governor, IO-Nice-Wert, Nice-Wert, CPU-Frequenz, VM Stat Polling Interval, Intel P-State-Treiber deaktiviert, Prozessorkerne isoliert, RCU deaktiviert, Scheduler deaktiviert.
  Standardabweichung: 0.2% Laufzeit: 54.02s

A.5. Alle Benchmarkergebnisse von Pyperf

Siehe Tabellen A.2 und A.3.

Durchlauf	Standardabweichung	Laufzeit
1	0,32%	47.1s
2	0,29%	47.1s
3	0,34%	47.2s
4	0,29%	47.1s
5	0,29%	47.1s
6	0,33%	47.1s
7	0,36%	47.1s
8	0,34%	47.1s
9	0,29%	47.1s
10	0,33%	47.1s

Tabelle A.2.: Alle Benchmarkergebnisse von *Pyperf* auf Computer B

Durchlauf	Standardabweichung	Laufzeit
1	0,40%	38.2s
2	0,40%	38.2s
3	0,54%	38.2s
4	0,33%	38.2s
5	0,21%	38.2s
6	0,31%	38.2s
7	0,27%	38.2s
8	0,34%	38.2s
9	0,25%	38.2s
10	0,33%	38.2s

Tabelle A.3.: Alle Benchmarkergebnisse von *Pyperf* auf Computer C

A.6. Alle Benchmarkergebnisse der LLVM-Tipps

Siehe Tabellen A.4 und A.5.

A.7. Weitere Ergebnisse des optimierten Algorithmus ohne Verhinderung von Multithreading

Siehe Tabelle A.6.

Durchlauf	Standardabweichung	Laufzeit
1	0,40%	23,8s
2	0,40%	23.8s
3	0,35%	23,9s
4	0,31%	23,8s
5	0,30%	23,8s
6	0,43%	23,8s
7	0,41%	23,8s
8	0,42%	23,8s
9	0,51%	23,9s
10	0,47%	23,9s

Tabelle A.4.: Alle Benchmarkergebnisse der *LLVM*-Tipps auf Computer B

Durchlauf	Standardabweichung	Laufzeit
1	0,56%	41.7s
2	0,62%	41.7s
3	0,52%	41.8s
4	0,60%	41.7s
5	0,54%	41.7s
6	0,48%	41.6s
7	0,60%	41.6s
8	0,62%	41.7s
9	0,61%	41.6s
10	0,50%	41.7s

Tabelle A.5.: Alle Benchmarkergebnisse der *LLVM*-Tipps auf Computer C

Durchlauf	Standardabweichung	Laufzeit
1	0,41%	12,4s
2	0,46%	12,49s
3	0,44%	12,6s
4	0,41%	12,5s
5	0,40%	12,5s
6	0,40%	12,8s
7	0,47%	12,3s
8	0,44%	15,5s
9	0,43%	12,5s
10	0,51%	12,5s

Tabelle A.6.: Ergebnisse des optimierten Algorithmus ohne Verhinderung von Multithreading auf Computer C