Karlsruher Institut für Technologie

Michael Nagel
michael.nagel@devzero.de

Autotuning im Umfeld von Google Go

– Diplomarbeit –

Fakultät für Informatik
Institut für Programmstrukturen und Datenorganisation

Gutachter:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	Dipl.-Inform. Thomas Karcher

Abgabedatum: 31. März 2014

Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst, und weder ganz oder in Teilen als Prüfungsleistung vorgelegt und keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die benutzten Werken im Wortlaut oder dem Sinn nach entnommen sind, habe ich durch Quellenangaben kenntlich gemacht. Dies gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen sowie für Quellen aus dem Internet.

Karlsruhe, den 31. März 2014

(Michael Nagel)

Kurzzusammenfassung

Diese Arbeit zeigt, wie durch Autotuning ausgewählte Aspekte im Umfeld von Go, Googles Programmiersprache mit Fokus auf Nebenläufigkeit und Parallelisierung, beschleunigt werden können. Parallele Programme profitieren besonders von automatischem Tuning, da in diesen oftmals eine Vielzahl wenig intuitiver Parameter gesetzt werden muss. Der vorliegende Ansatz verwendet einen bestehenden universellen Autotuner, der beliebige Go-Programme optimieren kann. Zwei Fallstudien untersuchen konkrete Möglichkeiten zur Beschleunigung, und zeigen wie ein Autotuner den Raum der möglichen Konfigurationen effizienter durchsuchen kann als ein menschlicher Entwickler. Gerade beim Onlinetuning können Laufzeitinformationen berücksichtigt werden, welche während der Entwicklung noch nicht zur Verfügung stehen. Anhand von Messungen mit verbreiteten Go-Bibliotheken wird die Leistungsfähigkeit der automatisch ermittelten Konfigurationen evaluiert. Darüber hinaus wird das Vorgehen des Autotuners im zugrundeliegenden Suchraum untersucht.

English Abstract

In this thesis we show how autotuning can speed up selected aspects in Go, Google's programming language with a focus on concurrency and parallelism. Parallel programs particularly benefit from automatic tuning, because they often require non-intuitive parameters to be set. In our approach, we use a universal autotuner that can optimize arbitrary Go programs. Two case studies explore concrete examples of program optimization, and show how an autotuner can search for configurations more efficiently than a human developer. Online-tuning can account for runtime information that is not available during development. We evaluate the performance of the optimized configurations using measurements based on widespread Go libraries. In addition, we analyze the behavior of autotuners in the underlying search space.

Inhaltsverzeichnis

1. Einleitung und Motivation

Dieses Kapitel motiviert die Verwendung von Autotuning zur automatischen Konfiguration von Software im Umfeld von Google Go. Zusätzlich bietet es einen kurzen Überblick über die Gliederung der weiteren Arbeit.

1.1 Automatische Konfiguration von Software

Moderne Hard- und Software wird immer komplexer, und nur durch eine gute Abstimmung der Software auf die zugrundeliegende Hardware ist die bestmögliche Leistung erreichbar [ABC+06]. Die Entwicklung weg von Einprozessorsystemen mit immer schneller getaktetem Kern hin zu Mehrkernsystemen mit vielen, aber langsamer getakteten Kernen, führt zu neuen Programmiermodellen [ABC+06].

Eine Vielzahl von Einstellungsmöglichkeiten erlaubt es, die Software auf die Hardware abzustimmen. Die optimale Konfiguration ist dabei oftmals nicht offensichtlich, sondern muss teilweise unter Zuhilfenahme von Expertenwissen aufwändig ermittelt werden. Die Anpassung auf ein bestimmtes Szenario ist dadurch teilweise schlichtweg unwirtschaftlich.

Eine Möglichkeit zur automatischen Ermittlung guter Konfigurationen – beispielsweise durch Autotuning – wird daher immer wichtiger. Autotuning ermittelt die Leistung einer Konfiguration und versucht durch Rekonfiguration der Software die Leistung zu verbessern. Autotuning kann kostengünstiger und oftmals schneller erfolgen, als eine Anpassung durch einen menschlichen Entwickler. Es kann darüber hinaus ganz oder teilweise im laufenden Betrieb stattfinden. Informationen, wie Eingabedaten oder die aktuelle Auslastung des Systems, die erst zur Laufzeit verfügbar sind, können mit in die Entscheidung einbezogen werden.

Die vorliegende Arbeit führt Autotuning in das Umfeld von Go ein. Go ist eine moderne Programmiersprache von Google mit Fokus auf Nebenläufigkeit und Parallelisierung. Die

Arbeit untersucht, inwiefern Autotuning helfen kann, die vorhandenen Hardwareressourcen bestmöglich mit den in Go gebotenen Sprachmittel auszunutzen.

Ein Schwerpunkt dieser Arbeit besteht in der Bestimmung und Analyse grundlegender Freiheitsgrade in Go-Bibliotheken und ihrer Umwandlung in autotuningfähige Parameter. Solche Freiheitsgrade zeichnen sich dadurch aus, dass sie in gewissen Szenarien eine signifikante Auswirkung auf die Geschwindigkeit haben, aber die optimale Konfiguration dabei nicht offensichtlich ist.

Der zweite Schwerpunkt ist die Untersuchung des Vorgehens eines bestehenden Autotuners bei der Optimierung der aus den Freiheitsgraden extrahierten Parameter. Dazu wird der der Userspace-Autotuner von Braun [Bra12] verwendet, weil dieser ohne fundamentale Änderungen an den Zielprogrammen in diese integriert werden kann.

Anhand von Messungen mit verbreiteten Go-Bibliotheken wird die Leistung der optimierten Versionen evaluiert und das Vorgehen des Autotuners untersucht und bewertet.

1.2 Gliederung dieser Arbeit

Kapitel 2 legt die Grundlagen zum Verständnis dieser Arbeit. Kapitel 3 zeigt den aktuellen Stand der Forschung auf und bietet Verweise auf verwandte Arbeiten. Kapitel 4 erläutert die Überlegungen vor Durchführung der Fallstudien. Die Fallstudien selbst sind in Kapitel 5 und Kapitel 6 dokumentiert. Kapitel 7 stellt die Implementierung vor. Die Diplomarbeit endet mit einer Zusammenfassung in Kapitel 8 und dem Ausblick auf weitere Forschungsmöglichkeiten.

2. Grundlagen

Dieses Kapitel stellt die Grundlagen zu den beiden großen Themengebieten dieser Diplomarbeit vor – Autotuning und Google Go.

Die Beispiele und Erläuterungen zu Autotuning in Abschnitt 2.2 bedienen sich bereits der Syntax von Go, deshalb führt der folgende Abschnitt zunächst die Grundlagen zu Go ein.

2.1 Die Sprache Go

Go [golb] ist eine moderne kompilierbare Programmiersprache mit automatischer Speicherbereinigung, welche zunächst als internes Projekt bei Google entwickelt wurde, seit der Veröffentlichung im November 2009 [GPT+] jedoch in einem offenen Prozess weiterentwickelt wird. Go ist auf eine kompakte Grammatik und schnelle Kompilation ausgelegt [gof].

Die folgende Einführung kann nur einen kurzen Überblick über die Sprache bieten. Darüber hinaus existiert eine offizielle interaktive Einführung in die Sprache [GoT] sowie diverse Bücher [Dox12, FB11, Sum12], die sich mit der Sprache beschäftigen.

2.1.1 Syntax von Go

Als eine „curly braces“-Programmierspache weist die Syntax von Go große Ähnlichkeit zu C auf. Ein syntaktischer Unterschied ist, dass in Go das Semikolon am Anweisungs- bzw. Zeilenende weggelassen werden kann, weil beides fast immer zusammenfällt [eff].

Wie in C ist = der Zuweisungsoperator, `:=` erlaubt die Deklaration einer neuen Variable, deren statischer Typ automatisch aus dem Ausdruck auf der rechten Seite bestimmt wird. Mehrfachzuweisungen der Form `x,y = 5,7` oder `x,y := 5,7` sind möglich.

Felder haben eine feste Länge. Ein Feld `buffer` mit Kapazität für 256 `byte`-Werte wird folgendermaßen deklariert:

```
1 var buffer [256]byte
```

Slices sind Ausschnitte aus einem Feld und haben im Gegensatz zu Feldern eine variable Länge. Die Deklaration und die Erstellung eines Slices erfolgen wie im folgenden Beispiel.

```
1 var buf []byte              // Deklaration
2 buf = make([]byte, length) // Erstellung
```

Typdeklarationen werden in Go immer von links nach rechts gelesen [Pik]. Eine Funktion `f`, welche eine Funktion `(int,int)->int` und eine Ganzzahl als Parameter übergeben bekommt und eine Funktion `(int,int)->int` zurückgibt, wird daher in Go folgendermaßen deklariert:

```
1 f func( func(int, int) int, int )  func(int, int) int
```

In C wird ein Funktionspointer `fp` mit vergleichbarer Funktionalität hingegen ohne klare Leserichtung deklariert [Pik].

```
1 int (*(*fp)(int (*)(int, int), int))(int, int)
```

Unter Beachtung dieser und einiger weiterer syntaktischer Unterschiede – sowie der automatischen Speicherbereinigung – können prozedurale C-Programme in Go nachprogrammiert und einfache Go-Programme verstanden werden.

Weil die Konzepte der Nebenläufigkeit und der Objektorientierung jedoch integraler Bestandteil von Go sind, werden Programme in Go oftmals anders strukturiert.

2.1.2 Nebenläufigkeit

Go legt ein starkes Augenmerk auf Nebenläufigkeit. Nebenläufigkeit bezeichnet die Bearbeitung unabhängiger Teilaufgaben. Nebenläufigkeit wird strikt von Parallelität unterschieden, und umfasst explizit nicht nur Multithreading der CPU sondern auch alle Arten von Asynchronität, wie zum Beispiel I/O-Operationen.

Nebenläufigkeit in Go [eff] basiert auf der von Tony Hoare entwickelten Theorie der CSP (Communicating Sequential Processes). Die Prozesse werden in Go als Goroutinen bezeichnet und können auch als leichtgewichtige Userspace-Threads angesehen werden. Der Datenaustausch zwischen Goroutinen erfolgt über Kommunikation durch sogenannte Channel, welche mit typisierten Unix-Pipes vergleichbar sind.

Der folgende Code startet in den Zeilen 3 und 4 mit dem Schlüsselwort `go` zwei Goroutinen. Die beiden neuen Goroutinen sowie die Goroutine, die `main` ausführt, laufen dann nebenläufig und ohne definierte Reihenfolge. Die Haupt-Goroutine wartet in Zeile 6 zur Synchronisierung auf zwei Nachrichten aus dem übergebenen Channel, ohne den Inhalt der Nachrichten weiter zu beachten. Beide Goroutinen senden in Zeile 11 jeweils eine Nachricht dorthin, und erst wenn beide gesendet und empfangen wurden, kann die `main`-Methode beendet werden.

```
1  func main() {
2    c := make(chan int)
3    go run(c)
4    go run(c)
5    for i := 0; i < 2; i++ {
6      <- c
7    }
8  }
9
10 func run(c chan int) {
11   c <- 1
12 }
```

2.1.3 Objektorientierung

Go unterstützt Objektorientierung in Form von Typen und Schnittstellen, bietet aber keine Klassen. Statt Vererbung wird Einbettung [eff] verwendet, eine Form der Komposition. Schnittstellen erlauben dynamisches Binden und dadurch Polymorphie [gof]. Pakete ähnlich zu jenen in Java erlauben eine Modularisierung des Codes [GoC].

Go definiert beispielsweise im Paket `sort` eine Sortierschnittstelle `sort.Interface`.

```
1  type Interface interface {
2    // Len is the number of elements in the collection.
3    Len() int
4    // Less reports whether the element with
5    // index i should sort before the element with index j.
6    Less(i, j int) bool
7    // Swap swaps the elements with indexes i and j.
8    Swap(i, j int)
9  }
```

Werden für einen neuen Datentypen `Sequence` [eff] alle drei Methoden definiert, erfüllt `Sequence` die `sort`-Schnittstelle, ohne dass dies explizit spezifiziert werden muss.

```
1  type Sequence []int
2  func (s Sequence) Len() int {
3    return len(s)
4  }
5  func (s Sequence) Less(i, j int) bool {
6    return s[i] < s[j]
7  }
8  func (s Sequence) Swap(i, j int) {
9    s[i], s[j] = s[j], s[i]
10 }
```

Die `Sort`-Methode, welche auf der Schnittstelle aufbaut, kann dann ohne weitere Angaben folgendermaßen verwendet werden.

```
1 func main() {
2   s := Sequence{3,4,1,5,2}
3   sort.Sort(s)
4   fmt.Println(s)
5 }
6
7 // Ausgabe:
8 // [1 2 3 4 5]
```

2.1.4 Compiler und Tools

Es existieren zwei bedeutende Compiler für Go: der offizielle Go-Compiler (je nach Architektur `8g` auf $x86$, `6g` auf $x86\_64$ und `5g` auf ARM) sowie `gccgo`, ein Go-Frontend für die GNU Compiler Collection (`gcc`). Im weiteren Verlauf der Arbeit wird, soweit nicht anderweitig gekennzeichnet, der offizielle `6g`-Compiler in Version 1.2 verwendet.

2.1.5 Ausgewählte Aspekte der Sprache Go

Der folgende Unterabschnitt stellt einige Aspekte der Sprache Go vor, die im weiteren Verlauf der Arbeit von besonderer Bedeutung sind.

2.1.5.1 GOMAXPROCS

Auf Mehrkernsystemen mit **k** Rechenkernen können jederzeit bis zu **k** Threads gleichzeitig ausgeführt werden, einer auf jedem Rechenkern. Das Betriebssystem verwaltet unabhängig davon **p** Prozesse gleichzeitig, wobei oftmals $\mathbf{p} \gg \mathbf{k}$ gilt. Der Scheduler des Betriebssystems sorgt per Zeitmultiplexing der Rechenkerne dafür, dass die Prozesse reihum (oder nach komplexeren Schedulingstrategien) auf den Rechenkernen zur Ausführung kommen.

Go bietet unabhängig von den Prozessen im Betriebssystem leichtgewichtige Threads im Userspace, die Goroutinen genannt werden [gof, eff]. Analog zum Multiplexing der Prozesse auf die Rechenkerne durch das Betriebssystem werden die Goroutinen von der Go-Laufzeitumgebung auf Prozesse im Betriebssystem verteilt. Das Umschalten zwischen verschiedenen Goroutinen ist allerdings um Größenordnungen schneller als das Umschalten zwischen Prozessen im Betriebssystem, weil Goroutinen eines Programms kooperativ in einem einzigen Adressraum laufen [Vyu].

Die Variable `GOMAXPROCS` der Go-Laufzeitumgebung bezeichnet die maximale Anzahl der zur Ausführung von Goroutinen genutzten Betriebssystem-Prozesse. Passend zur Variablen `GOMAXPROCS` existiert auch eine gleichnamige Methode zum Setzen der Variablen.

> `func GOMAXPROCS(n int) int`
>
> `GOMAXPROCS` sets the maximum number of CPUs that can be executing simultaneously and returns the previous setting. If $n < 1$, it does not change the current setting. The number of logical CPUs on the local machine can be queried with `NumCPU`. This call will go away when the scheduler improves.
>
> Quelle: [gomb]

Der Aufruf wird trotz der Ankündigung auf absehbare Zeit nicht verschwinden, da für Go 1.X API-Kompatibilität garantiert ist [go1a]. Der Wert kann auch beim Programmstart über die gleichnamige Umgebungsvariable festgelegt werden. Ein Wert von `GOMAXPROCS = 1` unterbindet Parallelität im Programm. Diese Einstellung ist momentan der Standard, sodass Aufrufe der `GOMAXPROCS`-Methode in vielen Programmen vorkommen.

2.1.5.2 C-Anbindung

Go erlaubt die direkte Einbindung von C-Code in Go-Programme mittels `cgo`.

> **`cgo`** enables the creation of Go packages that call C code.
>
> [...] The input file.go is a syntactically valid Go source file that imports the pseudo-package `"C"` and then refers to types such as `C.size_t`, variables such as `C.stdout`, or functions such as `C.putchar`.
>
> If the import of `"C"` is immediately preceded by a comment, that comment, called the preamble, is used as a header when compiling the C parts of the package.
>
> Quelle: [cgo]

Das folgende Programm gibt beispielsweise `1024` ($= 2^{5+5}$) aus und zeigt Aufrufe von Go nach C, die Einbindung von C-Bibliotheken sowie die Umwandlung primitiver Datentypen.

```
1  package main
2  // #cgo LDFLAGS: -lm
3  // #include <math.h>
4  // static double power(int a, int b) {
5  //  return pow((double)a, (double)b);
6  // }
7  import "C"
8
9  func main() {
10   x := 5+5
11   res := C.power(2, C.int(x))
12   print(int(res))
13 }
```

2.1.5.3 First-Class-Funktionen

Go unterstützt First-Class-Funktionen. Funktionen können zur Laufzeit erstellt, in Variablen gespeichert und als anonyme Funktionen verwendet werden.

Das folgende Programm gibt `8` und `15` aus.

```
1  func main() {
2    add := func(a, b int) int {
3      return a+b
4    }
5    x, y := 5, 3
6
7    print(add(x,y))
8    print(func(a,b int) int {
9      return a * b;
10   }(x,y))
11 }
```

`add` ist eine Variable, die eine Funktion zum Addieren zweier Zahlen enthält. Die Variable wird in Zeile 2ff deklariert und die Funktion wird in Zeile 7 aufgerufen. Die Funktion zur Multiplikation aus Zeile 8ff wird als anonyme Funktion in Zeile 10 direkt mit den Parametern `x,y` aufgerufen.

2.1.5.4 defer-Anweisung

`defer` dient der Ausführung einer Anweisung beim Verlassen der aktuellen Funktion. `defer` wird insbesondere dazu genutzt, Ressourcen beim Verlassen einer Funktion wieder freizugeben, unabhängig auf welchem Wege die Funktion verlassen wird.

```
1  func CopyFile(dstName, srcName string) (written int64, e error) {
2    src, e := os.Open(srcName)
3    if e != nil {
4      return
5    }
6    defer src.Close()
7
8    dst, e := os.Create(dstName)
9    if e != nil {
10     return
11   }
12   defer dst.Close()
13
14   return io.Copy(dst, src)
15 }
```

Der Code im obigen Beispiel stammt von der Go-Webseite [Ger] und stellt sicher, dass alle geöffneten Dateien auch wieder geschlossen werden. Er funktioniert korrekt, auch wenn es zu einem Fehler kommt, nachdem eine oder beide Dateien bereits geöffnet wurden.

Dadurch, dass zusammengehörige `Open`- und `Close`-Aufrufe auch im Code zusammenstehen, ist es einfach zu überprüfen, ob alle Ressourcen korrekt freigegeben werden.

2.2 Autotuning

Der folgende Abschnitt stellt die Grundlagen zu Autotuning vor. Autotuning ermöglicht die automatische Anpassung einer Anwendung innerhalb festgelegter Grenzen. Das Ziel des Autotunings ist die Beschleunigung einer Anwendung. Die Idee geht auf Veröffentlichungen von Bilmes et.al. sowie von Frigo und Johnson aus den späten Neunziger Jahren zurück, siehe dazu auch Abschnitt 3.1.2.

2.2.1 Funktionsweise von Autotuning

Software besitzt oftmals Freiheitsgrade in Form von impliziten oder expliziten Parametern, welche zwar keinen Einfluss auf das Ergebnis des Programms haben, dabei die Laufzeit des Programms aber mehr oder weniger entscheidend beeinflussen.

Abhängig davon, wie bewusst sich der Programmierer über die Bedeutung eines angemessenen Wertes für den Parameter ist, kann es sich dabei um eine magische Zahl[1] im Code handeln, ein `#define` für den Präprozessor, oder einen Parameter, welcher per Kommandozeile oder in einer Konfigurationsdatei gesetzt wird.

Wie dieser Wert konkret gesetzt wird, könnte per Würfelwurf entschieden worden sein, oder durch scharfes Nachdenken statisch oder in Form einer einfachen Heuristik, z.B. `Anzahl Threads := Anzahl Rechenkerne`. Je nach verfügbarer Zeit und Können des Entwicklers kann der Wert durch Profiling verbessert werden.

Beim Profiling lässt sich der Entwickler mit Hilfe geeigneter Tools Informationen dazu anzeigen, wie oft bestimmte Codeabschnitte aufgerufen wurden und wie lange die Ausführung gedauert hat – meist auf der Ebene einzelner Funktionen. Aus diesen Informationen kann der Entwickler Rückschlüsse auf mögliche Leistungsprobleme ziehen.

Autotuning unterscheidet sich von Profiling dadurch, dass der Optimierungsvorgang automatisch durchgeführt wird. Ein Programm, der Autotuner, übernimmt die Aufgabe, systematisch bestimmte Konfigurationen auszuprobieren, deren Laufzeit zu bestimmen und eine gute/optimale Konfiguration zu finden. Erforderlich ist dazu eine Möglichkeit zur Anpassung der Konfiguration und zur Rückmeldung der Laufzeit, sodass eine Heuristik im Autotuner eine neue Konfiguration setzen kann.

[1] „[Eine magische Zahl ist] ein im Sourcecode eines Programms auftauchender Zahlenwert[...], dessen Bedeutung sich nicht unmittelbar erkennen lässt.“ [Mag]

Ein wichtiger Vorteil von Autotuning gegenüber Profiling ist, dass nicht nur einmalig auf einer Entwicklermaschine mit willkürlichen Daten gemessen und optimiert wird. Es kann in der Produktivumgebung mit realistischen Daten oder sogar im laufenden Betrieb durchgeführt werden.

2.2.2 Kurzbeispiel: Autotuning eines Programms

Das Beispielprogramm wartet wiederholt in einer Schleife. Wie lange es jeweils wartet, hängt von den Werten der Variablen `x` und `y` ab. Angenommen, das Warten entspreche der gewünschten Funktionalität des Programms, aber die Parameter `x` und `y` seien Freiheitsgrade im Sinne des vorigen Unterabschnitts.

```
1 func main_Untuned() {
2   x, y := -200, -100
3
4   for i := 1; i <= ITERATIONS; i++ {
5     tictoc.Tic()
6     time.Sleep(time.Duration(
          100 + x*x/200 + y*y/100) * time.Millisecond)
7     tictoc.Toc()
8   }
9 }
```

Die Methoden `tictoc.Tic()` und `tictoc.Toc()` dienen dem Starten und Stoppen einer Stoppuhr. Sie sind an die Stoppuhr-Schnittstelle von MATLAB [tic] angelehnt.

Ohne Änderungen am Code findet noch keine Optimierung statt und die Ausgabe des Programms lautet wie folgt:

```
1  toc --- 0.400232 11111111111111111111
2  toc --- 0.400233 11111111111111111111
3  toc --- 0.400233 11111111111111111111
4  toc --- 0.400241 11111111111111111111
5  toc --- 0.400113 11111111111111111111
6  toc --- 0.400235 11111111111111111111
7  toc --- 0.400167 11111111111111111111
8  toc --- 0.400640 11111111111111111111
9  toc --- 0.400227 11111111111111111111
10 toc --- 0.400286 11111111111111111111
11 toc --- 0.400227 11111111111111111111
12 toc --- 0.400143 11111111111111111111
```

Die erste Zahl (circa $0,4 = (100 + (-200)^2/200 + (-100)^2/100) * 10^{-3}$) entspricht der Laufzeit in Sekunden, dahinter befindet sich eine Repräsentation als Balkendiagramm.

Die Optimierung der Laufzeit des Programms entspricht einer Minimierung der mathematischen Funktion

$$f(x, y) := 100 + \frac{x^2}{200} + \frac{y^2}{100}.$$

Ein Minimum wird angenommen für $(x, y) = (0, 0)$.

Die folgende abgeänderte Version des Programms ist autotuningfähig.

```
1  func main_GoFullWalker() {
2    x, y := -200, -100
3    tuner := CreateGoFullWalker()
4    tuner.AddParameter(&x, "x", -200, 200, 100)
5    tuner.AddParameter(&y, "y", -100, 100, 50)
6
7    for i := 1; i <= ITERATIONS; i++ {
8      tuner.Tic()
9      time.Sleep(time.Duration(
           100 + x*x/200 + y*y/100) * time.Millisecond)
10     tuner.Toc()
11   }
12   tuner.Destroy()
13 }
```

In Zeile 3 wird eine Autotuner-Instanz erstellt. Der Autotuner kann verschiedene Optimierungsstrategien verwenden. Der `GoFullWalker` ist eher ein Erkunder des Suchraums als ein echter Optimierer. Er durchläuft den Suchraum vollständig, und aktiviert systematisch alle darin befindlichen Konfigurationen.

In den Zeilen 4 und 5 werden die beiden Parameter beim Autotuner registriert. Dabei werden jeweils die folgenden Argumente übergeben:

- ein Zeiger auf den zu optimierenden Parameter
- eine kurze Beschreibung des Parameters für Protokollausgaben
- der zulässige Minimalwert
- der zulässige Maximalwert
- die Schrittweite, in welcher der Suchraum durchsucht werden soll

Die Zeilen 8 und 10 markieren den Anfang und das Ende des Messabschnitts, dessen Laufzeit optimiert werden soll. Beim Betreten des Messabschnitts hat der Autotuner die Möglichkeit, die gewünschte Konfiguration zu setzen.

In Zeile 12 wird der Autotuner schließlich beendet.

Die Ausgabe des angepassten Programms lautet wie folgt:

```
creating GoFullWalker
adding parameter x with values from -200..200 in steps of 100
adding parameter y with values from -100..100 in steps of 50
[ x   -200 ][ y   -100 ] --- 0.400225 11111111111111111111
[ x   -100 ][ y   -100 ] --- 0.250225 111111111111
[ x      0 ][ y   -100 ] --- 0.200229 1111111111
[ x    100 ][ y   -100 ] --- 0.250151 111111111111
[ x    200 ][ y   -100 ] --- 0.400321 11111111111111111111
[ x   -200 ][ y    -50 ] --- 0.325160 1111111111111111
[ x   -100 ][ y    -50 ] --- 0.175146 11111111
[ x      0 ][ y    -50 ] --- 0.125221 111111
[ x    100 ][ y    -50 ] --- 0.175130 11111111
[ x    200 ][ y    -50 ] --- 0.325220 1111111111111111
[ x   -200 ][ y      0 ] --- 0.300225 111111111111111
[ x   -100 ][ y      0 ] --- 0.150153 1111111
destroying GoFullWalker
```

Die Anzahl der Iterationen ist in diesem Beispiel nicht hoch genug um das globale Optimum zu finden, aber es ist zu erkennen, dass verschiedene Konfigurationen mit verschiedenen Laufzeiten aktiv waren. Die Konfiguration $(x, y) = (0, -50)$ war mit 0,13s Laufzeit beispielsweise deutlich schneller als andere Konfigurationen.

Eine Visualisierung des Suchraums ist in Abbildung 2.1 zu sehen.

2.2.3 Klassifikation von Autotuning-Systemen

Autotuning kommt in unterschiedlichen Ausprägungen vor. Autotuning-Systeme lassen sich anhand verschiedener Kriterien klassifizieren. Die folgende Aufstellung orientiert sich an Schaefers Klassifikation von Autotuning-Systemen [Sch10].

2.2.3.1 Tuningtyp

Der Tuningtyp gibt an, ob das Autotuning im laufenden Betrieb oder vor dem produktiven Programmlauf stattfindet. Es wird dementsprechend zwischen Offline- und Onlinetuning unterschieden.

Beim Offlinetuning wird eine Maschine (meist die konkrete Maschine, auf welcher das Programm später ausgeführt werden soll) genutzt, um mit Benchmarks zu ermitteln, welche Konfiguration sich für die erwartete Last am besten eignet. Diese wird aktiviert und erst dann startet der produktive Programmlauf.

Beim Onlinetuning wird die Messung im laufenden Betrieb durchgeführt, sodass ein vollständiger Regelkreis entsteht. Dadurch wird eine adaptive Anpassung an die konkret verwendeten Daten und die aktuelle Systemauslastung möglich.

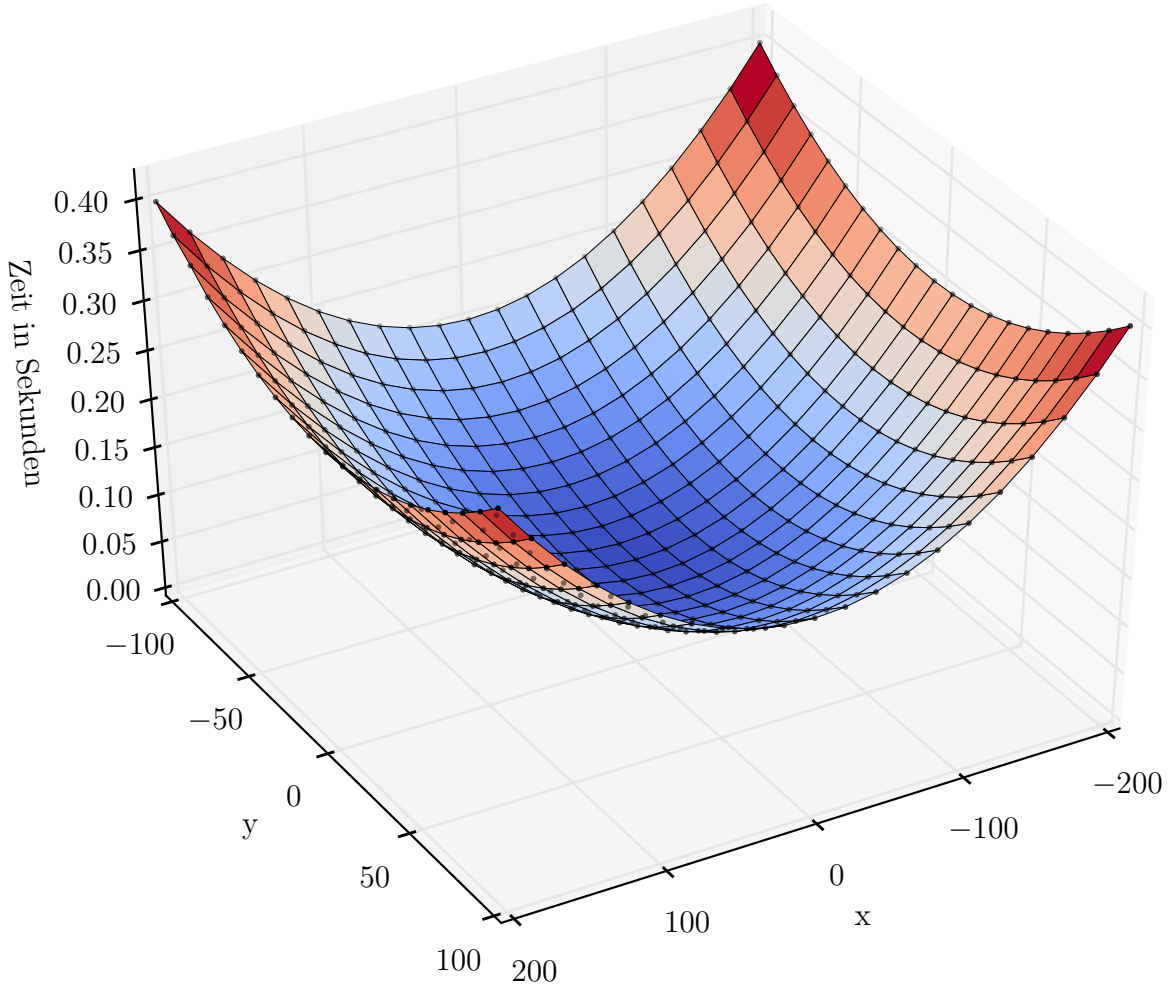


Abbildung 2.1: Visualisierung des Autotuner-Suchraums im Beispielprogramm. Um eine aussagekräftige Abbildung zu erhalten, wurde die Schrittweite reduziert und die Anzahl der Durchläufe erhöht.

2.2.3.2 Tuningstrategie

Die Art der Tuningstrategie gibt an, welche Rückschlüsse aus den Messwerten für die weitere Optimierung gezogen werden. Es wird zwischen modellbasierten und suchbasierten Ansätzen unterschieden.

Modellbasierte Ansätze bestimmen Kenngrößen des Systems und verfügen über Informationen darüber, wie sich die Größen auf die Laufzeit des zu optimierenden Programms auswirken. Nach Bestimmung der Kenngrößen kann direkt die bestmögliche Konfiguration berechnet werden, ohne dass weitere Konfigurationen getestet werden müssen.

Suchbasierte Ansätze messen diverse Konfigurationen und versuchen dabei durch Such- bzw. Optimierungsverfahren eine gute Konfiguration zu finden. Bei diesem Ansatz ist kein Wissen über die Bedeutung der verschiedenen Parameter erforderlich.

3. Stand der Forschung

Dieses Kapitel stellt den aktuellen Stand der Forschung zum Thema „Autotuning im Umfeld von Google Go“ vor. Abschnitt 3.1 erläutert zunächst verschiedene Autotuning-Ansätze, Abschnitt 3.2 geht auf die Beschleunigung von Go ein.

3.1 Autotuning

Autotuning wurde Ende der 90er Jahre entwickelt und die verschiedenen Ausprägungen von Autotuning sind auch heute noch Gegenstand aktueller Forschung.

3.1.1 Optimierende Compiler, Quelltexttransformationen

Ein wichtiger Schritt beim Kompilieren von Quellcode ist die Optimierung des erzeugten Programmcodes. Die Hilfeseite der GNU Compiler Collection [Opt] listet über 100 entsprechende Kommandozeilenparameter. Das Dragon Book (Compilers: Principles, Techniques, and Tools [ALSU07]) widmet der Thematik mehrere Kapitel.

Viele klassische Optimierungen, wie beispielsweise die Konstantenfaltung oder das Entfernen gemeinsamer Teilausdrücke,[1] sind kein Autotuning im Sinne dieser Arbeit, weil der modifizierte Code nicht ausgeführt wird, um zu messen, ob dadurch eine Beschleunigung erzielt wurde.

Einige Compiler, wie zum Beispiel der Intel Fortran Compiler, unterstützen neben klassischen Optimierungen auch PGO (Profile-Guided Optimizations) [Int]. PGO sind Optimierungen, bei denen das Programm nach einer ersten Kompilierung ausgeführt wird, um ein Anwendungsprofil zu erstellen. Die Informationen im Anwendungsprofil nutzt der

[1] Bei der Konstantenfaltung werden zur Kompilierzeit bekannte Werte so weit wie möglich propagiert. `r=1; pi=3.14; d=2*r; A=pi/4*d*d` wird ersetzt durch `A=3.14`.

Beim Entfernen gemeinsamer Teilausdrücke wird ein wiederholt verwendeter Wert zwischengespeichert, anstatt ihn doppelt zu berechnen. `b=sqrt(x)+sqrt(x)` wird ersetzt durch `a=sqrt(x); b=a+a`.

Compiler bei nachfolgenden Kompiliervorgängen, um schnelleren Code zu generieren, weil beispielsweise bessere Sprungvorhersagen möglich sind.

Die Verwendung von PGO kommt Autotuning näher als die klassischen Optimierungen. Einige weitergehende Ansätze, wie der Insieme-Compiler, entsprechen vollumfänglich dem in dieser Arbeit aufgezeigten Autotuning-Schema.

3.1.1.1 Insieme

Insieme [Tho13, Ins] von Fahringer et.al. an der Universität Innsbruck ist ein Quelltexttransformator für C und C++. Insieme ermöglicht die Beschleunigung von Code für verschiedene Parallelisierungsstandards wie OpenMP, OpenCL und MPI.

Eine INSPIRE genannte Zwischensprache ermöglicht sprachunabhängige Optimierungen. Der transformierte Code enthält Aufrufe der Insieme-Laufzeitumgebung. Diese enthält einen Loop-Scheduler und weitere Heuristiken, die zur Laufzeit für eine schnellstmögliche Abarbeitung des Programms sorgen.

Die Autotuning-Aspekte von Insieme sind modellbasiertes Onlinetuning.

3.1.2 Autotuning in numerischen Bibliotheken

Die ersten Autotuner in der Form, wie sie in dieser Arbeit untersucht werden, lassen sich auf Veröffentlichungen von Bilmes et.al. (PHiPAC) sowie von Frigo und Johnson (FFTW) zurückverfolgen. Beide Gruppen verwenden Autotuning zur Beschleunigung numerischer Bibliotheken.

3.1.2.1 FFTW

FFTW (Fastest Fourier Transform in the West) [FJ97, FJ98, ffta] von Frigo und Johnson am MIT ist ein in OCaml geschriebener Codegenerator, der effizienten Code zur Berechnung der schnellen Fourier-Transformation erzeugt.

Für Teilprobleme der schnellen Fourier-Transformation sind verschiedene Quellcodefragmente („codelets“) vordefiniert und FFTW ermittelt in einem Benchmarklauf eine günstige Kombination der Fragmente. Die ermittelten Informationen können in Form von „wisdom files“ exportiert und für spätere Berechnungen wieder importiert werden.

FFTW wird auch heute noch aktiv weiterentwickelt [FFTb] und kommt in bedeutenden numerischen Programmpaketen wie z.B. MATLAB zum Einsatz [MATa]. FFTW implementiert Offlinetuning und nutzt dabei in einem modellbasierten Ansatz viel Domänenwissen über die schnelle Fourier-Transformation.

3.1.2.2 PHiPAC

PHiPAC (Portable High Performance ANSI C) [BAD+96, BACD97, BAV+] von Bilmes et.al. an der UC Berkeley ist FFTW konzeptionell ähnlich, hat jedoch die Beschleunigung der BLAS [LHKK79] zum Ziel.

PHiPAC ist ein Codegenerator, der ein Maschinenmodell („machine specifications file“) mit Angaben wie `num_double_prec_registers` oder `L1_cache_size` verwendet, um auf die Maschine abgestimmte BLAS zu generieren.

PHiPAC wird im Gegensatz zu FFTW nicht mehr aktiv verwendet, ist aber als Vorläufer von ATLAS von Bedeutung. PHiPAC implementiert Offlinetuning in einem modellbasierten Ansatz.

3.1.2.3 ATLAS

ATLAS (Automatically Tuned Linear Algebra Software) [WD98, Aut] von Whaley und Dongarra an der Universität Tennessee ist eine Bibliothek, welche die BLAS für dichtbesetzte Matrizen implementiert und dabei Autotuning zur Leistungsverbesserung nutzt.

ATLAS verfährt nach einem als AEOS (Automated Empirical Optimization of Software) bezeichneten Ansatz. AEOS erfordert, dass geschwindigkeitskritische Routinen extrahiert werden, die Software über Parameter an unterschiedliche Umfelder anpassbar sein muss, sowie dass verlässliche Timer und angemessene Suchheuristiken bereitgestellt werden. Daraufhin kann ein System wie ATLAS automatisch optimiert werden.

ATLAS implementiert Offlinetuning, und die verwendeten Parameter resultieren ähnlich wie bei PHiPAC in der Erstellung eines Maschinenmodells.

3.1.2.4 OSKI

OSKI (Optimized Sparse Kernel Interface) [VDY05] von Vuduc, Demmel und Yelick an der UC Berkeley implementiert die BLAS für dünnbesetzte Matrizen.

Die Berechnungen der BLAS hängen bei dünnbesetzten Matrizen viel stärker von den verwendeten Daten ab als dies bei dichtbesetzten Matrizen der Fall ist. Daher verwendet OSKI einen Onlinetuningansatz. Das Tuning erfolgt mit den bei den Berechnungen verwendeten Matrizen. Es erfolgt allerdings nicht implizit, sondern in einem expliziten Tuningschritt vor der eigentlichen Berechnung.

3.1.3 Universell einsetzbare Autotuner

Universell einsetzbare Autotuner sind so konzipiert, dass Entwickler von beliebigen Anwendungsprogrammen Tuningfähigkeiten in ihre Software einbauen können. Oftmals wird dies realisiert, indem ein Entwickler Freiheitsgrade und Messabschnitte im eigenen Programm markiert und der Autotuner daraufhin die Optimierung übernimmt.

3.1.3.1 Active Harmony

Active Harmony [TCH02] von Tapus, Chung und Hollingsworth an der University of Maryland ist ein Client-Server-Autotuner, der Onlinetuning hinsichtlich benutzerdefinierter Qualitätsmetriken erlaubt.

Active Harmony wird insbesondere zum Autotuning auf höherer Ebene genutzt, wobei ganze Bibliotheken durch alternative Implementierung mit anderem Laufzeitverhalten ausgetauscht werden. Mittels einer Spezifizierungssprache können kompatible Bibliotheken deklariert werden.

Active Harmony arbeitet suchbasiert im Online-Modus. Dabei kommt mit Parallel Rank Ordering ein speziell auf den Clusterbetrieb zugeschnittener Optimierungsalgorithmus zum Einsatz.

3.1.3.2 MATE

MATE (Monitoring, Analysis and Tuning Environment) [MMML04] von Morajko, Morajko, Margalef und Luque an der Universität Barcelona ist ein universell einsetzbarer Onlinetuner. Der Schwerpunkt von MATE liegt auf der Optimierung von verteilten wissenschaftlichen Anwendungen, welche PVM (Parallel Virtual Machine) [pvm] verwenden.

MATE kann in zwei Modi betrieben werden. Der erste, automatische Modus erfordert keine Kooperation der Anwendung. Er erlaubt die Anpassung von Parametern des Betriebssystems, der Standardbibliotheken sowie der PVM-Konfiguration. Im zweiten, kooperativen Modus können darüber hinaus noch anwendungsspezifische Parameter wie die Konfiguration des Arbeiterpools angepasst werden.

MATE führt Onlinetuning durch. Der automatische Ansatz ist modellbasiert, der kooperative Ansatz ist suchbasiert.

3.1.3.3 Perpetuum und Userspace-Autotuner

Der in dieser Arbeit verwendete Userspace-Autotuner [Bra12] von Braun basiert auf dem im Betriebssystemkern implementierten Perpetuum [Sch09] von Schwedes. Beide wurden am Institut für Programmstrukturen und Datenorganisation am Karlsruher Institut für Technologie entwickelt. Der Userspace-Autotuner zeichnet sich dadurch aus, dass er von überschaubarem Umfang ist und ohne fundamentale Änderungen an den Zielprogrammen in diese integriert werden kann.

Der Userspace-Autotuner unterstützt Onlinetuning in einem suchbasierten Ansatz. Bei der Suche der besten Konfiguration im Raum der möglichen Konfigurationen stehen drei verschiedene Optimierungsstrategien zur Verfügung.

- Der `FullOptimizer` aktiviert systematisch jede Konfiguration im Suchraum.

- Der `HeuristicOptimizer` durchsucht den Suchraum zunächst zufällig und führt dann eine Optimierung anhand des Bergsteigeralgorithmus durch.
- Der `NelderMeadOptimizer` verwendet ein an das Nelder-Mead- bzw. Downhill-Simplex-Verfahren [NM65] angelehntes Verfahren. Der Userspace-Autotuner konstruiert von einem Startpunkt ausgehend zunächst das Startsimplex zufällig. Die Ecken des Simplex werden dann entsprechend der für das Verfahren charakteristischen Regeln solange verschoben, bis keine weitere Verbesserung mehr möglich ist. Das Verfahren ist ursprünglich für kontinuierliche Suchräume gedacht; der Userspace-Autotuner arbeitet in geschlossenen, diskreten Suchräumen. Er führt daher abschließend lokale Suche durch, um ganzzahlige Konfigurationen in der Nähe des berechneten Optimums zu prüfen.

3.1.3.4 XJava

XJava [Ott13] von Otto am Karlsruher Institut für Technologie ist eine Java-Dialekt zur datenstromorientierten Programmierung. XJava dient der Entwicklung und Optimierung von Software für Multikernrechner.

In der datenstromorientierten Programmierung werden Daten von Filterknoten transformiert. Die Ein- und Ausgänge der Filterknoten werden von einem Fließband miteinander verbunden, um den Datenstrom zu leiten. XJava definiert parallele Filter, deren Parameter – wie Threadanzahl oder Replikationsanzahl – von der Laufzeitumgebung automatisch im Produktivbetrieb optimiert werden.

XJava ist suchbasiertes Onlinetuning, wobei die zu optimierenden Parameter statisch vorgegeben sind.

3.1.3.5 Atune

Atune [SPT09] von Schaefer am Karlsruher Institut für Technologie ist ein suchbasierter Offlinetuner, der mit der Annotationssprache Atune-IL versehene Programme optimiert.

Atune-IL ist eine sprachunabhängige Pragma-Sprache, die von einem Präprozessor verarbeitet wird. Dieser generiert mehrere Codevarianten, die offline optimiert werden. Atune erreicht Suchraumreduktion durch entsprechende Annotationen in Atune-IL und durch die Deklaration unabhängiger Sektionen, die unterschiedliche Parameter optimieren.

3.1.4 Vergleich der Autotuning-Systeme

Die folgende Tabelle bietet einen schnellen Überblick über die Merkmale der verschiedenen Autotuning-Systeme. Die Merkmale orientieren sich an der in Abschnitt 2.2.3 vorgestellten Klassifikation von Autotuning-Systemen nach Schaefer [Sch10].

Die Spalte „Typ“ gibt an, ob ein Autotuner Online- oder Offlinetuning implementiert. Die Spalte „Strategie“ gibt an, ob es sich um einen modell- oder suchbasierten Autotuner handelt. In der letzten Spalte wird der Anwendungsbereich kurz umrissen.

Autotuner	Typ	Strategie	Anwendungsbereich
Insieme	Online	Modell	Parallele C/C++ Programme
FFTW	Offline	Modell	Fast Fourier Transformation
ATLAS	Offline	Modell	BLAS für dichtbesetzte Matrizen
OSKI	Online	Modell	BLAS für dünnbesetzte Matrizen
Active Harmony	Online	Suche	universell
MATE	Online	Modell, Suche	PVM-Programme, bedingt universell
Userspace-Autotuner	Online	Suche	universell, Fokus C/C++
XJava	Online	Modell	Java-Dialekt
Atune	Offline	Suche	universell

3.2 Beschleunigung von Go

In Go geschriebene Programme sind bereits vergleichsweise schnell [ben]; dennoch gehen mit neuen Veröffentlichungen der Sprache teilweise signifikante Beschleunigungen der Laufzeitumgebung und der Standardbibliothek einher [go1b]. Dies ist auf das vergleichsweise geringe Alter der Sprache zurückzuführen.

Neben der allgemeinen Beschleunigung von Go in neueren Versionen der Sprache gibt es verschiedene Ansätze zur Beschleunigung von einzelnen Go-Anwendungen.

3.2.1 Testing-Bibliothek und Benchmarks

Das Paket `testing` [gola] aus der Go-Standardbibliothek implementiert Benchmarking-Methoden. Diese erlauben die Definition von Messabschnitten, die dann adaptiv so oft ausgeführt werden, bis eine verlässliche Messung der Laufzeit möglich ist. Eine Anpassung beziehungsweise ein Autotuning der ausgeführten Codeabschnitte ist jedoch nicht vorgesehen.

3.2.2 Profiling in Go

Profiling zur Verbesserung der Laufzeit wird auch im Umfeld von Go betrieben. Exemplarisch sei hier auf eine Fallstudie [Pow] von Powers verwiesen. In dieser beschreibt er, welche Veränderungen er an einem Go-Programm vornimmt, um die Ausführungsgeschwindigkeit zu verbessern. Alle vorgenommenen Änderungen sind jedoch dauerhaft und die Möglichkeit einer dynamischen Umschaltung wird nicht in Betracht gezogen. Der Ansatz ist nicht automatisiert und stark an das untersuchte Anwendungsprogramm angepasst.

3.2.3 Autotuning in Go

Insbesondere über die C-Schnittstelle in Go ist es möglich, bestehende universell einsetzbare Autotuner an Go-Code anzubinden. Bislang sind jedoch keine Arbeiten veröffentlicht, die sich mit Onlinetunern speziell im Umfeld von Go beschäftigen.

Die vorliegende Arbeit soll diese Lücke schließen.

4. Autotuning in Go

Dieses Kapitel stellt die im weiteren Verlauf der Arbeit genutzte Nomenklatur vor. Außerdem werden Kriterien für die Suche nach relevanten Freiheitsgraden vorgestellt.

Die Konzeption der Fallstudien, welche an verschiedenen Beispielen die Umwandlung eines Freiheitsgrades in einen autotuningfähigen Parameter beschreiben, sind als Teil der jeweiligen Fallstudie in den nachfolgenden Kapitel zu finden.

4.1 Autotuning-Nomenklatur

Dieser Abschnitt definiert die Begriffe „Freiheitsgrad“, „Parameter“, „Konfiguration“ und „Rekonfiguration“ und grenzt sie gegeneinander ab.

4.1.1 Freiheitsgrad

Das abstrakte Konzept einer Entscheidungsmöglichkeit, die Einfluss auf die Geschwindigkeit, aber nicht das Ergebnis eines Programmes hat, wird als **Freiheitsgrad** bezeichnet. Die folgenden drei Situationen liefern Beispiele für Freiheitsgrade.

- Verschiedene Teilaufgaben in einem Programm können oftmals nacheinander oder aber gleichzeitig bearbeitet werden. Zur Umsetzung der Gleichzeitigkeit können beispielsweise Threads verwendet werden. Die Entscheidung, ob ein Programm sequentiell oder aber durch eine bestimmte Anzahl Threads parallel abgearbeitet werden soll, stellt einen Freiheitsgrad dar.
- Programme können Informationen auf verschiedene Art und Weise vorhalten. So kann manchmal entweder eine verkettete Liste oder aber ein Feld zur Datenhaltung genutzt werden. Die Entscheidungsmöglichkeit bei der Wahl der Datenstruktur ist ein Freiheitsgrad.

- Manche Stapelverarbeitungsprogramme können eine variable Anzahl von Datensätzen auf einmal einlesen, diese Datensätze bearbeiten und die Ergebnisse gemeinsam zurückschreiben, bevor die nächsten Datensätze eingelesen werden. Bei der Bündelung der Datensätze existiert ein Freiheitsgrad.

4.1.2 Parameter

Eine einfache Variable, die einen Freiheitsgrad steuert, wird als **Parameter** bezeichnet. Die folgenden drei Variablen sind Beispiele für Parameter, sofern eine Rekonfiguration die angedeutete Verhaltensänderung im Programm bewirkt.

- Integer-Variable `thread_count`.
- Boolean-Flag `use_array`.
- Integer-Variable `blocksize`.

4.1.3 Konfiguration

Eine konkrete Parameterbelegung wird als **Konfiguration** bezeichnet.

Beispiel(e):

- `thread_count=3`
- `use_array=true`
- `blocksize=5`

Diese drei Werte können drei Konfigurationen in drei eindimensionalen Parameterräumen darstellen. Es können allerdings Abhängigkeiten zwischen den Parametern bestehen, sodass die Kombination der drei Werte auch eine einzelne Konfiguration in einem dreidimensionalen Parameterraum darstellen kann.

4.1.4 Rekonfiguration

Wenn der Autotuner entscheidet, dass die Konfiguration gewechselt werden soll, findet daraufhin eine sogenannte **Rekonfiguration** statt. Dies bedeutet zunächst einmal, dass die entsprechenden Parameter abgeändert werden. Der Wechsel ist jedoch im Allgemeinen mit zusätzlichem Aufwand verbunden. Die folgenden Beispiele verdeutlichen möglichen Rekonfigurationsaufwand.

- Änderungen der Threadanzahl erfordern das Erstellen oder Vernichten von Threads.
- Änderungen der Datenstruktur erfordern das Erstellen und Befüllen der neuen Datenstruktur und die Freigabe der bestehenden Datenstruktur.
- Änderungen der Blockgröße erfordern gegebenenfalls nur eine einfache Abänderung der Schleifengrenzen für die nachfolgenden Durchläufe.

Der Rekonfigurationsaufwand kann unmittelbar oder mittelbar sein. Unmittelbarer Rekonfigurationsaufwand ist Laufzeit, die in der Rekonfigurationsmethode verbracht wird. Mittelbarer Rekonfigurationsaufwand ist Laufzeit, die später verloren geht, beispielsweise durch eine Verlangsamung aufgrund von geleerten Caches.

4.2 Explizite Rekonfiguration

Der Userspace-Autotuner geht bislang implizit davon aus, dass beim Betreten eines Messabschnitts eine neue Konfiguration aktiviert wird. Die explizite Rekonfiguration trennt die Rekonfiguration von der Messwertbestimmung. Die Zeitmessung erfolgt weiterhin über das Betreten und Verlassen eines Messabschnitts. Die Rekonfiguration der Anwendung erfolgt unabhängig davon durch einen expliziten Aufruf der `Reconfigure`-Methode des Autotuners.

In der `Reconfigure`-Methode ändert der Autotuner die Konfiguration ab und ruft zudem einen `OnReconfigure`-Callback in der Anwendung auf. Der Code zur Anpassung der Anwendung an die neue Konfiguration findet in diesem Callback einen eindeutigen Platz.

Die explizite Rekonfiguration bietet zudem eine gute Möglichkeit, zu kontrollieren, inwiefern Zeitmessungen durch mitgestoppten Rekonfigurationsaufwand verfälscht werden. Mit der bestehenden C-Schnittstelle des Userspace-Autotuners findet die Rekonfiguration zwangsläufig nach dem Betreten eines Messabschnitts statt. Die Rekonfiguration ist mit Aufwand verbunden, der der neuen Konfiguration zugeschlagen wird. Das funktioniert gut, solange der Aufwand gering und gleichmäßig verteilt ist. Im Allgemeinen ist der Aufwand jedoch nicht unbedingt gering und hängt nicht nur von der neuen Konfiguration ab, sondern auch von der vorhergegangenen, gegebenenfalls sogar von allen vorangegangenen.

Zur Veranschaulichung sei ein Parameter mit drei möglichen Konfigurationen **A**, **B** und **C** gegeben. Die Ausführungskosten und der Rekonfigurationsaufwand sind in Tabelle 4.1 und Tabelle 4.2 aufgeführt.

In diesem Szenario misst ein Autotunerdurchlauf, der die Konfigurationen in der Reihenfolge **A**→**B**→**C** aktiviert, bei **B** Kosten von $19 = 11 + 8$ und bei **C** Kosten von $15 = 12 + 3$. Ein Autotunerdurchlauf **A**→**C**→**B** misst bei **B** Kosten von 13 und bei **C** Kosten von 20. Konfiguration **A** ist jedoch die Konfiguration, die vermieden werden sollte.

Die Einträge auf der Diagonalen der Aufwandsmatrix sind im Allgemeinen niedrig, denn das Beibehalten einer Konfiguration sollte keine Kosten verursachen. Für die Einträge neben der Diagonalen gilt dies allerdings nicht zwangsläufig. Insbesondere bei mehrdimensionalen Suchräumen kann es vorkommen, dass unterschiedliche Parameter mit unterschiedlichem Rekonfigurationsaufwand verbunden sind. Eine Messung könnte den langwierigen Wechsel des Datenbanksystems umfassen während in einer anderen Messung nur ein weiterer Thread gestartet wird. Der Aufwand wird jeweils der neuen Konfiguration angelastet.

Konfiguration	Kosten
A	18
B	11
C	12

Tabelle 4.1: Ausführungskosten der drei Konfigurationen.

	→**A**	→**B**	→**C**
A→	0,1	8	8
B→	3	0,1	3
C→	2	2	0,1

Tabelle 4.2: Rekonfigurationsaufwand beim Wechsel der Konfiguration. In der Zeile steht die Ausgangskonfiguration, in der Spalte die Zielkonfiguration. Der Wechsel von **A** nach **B** kostet 8, der Wechsel von **B** nach **A** kostet 3.

Der Userspace-Autotuner beachtet die Rekonfiguration und ihren Aufwand nicht. Er geht implizit von einheitlichem, geringem Rekonfigurationsaufwand aus. Er wechselt beliebig oft die Konfiguration und verwendet die Zeitmessungen aller Durchläufe. Zur Steigerung der Zuverlässigkeit sollten nur Messungen auf der Diagonalen mit minimalem Rekonfigurationsaufwand verwendet werden, beispielsweise indem Konfigurationen zwei Durchläufe lang aktiviert bleiben und nur der zweite Durchlauf zur Zeitmessung verwendet wird.

Bei einer Rekonfiguration muss zusätzlicher Rekonfigurationsaufwand abgearbeitet werden. Ist die Anzahl der Durchläufe beschränkt, stellt sich die Frage, ob sich eine Rekonfiguration jemals amortisieren wird. Manchmal sollte eine gute, aber nicht optimale Konfiguration aktiviert bleiben, anstatt eine Rekonfiguration durchzuführen. Neben dem zu verrichtenden Rekonfigurationsaufwand besteht bei der Suche nach der optimalen Konfiguration zudem das Risiko, eine schlechte Konfiguration zu aktivieren, die den gesamten Ablauf verzögert. Die explizite Rekonfiguration kann dabei helfen, ungewünschte Rekonfigurationen zu vermeiden und gegebenenfalls die Optimierungsstrategie beeinflussen.

4.3 Autotuner-Schnittstelle

Der Entwurf einer Go-Autotuner-Schnittstelle ist Teil dieser Arbeit. Die Go-Schnittstelle orientiert sich an der C-Schnittstelle des Userspace-Autotuners [Bra12] und ist mit dieser kompatibel, sodass der Userspace-Autotuner aus Go heraus über einen Wrapper angesprochen werden kann. Sie ist darüber hinaus aber auch für andere Autotuner nutzbar; insbesondere gilt dies für den in Go geschriebenen `GoFullWalker` zur Exploration von Suchräumen.

Im Vergleich zur bestehenden C-Schnittstelle des Userspace-Autotuners wurde eine textuelle Beschreibung der Parameter für Protokollierungszwecke hinzugefügt. Die Rekonfiguration ist wie in Abschnitt 4.2 beschrieben nun ein expliziter Schritt.

Die Schnittstelle erfordert die Implementierung der folgenden Methoden:

- `AddParameter`
  Die `AddParameter`-Methode dient dazu, dem Autotuner einen weiteren Parameter mitzuteilen. Dieser wird in darauffolgenden Durchläufen berücksichtigt.
- `GetParameters`
  Die `GetParameters`-Methode liefert die Liste der registrierten Parameter zurück.
- `Tic`
  Die `Tic`-Methode markiert den Beginn eines Messabschnitts. Der Methodenname ist an die Stoppuhr-Schnittstelle von MATLAB [tic] angelehnt.
- `Toc`
  Die `Toc`-Methode markiert das Ende eines Messabschnitts. Der Methodenname ist ebenfalls an die Stoppuhr-Schnittstelle von MATLAB angelehnt.
- `Reconfigure`
  Die `Reconfigure`-Methode weist den Autotuner an, die nächste zu untersuchende Konfiguration zu bestimmen und zu aktivieren. Nach der Aktivierung der neuen Konfiguration erfolgt der Aufruf des `OnReconfigure`-Callbacks.
- `SetOnReconfigure`
  Die `SetOnReconfigure`-Methode dient dazu, einen `OnReconfigure`-Callbacks zu hinterlegen, der bei jeder Rekonfiguration aufgerufen wird. Die hinterlegte Methode ist dafür verantwortlich, das Programm an die vom Autotuner eingestellte Konfiguration anzupassen.
- `Destroy`
  Die `Destroy`-Methode führt gegebenenfalls erforderliche Aufräumarbeiten beim Beenden des Autotuners durch.

Der Datentyp `Parameter` speichert Informationen über einen einzelnen Autotuning-Parameter. Der Datentyp umfasst die im Folgenden erläuterten Attribute. Im Vergleich zur bestehenden C-Schnittstelle des Userspace-Autotuners trägt jeder Parameter zusätzlich noch einen sprechenden Namen `description`.

- `where`
  `where` ist ein Zeiger auf den Speicherbereich, in dem der Parameter gespeichert ist. Dadurch kann der Autotuner direkt schreibend auf den Parameter zugreifen und das Speicherlayout der Anwendung bleibt erhalten.
- `min`
  `min` definiert den zulässigen Minimalwert für diesen Parameter.
- `max`
  `max` definiert den zulässigen Maximalwert für diesen Parameter.

- `step`
  `step` definiert die Schrittweite, in welcher der Autotuner das Intervall zwischen `min` und `max` durchsuchen soll.

- `description`
  `description` ist der sprechende Name des Parameters, der in Protokollausgaben verwendet wird.

4.4 Identifikation und Bewertung von Freiheitsgraden

Der folgende Abschnitt beschreibt das Vorgehen und die Kriterien bei der Suche nach relevanten Freiheitsgraden, die den nachfolgenden Fallstudien zugrunde liegen.

4.4.1 Suche nach Freiheitsgraden

Üblicherweise soll beim Autotuning das Optimierungspotential in einem bestimmten Programm ausgenutzt werden. Zur Durchführung der Fallstudien ist es erforderlich, Programme mit möglichst großem Optimierungspotential zu finden.

Alle in Konfigurationsdateien oder Kommandozeilenparameter ausgelagerten Freiheitsgrade sind Kandidaten für das Autotuning. Auch Konstanten im Code können auf eine Konfigurationsmöglichkeit hindeuten, die momentan noch nicht ausgenutzt wird. Konstanten können entweder als `const BEZEICHNER = 1234` im Code auftauchen oder in Form expliziter Argumente bei Funktionsaufrufen. Das Durchsuchen von Code nach `buffer`, `pool`, `config`, `setting` und `default` zeigt ebenfalls Kandidaten auf.

Bei der Suche muss abgeschätzt werden, ob ein Kandidat Auswirkungen auf die Geschwindigkeit des Programms hat. Außerdem muss sichergestellt werden, dass der Parameter das Ergebnis des Programms nicht verändert, weil er sonst nicht für das Autotuning verwendet werden kann.

4.4.2 Bewertungskriterien

Für das Autotuning gut geeignete Freiheitsgrade zeichnen sich wie nachfolgend beschrieben aus. Der Einsatz von Autotuning ist umso vielversprechender, je eher ein Freiheitsgrad diese Kriterien erfüllt.

- signifikante Auswirkung auf die Geschwindigkeit in gewissen Szenarien

  Nur wenn der Freiheitsgrad eine Auswirkung auf die Ausführungsgeschwindigkeit des Programms hat, kann eine Änderung auch eine Beschleunigung des Programms bewirken. Die Erhöhung der Ausführungsgeschwindigkeit ist momentan das alleinige Ziel des Autotunings.

- optimale Konfiguration ist nicht offensichtlich

  Die optimale Konfiguration sollte nicht offensichtlich sein, denn andernfalls kann die optimale Konfiguration direkt eingestellt werden, ohne dass eine Suche erforderlich ist.

- optimale Konfiguration ist nicht statisch, sondern Hardware- bzw. Daten-abhängig

  Falls die optimale Konfiguration zwar nicht offensichtlich, aber statisch ist, kann diese zur Entwicklungszeit durch einmaliges Profiling ermittelt werden. Hängt die Konfiguration von der Hardware ab, sind Messungen auf der verwendeten Hardware erforderlich. Wenn zusätzlich Datenabhängigkeiten oder die Systemlast berücksichtigt werden sollen, ist Onlinetuning zur Laufzeit erforderlich.

- weite Verbreitung

  Vorrangig sollen Freiheitsgrade mit einer weiten Verbreitung untersucht werden, sodass die Ergebnisse möglichst vielen Anwendungen zugutekommen. Im Optimalfall ergibt sich die Autotuningfähigkeit der Anwendungen implizit durch Anpassungen der Sprache, der Laufzeitumgebung oder an Bibliotheken, ohne dass Änderungen an den einzelnen Programmen erforderlich sind.

Freiheitsgrade mit direktem Bezug zur Parallelität werden bevorzugt behandelt.

4.4.3 Autotuning-Kandidaten

Neben den in den Fallstudien in Kapitel 5 und Kapitel 6 genauer vorgestellten Freiheitsgraden wurden auch die folgenden Freiheitsgrade in Betracht gezogen.

4.4.3.1 Asynchrone Channel

Go bietet neben synchronen, blockierenden Channeln auch asynchrone Channel [eff] mit einem Puffer. Diese blockieren erst, wenn eine entsprechende Anzahl von Nachrichten gesendet, aber noch nicht empfangen wurde. Programme, die asynchrone Channel als Fließband verwenden, könnten von automatischen Anpassungen der Puffergröße profitieren, gerade beim parallelen Zugriff auf die Channel.

Da Channel ein zentrales Konzept in Go sind, würden viele Programme implizit autotuningfähig. Problematisch sind Anwendungen, die asynchrone Channel als Semaphore nutzen. Dabei beeinflusst die Puffergröße die Synchronisierungslogik und Änderungen führen zu Fehlern in der Anwendung.

4.4.3.2 Gepufferte Ein-/Ausgabe

Go definiert die Schnittstellen `Reader` und `Writer`, welche dem Lesen und Schreiben von Daten dienen. Viele Standardoperationen in Go, wie der Zugriff auf Dateien, Netzwerkoperationen, aber auch das (De)Komprimieren von Datenströmen oder die Formatierung (beispielsweise im XML-Format) erwarten `Reader` und `Writer` zur Ein- bzw. Ausgabe.

Gepufferte Ein-/Ausgabeoperationen werden in Go durch das `bufio`-Paket [buf] ermöglicht. `bufio` stellt nach dem Entwurfsmuster des Dekorierers Pufferung für `Reader` und `Writer` bereit. Die Puffergröße kann entweder bei der Erstellung explizit festgelegt werden, oder es wird eine Standardgröße verwendet.

Die Puffergröße könnte vom Autotuner gesetzt werden. Die Optimierung verliehe allen Go-Programmen implizite Autotuningfähigkeiten, sofern sie Gebrauch von den Go-Standardmethoden zur Pufferung machen.

4.4.3.3 Puffer in der Standardbibliothek

An verschiedenen Stellen in der Go-Standardbibliothek werden Puffer mit Standardgrößen alloziert. Diese Größen könnten automatisch angepasst werden.

In der `MatchAll`-Methode der Bibliothek für reguläre Ausdrücke [reg] wird beispielsweise ein Puffer für die Treffer reserviert. Dieser bietet zunächst für zehn Treffer Platz und wird bei Bedarf vergrößert. Werden regelmäßig weniger oder mehr Treffer gefunden, ist eine Größenanpassung empfehlenswert. Der Einfluss auf die Gesamtleistung ist allerdings unklar und vermutlich eher gering.

Das Paket `index/suffixarray` [suf] bietet Methoden zur Suche nach Substrings. Auch dieses alloziert Puffer, deren Größe automatisch angepasst werden könnte. Auch hier ist der Einfluss auf die Geschwindigkeit vermutlich eher gering.

Das Paket `math/big` [big] bietet Datentypen zum Rechnen mit Ganzzahlen beliebiger Größe. Diese werden in Felder von Maschinenwörtern abgelegt. Bei der Allokation der Felder wird mehr Speicherplatz reserviert als unmittelbar erforderlich, um spätere teure Neuallokationen zu vermeiden. Die Menge des reservierten Speicherplatzes ist autotuningfähig. Bei numerischem Code ist hier ein nennenswerter Einfluss auf die Geschwindigkeit denkbar.

4.4.3.4 Nebenläufige Warteschlangen

Elbre vergleicht verschiedene Möglichkeiten, nebenläufige Warteschlangen zu implementieren [Elb]. Das Wechseln der verwendeten Warteschlange durch den Autotuner ist denkbar. Bislang sind allerdings keine Go-Programme veröffentlicht, die diese Warteschlangen verwenden. Eventuell können die implementierten Sperrkonzepte auch in anderen Fällen automatisch getauscht werden, um auf unterschiedliche Zugriffsmuster zu reagieren.

4.4.3.5 Eine Million Domänen

Die schnelle parallele Auflösung vieler DNS-Einträge [Maj] ist vergleichbar mit dem Szenario in der Fallstudie in Kapitel 6, da ebenfalls Anfragen an eine externe Datenbank gestellt werden. Im Falle des DNS ist der Einfluss der Netzwerklatenz und das Chaching des DNS-Servers wohl noch einflussreicher.

Die Übertragbarkeit der Ergebnisse auf andere Szenarien ist unklar.

4.4.3.6 Goroutinen mit Ein-/Ausgabe

Die Fallstudie in Kapitel 5 untersucht Goroutinen, die durch ihren Rechenaufwand limitiert werden. Ein- und Ausgabeoperationen sind explizit von der Zeitmessung ausgenommen. Für verschiedene Verhältnisse von langsamen Ein- bzw. Ausgabeoperationen zu schnellen Rechenoperationen ergeben sich vermutlich interessante Suchräumen von großer Generalität.

4.4.3.7 gccgo Kommandozeilenparameter

`gccgo` ist ein `gcc`-Frontend für Go. Viele der in `gcc` gebotenen Codeoptimierungen sind verfügbar und können über Kommandozeilenparameter aktiviert und konfiguriert werden. Die Menge der zu aktivierenden Optimierungen und ihre Konfiguration könnte vom Autotuner ermittelt werden. Mit diesem Ansatz sind vermutlich signifikante Leistungssteigerungen bei vielen Programmen möglich, aber der Einsatz von Onlinetuning ist wahrscheinlich nur schwierig umsetzbar.

Der Ansatz ist wenig Go-spezifisch und wurde beispielsweise für den Glasgow Haskell Compiler schon vergleichbar von Stewart durchgeführt [Ste].

4.4.3.8 Sortieralgorithmen

Der Quicksort-Sortieralgorithmus in Go [sor] fällt nach einer festgelegten Anzahl von Rekursionsschritten auf Heapsort zurück. Sowohl die Anzahl der Rekursionsschritte als auch generell der verwendete Sortieralgorithmus könnte automatisch angepasst werden. Bestimmte Eigenschaften der Eingabedaten beeinflussen die Leistungsfähigkeit verschiedener Sortieralgorithmen unterschiedlich. Momentan findet in der Go-Standardbibliothek keine Parallelisierung der Sortierung statt, auch diese könnte hinzugefügt werden.

Die Thematik ist relativ wenig Go-spezifisch und die theoretischen Aspekte sind gut erforscht. Autotuning würde weniger zur Anpassung an die Hardware genutzt sondern stärker zur Anpassung an die Eigenschaften der Algorithmen und der Eingabedaten.

4.5 Die k-Proben-Heuristiken

Bei der Bewertung von Autotuner-Heuristiken dienen zufallsbasierte k-Proben-Heuristiken als Referenz. Für jede natürliche Zahl $k \in \mathbb{N}$ ist eine k-Proben-Heuristik definiert. Diese Heuristiken gehen in zwei Phasen vor, der Probephase und der produktiven Phase.

Zunächst erfolgt die Probephase:

```
Eingaben:
  Anzahl k der Proben
  Menge x der zulässigen Konfigurationen
Initialisiere:
    Beste Laufzeit ztop: unendlich
    Beste Konfiguration ptop: undefiniert
Wiederhole k Mal:
    Wähle Probe p zufällig aus x
    Aktiviere Konfiguration p
    Messe Laufzeit z der Konfiguration p
    Falls z < ztop:
        Setze ztop auf z
        Setze ptop auf p
Ergebnis der Probephase:
    Konfiguration ptop
```

Daraufhin folgt die produktive Phase:

```
Wiederhole bis Programmende:
    Aktiviere Konfiguration ptop
```

Das heißt, dass in der Probephase zunächst k zufällige Konfigurationen verwendet werden, und danach in der produktiven Phase die beste Konfiguration aus der Probephase bis zum Programmende beibehalten wird.

Bei der Analyse der k-Proben-Heuristik helfen die folgenden Definitionen:

- n sei die Anzahl der Konfigurationen im Suchraum.
- $x = (x_1, \cdots, x_n)^T$ sei der Spaltenvektor der Konfigurationen im Suchraum.
- $z = (z_1, \cdots, z_n)^T$ sei der Spaltenvektor mit der jeweiligen Laufzeit der Konfigurationen in x. Im Rahmen der Analyse werden diese als bekannt vorausgesetzt. Ohne Beschränkung der Allgemeinheit sei x zudem so angeordnet, dass z aufsteigend sortiert ist.

Weiterhin sei $p^k = (p_1^k, \cdots, p_n^k)$ der Zeilenvektor, der die Wahrscheinlichkeiten p_i^k angibt, mit der x_i das Ergebnis der Probephase der k-Proben-Heuristik ist. p^1 ergibt sich zu $p^1 = (\frac{1}{n}, \cdots, \frac{1}{n})$, weil jede Konfiguration mit der gleichen Wahrscheinlichkeit gezogen wird und als einzig getestete Konfiguration automatisch das Ergebnis der Heuristik ist.

Das Durchführen weiterer Proben ergibt eine Markow-Kette, in der sich p^{k+1} aus dem aktuellen p^k durch Multiplikation mit der folgenden Übergangsmatrix $A = (a_{ij}) \in \mathbb{R}^{n \times n}$ berechnet.

$$A = \begin{bmatrix} \frac{n+1-i}{n} & 0 & \cdots & \cdots & 0 \\ \frac{1}{n} & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ \frac{1}{n} & \cdots & \cdots & \frac{1}{n} & \frac{n+1-i}{n} \end{bmatrix}$$

Es gilt $p^{k+1} = p^k \cdot A$ sowie $p^{k+1} = p^1 \cdot A^k$.

Anschaulich ergibt sich A durch die Betrachtung aller möglichen Kombinationen von „Was war bisher die beste Probe?“ und „Welche Probe wird gezogen?“. Angenommen, es existieren die vier Konfigurationen x_1, x_2, x_3, x_4 mit den Laufzeiten $1, 2, 3, 4$. Wenn bislang x_4 die beste Probe war, wird jetzt mit Wahrscheinlichkeit $\frac{1}{4}$ die Konfiguration x_1 als Probe gezogen und damit zur bisher besten Probe. Ebenfalls mit Wahrscheinlichkeit $\frac{1}{4}$ wird x_2 als Probe gezogen und zur besten Probe. Auch x_3 wird mit Wahrscheinlichkeit $\frac{1}{4}$ als Probe gezogen und zur besten Probe. Mit Wahrscheinlichkeit $\frac{1}{4}$ wird erneut x_4 gezogen und x_4 bleibt die beste Probe. $(\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ ist die unterste Zeile der A-Matrix, denn diese wird mit dem letzten Eintrag des p-Vektors multipliziert, der angibt, wie wahrscheinlich x_4 bislang die beste Probe war.

Wenn x_3 bisher die beste Probe war, wird mit $w = \frac{1}{4}$ die Konfiguration x_1 als Probe gezogen und zur besten Probe. Ebenfalls mit $w = \frac{1}{4}$ wird x_2 als Probe gezogen und zur besten Probe. Mit $w = \frac{2}{4}$ wird x_4 oder x_3 gezogen und x_3 bleibt die beste Probe. x_4 wird nicht zur besten Probe werden, wenn schon x_3 gefunden wurde. $(\frac{1}{4}, \frac{1}{4}, \frac{2}{4}, 0)$ ist die dritte Zeile der A-Matrix, denn diese wird mit dem dritte Eintrag des p-Vektors multipliziert, der angibt, wie wahrscheinlich x_3 bislang die beste Probe war.

Wenn x_2 bisher die beste Probe war, wird mit $w = \frac{1}{4}$ die Konfiguration x_1 als Probe gezogen und zur besten Probe. Mit $w = \frac{3}{4}$ bleibt x_2 die beste Probe. $(\frac{1}{4}, \frac{3}{4}, 0, 0)$ ist die zweite Zeile der A-Matrix, denn diese wird mit dem zweiten Eintrag des p-Vektors multipliziert, der angibt, wie wahrscheinlich x_2 bislang die beste Probe war.

Wenn x_1 bisher die beste Probe war, bleibt x_1 auch nach einer weiteren Probe die beste Probe. $(1, 0, 0, 0)$ ist die oberste Zeile der A-Matrix, denn diese wird mit dem ersten Eintrag des p-Vektors multipliziert, der angibt, wie wahrscheinlich x_1 bislang die beste Probe war.

Zeilenweise betrachtet kann die Übergangsmatrix also folgendermaßen interpretiert werden: jedes $\frac{1}{n}$ links der Diagonalen steht für die Wahrscheinlichkeit, dass die neue Probe diese neue beste Konfiguration findet. Die Einträge auf der Diagonalen stellen die Gegenwahrscheinlichkeit dar – dass die aktuelle Probe keine neue beste Konfiguration findet. Die

Nullen rechts der Diagonalen ergeben sich aus der Tatsache, dass eine schlechte Probe das Ergebnis nicht verschlechtert, sondern einfach weiterhin die schon bekannte Konfiguration verwendet wird. Als zeilenstochastische Matrix ist die Zeilensumme jeweils 1.

In p^1 sind alle Konfigurationen gleich wahrscheinlich. Je größer k wird, desto weiter verschiebt sich die Verteilung zugunsten von Konfigurationen mit besserer Laufzeit. Der Erwartungswert $E_{best}(k)$ für die Laufzeit der besten gefundenen Konfigurationen bei Verwendung der k-Proben-Heuristik ergibt sich durch Multiplikation der Wahrscheinlichkeiten p^k mit den jeweiligen Laufzeiten z und ist monoton fallend mit steigendem k.

$$E_{best}(k) = p^k \cdot z$$

Für den Erwartungswert $E_{avg}(k, h)$ der durchschnittlichen Laufzeit nach h Durchläufen gilt: er berechnet sich aus dem gewichteten Mittel der erwarteten Laufzeiten der h Durchläufe. Die ersten k Messungen werden mit zufälligen Konfigurationen durchgeführt. Alle weiteren Messungen werden mit der besten bis dahin gefundenen Konfiguration durchgeführt.

$$E_{avg}(k,h) = \begin{cases} E_{best}(1) & h \leq k \\ (k \cdot E_{best}(1) + (h-k) \cdot E_{best}(k))/h & \text{sonst} \end{cases}$$

Die Erwartungswerte hängen über z von der Struktur des Suchraums ab. Abbildung 4.1 zeigt die Erwartungswerte $E_{best}(h)$ und $E_{avg}(k, h)$ für ausgewählte k über die ersten dreißig Durchläufe mit z-Werten, wie sie sich bei der Fallstudie in Kapitel 5 ergeben.

Mit steigender Probenzahl sinkt die erwartete beste Laufzeit. Heuristiken mit unterschiedlich vielen Proben verhalten sich unterschiedlich. Heuristiken mit hohem k haben eine lange Probephase mit schlechtem Erwartungswert, aber ein gutes Endergebnis, wenn sie lange genug laufen dürfen, dass die Probephase nicht mehr dominiert. Heuristiken mit niedrigem k haben eine kürzere Probephase, fallen danach zunächst auch schneller, haben aber ein schlechteres erwartetes Endergebnis.

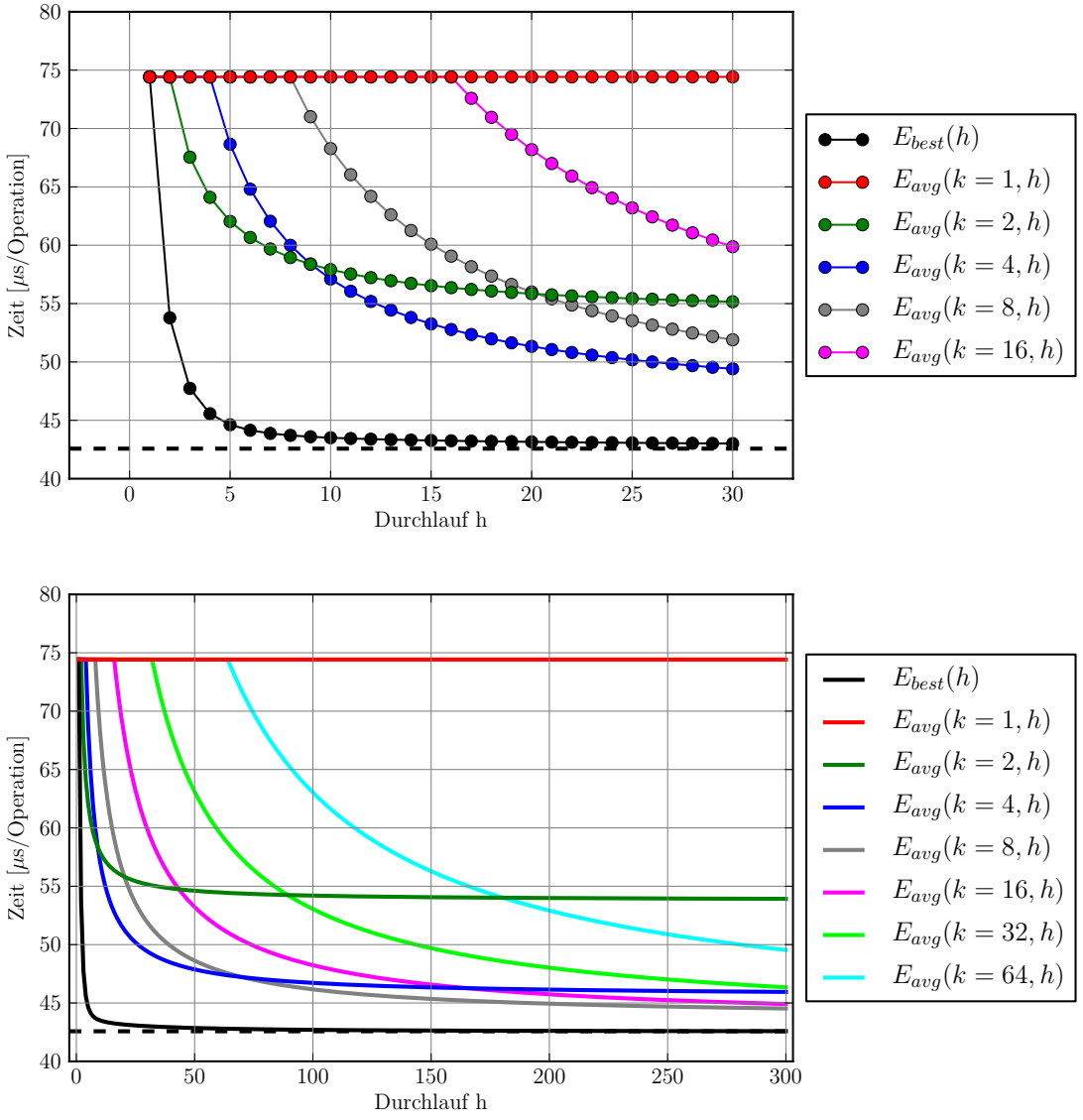


Abbildung 4.1: Erwartete Laufzeit bei Verwendung der k-Proben-Heuristik. Der Erwartungswert $E_{best}(k)$ für die beste gefundene Konfiguration fällt mit steigendem k schnell ab. Die durchschnittliche Laufzeit einer k-Proben-Heuristik nähert sich nach Ablauf der zufälligen Probephase langsam der gefundenen Minimallaufzeit an. Die gestrichelte Linie markiert die Laufzeit der schnellsten Konfiguration.

5. Fallstudie: Threadpool in nfnt/resize

`nfnt/resize` [nfn] ist eine vollständig in Go geschriebene Bibliothek zur Skalierung von Pixelgrafiken. Sie unterstützt verschiedene Interpolationsmethoden, wie z.B. bilineare und bikubische Interpolation sowie Lanczos-Interpolationen.

5.1 Der Freiheitsgrad

`nfnt/resize` berechnet das skalierte Bild pixelweise aus dem Ausgangsbild. Zur Berechnung eines Pixels im skalierten Bild werden nur Daten aus dem Ausgangsbild verwendet, sodass das skalierte Bild in mehrere Streifen zerschnitten werden kann, die nebenläufig berechnet werden können. In `nfnt/resize` wird jeder Streifen in einer separaten Goroutine verarbeitet, um mehrere Rechenkerne gleichzeitig nutzen zu können.

Das Gesamtkonstrukt stellt – ohne dies explizit zu kennzeichnen – einen Threadpool dar. Die Anzahl der Betriebssystem-Threads wird von der Go-Laufzeitumgebung bestimmt und ist über `GOMAXPROCS` steuerbar. Die Anzahl der Arbeitspakete entspricht der Anzahl der Streifen und wird in `nfnt/resize` von der Heuristik `numJobs` festgelegt.

Die entscheidende Methode bei der Skalierung ist die `Resize`-Funktion, deren Struktur im Folgenden wiedergegeben ist.

```
1  func Resize(w, h uint, img image.Image, ...) image.Image { ...
2    n := numJobs(b.Dy())
3    c := make(chan int, n)
4    for i := 0; i < n; i++ {
5      x1, x2, y1, y2 := ... // corners of section depending on i
6      ...
7      go func(b image.Rectangle, c chan int) {
8        ...
9        for y := b.Min.Y; y < b.Max.Y; y++ {
10         for x := b.Min.X; x < b.Max.X; x++ {
11           ... // calculate and store result pixel
12         }
13       }
14       c <- 1
15     }(image.Rect(x1, y1, x2, y2), c)
16   }
17   for i := 0; i < n; i++ {
18     <-c
19   }
20   return resizedImg
21 }
```

In Zeile 4 beginnt eine Schleife über die **n** Streifen, deren jeweilige Position in Zeile 5 bestimmt wird.[1] In Zeile 7ff wird für jeden Streifen eine anonyme Funktion zur Berechnung der Pixel im skalierten Bild definiert und in einer separaten Goroutine gestartet. Beim Start der Goroutine in Zeile 15 wird der jeweilige Streifen übergeben. Dieser Streifen wird von den beiden verschachtelten Schleifen, die in den Zeilen 9 und 10 beginnen, pixelweise berechnet. Der Channel **c** dient der Synchronisation der Goroutinen.

Die Anzahl **n** der Streifen wird in Zeile 2 von der Heuristik `numJobs` ermittelt.

```
1 func numJobs(d int) (n int) {
2   n = runtime.NumCPU()
3   if n > d {
4     n = d
5   }
6   return
7 }
```

Die Anzahl **n** der verwendeten Streifen wird auf die Anzahl `NumCPU` der im Rechner vorhandenen Rechenkerne [gomb] gesetzt, sofern diese die Anzahl der Zeilen **d** in der Grafik nicht übersteigt. Andernfalls werden **d** Streifen verwendet und jede Zeile wird in

[1] Im Originalcode werden die Koordinaten direkt beim Funktionsaufruf in Zeile 15 berechnet. Aus Gründen der Übersichtlichkeit ist die Berechnung hier in einen eigenen Schritt ausgegliedert.

einer eigenen Goroutine bearbeitet.

Da nicht explizit angegeben wird, wie viele Goroutinen gleichzeitig bearbeitet werden sollen, bestimmt die Umgebungsvariable `GOMAXPROCS` die Anzahl der von der Laufzeitumgebung genutzten Betriebssystem-Threads. Der Standardwert für `GOMAXPROCS` ist dabei `1`, was bedeutet, dass keine parallele Ausführung der Goroutinen stattfindet.

Für jeden Rechenkern eine Goroutine zu starten, ist für rechenintensive, parallelisierbare Probleme insgesamt eine gute Entscheidung. Die Evaluation wird zeigen, dass sie in vielen, aber nicht allen Fällen zu guten Laufzeiten führt. Dass die Umgebungsvariable `GOMAXPROCS` für die maximale Anzahl der gleichzeitigen Threads respektiert wird, ist aus softwaretechnischer Sicht vernünftig, weil das Gesamtsystem so wartbar bleibt. Abweichende Konfigurationen können jedoch eine bessere Ausführungsgeschwindigkeit erreichen.

Die beiden Freiheitsgrade *Anzahl Arbeitspakete* (d.h. die Anzahl der Streifen/Goroutinen) und *Anzahl Threads* (d.h. der Wert von `GOMAXPROCS`) sind getrennt konfigurierbar, aber es bestehen Abhängigkeiten bezüglich guter Konfigurationen. Es handelt sich daher um einen zweidimensionalen, autotuningfähigen Suchraum.

5.2 Definition der Messabschnitte

Der Messabschnitt, welcher den zu beschleunigenden Codeabschnitt markiert, umfasst den kompletten Rumpf der `Resize`-Methode. Die Methode wird wie folgt angepasst:

```
1  func Resize(...) ... {
2    tuner.Tic()
3    // after Tic for compatibility with Userspace-Autotuner
4    tuner.Reconfigure()
5
6    // original method body
7
8    tuner.Toc(...)
9    return resizedImg
10 }
```

Zur Normierung der gestoppten Zeit wird im Aufruf von `Toc` der Aufwand des Messabschnitts angegeben. So können Messungen mit unterschiedlich großen Bildern verglichen werden. Als Maß für den Aufwand dient die Anzahl der Pixel im skalierten Bild. Diese Abschätzung ist allerdings nicht perfekt, weil auch die Größe des Ausgangsbildes, der Skalierungsfaktor, sowie die Seitenverhältnisse eine Rolle spielen. Es besteht kein einfacher linearer Zusammenhang zwischen diesen Größen und dem Aufwand. Für annähernd gleichgroße Eingabebilder ergibt die Größe des skalierten Bildes jedoch einen guten Korrekturfaktor.

5.3 Extraktion der Parameter

Alle Änderungen für das Autotuning finden in der Bibliothek statt. Das Anwendungsprogramm, welches `nfnt/resize` verwendet, bleibt komplett unverändert.

Durch diese Struktur profitieren alle Anwendungen, welche die Bibliothek verwenden, von der Modifikation und erhalten Autotuningfähigkeiten. Sie werden autotunbar, ohne dass der Anwendungsentwickler explizite Schritte dazu unternimmt.

In der Bibliothek werden Variablen für den Autotuner und die Parameter angelegt:

```
1 var tuner = fastergo.CreateGoFullWalker()
2 var jobs, jobs_min, jobs_max, jobs_step
      = 1, 1, 10, 1
3 var threads, threads_min, threads_max, thread_step
      = 1, 1, 10, 1
4
5 func init() {
6   tuner.AddParameter(&jobs, "#jobs",
       jobs_min, jobs_max, jobs_step)
7   tuner.AddParameter(&threads, "#threads",
       threads_min, threads_max, thread_step)
8 }
```

Die Methode `numJobs` wird so angepasst, dass sie nicht mehr die Anzahl der Rechenkerne und die Anzahl der Zeilen beachtet, sondern auf eine Variable zugreift, die unter der Kontrolle des Autotuners steht.

```
1 func numJobs(d int) (n int) {
2   return jobs
3 }
```

Im `OnReconfigure`-Callback wird unter Missachtung von `GOMAXPROCS` die maximale Anzahl der von der Laufzeitumgebung verwendeten Threads gesetzt. Hierfür wird ebenfalls ein vom Autotuner vorgegebener Wert verwendet.

```
1 f := func() { runtime.GOMAXPROCS(threads) }
2 tuner.SetOnReconfigure(f)
```

5.4 Evaluation

Das folgende Szenario bildet den Hintergrund der Evaluation: Im Rahmen der Organisation einer Fotosammlung werden Vorschaubilder von vielen Bildern generiert. Die Skalierung eines Einzelbildes wird in in diesem Szenario vielfach durchgeführt, sodass diese Operation als wiederholt ausgeführter Messabschnitt gesehen werden kann.

Bei der Evaluation wird anstelle einer Fotosammlung mit unterschiedlichen Bildern bei jeder Messung dasselbe Originalbild verwendet, um einen eventuellen Einfluss des Bildes

auszuschließen. Das Originalbild hat eine Größe von 2000x843 Pixel, das Vorschaubild ist 80x34 Pixel groß. Ausschließlich der Skalierungsvorgang wird gestoppt; das Einlesen aus dem Dateisystem, das Dekodieren des JPG-Bildes, sowie das Kodieren und Rückschreiben des Ergebnisses ist von der Messung ausgenommen.

Im Rahmen der Evaluation wurden Messungen auf unterschiedlichen Systemen durchgeführt. Die verwendete Hardware und Software ist in Tabelle 5.1 beschrieben.

	System A	**System B**
Anzahl Kerne	2	1-4 virtuelle CPUs
Betriebssystem	Ubuntu 12.04 64bit	Ubuntu 12.04 64bit
Go-Version	1.2	1.2
VM-Hypervisor	-	VirtualBox 4.2
VM-Host	-	System mit 4 Kernen

Tabelle 5.1: In der Evaluation verwendete Hardware und Software.

Da Zeitmessungen immer Schwankungen unterworfen sind, wurden die Messungen zehn Mal wiederholt und der Median der Messungen verwendet. Zur Simulation von Last auf den Systemen wird `cpuburn` [Bur] verwendet.

Die Autotunervariable `jobs` steuert die Anzahl der Arbeitspakete. Jedes Arbeitspaket wird in einer Goroutine abgearbeitet. Die Begriffe `jobs`, Arbeitspaket und Goroutine werden im Rahmen der Evaluation synonym verwendet. Die Autotunervariable `threads` steuert die Variable `GOMAXPROCS` der Laufzeitumgebung, welche die (maximalen) Anzahl der Threads im Betriebssystem entspricht. Die Begriffe `threads`, `GOMAXPROCS` und Threads werden im Rahmen der Evaluation ebenfalls synonym verwendet.

5.4.1 Der Suchraum

Durch eine erschöpfende Suche des `GoFullWalker` kann der Suchraum in `nfnt/resize` erkundet werden. In Abbildung 5.1 ist eine daraus generierte Visualisierung des Suchraums zu sehen. In dem Graphen sind alle Einzelmessungen als Punkt verzeichnet. Durch die Mediane der jeweils wiederholt durchgeführten Messungen ist eine Ebene gelegt.

Insgesamt wurden 100 unterschiedliche Konfigurationen gemessen: jede Kombination von 1-10 Threads und 1-10 Arbeitspaketen. Auf System A ist die `nfnt/resize`-Standardeinstellung von zwei Threads und zwei Arbeitspaketen unter Laborbedingungen akzeptabel, aber nicht optimal. Mit 149,67 Mikrosekunden pro Operation liegt der Median dieser Durchläufe auf Platz 26 der 100 Mediane und ist 1,3% langsamer als der beste beobachtete Median (7 Arbeitspakete, 7 Threads und 147,81 Mikrosekunden pro Operation).

Konfigurationen mit einem Arbeitspaket sowie Konfigurationen mit einem Thread sind unter den gegebenen Rahmenbedingungen nicht optimal, da keine Parallelisierung stattfinden kann. Konfigurationen mit sowohl zwei oder mehr Threads als auch zwei oder mehr Arbeitspaketen laufen schneller, da dann eine effektive Parallelisierung stattfinden kann.

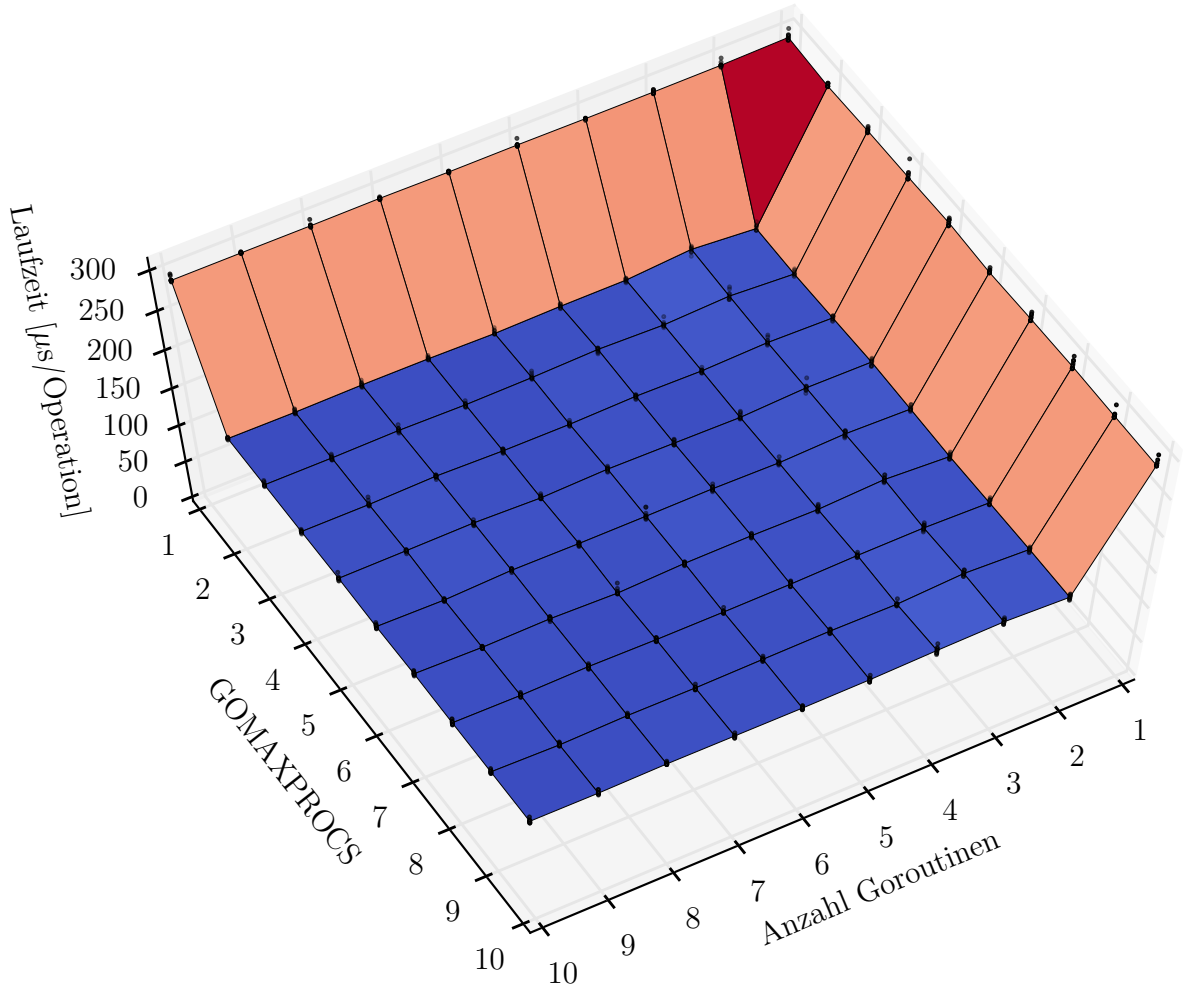
Bei Verwendung von Go 1.1 ist in Abbildung 5.2 ein Verschlechterung der Laufzeit bei Konfigurationen mit drei Arbeitspaketen zu beobachten. Die Messungen wurden auf System A mit zwei Rechenkernen durchgeführt. Das dritte Arbeitspaket wird erst abgearbeitet, nachdem die ersten beiden parallel bearbeitet wurden. Während der Bearbeitung des dritten Arbeitspakets kann keine Parallelisierung mehr stattfinden. Die Verwendung von drei Threads ist unproblematisch, da diese vom präemptiven Betriebssystemscheduler in kurzen Zeitscheiben durchgewechselt werden. Dadurch wird verhindert, dass ein einzelner Thread nach Abarbeitung der anderen zurückbleibt. Der Effekt ist auch bei Konfigurationen mit 5, 7 und 9 Arbeitspaketen zu beobachten, allerdings weniger ausgeprägt. Das immer kleiner werdende Arbeitspaket fällt dann immer weniger ins Gewicht.

Seit Version 1.2 bietet Go einen Scheduler, der Präemption an gewissen Punkten ermöglicht [go1c]. Der Vergleich von Abbildung 5.2 mit Abbildung 5.1 zeigt, dass die Situation im vorliegenden Fall dadurch deutlich verbessert wird.

Abbildung 5.1 bis Abbildung 5.4 verdeutlichen, dass die beiden Parameter `jobs` und `threads` jeweils mindestens auf die Anzahl der Rechenkerne gesetzt werden sollten. Eine virtuelle Maschine verhält sich hierbei genauso wie echte Hardware mit entsprechend vielen Rechenkernen.

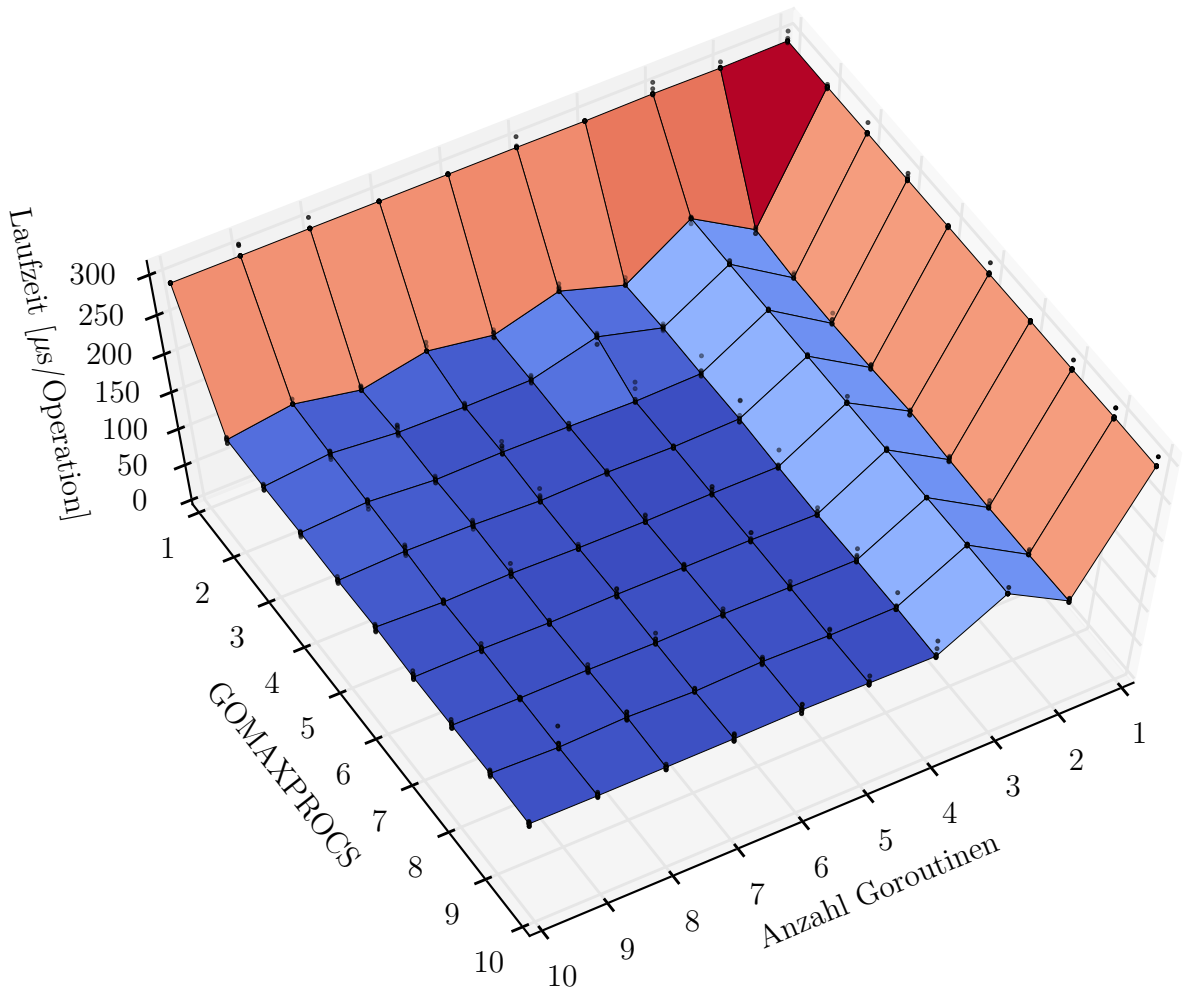
Wenn ein Rechenkern durch einen anderen Prozess belegt ist, wie in Abbildung 5.5, ergeben sich Laufzeiten wie auf einer Maschine mit einem Rechenkern weniger. Interessant ist dies insbesondere in Kombination mit der Beobachtung, dass bei Verwendung von Go 1.1 Konfigurationen langsam sind, die ein Arbeitspaket mehr erstellen als Rechenkerne verfügbar sind. Die in `nfnt/resize` verwendete Heuristik erstellt dann `NumCPU` Arbeitspakete, also so viele, wie insgesamt Rechenkerne vorhandenen sind. Dies führt zu schlechten Ergebnissen, da genau Rechenkern weniger für Berechnungen zur Verfügung steht.

In solchen Fällen hat Onlinetuning Vorteile gegenüber Offlinetuning, welches nur die Anzahl der Kerne, nicht aber deren Auslastung berücksichtigen kann.



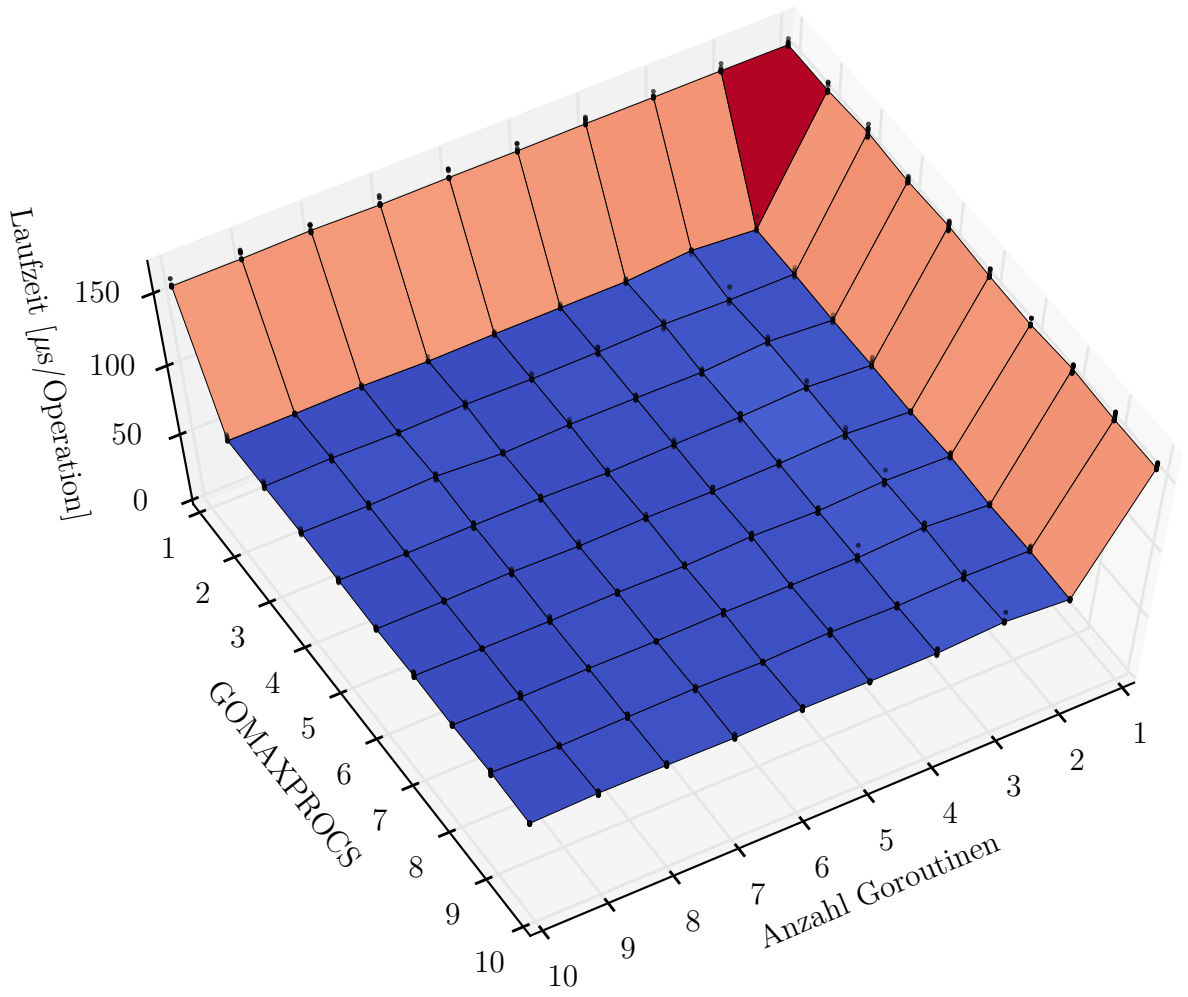
Testsystem	**System A**
Anzahl Kerne	2
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	-
VM-Host	-

Abbildung 5.1: Suchraum des Autotuners im Threadpool von `nfnt/resize`. Alle Messwerte sind durch einen Punkt markiert, die Fläche wurde durch die Mediane der Messwerte an jeder Messstelle gelegt. Bei den Messstellen handelt es sich um diskrete Werte, die Fläche dazwischen dient lediglich als optische Unterstützung um Messwerte einfacher vergleichen zu können.



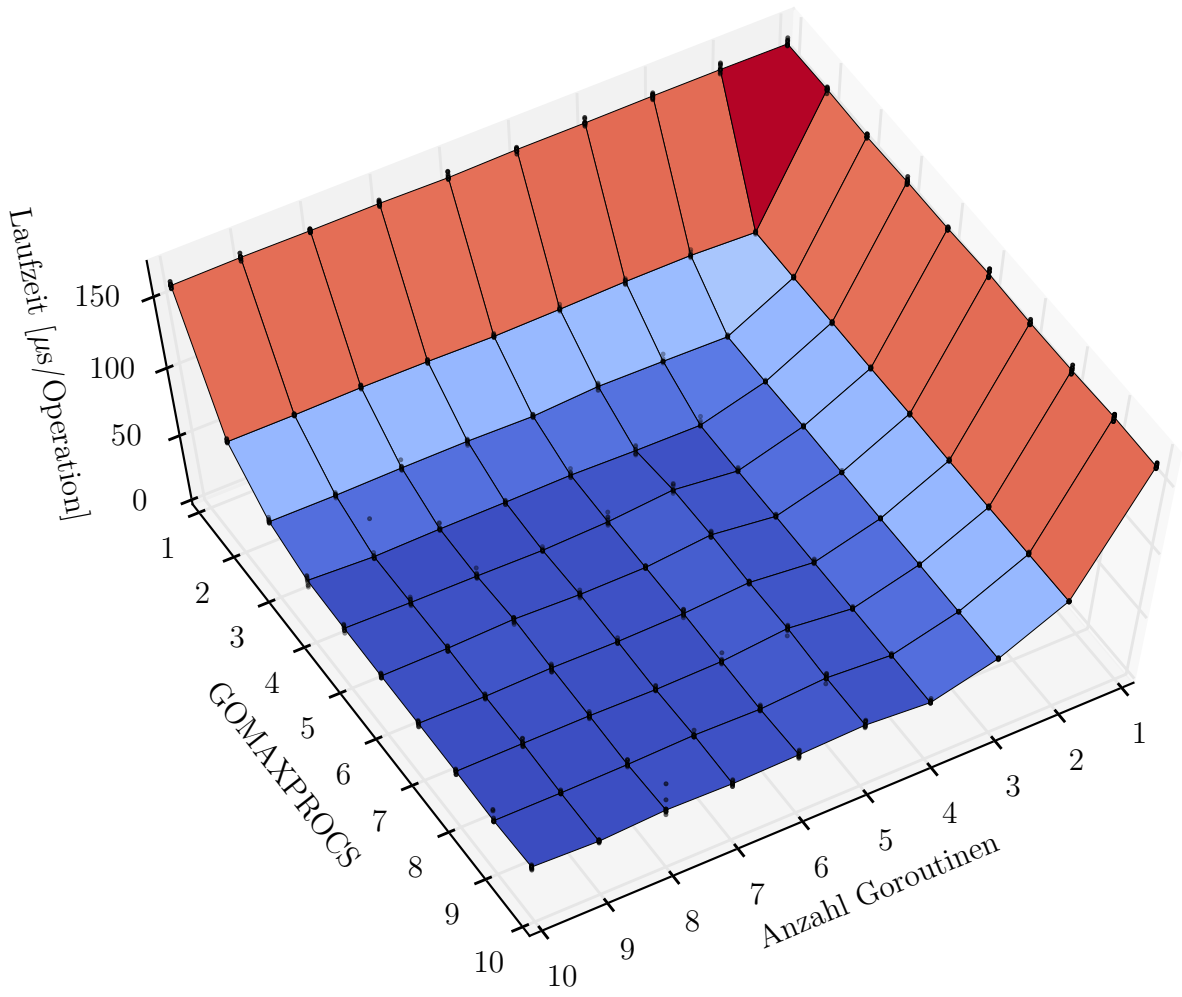
Testsystem	**System A**
Anzahl Kerne	2
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.1
VM-Hypervisor	-
VM-Host	-

Abbildung 5.2: Suchraum des Autotuners im Threadpool von `nfnt/resize`. Im Gegensatz zur vorherigen Messung wird nun Go 1.1 verwendet, Konfigurationen mit drei Goroutinen laufen damit langsamer.



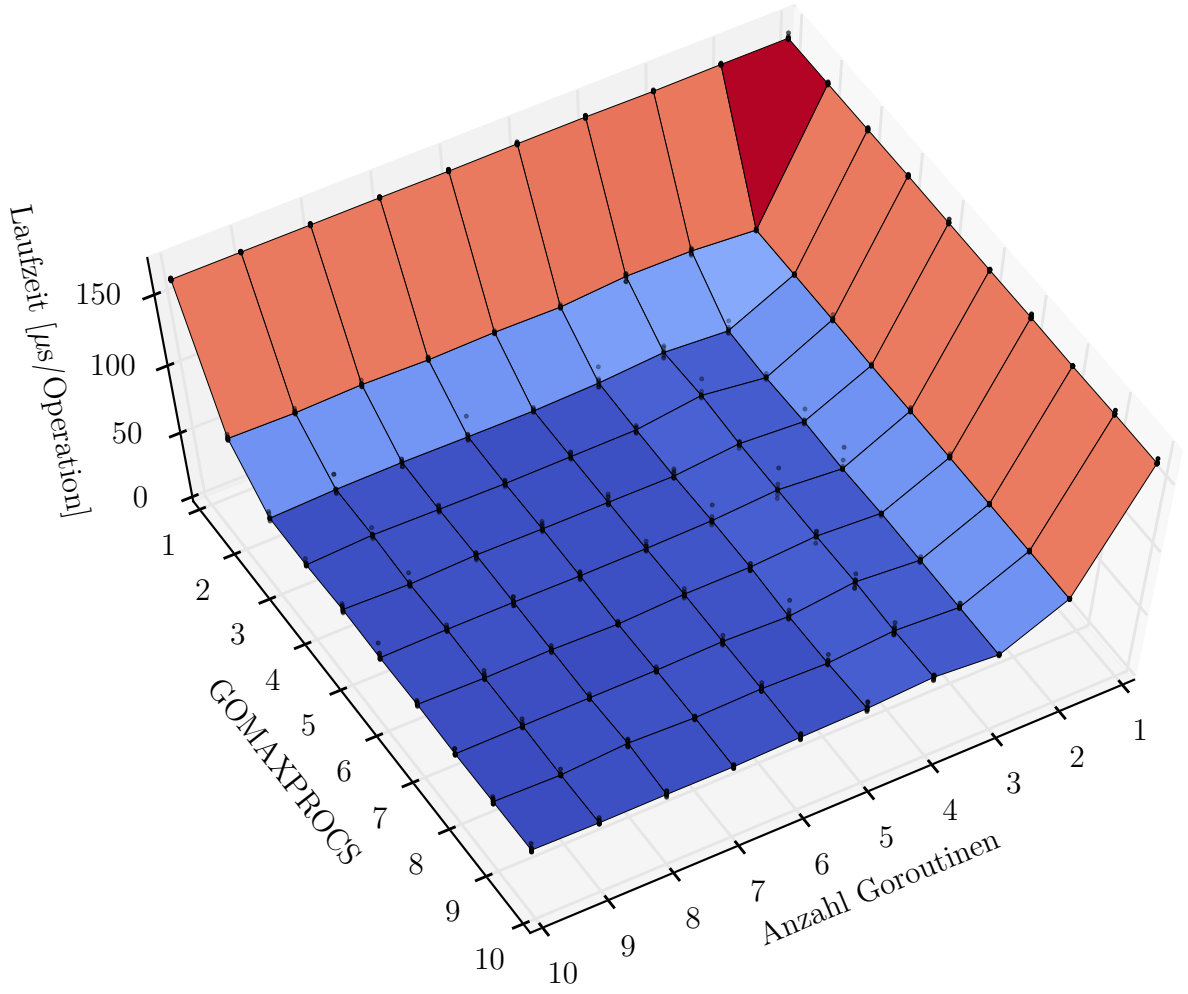
Testsystem	**System B**
Anzahl Kerne	2
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 5.3: Suchraum des Autotuners im Threadpool von `nfnt/resize` auf einem anderen Testsystem. Diese Messung verwendet wieder Go 1.2 und wurde in einer virtuellen Maschine durchgeführt. Der Suchraum unterscheidet qualitativ sich nur unwesentlich von dem in Abbildung 5.1 gezeigten.



Testsystem	**System B**
Anzahl Kerne	4
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 5.4: Suchraum des Autotuners im Threadpool von `nfnt/resize`. Wenn vier Rechenkerne zur Verfügung stehen ist im Vergleich zu den vorherigen Messungen eine weitergehende Beschleunigung möglich.



Testsystem	**System B**
Anzahl Kerne	4
cpuburn-Instanzen	1
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 5.5: Suchraum des Autotuners im Threadpool von `nfnt/resize`. Wenn eine Instanz von `cpuburn` läuft, erlaubt die vierte Goroutine bzw. der vierte Thread keine Beschleunigung mehr.

5.4.2 Vorgehen des Autotuners

Um das Vorgehen des Userspace-Autotuners bei der Optimierung des `nfnt/resize`-Threadpools zu untersuchen, wurden zehn Messreihen durchgeführt, bei denen der Autotuner eine automatische Parallelisierung durchführt. Er geht jeweils vom sequentiellen Fall `threads=1`, `jobs=1` aus und führt im Nelder-Mead-Modus dreißig Messungen und zugehörige Rekonfigurationen durch. Die Messungen werden auf System B durchgeführt. Die Rahmenbedingungen sind dieselben wie bei der Messung zu Abbildung 5.4.

Exemplarisch sind in Tabelle 5.2 die dreißig Messungen einer Messreihe aufgeführt. In Abbildung 5.6 sind die vom Userspace-Autotuner gewählten Messstellen im Suchraum markiert. Die konkrete Auswahl der aktivierten Konfigurationen hängt insbesondere vom Zufall, dem gewählten Startwert sowie von den zuvor gemessenen Zeiten ab. Messreihen sind daher nicht reproduzierbar.

	Messungen 1-15 Jobs, Threads, μs		Messungen 16-30 Jobs, Threads, μs
1	1, 1, 157.07	16	10, 2, 81.05
2	8, 10, 47.24	17	6, 4, 44.67
3	8, 10, 43.72	18	6, 4, 44.48
4	8, 10, 44.60	19	6, 4, 43.74
5	8, 10, 42.70	20	6, 5, 49.23
6	8, 10, 47.20	21	8, 4, 45.74
7	5, 5, 48.86	22	8, 4, 45.75
8	5, 5, 47.51	23	8, 4, 42.81
9	8, 3, 57.27	24	7, 5, 43.82
10	8, 3, 54.88	25	7, 5, 45.41
11	2, 7, 80.19	26	7, 5, 44.84
12	2, 7, 79.93	27	7, 3, 56.84
13	7, 4, 44.89	28	7, 4, 45.65
14	7, 4, 43.96	29	7, 4, 43.54
15	7, 4, 44.28	30	7, 4, 42.57

Tabelle 5.2: Beispiel für eine Messreihe mit dem Userspace-Autotuner im Nelder-Mead-Modus im Suchraum zum `nfnt/resize`-Threadpool.

Auffällig ist, dass beispielsweise in den Messungen 2-6 mehrfach nacheinander dieselbe Konfiguration gewählt wird, obwohl der Autotuner keine wiederholten Messungen zur Steigerung der Konfidenz durchführt. Insgesamt wurden in den dreißig Durchläufen nur 12 verschiedene Konfigurationen aktiviert.

Dies liegt daran, dass das Nelder-Mead-Verfahren ursprünglich für unbeschränkte, kontinuierliche Suchräume ausgelegt ist. Der Userspace-Autotuner verwendet es jedoch in beschränkten, diskreten Suchräumen und verwendet die nächstgelegene zulässige Messstelle, wenn die berechnete Messstelle außerhalb des Suchraums oder nicht direkt auf einem Gitterpunkt liegt. Wenn das Simplex im Vergleich zur Gitterweite klein wird, führt dies zu

signifikanten Abweichungen vom intendierten Verhalten und dazu, dass mehrere Ecken auf dieselbe Konfiguration gerundet werden.

Für die nachfolgenden Betrachtungen werden eine Referenzkonfiguration und eine zugehörige Referenzzeit festgelegt. Dies ist die Konfiguration `threads=4, jobs=5`, die bei der Exploration des Suchraums im Median die kürzeste Laufzeit erreicht hat. Als Referenzzeit wird dieser Median von $42.575\mu s$ pro Operation angesetzt. Es ist daher unwahrscheinlich, aber aufgrund des Rauschens möglich, dass einzelne Durchläufe mit Zeiten unter der Referenzzeit gemessen werden.

Ziel der Bewertung sind einerseits Aussagen über die Laufzeit der schnellsten gefundenen Konfiguration in Abhängigkeit der Anzahl der Durchläufe. Diese Angabe ist insbesondere beim Offlinetuning entscheidend, weil hier primär die beste gefundene Konfiguration interessiert. Außerdem wird untersucht, wie lange die Optimierung dauert, bzw. wie die aktivierten Konfigurationen im Durchschnitt laufen. Diese Angabe ist insbesondere beim Onlinetuning bedeutsam, weil dabei auch wichtig ist, dass der Messlauf, der dem Programmlauf entspricht, schnell beendet wird.

In Abbildung 5.7 bis Abbildung 5.9 ist die Laufzeit der Konfigurationen aufgetragen, welche der Userspace-Autotuner im Nelder-Mead-Modus aktiviert. Jede Linie entspricht einer Messreihe; das Experiment wurde zehn Mal wiederholt. In jeder Messreihe wurde der Messabschnitt dreißig Mal durchlaufen, dies ist auf der x-Achse aufgetragen.

In Abbildung 5.7 ist auf der y-Achse zunächst die Laufzeit der jeweils aktivierten Konfiguration aufgetragen. Die gestrichelte Linie bei 42.58 entspricht der Referenzzeit. Eine systematische Entwicklung der Messwerte ist nicht erkennbar. Erkennbar ist allerdings, dass in den meisten Durchläufen auch zu späteren Zeitpunkten wieder schlechtere Konfigurationen aktiviert wurden. Messreihe 4 entspricht der Messung aus Tabelle 5.2.

In Abbildung 5.8 ist auf der y-Achse die Laufzeit der bis dato schnellsten Konfiguration (pro Messreihe) aufgetragen. Per Definition sind alle Linien monoton fallend. Im gegebenen Suchraum ist die gewählte Startkonfiguration schlecht und benötigt mehr als das Dreifache der Referenzzeit. In allen zehn Messreihen wird sodann jedoch eine Konfiguration aktiviert, die höchstens 18% über der Referenzzeit liegt. Kurz darauf wird in allen Messreihen eine Konfiguration gemessen, deren Laufzeit nicht mehr signifikant von der Referenzzeit abweicht.

Dieses sehr gute Optimierungsverhalten ist damit zu erklären, dass in diesem Suchraum viele gute Konfigurationen vorliegen. Auch ein rein zufällig agierender Autotuner erreicht vergleichbare Ergebnisse, siehe dazu den folgenden Abschnitt.

Um einen Überblick über das Ausmaß der Fehltritte des Autotuners zu erhalten, bietet es sich an, die durchschnittliche Laufzeit der in einem Messlauf aktivierten Konfigurationen zu untersuchen. Diese ist proportional zur Gesamtdauer des Tuningvorgangs. In Abbildung 5.9 wurde für jeden Durchlauf die durchschnittliche Laufzeit aller bis dahin erfolgter Messungen der Messreihe aufgetragen. Im Nelder-Mead-Modus werden für dreißig

Durchläufe meist 18-62% mehr Zeit benötigt als mit der Referenzkonfiguration; in einer Messreihe sind es sogar 150%.

Die Referenzkonfiguration kann ein Offlinetuner vor dem Produktivbetrieb ermitteln. Onlinetuning hat den Vorteil, dass es sich an dynamische Veränderungen, wie beispielsweise die anderweitige Auslastung eines Rechenkerns, anpassen kann.

5.4.3 Vergleich mit den k-Proben-Heuristiken

Im Vergleich mit den k-Proben-Heuristiken zeigen sich die Stärken und Schwächen des Nelder-Mead-Verfahrens im Userspace-Autotuner. Abbildung 5.10 zeigt die Erwartungswerte $E_{best}(h)$ und $E_{avg}(k, h)$ der k-Proben-Heuristiken für ausgewählte k über die ersten dreißig Durchläufe. Die zuvor ermittelten Median-Laufzeiten der Konfigurationen werden als z-Werte verwendet.

Wie in Abbildung 5.8 und Abbildung 5.10 zu sehen ist, erzielen sowohl der Userspace-Autotuner im Nelder-Mead-Modus als auch Zufallsproben nach nur drei Durchläufen schon sehr gute Konfigurationen unter $50\mu s$. Dabei ist zu beachten, dass es sich bei der k-Proben-Heuristiken um Erwartungswerte handelt, bei den Messreihen zum Nelder-Mead-Verfahren um eine Stichprobe. Kein Verfahren kann diese Werte für weitere Messreihen garantieren.

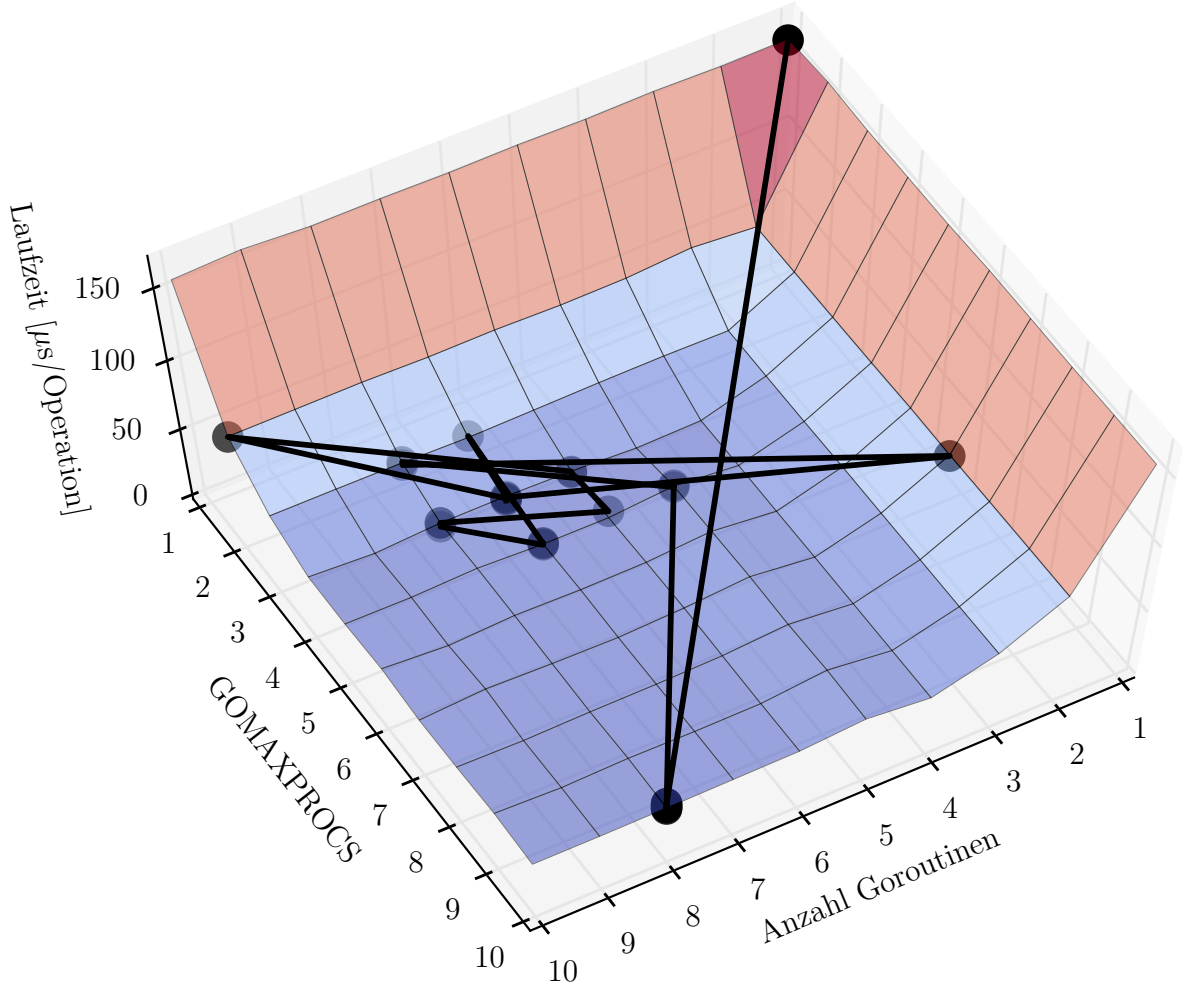
Die schlechte Leistung der Startkonfiguration ist ein wichtiger Grund, warum die Messreihen im Nelder-Mead-Modus nicht besser abschneiden. Der Startwert mit circa $160\mu s$ ist deutlich langsamer als der Erwartungswert $E_{best}(1) = 74.42\mu s$ einer einzelnen Probe.

Bei den k-Proben-Heuristiken hat die Anzahl k der zufälligen Stichproben großen Einfluss auf das erwartete Tuningergebnis $E_{best}(k)$ und die erwartete Tuningdauer $E_{avg}(k)$.

Heuristiken mit hohem k haben zwar eine lange Probephase mit jeweils (im Erwartungswert) langen Messungen, können aber ein gutes Tuningergebnis erwarten. Im Onlinetuning bzw. bei der erwartete Tuningdauer zahlt sich dies jedoch nur aus, wenn eine entsprechend lange produktive Phase folgt, in der die Informationen über die beste gefundene Konfiguration ausgenutzt werden können. Nähert sich k zu sehr an die Gesamtdurchlaufzahl an, dominiert die Probephase. Heuristiken mit niedrigem k haben eine kürzere Probephase, steigen danach zunächst auch schneller an, sind dann aber durch das schlechteres Ergebnis der Probephase erwartungsgemäß limitiert.

Ein gutes Endergebnis ist dann zu erwarten, wenn die Anzahl der Proben in einem angemessenen Verhältnis zu der Anzahl der weiteren Durchläufe steht. Die Werte in z sind nicht a priori bekannt, sodass die optimalen Probenanzahl k im Allgemeinen nicht vorab bestimmbar ist.

Im Nelder-Mead-Modus werden für dreißig Durchläufe 18-62% mehr Zeit benötigt als mit der Referenzkonfiguration; in einer Messreihe sind es sogar 150%. Bei einer Probenanzahl von 2-16 benötigen die k-Proben-Heuristiken 16-40% mehr Zeit als mit der Referenzkonfiguration.



Testsystem	**System B**
Anzahl Kerne	4
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 5.6: Messstellen des Userspace-Autotuners im Nelder-Mead-Modus im Suchraum zum `nfnt/resize`-Threadpool. Das Nelder-Mead-Verfahren arbeitet mit einem Simplex, d.h. im zweidimensionalen Fall mit einem Dreieck, und Punkte sollten weniger als Pfad in Richtung der Lösung, sondern mehr als Indikation für das betrachtete Gebiet gesehen werden.

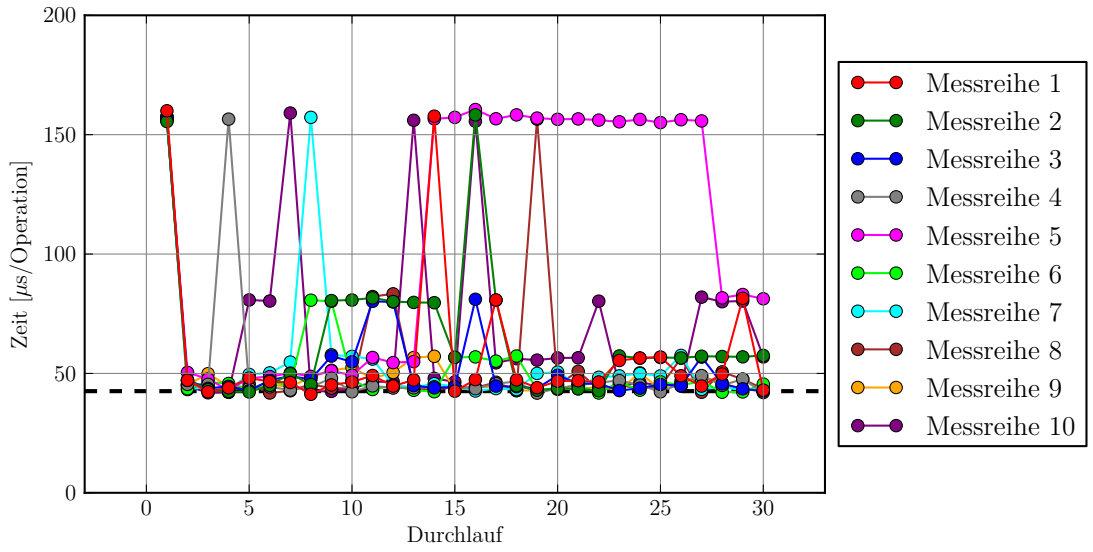


Abbildung 5.7: Laufzeit der jeweils gemessenen Konfiguration im `nfnt/resize`-Threadpool. Die Konfigurationen wurden von Userspace-Autotuner im Nelder-Mead-Modus bestimmt. In allen Messreihen wurden auch in späteren Durchläufen teilweise langsame Konfigurationen aktiviert. Die gestrichelte Linie markiert die Laufzeit der schnellsten Konfiguration.

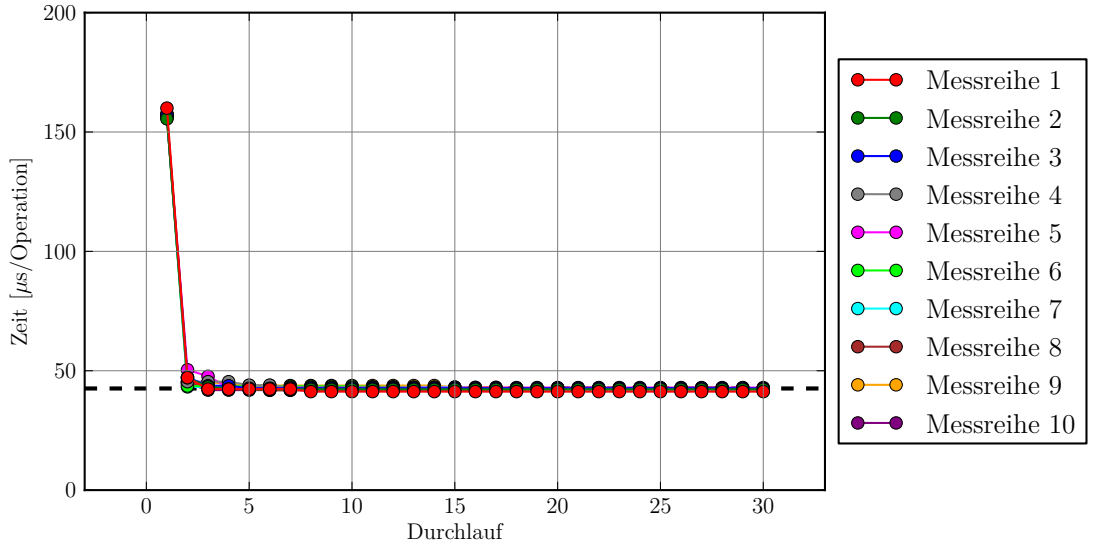


Abbildung 5.8: Laufzeit der schnellsten bis dahin gemessenen Konfiguration im `nfnt/resize`-Threadpool. In allen Messreihen wurde bereits nach wenigen Durchläufen eine sehr schnelle Konfiguration gefunden.

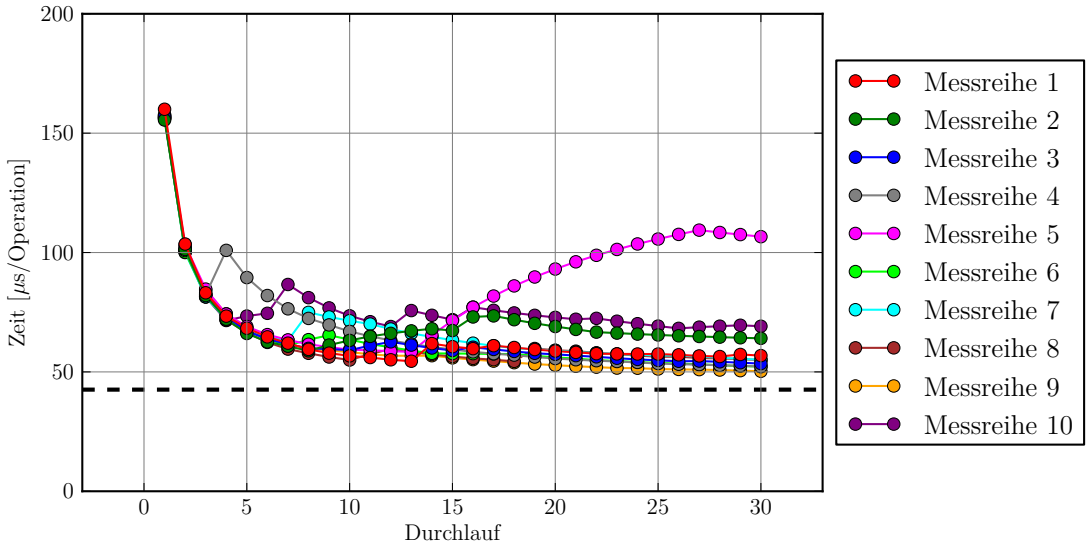


Abbildung 5.9: Durchschnittliche Laufzeit der bis dahin gemessenen Konfigurationen im `nfnt/resize`-Threadpool. Diese bleibt in allen Messreihen nach dreißig Durchläufen deutlich über der Referenzzeit. Meist wurde 18-62% mehr Zeit benötigt als mit der Referenzkonfiguration; in einer Messreihe jedoch 150%.

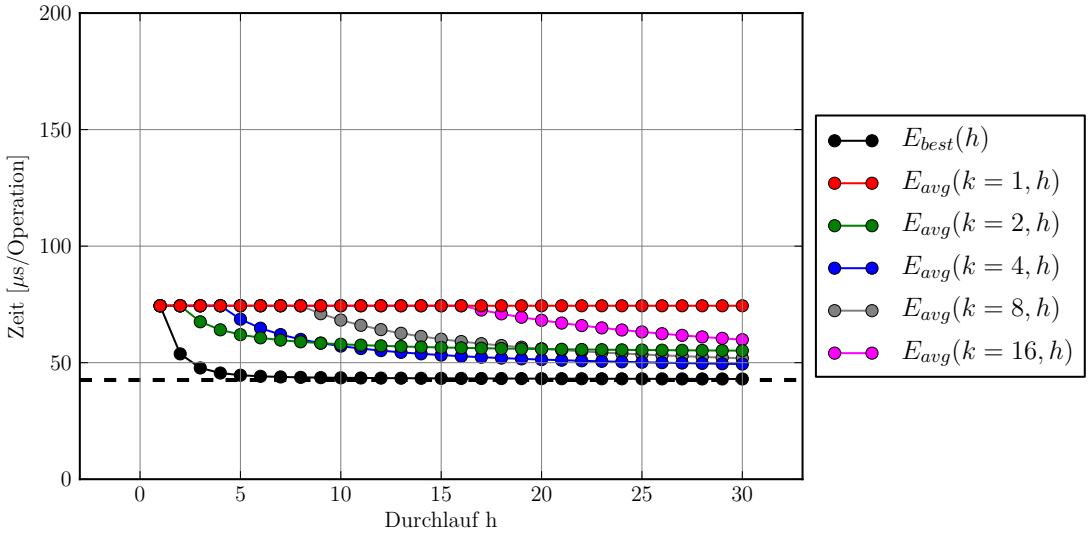


Abbildung 5.10: Erwartete Laufzeit der Konfigurationen im `nfnt/resize`-Threadpool bei Verwendung der k-Proben-Heuristik. Der Erwartungswert $E_{best}(k)$ für die beste gefundene Konfiguration fällt mit steigendem k schnell ab. Die durchschnittliche Laufzeit einer k-Proben-Heuristik nähert sich nach Ablauf der zufälligen Probephase langsam der gefundenen schnellsten Laufzeit an.

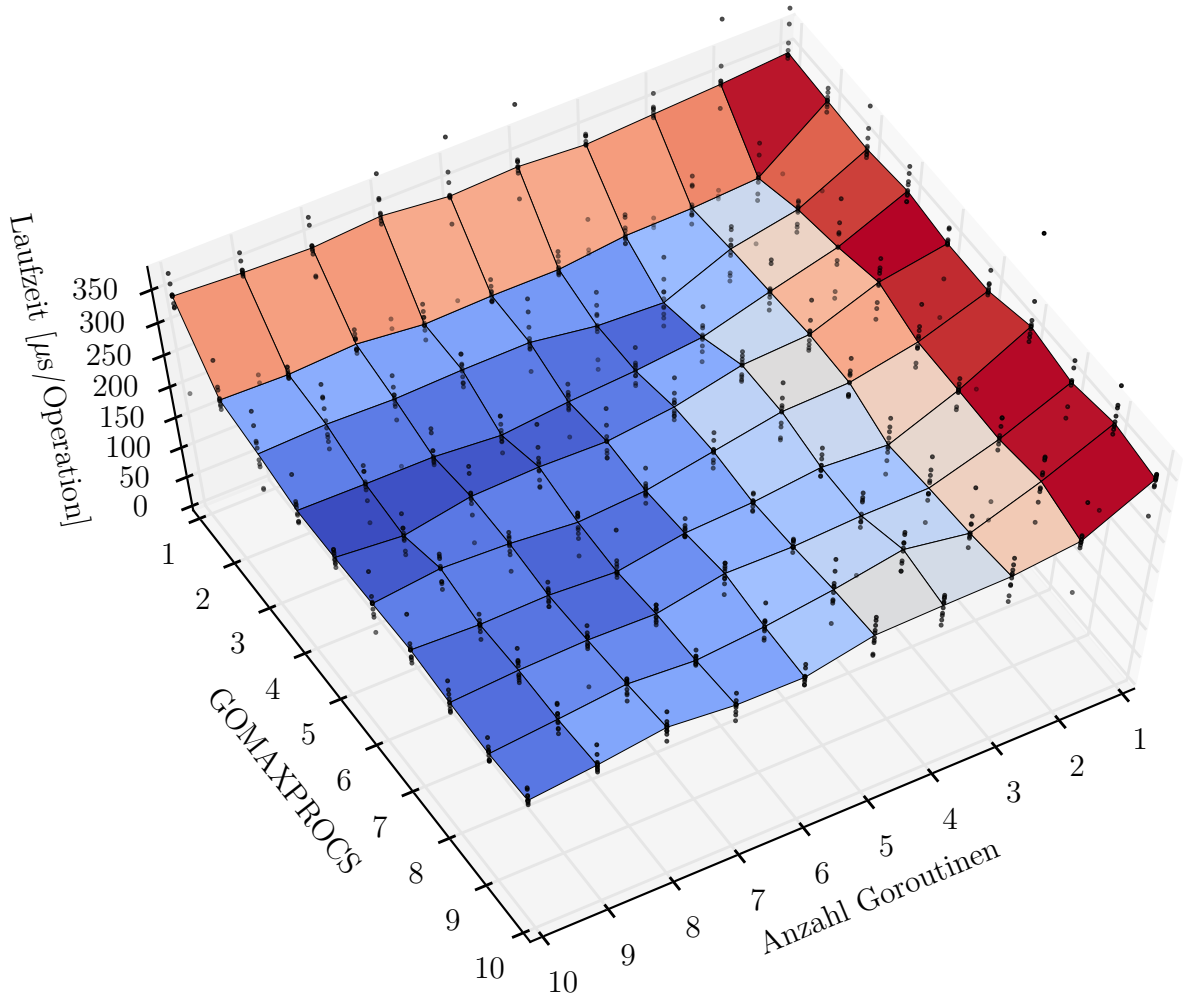
Die durchschnittliche Laufzeit bei Verwendung einer k-Proben-Heuristiken (mit mehr als einer Probe) ist nach der Probephase streng monoton fallend, da nur noch die beste Probe verwendet wird. Im Fall des Nelder-Mead-Modus kann sich die durchschnittliche Laufzeit jedoch jederzeit wieder verschlechtern. Dies ist bei Messreihe 5 deutlich zu erkennen.

5.4.4 Robustheit gegen Störungen

Auf Systemen mit niedriger sonstiger Auslastung, wie in den zuvor besprochenen Messungen ohne `cpuburn`, ist die Wiederholbarkeit der Messungen gut und die Streuung zwischen den einzelnen Messungen gering. Bei starker Auslastung des Systems wie in der Messung zu Abbildung 5.11 ist die Wiederholbarkeit jedoch schlecht und die Streuung hoch.

Es muss für jede Anwendung im Einzelfall geklärt werden, überhaupt eine Konfiguration als optimal bezeichnet werden kann und ob dies direkt die Konfiguration mit dem niedrigsten Median sein soll, oder ob das Rauschen berücksichtigt werden muss.

Autotuner wie der Userspace-Autotuner, die für jeden Messpunkt nur eine Messung vornehmen, werden kein reproduzierbares und vermutlich auch kein gutes Ergebnis finden, da sich bei Verwendung unterschiedlicher Messungen an den Messstellen unterschiedliche Suchräume ergeben, die weder quantitativ noch qualitativ übereinstimmen.



Testsystem	**System B**
Anzahl Kerne	4
cpuburn-Instanzen	4
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 5.11: Suchraum des Autotuners im Threadpool von `nfnt/resize`. Die horizontal übereinander liegenden Punkte zeigen die starke Streuung der Messwerte an den einzelnen Messstellen.

5.5 Validierung

Zur Validierung der Ergebnisse der `nfnt/resize`-Fallstudie wurden Messungen an einer zweiten Bibliothek durchgeführt, die ein ähnliches Prinzip zur Beschleunigung auf Mehrkernsystemen nutzt: `disintegration/imaging` [dis] ist wie `nfnt/resize` eine in Go geschriebene Bibliothek zur Skalierung von Pixelgrafiken.

5.5.1 Der Freiheitsgrad

Die `Resize`-Methode in `disintegration/imaging` streckt bzw. staucht das Bild in beiden Dimensionen nacheinander.

```
1 func Resize(img image.Image, w, h int, ...) *image.NRGBA { ...
2   var dst *image.NRGBA
3   ...
4   dst = resizeHorizontal(src, w, filter)
5   dst = resizeVertical(dst, h, filter)
6   ...
7   return dst
8 }
```

Der Code für die beiden Skalierungen ist gleich strukturiert. Im Folgenden ist daher exemplarisch nur die horizontale Streckung aufgeführt.

```
1  func resizeHorizontal(...) *image.NRGBA { ...
2    dst := image.NewNRGBA(image.Rect(0, 0, dstW, dstH))
3    ...
4    parallel(dstW, func(partStart, partEnd int) { ...
5      for dstX := partStart; dstX < partEnd; dstX++ { ...
6        for dstY := 0; dstY < dstH; dstY++ {  ...
7          ... // calculate and store result pixel
8        }
9      }
10   })
11   return dst
12 }
```

Wie auch in `nfnt/resize` wird das Ergebnisbild in zwei geschachtelten Schleifen pixelweise bestimmt. `disintegration/imaging` verwendet zur Parallelisierung eine dedizierte Methode `parallel`. Dieser wird als Parameter eine streifenweise agierende Bildbearbeitungsmethode in Form einer anonymen Methode übergeben.

`parallel` startet Goroutinen, welche die Bildbearbeitungsmethode ausführen. Die Bildbearbeitungsmethoden wiederum akzeptieren die Parameter `partStart` und `partEnd` zur Festlegung des zu bearbeitenden Ausschnitts, welche von der Methode `parallel` entsprechend gesetzt werden.

Anders als bei `nfnt/resize` wird nicht pro Goroutine ein Streifen bearbeitet, sondern die Anzahl der Streifen ist deutlich höher und jede Goroutine bearbeitet viele Streifen. `parallel` bestimmt die Anzahl der zu verwendenden Goroutinen `numGoroutines` anhand der folgenden einfachen Heuristik:

```
1 numProcs := runtime.GOMAXPROCS(0)
2 ...
3 numGoroutines =  numProcs
```

Für jeden von der Laufzeitumgebung genutzten Thread wird eine Goroutine gestartet. Für rechenintensive Probleme und unter der Annahme, dass innerhalb des Programms alle Goroutinen von `disintegration/imaging` zur Bildskalierung genutzt werden können, ist dies eine akzeptable Heuristik.

Die Breite eines Streifens `partSize` wird mit der folgenden Heuristik festgelegt:

```
1 partSize = dataSize / (numGoroutines * 100)
2 if partSize < 1 {
3   partSize = 1
4 }
```

Die Streifen werden so dünn geschnitten, dass sich circa 100 Streifen pro Goroutine ergeben, wobei ein Streifen immer mindestens einen Pixel breit ist.

Es folgt der Start der Goroutinen, die Bearbeitung der Streifen und das Warten auf die Beendigung aller Goroutinen. In `disintegration/imaging` wird eine `WaitGroup` anstelle der expliziten Channel zur Synchronisation verwendet.

```
1 var wg sync.WaitGroup
2 wg.Add(numGoroutines)
3 idx := uint64(0)
4
5 // Loop with Goroutines
6
7 wg.Wait()
```

In Zeile 2 wird per `wg.Add` die Anzahl der Aufrufe von `wg.Done` (innerhalb der Goroutinen) festgelegt, die erfolgen müssen, bevor die Barriere bei `wg.Wait` in Zeile 7 fällt.

```
1  // Loop with Goroutines
2  for p := 0; p < numGoroutines; p++ {
3    go func() {
4      defer wg.Done()
5      for {
6        partStart := int(
             atomic.AddUint64(&idx, uint64(partSize))
             ) - partSize
7        if partStart >= dataSize { // finished
8          break
9        }
10       partEnd := partStart + partSize
11       if partEnd > dataSize { // truncate last part
12         partEnd = dataSize
13       }
14       fn(partStart, partEnd)
15     }
16   }()
17 }
```

Die Schleife startet Goroutinen mit einer anonymen Funktion (Zeilen 3-16). Diese Funktion wiederum ruft in Zeile 14 wiederholt die übergebene Skalierungsfunktion mit den Streifengrenzen als Parameter auf.

In Zeile 6 wird der Anfang des nächsten unbearbeiteten Streifens bestimmt. Dazu wird die über alle Goroutinen geteilte Variable `idx` verwendet. Auf diese wird per atomare Operation zugegriffen und der Wert jeweils um eine Streifenbreite erhöht. Ist kein Streifen mehr verfügbar, wird die Schleife in Zeile 8 verlassen.

Die Zuteilung der Streifen auf die Goroutinen stellt einen Threadpool dar. Außerdem besteht weiterhin der Threadpool, der für die Zuteilung der Goroutinen auf Betriebssystem-Threads verantwortlich ist. Dieser zweite Threadpool soll automatisch optimiert werden um die Ergebnisse der `nfnt/resize`-Fallstudie zu überprüfen.

5.5.2 Definition der Messabschnitte

Der Messabschnitt, welcher den zu optimierenden Codeabschnitt markiert, umfasst den kompletten Rumpf der `Resize`-Methode mit beiden Aufrufen der `parallel`-Methode.

```
func Resize(img image.Image, w, h int, ...) *image.NRGBA { ...
  tuner.Tic()
  // after Tic for compatibility with Userspace-Autotuner
  tuner.Reconfigure()

  // original method body
  ...
  dst = resizeHorizontal(src, w, filter)
  dst = resizeVertical(dst, h, filter)
  ...

  tuner.Toc(...)
  return dst
}
```

5.5.3 Extraktion der Parameter

Autotuning-Variablen können analog zu denen in `nfnt/resize` eingeführt werden.

```
var tuner = fastergo.CreateGoFullWalker()
var jobs, jobs_min, jobs_max, jobs_step
     = 1, 1, 10, 1
var threads, threads_min, threads_max, thread_step
     = 1, 1, 10, 1

func init() {
  tuner.AddParameter(&jobs, "#jobs",
     jobs_min, jobs_max, jobs_step)
  tuner.AddParameter(&threads, "#threads",
     threads_min, threads_max, thread_step)
}
```

Die Initialisierung der Variablen `numProcs` in der Heuristik zur Bestimmung der Anzahl der Streifen muss so geändert werden, dass der Autotuner-Variablen verwendet wird.

```
- numProcs := runtime.GOMAXPROCS(0)
+ numProcs := jobs // runtime.GOMAXPROCS(0)
```

Im `OnReconfigure`-Callback wird unter Missachtung von `GOMAXPROCS` die maximale Anzahl der von der Runtime zu verwendenden Threads gesetzt, ebenfalls auf einen Wert, den der Autotuner vorgibt.

```
f := func() { runtime.GOMAXPROCS(threads) }
tuner.SetOnReconfigure(f)
```

5.5.4 Ergebnis

Abbildung 5.12 zeigt eine Visualisierung des Suchraums in `disintegration/imaging`. Qualitativ stimmt die Struktur des Suchraums mit dem in `nfnt/resize` überein. Die hohe Zahl von circa 100 Streifen pro Goroutine sorgt dafür, dass Arbeitspakete, die durch unglückliches Scheduling liegengeblieben sind, keine so großen Verzögerung verursachen wie in der `nfnt/resize`-Fallstudie. Insgesamt können die Ergebnisse der `nfnt/resize`-Fallstudie validiert werden.

5.6 Zusammenfassung

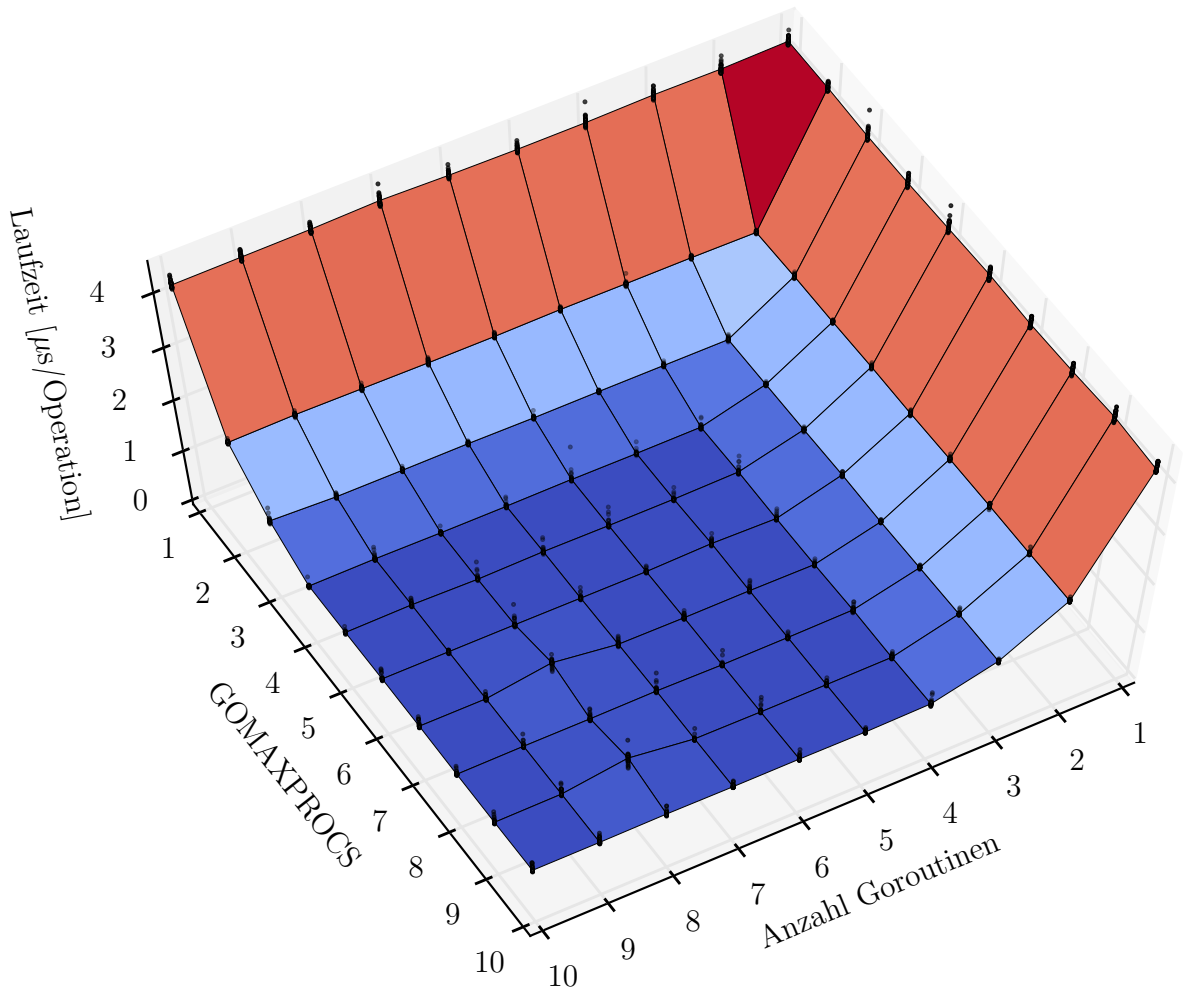
In dieser Fallstudie wurde erfolgreich Autotuning an einem Go-Programm demonstriert. Das Vorgehen umfasst das Erkennen des Freiheitsgrades, die Definition der Messabschnitte, die Extraktion der Parameter und die Anbindung an den Autotuner.

Goroutinen sind als zentrales Sprachkonzept tief in Go eingebettet und sorgen dafür, dass der dahinterstehende Threadpool und seine Parameter – allen voran `GOMAXPROCS` – allgegenwärtig sind. Die Hardware beeinflusst den Threadpool. Auch verschiedene Go-Versionen reagieren leicht unterschiedlich, sodass Anpassungen über Autotuning attraktiv werden. Gängige Heuristiken verwenden oftmals die Anzahl der Rechenkerne, und berücksichtigen nicht, ob diese überhaupt über freie Kapazitäten verfügen.

Das Rundungsverhalten des Userspace-Autotuners sollte genauer untersucht werden. Es ist ungeklärt, inwiefern die systematische Verletzung der Voraussetzungen des Nelder-Mead-Verfahrens die Korrektheit des Verfahrens beeinflusst. Die doppelten Messungen sollten unterlassen werden, oder aber dazu genutzt werden, die Konfidenz zu erhöhen und Schwankungen in den Messungen auszugleichen. Autotunerläufe sind nicht reproduzierbar, dies erschwert die Nachvollziehbarkeit insbesondere bei der Fehlersuche.

Auf Stichproben basierende Optimierungsheuristiken sind für den Autotuner schwer zu schlagen, wenn es im Suchraum hauptsächlich gute Konfigurationen gibt. Bei einem geringeren Anteil guter Konfigurationen und bei einem ausgeprägten Gradienten in Richtung des Optimums können Verfahren wie das Downhill-Simplex-Verfahren von Nelder und Mead ihre Vorteile besser ausspielen. Die Wahl der Startkonfiguration hat einen großen Einfluss auf die Dauer des Tuningvorgangs.

Die Validierung an einer unabhängigen Bibliothek zeigt, dass die beim Autotuning von `nfnt/resize` gewonnenen Erkenntnisse über den in jedem Go-Programm vorhandenen Threadpool der Goroutinen gut auf andere Szenarien übertragbar sind.



Testsystem	**System B**
Anzahl Kerne	4
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 5.12: Suchraum des Autotuners im Threadpool von `disintegration/imaging`. Der Suchraum stimmt qualitativ mit dem in Abbildung 5.4 gezeigten Suchraum im `nfnt/resize`-Threadpool überein.

6. Fallstudie: Paralleler Datenbankzugriff

Viele Go-Programme verwalten Daten in Datenbanken. Datenbanken erfordern meist den expliziten Aufbau einer persistenten Verbindung, über welche die weitere Kommunikation abläuft. Go bietet einen Verbindungspool, der mehrere Verbindungen zu einem Datenbankserver offenhält und diese bei der Bearbeitung von SQL-Befehlen nutzt.

In dieser Fallstudie soll der Durchsatz eines Programms, das viele Daten an eine Datenbank sendet, durch Anpassen der Verbindungsparameter erhöht werden, ohne dass die Struktur der Anfragen verändert wird. Die Fallstudie ist inspiriert von einem Experiment von Folkins [Fol]. Folkins optimiert den Datenbankzugriff durch manuelle Anpassungen. Diese Fallstudie überlässt die Anpassungen dem Autotuner.

6.1 Der Freiheitsgrad

Der Verbindungspool ist Teil der Go-Standardbibliothek [gos] und wird über den folgenden Befehl in ein Programm eingebunden:

```
1 import "database/sql"
```

Die Datenbanktreiber für die spezifischen Datenbanken [lib, goma, matb] sind nicht Teil der Go-Standardbibliothek, sondern werden separat von externen Quellen eingebunden:

```
1 import _ "github.com/lib/pq"
2 import _ "github.com/go-sql-driver/mysql"
3 import _ "github.com/mattn/go-sqlite3"
```

Der Befehl `sql.Open`, der die Datenbanknutzung initiiert, verfügt über zwei Parameter, die den zu verwendenden Treiber sowie die Zugangsdaten festlegen. `sql.Open` öffnet trotz des Namens nicht direkt eine Datenbankverbindung, sondern gibt einen Verbindungspool zurück, der Verbindungen nach Bedarf öffnet.

Die Anzahl der maximal gleichzeitig geöffneten Verbindungen im Pool ist durch den Aufruf `db.SetMaxOpenConns` konfigurierbar. Ebenso ist die Anzahl der Goroutinen einstellbar, aus denen die Anfragen erfolgen. Die beiden Freiheitsgrade *Poolgröße* und *Anzahl Goroutinen* bilden einen zweidimensionalen Suchraum.

Die in diesem Experiment durchgeführte Arbeit besteht aus dem Absenden vieler SQL-Befehle an die Datenbank. Solange der Aufwand der Anfragen verlässlich abgeschätzt werden kann, können beliebige Anfragen verwendet werden. Um sicherzustellen, dass jeder Durchlauf den gleichen Aufwand verursacht, wurden bei den Messungen immer gleich strukturierte Abfragen verwendet. Alle Anfragen verfügen über die folgende Struktur:

```
1 INSERT INTO `test` (`gopher_id`, `created`) VALUES ($1, $2);
```

Folkins verteilt die Arbeit in seinem Experiment auf mehrere Gopher.[1] Jeder Gopher entspricht einem Arbeiter und wird als eigene Goroutine ausgeführt. Die Anzahl der maximal gleichzeitig arbeitenden Gopher kann über `GOMAXPROCS` festgelegt werden, ist in dieser Fallstudie allerdings auf die Anzahl der Rechenkerne fixiert.

6.2 Definition der Messabschnitte

Ein Messabschnitt umfasst den Aufbau der Verbindung und das Abarbeiten aller SQL-Befehle, bis alle Einträge in die Datenbank geschrieben sind.

```
1  func measure(connection_type, connection_parameters string) {
2    tuner.Tic()
3    pool, _ = sql.Open(connection_type, connection_parameters)
4    // Reconfigure after Tic for UAT compatibility
5    // Reconfigure after sql.Open to affect the correct pool
6    tuner.Reconfigure()
7    run_all_gophers(gopher_count)
8    // dont defer Close, include it in Tic-Toc
9    pool.Close()
10   tuner.Toc(total_work)
11 }
```

Der Aufruf von `Reconfigure` erfolgt erst nach dem Aufbau der Verbindung, sodass der `OnReconfige`-Callback den korrekten Pool in der Größe anpassen kann.

`run_all_gophers` ist die Methode, in der die eigentliche Arbeit verrichtet wird.

6.3 Extraktion der Parameter

Zwei Autotuner-Variablen werden deklariert und initialisiert. `gopher_count` steht für die Anzahl der zu verwendenden Gopher. `conlim` steht für die Größe des Verbindungspools.

[1] englisch für Erdhörnchen, nach dem Maskottchen der Sprache Go

```
var tuner = fastergo.CreateGoFullWalker()
var gopher_count, gophers_min, gophers_max, gophers_step
     = 1, 1, 25, 1
var poolsize, poolsize_min, poolsize_max, poolsize_step
     = 1, 1, 25, 1

func init() {
  tuner.AddParameter(&gopher_count, "Anzahl Goroutinen",
     gophers_min, gophers_max, gophers_step)
  tuner.AddParameter(&poolsize, "MaxOpenConns",
     poolsize_min, poolsize_max, poolsize_step)
}
```

Die Autotuner-Variablen müssen im restlichen Code entsprechend verwendet werden. Die erste Anpassung ist der Einbau eines Aufrufs von `db.SetMaxOpenConns` mit der Kopplung an die Autotuner-Variable.

```
f := func() { db.SetMaxOpenConns(conlim) }
tuner.SetOnReconfigure(f)
```

Für die Anzahl der Goroutinen, die gleichzeitig Anfragen an die Datenbank stellen, existiert bereits eine Konstante. Diese Konstante wird gelöscht und stattdessen die Autotuner-Variable `gopher_count` verwendet.

```
func run_all_gophers(gopher_count int) {
  c := make(chan string)

  for i := 0; i < gopher_count; i++ { // use autotuner variable
    work := total_work / gopher_count
    if i+1 == gopher_count { // last iteration
      work = total_work - i * (total_work / gopher_count)
    }
    go run_gopher(i, work, c)
    defer func() { <-c }() // sync
  }
}
```

Der Code zur Anpassung der Poolgröße kann in `database/sql` platziert werden, sodass alle Go-Programme, die auf eine Datenbank zugreifen, direkt vom Autotuning profitieren. Für die Anzahl der Goroutinen, die auf die Datenbank zugreifen, gilt dies allerdings nicht. Als inhärente Eigenschaft des Anwendungsprogramms wird sie in diesem festgelegt. Daher wird im Rahmen der Fallstudie auch die Optimierung der Poolgröße im Anwendungsprogramm belassen um beide Freiheitsgrade gemeinsam zu optimieren.

6.4 Evaluation

Bei der Evaluation kommt wieder das in Tabelle 5.1 beschriebenen System B zum Einsatz. Als Datenbanksystem wird zunächst PostgreSQL [Pos] verwendet, später zum Vergleich auch MySQL [MyS] und SQLite [SQL].

Der Datenbankserver läuft mit dem zu optimierenden Go-Programm auf derselben Maschine. Diese beeinflussen sich gegenseitig in ihrer Leistungsfähigkeit. Der letztendliche Flaschenhals kann in der Go-Anwendung, im Datenbankserver oder auch in der Kommunikation zwischen den beiden liegen. Der Autotuner optimiert über die im Go-Programm zur Verfügung gestellten Parameter die Geschwindigkeit der definierten Messabschnitte und versucht dabei Leistungsengpässe zu vermeiden, unabhängig davon wo diese verursacht werden. Bei Nutzung eines entfernten Datenbankservers würde auch die Leistungsfähigkeit des entfernten Systems sowie der Anbindung dorthin berücksichtigt.

`GOMAXPROCS` ist auf die Anzahl der Rechenkerne fixiert.

6.4.1 Der Suchraum

Eine Visualisierung des mit dem `GoFullWalker` vermessenen Suchraums ist in Abbildung 6.1 zu sehen. Es werden die normierten Kehrwerte der gemessene Zeiten dargestellt, weil feine Unterschiede zwischen guten Konfigurationen so besser sichtbar sind. Alle Berechnungen erfolgen mit den Laufzeiten, da bei Rechnungen mit den Leistungen harmonische Mittel verwendet werden müssten.

Das Rauschen der Messungen ist in dieser Fallstudie deutlich stärker als in der ersten Fallstudie. Es ist dennoch erkennbar, dass auf dem Testsystem B ein Streifen an Konfigurationen mit 9-20 Goroutinen existiert, die einen besonders guten Durchsatz zu erreichen. Es existiert keine 1:1-Beziehung zur Anzahl der Kerne. Auch eine grenzenlose Erhöhung der Anzahl der Goroutinen beschleunigt nicht weiter, sondern führt wieder zu einer Verschlechterung.

Die Anzahl der Goroutinen ist nicht das einzige Kriterium, auch die Größe des Verbindungspools beeinflusst die Geschwindigkeit. Dieser Wert sollte mindestens so hoch wie die Anzahl der Goroutinen sein, ansonsten tritt eine Verlangsamung ein, wenn Goroutinen auf freie Verbindungen warten müssen. Anders betrachtet kann bei fester Poolgröße eine Reduktion der Anzahl der Goroutinen eine Beschleunigung des Programms bewirken.

Die Poolgröße und Anzahl der Goroutinen kann in der Praxis nicht beliebig erhöht werden, da es sonst zu Problemen mit dem Datenbankserver kommt. PostgreSQL reserviert pro möglicher Verbindung Ressourcen von signifikantem Ausmaß und stürzt ab, wenn diese nicht verfügbar sind. Das Verbindungslimit gilt zudem für alle Nutzer der Datenbank gemeinsam. Deswegen sollte die Anzahl der Verbindungen im Programm möglichst niedrig angesetzt werden, um viele gleichzeitige Nutzer zu ermöglichen.

Diese Nebenbedingung ist ein Fall, in dem weitere Optimierungskriterien außer der Laufzeit im Autotuner wünschenswert wären. Bei gleicher oder ähnlicher Laufzeit sollten Konfigurationen mit weniger Verbindungen bevorzugt werden.

Das Auswechseln des Datenbanksystems im laufenden Betrieb als Autotuning-Maßnahme ist prinzipiell umsetzbar, aber eher unrealistisch, da dies mit hohem Rekonfigurationsaufwand verbunden und oftmals auch aus administrativen Gründen[2] nicht gewünscht ist. Eine Software, die mit unterschiedlichen Datenbanksystemen betrieben werden kann und sich jeweils gut adaptiert, ist jedoch durchaus realistisch und bietet Möglichkeiten für Autotuning. Der Wechsel des Datenbanksystems entspricht dem Betrieb unter geänderten Rahmenbedingungen, wie der Wechsel der Hardware in der `nfnt/resize`-Fallstudie.

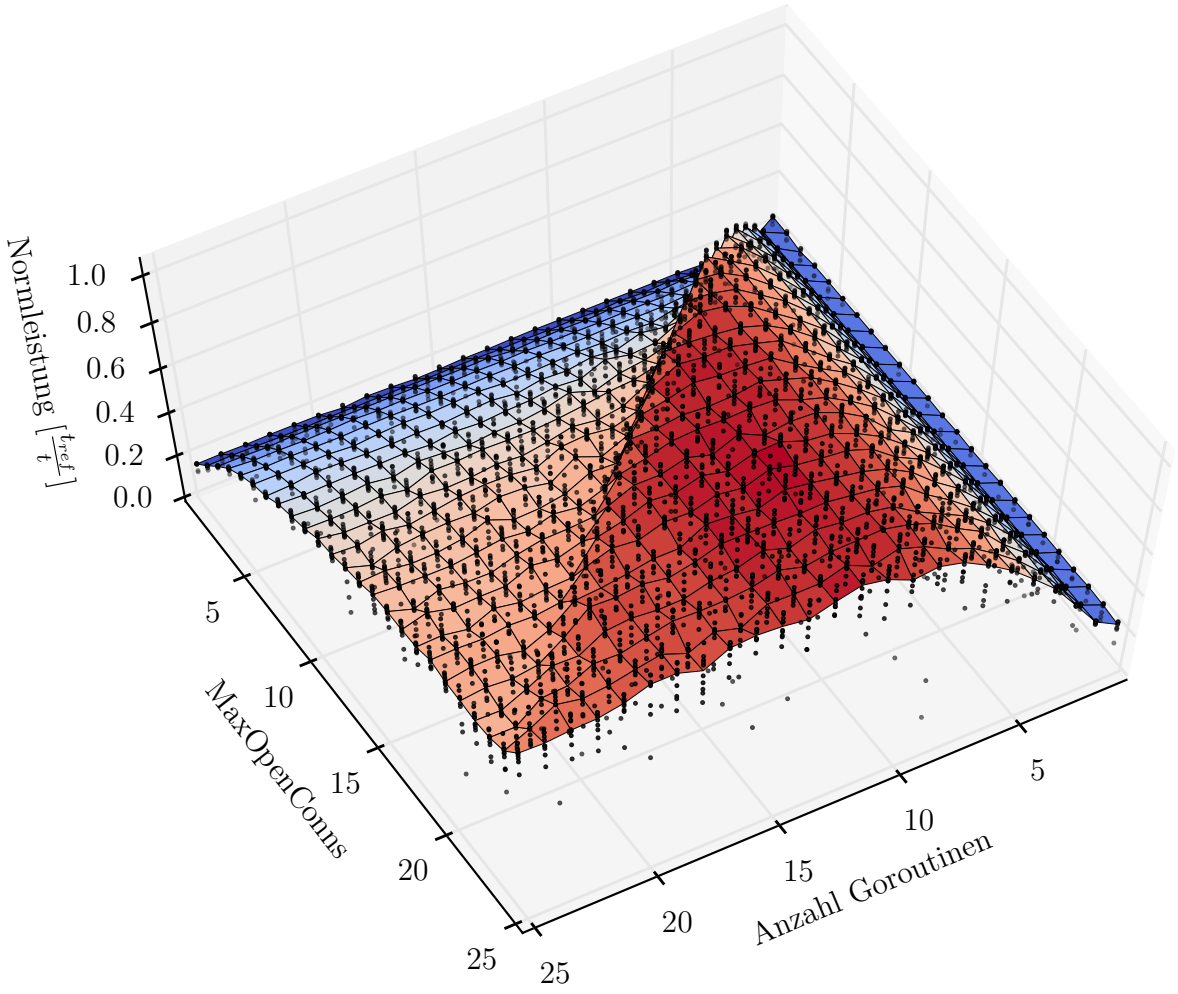
Wechselt man die PostgreSQL-Datenbank gegen eine MySQL-Datenbank aus, sind ähnliche Effekte zu beobachten wie zuvor (siehe Abbildung 6.2). Generell sind mehr Goroutinen erforderlich als bei PostgreSQL und auch die Tendenz, dass weitere Goroutinen auch bei großen Poolgrößen ab einem gewissen Punkt schaden, ist nicht mehr nachweisbar. Der negative Einfluss von zu kleinen Verbindungspools ist weiterhin vorhanden, aber weniger dramatisch. Das Rauschen nimmt weiter zu.

Die gemessenen Zeiten sollten nicht direkt zwischen den Datenbanken verglichen werden, um daraus Aussagen über die generelle Leistungsfähigkeit der Systeme abzuleiten. Es wurde keine Optimierung der Servereinstellungen vorgenommen, sondern jeweils die Standardkonfiguration verwendet. Nur die Software wurde per Autotuning an die Datenbank angepasst. Möglicherweise ist eine der beiden Standardkonfigurationen besser an die getesteten Anfragen angepasst oder führt weniger Sicherheitsmaßnahmen durch. Eine Arbeit von Bonn [Bon12] untersucht den umgekehrten Ansatz, die Datenbank für eine Anwendung zu optimieren.

Auch SQLite kann über die `database/sql`-Schnittstelle angebunden werden. SQLite zeigt andere Charakteristiken als PostgreSQL und MySQL, wie in Abbildung 6.3 zu sehen ist. Ein im Vergleich zur Anzahl der Goroutinen zu kleiner Verbindungspool wirkt sich zwar auch hier negativ auf die Geschwindigkeit aus, aber der sequentielle Betrieb ist bei Verwendung von SQLite generell schneller als der parallele Zugriff. In Bezug auf die Anzahl der Goroutinen gilt: „Je mehr, desto langsamer.“ Gute Laufzeiten werden erreicht, wenn der Pool auf eine Verbindung beschränkt wird. Die beste Laufzeit wird bei Verwendung einer einzigen Goroutine erreicht. Die Poolgröße ist dabei irrelevant, da ohnehin nur eine Verbindung verwendet wird. Das Rauschen nimmt bei Verwendung von SQLite weiter zu.

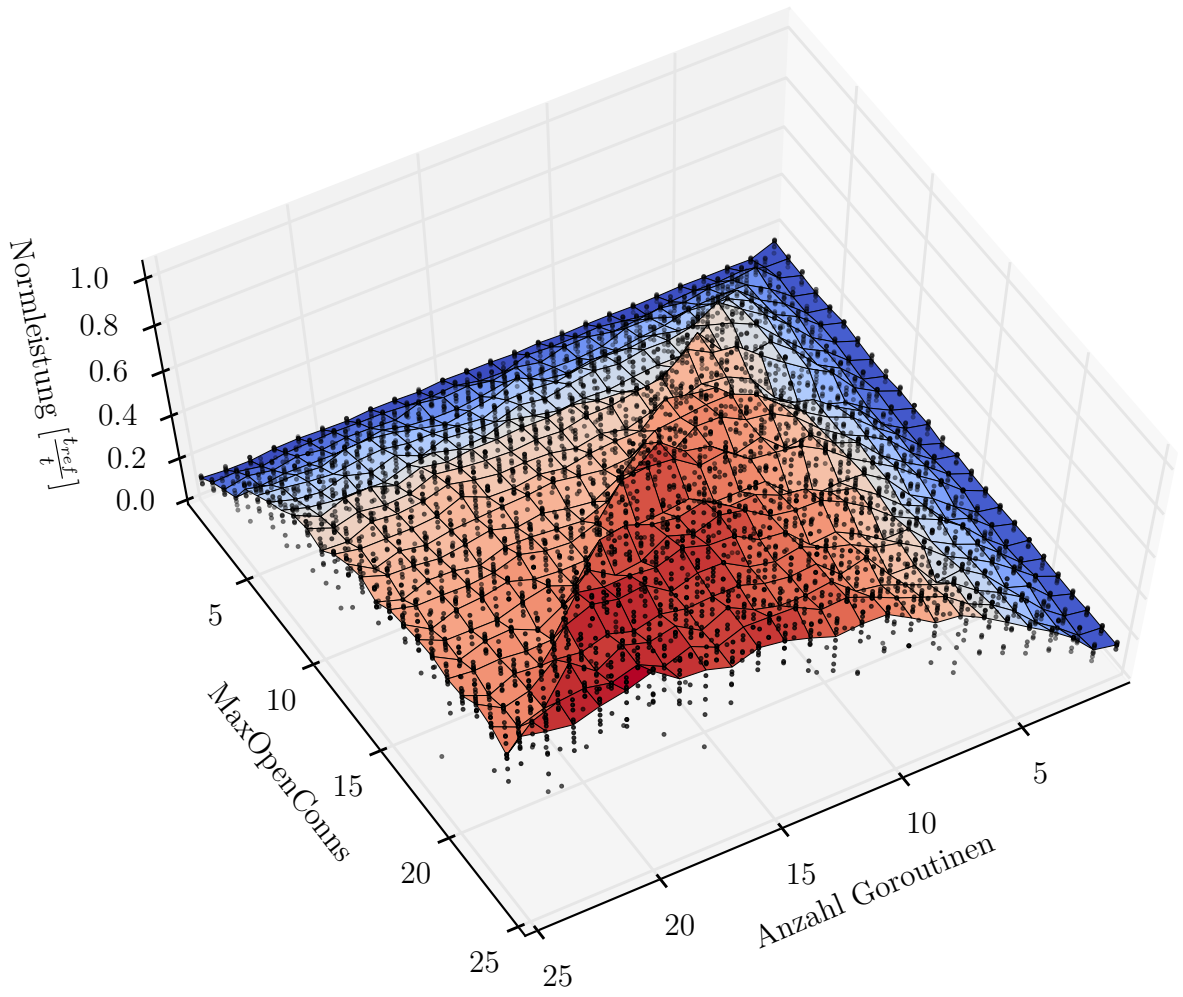
Der Autotuner kann Anwendungen an derartige Eigenheiten eines Datenbanksystems anpassen. Nur ein funktionsfähiger, vom Autotuning unabhängiger Datenbanktreiber ist erforderlich. Der Entwickler der Software muss die konkreten Eigenschaften der Datenbank nicht kennen, um vom Autotuning profitieren zu können.

[2] Backup-Richtlinien und dergleichen



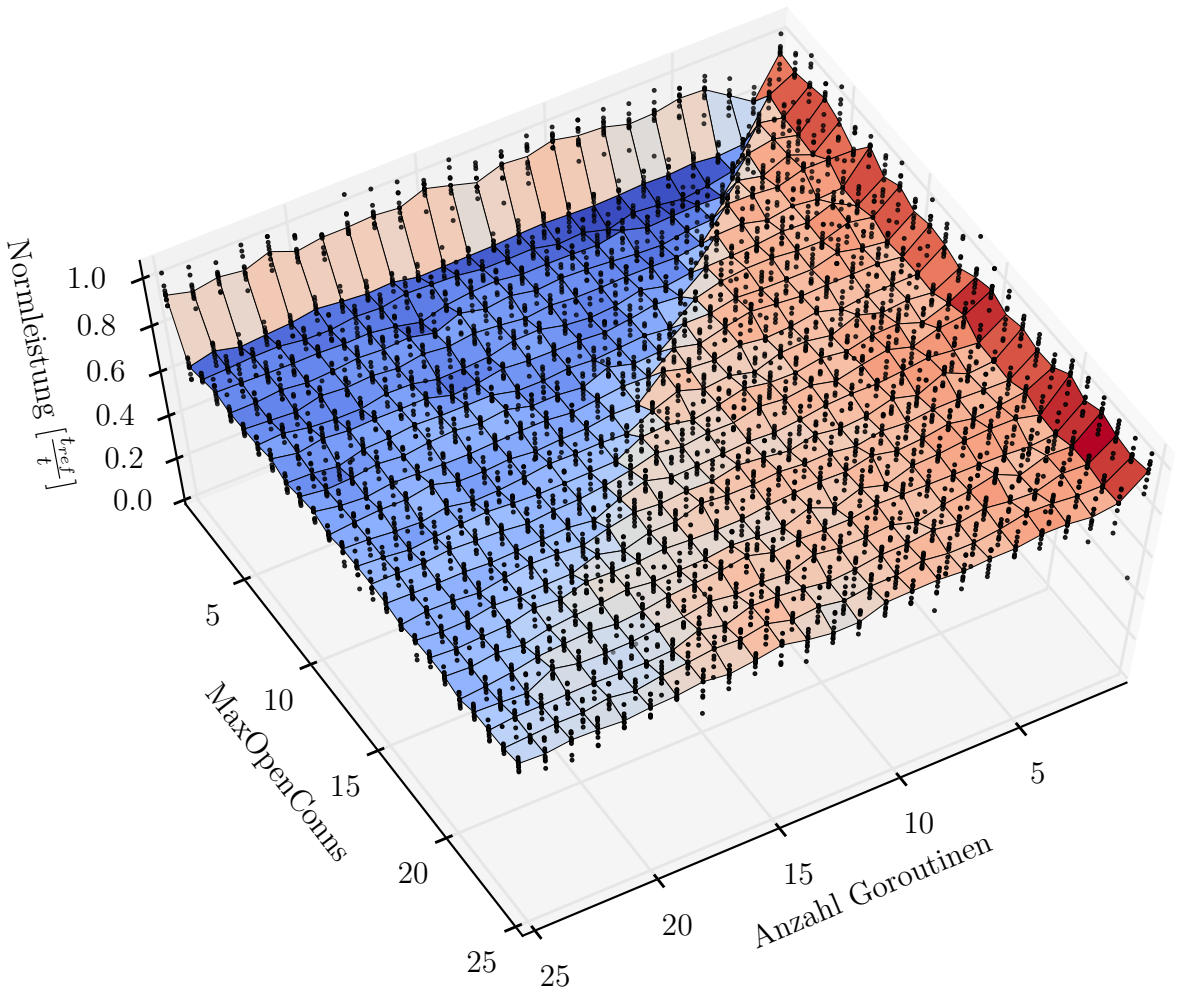
Testsystem	**System B**
Anzahl Kerne	4
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen
Datenbanksystem	PostgreSQL 9.1

Abbildung 6.1: Suchraum des Autotuners beim parallelen Datenbankzugriff auf eine PostgreSQL-Datenbank. Niederige Werte entsprechen hohen Laufzeiten. Zu wenig Goroutinen sowie zu kleine Poolgrößen verhindern gute Laufzeiten.



Testsystem	**System B**
Anzahl Kerne	4
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen
Datenbanksystem	MySQL 5.5

Abbildung 6.2: Suchraum des Autotuners beim parallelen Datenbankzugriff auf eine MySQL-Datenbank. Die Messungen rauschen stärker als bei PostgreSQL, bei niedrigen Parameterwerten sind Unterschiede zu erkennen. Das Optimum verschiebt sich zu Konfigurationen mit mehr Goroutinen.



Testsystem	**System B**
Anzahl Kerne	4
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen
Datenbanksystem	SQLite 3.7

Abbildung 6.3: Suchraum des Autotuners beim parallelen Datenbankzugriff auf eine SQLite-Datenbank. Der Suchraum rauscht stark und unterscheidet sich grundlegend von den beiden vorherigen. Der sequentielle Zugriff ist schneller als der parallele Zugriff. Der Leistungssprung entlang der Diagonalen ist stark ausgeprägt.

6.4.2 Vorgehen des Autotuners

Um das Vorgehen des Autotuners bei der Optimierung des `disintegration/imaging`-Threadpools zu untersuchen, wurden zehn Messreihen durchgeführt. Pro Messreihe führte der Userspace-Autotuner jeweils vom sequentiellen Fall `gopher_count=1`, `poolsize=1` ausgehend dreißig Messungen und Rekonfigurationen im Nelder-Mead-Modus durch.

Exemplarisch sind in Tabelle 6.1 die Messungen einer Messreihe aufgeführt. In Abbildung 6.4 sind die gewählten Messstellen im Suchraum markiert. Die konkrete Auswahl der aktivierten Konfigurationen hängt insbesondere vom Zufall, dem gewählten Startwert sowie von den zuvor gemessenen Zeiten ab. Messreihen sind daher nicht reproduzierbar.

	Messungen 1-15 Gopher, Poolgröße, μs		Messungen 16-30 Gopher, Poolgröße, μs
1	1, 1, 857.30	16	15, 25, 223.95
2	14, 11, 318.91	17	15, 25, 247.69
3	14, 11, 293.15	18	5, 7, 328.25
4	14, 11, 325.12	19	11, 20, 226.62
5	25, 21, 276.39	20	11, 20, 219.31
6	25, 21, 408.88	21	7, 10, 271.73
7	25, 21, 297.58	22	7, 10, 260.31
8	5, 15, 332.02	23	7, 10, 265.01
9	5, 15, 322.83	24	5, 11, 589.40
10	5, 15, 331.85	25	5, 11, 327.87
11	5, 15, 321.32	26	5, 11, 337.75
12	18, 10, 312.34	27	10, 13, 506.44
13	18, 10, 338.55	28	8, 18, 245.26
14	18, 10, 340.28	29	8, 18, 236.98
15	15, 25, 230.44	30	8, 18, 248.96

Tabelle 6.1: Beispiel für eine Messreihe mit dem Userspace-Autotuner im Nelder-Mead-Modus im Suchraum beim parallelen Datenbankzugriff.

Auch in dieser Messreihe wird zum Beispiel in den Messungen 8 bis 11 mehrfach nacheinander dieselbe Konfiguration gewählt, obwohl der Autotuner keine Wiederholungen zur Steigerung der Konfidenz durchführt.

Für die nachfolgenden Betrachtungen wird eine Referenzkonfiguration mit zugehöriger Referenzzeit festgelegt. Dies ist die Konfiguration `gopher_count=12`, `poolsize=19`, die bei der zuvor durchgeführten erschöpfenden Exploration des Suchraums im Median die kürzeste Laufzeit erreicht hat. Als Referenzzeit wird dieser Median von $223.495\mu s$ pro Operation gesetzt. Aufgrund des starken Rauschens werden zahlreiche Durchläufe mit Zeiten unter der Referenzzeit gemessen.

In Abbildung 6.5 ist auf der y-Achse zunächst die Laufzeit der im jeweiligen Durchlauf aktivierten Konfiguration aufgetragen. Eine systematische Entwicklung der Laufzeiten ist

nicht erkennbar. Erkennbar ist allerdings, dass in den meisten Messreihen auch in späteren Durchläufen wieder schlechtere Laufzeiten gemessen wurden. Messreihe 9 entspricht der Messung aus Tabelle 6.1.

In Abbildung 6.6 ist auf der y-Achse die Laufzeit der bis dato schnellsten Konfiguration (pro Messreihe) im Verhältnis zur Referenzkonfiguration aufgetragen. Im gegebenen Suchraum ist die gewählte Startkonfiguration schlecht, und läuft fast viermal so lange wie die Referenzkonfiguration. Die gefundenen Konfigurationen verbessern sich jedoch schnell. Der Anstieg ist weniger rapide als bei `nfnt/resize`, aber Messungen unterhalb der Referenzzeit sind aufgrund des Rauschens häufiger. Nach 20 Rekonfigurationen haben alle Messreihen eine Messung schneller als die Referenzzeit durchgeführt.

In Abbildung 6.7 wurde zu jedem Zeitpunkt die durchschnittliche Laufzeit der bis dahin in der Messreihe erfolgten Messungen aufgetragen. Für die dreißig Durchläufe wird 31-84% mehr Zeit benötigt als mit der Referenzkonfiguration.

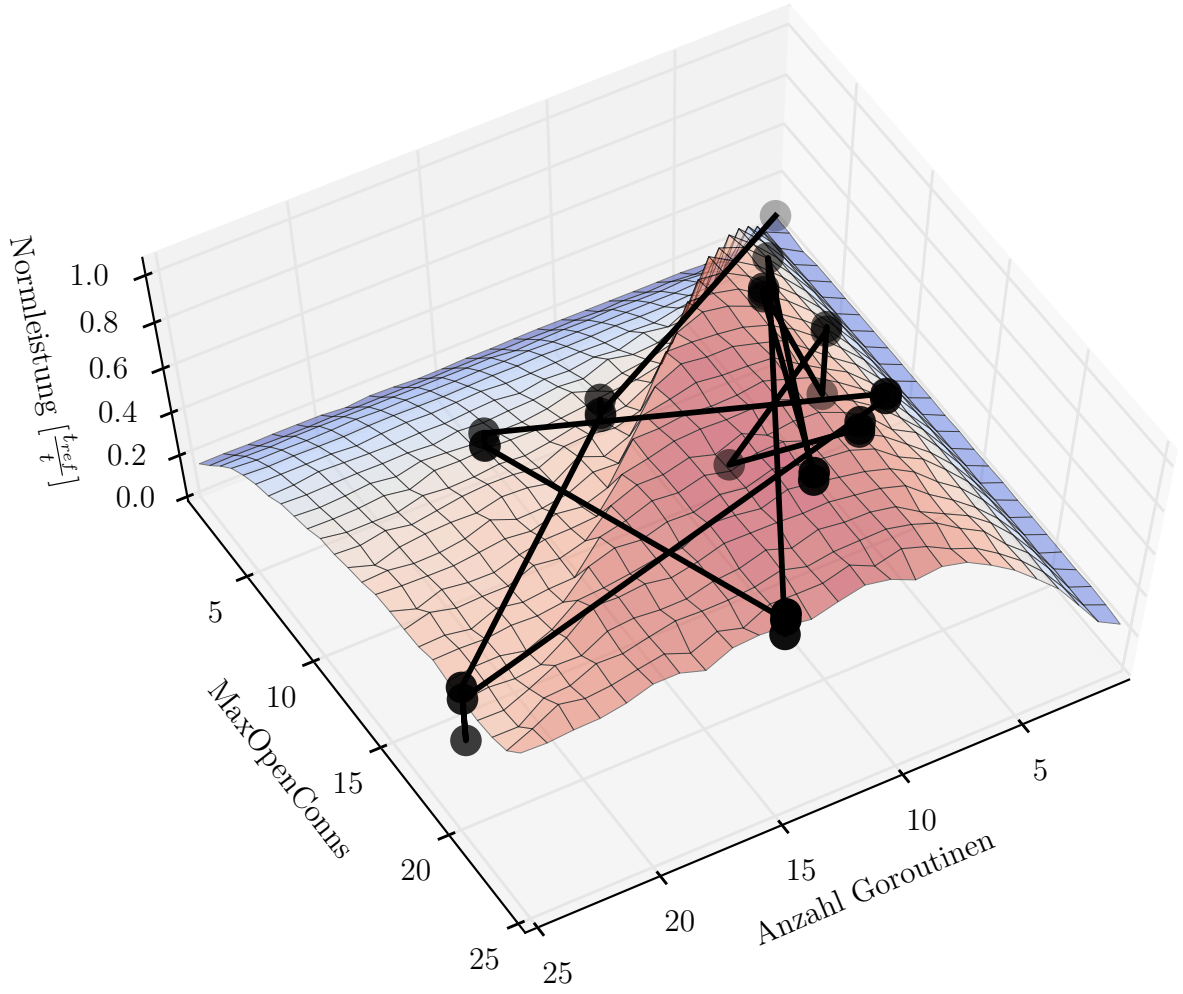
6.4.3 Vergleich mit den k-Proben-Heuristiken

Der Vergleich mit den k-Proben-Heuristiken erfolgt analog zur `nfnt/resize`-Fallstudie. Abbildung 6.8 zeigt die Erwartungswerte $E_{best}(h)$ und $E_{avg}(k, h)$ für ausgewählte k über die ersten dreißig Durchläufe. Als z-Werte werden die Mediane der beim parallelen Zugriff auf die PostgreSQL-Datenbank gemessenen Laufzeiten verwendet. Bei Gegenüberstellung von Abbildung 5.10 und Abbildung 6.8 ist zu erkennen, dass sich das Verhalten der k-Proben-Heuristiken in beiden Fallstudien kaum unterscheidet.

Wie in Abbildung 6.6 und Abbildung 6.8 zu sehen ist, finden im gegebenen Suchraum sowohl der Userspace-Autotuner im Nelder-Mead-Modus als auch die Zufallsproben nach nur wenigen Durchläufen sehr gute Konfigurationen.

Im Nelder-Mead-Modus werden für dreißig Durchläufe 31-84% mehr Zeit benötigt als mit der Referenzkonfiguration. Bei einer Probenanzahl von 2-16 benötigen die k-Proben-Heuristiken 27-52% mehr Zeit als die Referenzkonfiguration. Der Startwert mit circa $900\mu s$ ist wieder deutlich langsamer als der Erwartungswert $E_{best}(1) = 438.51\mu s$ einer einzelnen Probe und zieht den Durchschnittswert für das Nelder-Mead-Vorgehen nach oben.

Abbildung 6.9 verdeutlicht den Effekt, dass sich bei k-Proben-Heuristiken mit langer Probephase der Erwartungswert der durchschnittlichen Laufzeit an die Laufzeit der optimalen Konfiguration annähert.



Testsystem	**System B**
Anzahl Kerne	4
cpuburn-Instanzen	0
Betriebssystem	Ubuntu 12.04 64bit
Go-Version	1.2
VM-Hypervisor	VirtualBox 4.2
VM-Host	System mit 4 Kernen

Abbildung 6.4: Messstellen des Userspace-Autotuners im Nelder-Mead-Modus im Suchraum beim parallelen Datenbankzugriff. Das Nelder-Mead-Verfahren arbeitet mit einem Simplex, d.h. im zweidimensionalen Fall mit einem Dreieck, und Punkte sollten weniger als Pfad in Richtung der Lösung, sondern mehr als Indikation für das betrachtete Gebiet gesehen werden.

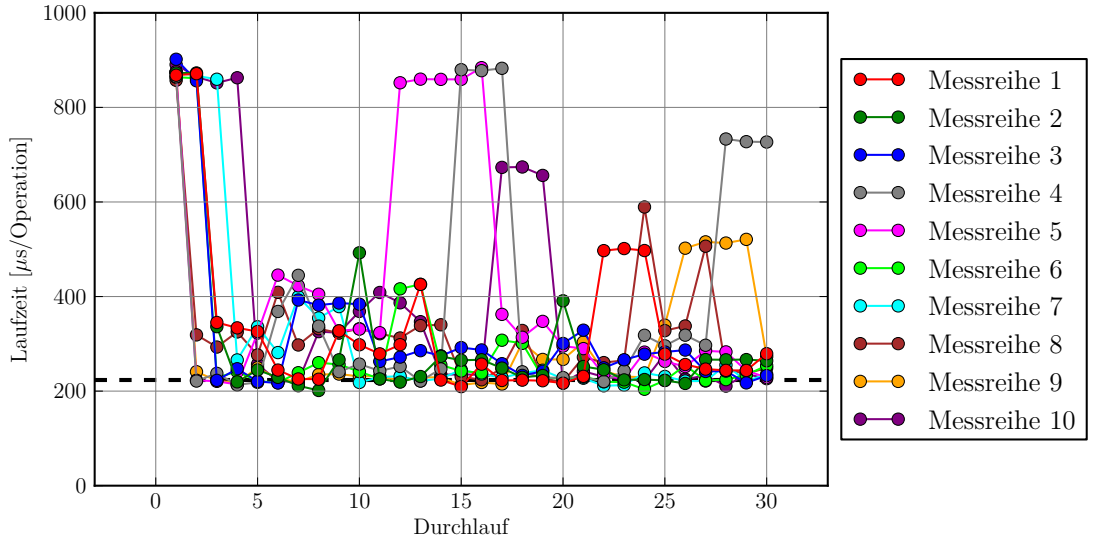


Abbildung 6.5: Laufzeit der jeweils gemessenen Konfigurationbeim parallelen Datenbankzugriff. Die Konfigurationen wurden von Userspace-Autotuner im Nelder-Mead-Modus bestimmt. In allen Messreihen wurden auch in späteren Durchläufen teilweise langsame Konfigurationen aktiviert. Die gestrichelte Linie markiert die Laufzeit der schnellsten Konfiguration.

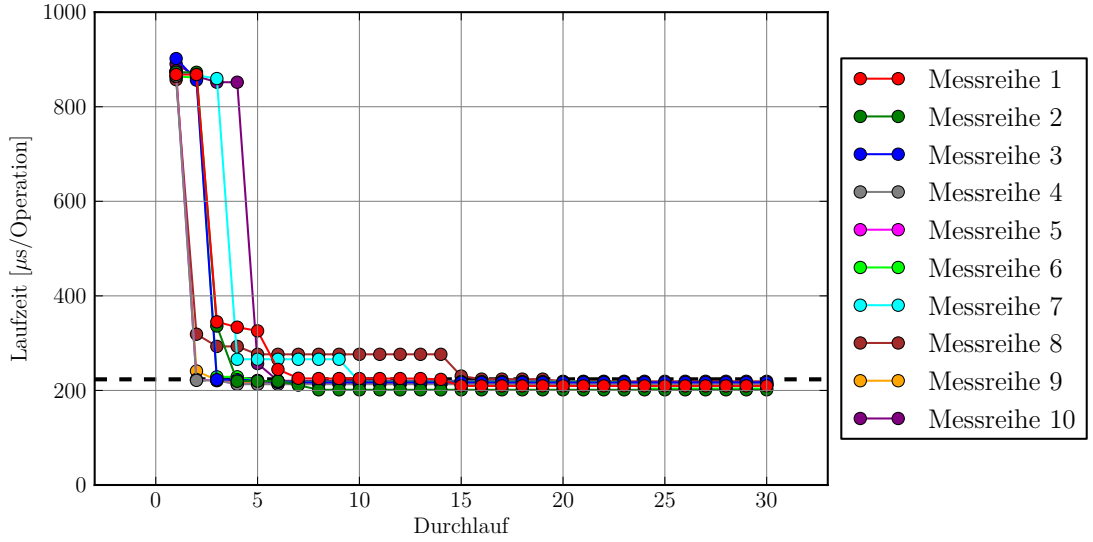


Abbildung 6.6: Laufzeit der schnellsten bis dahin gemessenen Konfiguration parallelen Datenbankzugriff. Nach spätestens 20 Durchläufen haben alle Messreihen eine Messung schneller als die Referenzzeit durchgeführt.

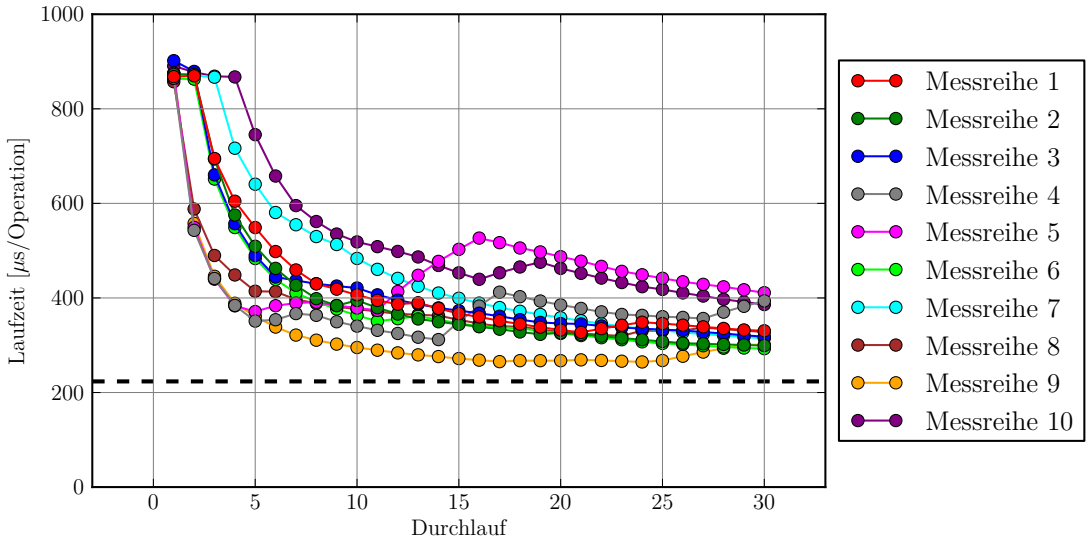


Abbildung 6.7: Durchschnittliche Laufzeit der bis dahin gemessenen Konfigurationen beim parallelen Datenbankzugriff. Diese bleibt in allen Messreihen nach dreißig Durchläufen deutlich über der Referenzzeit. Es wird 31-84% mehr Zeit benötigt als mit der Referenzkonfiguration.

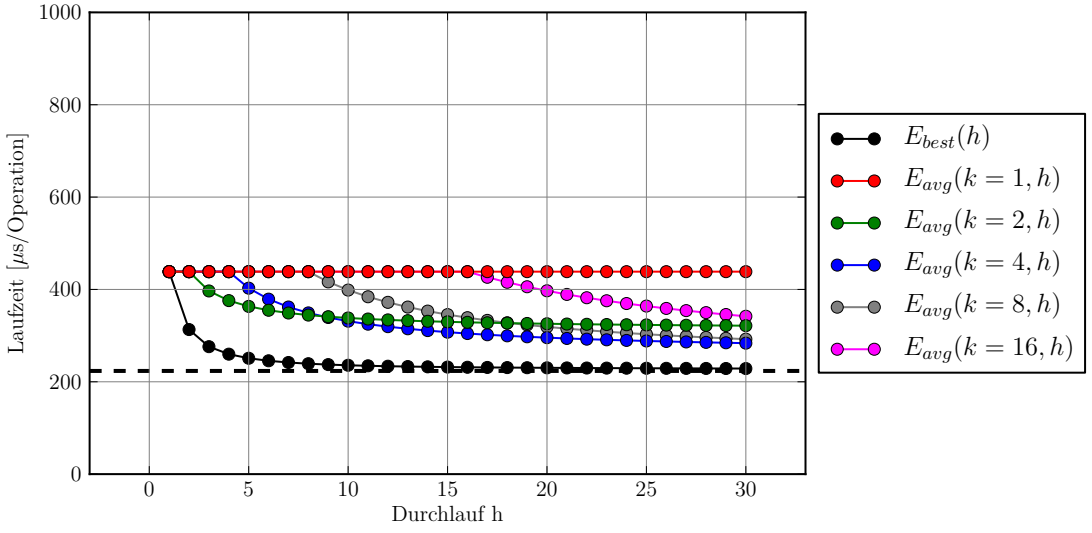


Abbildung 6.8: Erwartete Laufzeit der Konfigurationen beim parallelen Datenbankzugriff bei Verwendung der k-Proben-Heuristik. Der Erwartungswert $E_{best}(k)$ für die beste gefundene Konfiguration fällt mit steigendem k schnell ab. Die durchschnittliche Laufzeit einer k-Proben-Heuristik nähert sich nach Ablauf der zufälligen Probephase langsam der gefundenen schnellsten Laufzeit an.

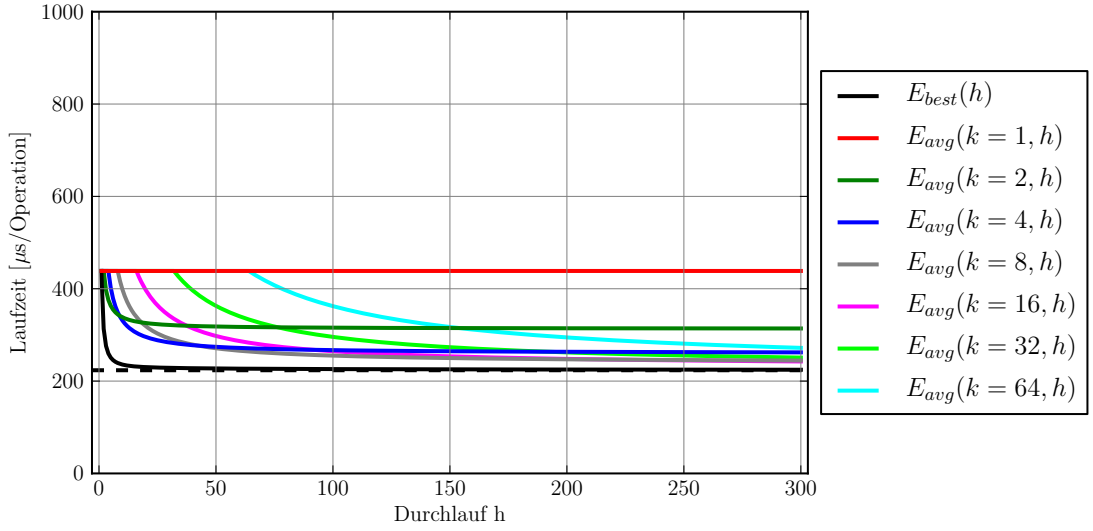


Abbildung 6.9: Erwartete Laufzeit der Konfigurationen beim parallelen Datenbankzugriff bei Verwendung der k-Proben-Heuristik. Die Abbildung zeigt längere Messreihen als Abbildung 6.8, sodass sich weitere Stichproben amortisieren.

6.5 Zusammenfassung

In dieser Fallstudie wurde erneut Autotuning an einem Go-Programm demonstriert. Die Besonderheit dabei ist, dass dessen Laufzeit stark von einem externen Programm beeinflusst wird, über das der Autotuner keine Kontrolle hat. Das Prinzip der automatischen Leistungsverbesserung funktioniert dennoch; aus Sicht des Autotuners ist die Datenbank Teil der Rahmenbedingungen, ähnlich wie die Hardware. In dem resultierenden Suchraum, dessen Optimum nicht durch bekannte Heuristiken oder einfache Überlegungen abgeschätzt werden kann, findet Autotuning schnell gute Konfigurationen.

Die Messungen schwanken in diesem Suchraum vergleichsweise stark, sodass effektives Autotuning das Rauschen mit in seine Entscheidungen einbeziehen sollte. Autotuner sollten weitere Optimierungsziele neben der Laufzeit unterstützen. Das Öffnen weiterer Datenbankverbindungen sollte vermieden werden, auch wenn diese sich noch nicht negativ auf die Geschwindigkeit auswirken.

Der Vergleich mit den k-Proben-Heuristiken zeigt erneut, dass die Wahl der Startkonfiguration einen großen Einfluss auf die Dauer des Tuningvorgangs hat.

7. Implementierung

Dieses Kapitel beschreibt den im Rahmen dieser Diplomarbeit geänderten und neu erstellten Code. Dies umfasst insbesondere die Anbindung des Autotuners an Go, den `GoFullWalker` zur Suchraumerkundung in Go, sowie die Werkzeuge zur Evaluation. Die Änderungen an den Anwendungsprogrammen im Rahmen der Fallstudien sind innerhalb der jeweiligen Fallstudie beschrieben.

7.1 Go-Schnittstelle für Autotuner

Autotuner in Go müssen die in Abschnitt 4.3 beschriebene Schnittstelle erfüllen.

```
1 type AutoTunerPointer interface {
2   AddParameter(where *int, desc string,
       min int, max int, step int)
3   GetParameters() []Parameter
4   Tic()
5   Toc()
6   Reconfigure()
7   SetOnReconfigure(callback func())
8   Destroy()
9 }
```

Der `Parameter`-Datentyp umfasst die folgenden Felder.

```
1 type Parameter struct {
2   where *int
3   min, max, step int
4   description string
5 }
```

7.2 Autotuner in Go

Dieser Abschnitt beschreibt drei verschiedene Ansätze, in Go einen Autotuner zu verwenden. Die ersten beiden behandeln die Anbindung des Userspace-Autotuners, der dritte Ansatz ist der rein in Go geschriebenen `GoFullWalker`.

7.2.1 SWIG

SWIG (Simplified Wrapper and Interface Generator) [Bea] ist ein Programmierwerkzeug, mit dessen Hilfe C und C++ Code aus anderen Programmiersprachen heraus aufgerufen werden kann. SWIG autogeneriert dazu Codefragmente („Wrapper“) der jeweiligen Zielsprache. Diese rufen die bestehende externe Funktionalität in C oder C++ auf und bieten eine Schnittstelle, die den Techniken und Standards der Zielsprache entsprechen.

SWIG unterstützt Go, aber der Versuch, den in C++ geschriebenen Userspace-Autotuner aus dem Go-Code heraus anzusprechen, war zunächst nicht von Erfolg gekrönt. Die Fehlersuche gestaltete sich aufgrund der spärlichen Dokumentation schwierig. Da sich überdies schnell abgezeichnet hat, dass die C-Anbindung gut funktioniert, wurde dieser Ansatz nicht weiter verfolgt.

7.2.2 C-Import

Aufgrund von vorherigem Autotuning von C-Code durch Braun [Bra12], existiert im Userspace-Autotuner bereits eine Anbindung an puren C-Code. Da Go nativ eine Anbindung an C mitliefert, kann diese zur indirekten Anbindung des Userspace-Autotuners an Go genutzt werden.

In Go ist, wie in Abschnitt 2.1.5.2 beschrieben, ein besonderer `import` der C-Bibliothek erforderlich, um C-Code aufrufen zu können. Insbesondere müssen die Suchpfade für Compiler und Linker gesetzt werden.

```
1 // #cgo LDFLAGS: -lctuner -L/home/mn/da/uat/Client/ -lrt
2 // #cgo CFLAGS: -I.
3 // #include "/home/mn/da/uat/Client/tunerWrapper.h"
4 import "C"
```

Mit dieser Präambel kann der Autotuner aus Go heraus angesprochen werden, allerdings sind die importierten Aufrufe nur im lokalen Paket sichtbar. Das gilt sogar für die primitiven Datentypen wie `C.int`. Daher ist ein Wrapper erforderlich, der die Aufrufe anderen Paketen zur Verfügung stellt. Dieser Wrapper sollte auch das Umwandeln von Go- in C-Datentypen und umgekehrt übernehmen.

Ein Beispiel für einen solchen Wrapper ist der Wrapper um `AddParameter`.

```
1  func (t *CTuner) AddParameter(where *int, desc string,
       min int, max int, step int, array []int) {...
2    newp := Parameter{...}
3    where_c := (*C.int)(unsafe.Pointer(newp.where))
4    min_c := C.int(newp.min)
5    max_c := C.int(newp.max)
6    step_c := C.int(newp.step)
7
8    C.ctuner_register_parameter(
       t.pointer_c, where_c, min_c, max_c, step_c)
9    C.ctuner_register_section_parameter(t.pointer_c, 1, where_c)
10 }
```

In Zeile 1 erwartet der Wrapper beim Aufruf Go-Datentypen. In den Zeilen 3 bis 6 findet die Umwandlung in C-Datentypen statt. In den Zeilen 8 und 9 findet der Aufruf der C-Methoden mit C-Datentypen statt.

Entsprechende Wrapper existieren auch für die anderen Autotuner-Methoden.

7.2.3 Go-Autotuner

Für einfache Experimente und Plausibilitätsprüfungen bietet sich ein komplett in Go geschriebener Autotuner an. Der `GoFullWalker` ist sowohl im Umfang als auch von den gebotenen Funktionen minimal. Er stellt als Suchstrategie nur die erschöpfende Suche zur Verfügung. Der Code des `GoFullWalker` umfasst circa 60 Zeilen.

Die Schnittstelle des `GoFullWalker` ist kompatibel mit dem Wrapper um den Userspace-Autotuner, sodass beide Implementierungen einfach gegeneinander ausgetauscht werden können.

Als permanente Daten werden gespeichert:

- ein Slice mit den verwalteten Parametern
- der Zeitpunkt, zu dem der letzte Messabschnitt betreten wurde
- die Methode, die bei einer Rekonfiguration aufgerufen wird

```
1 type GoFullWalker struct {
2   Parameters []Parameter
3   TicTime time.Time
4   OnReconfigure func()
5 }
```

Alle Methoden der Autotuner-Schnittstelle aus Abschnitt 7.1 werden erfüllt.

Im Konstruktor werden die Datenfelder initialisiert.

```
1 func CreateGoFullWalker() AutoTunerPointer {
2   fmt.Println("creating GoFullWalker")
3   t := GoFullWalker{}
4   t.Parameters = make([]Parameter, 0, MAXPARAMS)
5   t.OnReconfigure = func() {}
6   return &t
7 }
```

Die `Destroy`-Methode dient der Erfüllung der Schnittstelle, bietet darüber hinaus aber keine wirkliche Funktionalität.

```
1 func (t *GoFullWalker) Destroy() {
2   fmt.Println("destroying GoFullWalker")
3 }
```

Beim Hinzufügen eines Parameters mit der `AddParameter`-Methode werden die Informationen in die internen Datenstrukturen kopiert.

```
1 func (t *GoFullWalker) AddParameter(where *int, desc string,
        min int, max int, step int) {
2   t.Parameters = t.Parameters[0:len(t.Parameters)+1]
3   newp := Parameter{}
4   newp.where = where
5   newp.min = min
6   newp.max = max
7   newp.step = step
8   newp.description = desc
9   t.Parameters[len(t.Parameters)-1] = newp
10  fmt.Printf("adding parameter %s with values from %d..%d in
       steps of %d\n", desc, min, max, step)
11 }
```

Die `GetParameters`-Methode liefert die interne Datenstruktur zurück, in der die Parameter gespeichert sind.

```
1 func (t *GoFullWalker) GetParameters() []Parameter {
2   return t.Parameters
3 }
```

Die `Tic`-Methode protokolliert den Zeitpunkt, zu dem der Messabschnitt betreten wird.

```
1 func (t *GoFullWalker) Tic() {
2   t.TicTime = time.Now()
3 }
```

Die `Toc`-Methode bestimmt die Zeitdifferenz zum Aufruf der `Tic`-Methode, skaliert sie entsprechend der geleisteten Arbeit und gibt sie in einem standardisierten Format aus.

```
1 func (t *GoFullWalker) Toc(work int) {
2   dur := time.Now().Sub(t.TicTime).Seconds() / float64(work)
3   tictoc.PrintTocMachine(ToString(t), dur)
4 }
```

Die `Reconfigure`-Methode wählt die nächste Konfiguration indem sie – ähnlich wie beim Zählen in einem Stellenwertsystem – in Zeile 6 den Parameter mit dem niedrigsten Index um einen Schritt erhöht und bei einem Überlauf in Zeile 8ff zum nächsthöheren Index fortschreitet.

Außerdem ruft `Reconfigure` den `OnReconfigure`-Callback auf.

```
1  func (t *GoFullWalker) Reconfigure() {
2    p := t.Parameters
3    i := 0
4    for i < len(p) {
5      if *p[i].where < p[i].max {
6        *p[i].where += p[i].step
7        break
8      } else {
9        *p[i].where = p[i].min
10       i += 1
11       if i > len(p) {
12         break
13       }
14     }
15   }
16   t.OnReconfigure()
17 }
```

Die `SetOnReconfigure`-Methode hinterlegt die Rekonfigurationsmethode. Diese wird bei jeder Rekonfiguration aufgerufen.

```
1 func (t *GoFullWalker) SetOnReconfigure(callback func()) {
2   t.OnReconfigure = callback
3 }
```

7.3 Visualisierung der Messdaten

Die `ToString`-Methode für Autotuner dient der Ausgabe der Konfiguration in einem Format, das zusammen mit der gestoppten Zeit mittels `numpy` und `matplotlib` [JOP+01] visualisiert werden kann.

```
func ToString(t AutoTunerPointer) string {
  s := ""
  for _, p := range t.GetParameters() {
    s += fmt.Sprintf("%s, %6d, ", p.description, *p.where)
  }
  return s
}
```

Das Python-Skript zur Erstellung der Visualisierungen gliedert sich in drei Abschnitte:

- Einlesen der Daten
- Konvertieren der Daten
- Aufruf der Plotting-Bibliothek

7.3.1 Einlesen der Daten

Die Klasse `DictReader` aus der Standardbibliothek erlaubt das Einlesen einer CSV-Datei.

```
r = csv.DictReader(open(sys.argv[1]), delimiter=',')
i2n = r.fieldnames

x,y,z = [],[],[]
for line in r:
    x.append(float(line[i2n[0]]))
    y.append(float(line[i2n[1]]))
    z.append(float(line[i2n[2]]))
```

7.3.2 Konvertieren der Daten

numpy-Felder erlauben die effiziente Manipulation der Daten über Matrixoperationen.

```
ux = numpy.unique(x)
uy = numpy.unique(y)

X, Y = numpy.meshgrid(ux, uy)
zz = numpy.array(z)

# reshape to grid with n measurements for every grid point
# Fortran order is great, because the first index varies fastest
# i.e. we rotate x, then y, and finally we repeat measurements
zz_folded = zz.reshape(len(ux), len(uy),-1, order='F')

# calculate medians along the "repeated measurements axis"
zz_medians = numpy.median(zz_folded, axis=2)
# transpose for plotting
zz_medians = numpy.transpose(zz_medians)
```

In den Zeilen 1 und 2 werden die Gitterpunkte bestimmt. In Zeile 4 wird das Gitter erzeugt. Zeile 10 fasst die Messwerte der wiederholten Messungen an einem Gitterpunkt zusammen. In den Zeilen 13 und 15 werden die Mediane der Messungen bestimmt.

7.3.3 Aufruf der Plotting-Bibliothek

`matplotlib` wird zur Generierung der Graphen verwendet.

```
1 figure = plot.figure()
2 subplot = figure.add_subplot(111, projection='3d')
3 subplot.scatter(x,y,z, c="k", marker="o", s=1, antialiased=False)
```

In Zeile 3 wird mit dem Befehl `scatter` die Punktwolke erstellt.

`plot_surface` legt die Fläche durch die Mediane.

```
1  subplot.plot_surface(
2      X,
3      Y,
4      zz_medians,
5      rstride=1,
6      cstride=1,
7      cmap=cm.coolwarm,
8      linewidth=0.1,
9      antialiased=True,
10     alpha=1.0
11 )
```

Die folgenden Aufrufe sorgen für die korrekte Beschriftung und Ausrichtung der Achsen.

```
1 subplot.set_xlabel(r.fieldnames[0])
2 subplot.set_ylabel(r.fieldnames[1])
3 subplot.set_zlabel(r.fieldnames[2])
4
5 subplot.view_init(elev=65.0, azim=60.0)
```

Ein Beispiel für eine Visualisierung ist in Abbildung 2.1 zu sehen.

8. Fazit und Ausblick

Dieses Kapitel zieht ein kurzes Fazit und zeigt Ansätze für weitere Forschungsmöglichkeiten im Bezug auf Autotuning im Umfeld von Google Go.

8.1 Fazit

Moderne Soft- und Hardware werden immer komplexer, und es wird immer schwieriger, als Anwender oder Entwickler aus einer gegebenen Kombination von Hard- und Software die optimale Leistung herauszuholen. Eine große Anzahl Einstellungsmöglichkeiten existiert und nur bestimmte Konfigurationen führen zu guten Ergebnissen. Es wird daher immer wichtiger, Möglichkeiten zu finden, diese guten Konfigurationen automatisiert zu finden – und Autotuning ist eine Möglichkeit dazu.

Im Umfeld von Go bestehen viele Möglichkeiten zu Autotuning, denn gerade parallele Programme profitieren besonders von automatischem Tuning. In diesen muss oftmals eine Vielzahl wenig intuitiver Parameter gesetzt werden. Es ist möglich, einen bestehenden Autotuner an Go anzubinden. Der Einsatz eines universellen Autotuners ermöglicht Autotuning in beliebigen Go-Programmen. Die in den Fallstudien beschriebenen Suchräume bestätigen teilweise bewährte Heuristiken, wie `Anzahl Threads := Anzahl Rechenkerne`, zeigen aber auch Potential für weitere Beschleunigung. Insbesondere Onlinetuning hat hier Potential, da es Zugriff auf den tatsächlichen Suchraum zur Laufzeit hat.

Andererseits sind bestehende Systeme noch nicht in allen Situationen besonders effizient und schlagen bewährte Heuristiken oder die zufällige Suche – wenn überhaupt – nur knapp.

8.2 Ausblick

Der Userspace-Autotuner verwendet die normierte Laufzeit als alleiniges Kriterium zur Bewertung der Güte einer Konfiguration. Oftmals ist jedoch eine Kombination mehrerer

Kriterien ausschlaggebend. Bei Packprogrammen kann hohe Geschwindigkeit beispielsweise mit schlechter Komprimierung erkauft werden – eine Optimierung, die nicht unbedingt gewünscht ist.

Der Userspace-Autotuner misst jede Konfiguration nur ein einziges Mal, obwohl die Messungen teilweise mit starken Schwankungen belegt sind. Der Autotuner sollte mehrfach messen, um die Konfidenz zu erhöhen.

Die Heuristiken, ob und wann rekonfiguriert werden soll, und mit welchem Zweck (Exploration des Suchraums, lokale Optimierung, Aktivieren einer besseren Konfiguration) sollten genauer untersucht werden, um unüberwachtes Tuning im Produktivbetrieb zu ermöglichen. Das Rundungsverhalten im Zusammenhang mit diskreten Suchräumen der Userspace-Autotuner-Variante des Nelder-Mead-Verfahrens sollte auf Korrektheit untersucht werden.

Die Bekanntheit und Akzeptanz von Onlinetuning könnte weiter erhöht werden, z.B. indem bekannte Serverprogramme autotuningfähig gemacht werden, um so in Rechenzentren bzw. in der Cloud eine Kostenreduktion zu erreichen.

9. Literaturverzeichnis

[ABC+06] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams und Katherine A. Yelick: *The landscape of parallel computing research: a view from Berkeley.* Technischer Bericht UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf`.

[ALSU07] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 2. Auflage, 2007.

[Aut] *Automatically Tuned Linear Algebra Soft. | Free software downloads at SourceForge.net.* `http://sourceforge.net/projects/math-atlas`, besucht: 08.01.2014.

[BACD97] Jeff Bilmes, Krste Asanovic, Chee Whye Chin und Jim Demmel: *Optimizing Matrix Multiply Using PHiPAC: A Portable, High-performance, ANSI C Coding Methodology.* In: *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, Seiten 340–347, New York, NY, USA, 1997. ACM, ISBN 0-89791-902-5. `http://doi.acm.org/10.1145/263580.263662`.

[BAD+96] J. Bilmes, K. Asanović, J. Demmel, D. Lam und C.W. Chin: *PHiPAC: A Portable, High-Performance, ANSI C Coding Methodology and its application to Matrix Multiply.* LAPACK Working Note 111, University of Tennessee, 1996.

[BAV+] Jeff A Bilmes, Krste Asanovic, Rich Vuduc, Sriram Iyer, Jim Demmel, CheeWhye Chin und Dominic Lam: *PHiPAC Fast Matrix Multiply Home Page.* `http://www1.icsi.berkeley.edu/~bilmes/phipac`, besucht: 10.10.2013.

[Bea] David M. Beazley: *SWIG : An Easy to Use Tool For Integrating Scriptint Languages with C and C++*. http://www.swig.org/papers/Tcl96/tcl96.html, besucht: 28.10.2013.

[ben] *Which programs are fastest? | Computer Language Benchmarks Game*. http://benchmarksgame.alioth.debian.org/u32q/benchmark.php?test=all&lang=all&lang2=gcc&data=u32q, besucht: 18.10.2013.

[big] *big - The Go Programming Language*. http://golang.org/pkg/math/big/, besucht: 23.03.2014.

[Bon12] Sebastian Bonn: *Autotuning von MySQL*. Diplomarbeit, IPD Tichy, Karlsruhe Institute of Technology, Germany, 2012.

[Bra12] Markus Braun: *Kernelfreies Autotuning*. Diplomarbeit, IPD Tichy, Karlsruhe Institute of Technology, Germany, 2012.

[buf] *bufio - The Go Programming Language*. http://golang.org/pkg/bufio/, besucht: 23.03.2014.

[Bur] *Ubuntu Manpage: cpuburn, burnBX, burnK6, burnK7, burnMMX, burnP5, burnP6 - a collection*. http://manpages.ubuntu.com/manpages/precise/man1/cpuburn.1.html, besucht: 15.01.2014.

[cgo] *cgo - The Go Programming Language*. http://golang.org/cmd/cgo, besucht: 12.11.2013.

[dis] *disintegration/imaging · GitHub*. https://github.com/disintegration/imaging, besucht: 03.03.2014).

[Dox12] Caleb Doxsey: *An Introduction to Programming in Go*. CreateSpace Independent Publishing Platform, 2012, ISBN 978-1478355823. http://www.golang-book.com/.

[eff] *Effective Go - The Go Programming Language*. http://golang.org/doc/effective_go.html, besucht: 07.11.2013.

[Elb] Egon Elbre: *egonelbre/goqueuestest*. https://github.com/egonelbre/goqueuestest, besucht: 23.03.2014.

[FB11] Rainer Feike und Steffen Blass: *Programmierung in Google Go*. Open source library. Addison-Wesley, München [u.a.], 2011, ISBN 978-3-8273-3009-3.

[ffta] *FFTW Home Page*. http://www.fftw.org, besucht: 10.10.2013.

[FFTb] *FFTW/fftw3*. https://github.com/FFTW/fftw3, besucht: 10.10.2013.

[FJ97] Matteo Frigo und Steven G. Johnson: *The Fastest Fourier Transform in the West. MIT-LCS-TR-728*. In: *the Proceedings of the 1998 International Conference on Acoustics, Speech, and Signal Processing, ICASSP '98*, 1997.

[FJ98] Matteo Frigo und Steven G. Johnson: *FFTW: An adaptive software architecture for the FFT*. In: *Proc. 1998 IEEE Intl. Conf. Acoustics Speech and Signal Processing*, Band 3, Seiten 1381–1384. IEEE, 1998.

[Fol] Jared Folkins: *How To: Shove data into Postgres using Goroutines(Gophers) and GoLang | acloudtree*. `http://www.acloudtree.com/2013/10/29/`, besucht: 03.03.2014.

[Ger] Andrew Gerrand: *Defer, Panic, and Recover - The Go Blog*. `http://blog.golang.org/defer-panic-and-recover`, besucht: 2014-03-02.

[go1a] *Go 1 and the Future of Go Programs - The Go Programming Language*. `http://golang.org/doc/go1compat`, besucht: 19.12.2013.

[go1b] *Go 1.1 Release Notes - The Go Programming Language*. `http://golang.org/doc/go1.1#performance`, besucht: 12.03.2014.

[go1c] *Go 1.2 Release Notes - The Go Programming Language*. `http://golang.org/doc/go1.2#preemption`, besucht: 22.03.2014.

[GoC] *How to Write Go Code - The Go Programming Language*. `http://golang.org/doc/code.html`, besucht: 17.03.2014.

[gof] *Frequently Asked Questions (FAQ) - The Go Programming Language*. `http://golang.org/doc/faq`, besucht: 07.11.2013.

[gola] *testing - The Go Programming Language*. `http://golang.org/pkg/testing`, besucht: 10.10.2013.

[golb] *The Go Programming Language*. `http://golang.org`, besucht: 07.11.2013.

[goma] *go-sql-driver/mysql · GitHub*. `https://github.com/go-sql-driver/mysql/`, besucht: 03.03.2014.

[gomb] *runtime - The Go Programming Language*. `http://golang.org/pkg/runtime/`, besucht: 12.11.2013.

[gos] *sql - The Go Programming Language*. `http://golang.org/pkg/database/sql/`, besucht: 03.03.2014.

[GoT] *A Tour of Go*. `http://tour.golang.org/#1`, besucht: 17.03.2014.

[GPT+] Robert Griesemer, Rob Pike, Ken Thompson, Ian Taylor, Russ Cox, Jini Kim und Adam Langley: *Hey! Ho! Let's Go!* `http://google-opensource.blogspot.de/2009/11/hey-ho-lets-go.html`, besucht: 18.10.2013.

[Ins] *Insieme Compiler Project*. `http://www.dps.uibk.ac.at/insieme`, besucht: 22.01.2014.

[Int] *Intel® Fortran Compiler XE 13.1 User and Reference Guides.* http://software.intel.com/sites/products/documentation/doclib/stdxe/2013/composerxe/compiler/fortran-win/index.htm#GUID-9C4692D9-BBDE-4F36-95B1-58295EAFA5B5.htm, besucht: 15.01.2014.

[JOP+01] Eric Jones, Travis Oliphant, Pearu Peterson *et al.*: *SciPy: Open source scientific tools for Python*, 2001. http://www.scipy.org.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid und F. T. Krogh: *Basic Linear Algebra Subprograms for Fortran Usage.* ACM Trans. Math. Softw., 5(3):308–323, September 1979, ISSN 0098-3500. http://doi.acm.org/10.1145/355841.355847.

[lib] *lib/pq · GitHub.* https://github.com/lib/pq, besucht: 03.03.2014.

[Mag] *Magische Zahl (Informatik) – Wikipedia.* http://de.wikipedia.org/wiki/Magische_Zahl_(Informatik)#Magische_Zahlen_in_Code, besucht: 16.03.2014.

[Maj] Marek Majkowski: *How to resolve a million domains — Idea of the day.* https://idea.popcount.org/2013-11-28-how-to-resolve-a-million-domains/, besucht: 23.03.2014.

[MATa] *Interface to FFTW library run-time algorithm tuning control - MATLAB fftw - MathWorks Deutschland.* http://www.mathworks.de/de/help/matlab/ref/fftw.html, besucht: 10.10.2013.

[matb] *mattn/go-sqlite3 · GitHub.* https://github.com/mattn/go-sqlite3, besucht: 03.03.2014.

[MMML04] Anna Morajko, Oleg Morajko, Tomàs Margalef und Emilio Luque: *MATE: Dynamic Performance Tuning Environment.* In: Marco Danelutto, Marco Vanneschi und Domenico Laforenza (Herausgeber): *Euro-Par 2004 Parallel Processing*, Band 3149 der Reihe *Lecture Notes in Computer Science*, Seite 98–107. Springer Berlin Heidelberg, 2004, ISBN 978-3-540-22924-7. http://dx.doi.org/10.1007/978-3-540-27866-5_13.

[MyS] *MySQL :: The world's most popular open source database.* http://www.mysql.com/, besucht: 03.03.2014.

[nfn] *nfnt/resize.* https://github.com/nfnt/resize, besucht: 28.10.2013.

[NM65] John A Nelder und Roger Mead: *A simplex method for function minimization.* Computer Journal, 7(4):308–313, 1965.

[Opt] *Optimize Options - Using the GNU Compiler Collection (GCC).* http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, besucht: 15.01.2014.

[Ott13] Frank Otto: *Objektorientierte Stromprogrammierung.* Dissertation, IPD Tichy, Karlsruhe Institute of Technology, Germany, 2013.

[Pik] Rob Pike: *Go's Declaration Syntax - The Go Blog.* http://blog.golang.org/gos-declaration-syntax, besucht: 07.11.2013.

[Pos] *PostgreSQL: The world's most advanced open source database.* http://www.postgresql.org/, besucht: 03.03.2014.

[Pow] Bobby Powers: *Optimizing Real World Go.* http://bpowers.github.io/weblog/2013/01/05/optimizing-real-world-go, besucht: 10.10.2013.

[pvm] *PVM: Parallel Virtual Machine.* http://www.csm.ornl.gov/pvm, besucht: 08.01.2014.

[reg] *regexp - The Go Programming Language.* http://golang.org/pkg/regexp/, besucht: 23.03.2014.

[Sch09] Sascha Schwedes: *Betriebssystemintegration eines automatischen Performanzoptimierers für parallele Anwendungen.* Diplomarbeit, IPD Tichy, Karlsruhe Institute of Technology, Germany, 2009.

[Sch10] Christoph A. Schaefer: *Automatisierte Performanzoptimierung Paralleler Architekturen.* Dissertation, IPD Tichy, Karlsruhe Institute of Technology, Germany, 2010.

[sor] *src/pkg/sort/sort.go - The Go Programming Language.* http://golang.org/src/pkg/sort/sort.go, besucht: 23.03.2014.

[SPT09] Christoph A. Schaefer, Victor Pankratius und Walter F. Tichy: *Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications.* In: Henk Sips, Dick Epema und Hai Xiang Lin (Herausgeber): *Euro-Par 2009 Parallel Processing*, Band 5704 der Reihe *Lecture Notes in Computer Science*, Seiten 9–20. Springer Berlin Heidelberg, 2009, ISBN 978-3-642-03868-6. http://link.springer.com/chapter/10.1007/978-3-642-03869-3_5.

[SQL] *SQLite Home Page.* http://www.sqlite.org/, besucht: 03.03.2014.

[Ste] Don Stewart: *ghc-gc-tune: Tuning Haskell GC settings for fun and profit | Control.Monad.Writer.* http://wp.me/pqcsb-9k, besucht: 26.03.2014.

[suf] *suffixarray - The Go Programming Language.* http://golang.org/pkg/index/suffixarray/, besucht: 23.03.2014.

[Sum12] Mark Summerfield: *Programming in Go : creating applications for the 21st century.* Developers library. Addison-Wesley, Upper Saddle River, NJ, 2012, ISBN 978-0-321-77463-7; 0-321-77463-9.

[TCH02] Cristian Tapus, I Hsin Chung und Jeffrey K. Hollingsworth: *Active harmony: towards automated performance tuning.* In: Roscoe C. Giles, Daniel A. Reed und Kathryn Kelley (Herausgeber): *SC*, Seiten 1–11. ACM, 2002. `http://dl.acm.org/citation.cfm?id=762771`.

[Tho13] Peter Thoman: *Insieme-RS A Compiler-supported Parallel Runtime System.* Dissertation, Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck, 2013.

[tic] *Start stopwatch timer - MATLAB tic - MathWorks Deutschland.* `http://www.mathworks.de/de/help/matlab/ref/tic.html`, besucht: 19.12.2013.

[VDY05] Richard Vuduc, James W. Demmel und Katherine A. Yelick: *OSKI: A library of automatically tuned sparse matrix kernels.* In: *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, Juni 2005. Institute of Physics Publishing. `http://bebop.cs.berkeley.edu/pubs/jop2005-SciDAC-OSKI.pdf`.

[Vyu] Dmitry Vyukov: *Goroutines vs OS threads - Google Groups.* `https://groups.google.com/forum/#!msg/golang-nuts/j51G7ieoKh4/aDoTNvIpXhQJ`, besucht: 18.03.2014.

[WD98] R. Clint Whaley und Jack J. Dongarra: *Automatically Tuned Linear Algebra Software.* In: *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, Supercomputing '98, Seiten 1–27, Washington, DC, USA, 1998. IEEE Computer Society, ISBN 0-89791-984-X. `http://dl.acm.org/citation.cfm?id=509058.509096`.