



Master-Thesis

Name: Andreas Schwarz

Thema: STM-basierte Futures und Promises in Scala

Arbeitsplatz: Hochschule Karlsruhe, Karlsruhe

Referent: Prof. Dr. Sulzmann

Korreferent: Prof. Dr. Körner

Abgabetermin: 29.02.2016

Karlsruhe, 01.09.2015

Der Vorsitzende des
Prüfungsausschusses

Prof. Dr. Ditzinger

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die hier vorgelegte Masterthesis selbstständig und ausschließlich unter Verwendung der angegebenen Literatur und sonstigen Hilfsmittel verfasst habe. Die Arbeit wurde in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt.

Karlsruhe, den 25.02.2016

.....
(Andreas Schwarz)

Kurzfassung

Das Ziel der vorliegenden Arbeit ist die Implementierung von Futures und Promises in Scala. Hierzu wird ausschließlich das funktionale Programmierparadigma eingesetzt. Die Verwaltung des Resultats eines Futures obliegt dabei einem STM. Dies vereinfacht die Realisierung des blockenden Zugriffs auf das Resultat und erlaubt die Implementierung komplexer Funktionen, die auf eine Menge von Futures oder Promises angewendet werden können. STM garantiert dabei die korrekte Ausführung der Operation.

Da Scala eine Standardimplementierung der beiden Abstraktionen besitzt, wird die eigene Umsetzung mit dieser verglichen. Eine Gegenüberstellung der Lines of Code zeigt, dass die eigene Realisierung etwa die Hälfte an Codezeilen benötigt, um dieselbe Funktionalität bereitzustellen. Des Weiteren werden Benchmarks ausgeführt, um die Performance der Implementierungen zu vergleichen. Während das sequentielle und nebenläufige Lesen eines Resultats in etwa gleich schnell ist, ist die Performance und Skalierung der eigenen Implementierung bei der Verschachtelung mehrerer Futures deutlich schlechter.

Nach der Analyse der Laufzeitumgebung konnten die Schwachstellen der eigenen Implementierung identifiziert werden. Ein exzessives Starten und sofortiges Pausieren von Threads sorgt für enormen Overhead. Dies beeinflusst die Laufzeiten maßgeblich. Basierend auf den Ergebnissen der Analyse wurde die Implementierung optimiert. Bei der erneuten Ausführung der Benchmarks zeigt sich, dass die Leistungsdefizite behoben werden konnten und die eigene Implementierung dieselben Performancecharakteristiken wie das native Gegenstück aufweist.

Abstract

The goal of this thesis is the implementation of Futures and Promises in Scala. The realization uses exclusively the functional programming paradigm. STM is responsible for the management of the result of Futures. This simplifies the implementation of the blocking function that access the result and allows the implementation of more complex functions, which operate on a set of Futures or Promises. STM guarantees the correct execution of these functions.

Scala itself has a default implementation of both abstractions. Therefore, a comparison takes place. The comparison of the lines of code shows that the own realization needs about half of the code to provide the same functionality. Furthermore, the execution of benchmarks takes place to compare the performance of both implementations. While the sequential and concurrent reading of a result is equally fast, the performance and scalability of the own implementation for nesting multiple futures is significantly worse.

After analyzing the runtime environment, the weak points were identified. An excessive starting and immediately pausing of threads causes vast overhead. This again affects the runtimes significantly. The analysis leads to the optimization of the implementation. The rerunning of the benchmarks shows that the optimization corrected the performance deficits. It shows further that the own implementation has the performance characteristics as the native counterpart.

Inhaltsverzeichnis

Abkürzungsverzeichnis	vi
Abbildungsverzeichnis	vii
Tabellenverzeichnis	x
1 Einleitung	1
1.1 Motivation	3
1.2 Ziel der Arbeit	3
1.3 Vorgehensweise	4
2 Software Transactional Memory	5
2.1 Leitmotiv von STM	5
2.2 STM Features in Scala	6
2.2.1 Atomare Blöcke	7
2.2.2 Retry	7
2.2.3 orElse	8
2.3 Vorteile STM	9
2.4 Nachteile STM	13
2.5 STM Implementierungen in Scala	15
2.5.1 RadonSTM	15
2.5.2 MUTS	17
2.5.3 ScalaSTM	19
2.6 Verwendete Concurrency Mechanismen in Scala	22
2.6.1 Suche nach Scala Repositories	22
2.6.2 Durchsuchen der Repositories	23
2.6.3 Ergebnis	25
3 Futures mit STM	28
3.1 Auswahl des STMs	28
3.2 Implementierung der Futures und Promises	29
3.2.1 Datenstrukturen und Hilfsfunktionen	30
3.2.2 Future	33
3.2.3 Promise	40
4 Vergleich mit der nativen Implementierung	46
4.1 Vergleich des Aufbaus	46
4.2 Vergleich der Lines of Code	50
4.3 Benchmarking der Implementierungen	52
4.3.1 Aufbau der Benchmarks	53
4.3.2 Zugriff auf das Ergebnis eines Futures	55

4.3.3	Verschachtelung der Kombinatoren	61
4.3.4	Sleeping Barber mit Promises	71
4.3.5	Santa Clause mit Promises	73
4.3.6	Dining Philosophers mit Promises	75
5	Optimierung der Implementierung	80
5.1	Analyse der Schwachstellen	80
5.2	Optimierung	81
5.2.1	Erster Optimierungsversuch	81
5.2.2	Zweiter Optimierungsversuch	82
5.2.3	Validierung der Optimierung	86
6	Fazit	91
6.1	Zusammenfassung	91
6.2	Ausblick	93
	Literatur	94

Abkürzungsverzeichnis

STM: Software Transactional Memory

JVM: Java Virtual Machine

MUTS: Manchester University Transactions for Scala

EJB: Enterprise Java Beans

AST: Abstract Syntax Tree

API: Application Programming Interface

ADT: Abstract Data Type

GC: Garbage Collection

Abbildungsverzeichnis

1	Atomare Blöcke	7
2	Einsatz von retry, um die Transaktion erneut auszuführen	8
3	Angabe alternativer Berechnungspfade (orElse)	8
4	Funktionsweise von orElse	9
5	Withdraw und Deposite mit Locks	9
6	Funktion zur Durchführung eines Geldtransfers	10
7	Geldtransfer mit grobgranularen Locks	11
8	Geldtransfer mit Locks für jedes Konto	11
9	Geldtransfer mit Locks für jedes Konto und Ordnungsfunktion	12
10	Withdraw und Deposite mit STM	12
11	Geldtransfer mit STM	13
12	Transaktion mit Seiteneffekten	14
13	RadonSTM Syntax	16
14	Transaktionale Schleife	16
15	Verschachtelte Transaktionen	17
16	MUTS Transformation eines atomic-Blocks	18
17	Aufbau von ScalaSTM	19
18	ScalaSTM Syntax	20
19	Aggregation der Ergebnis	25
20	Algebraischer Datentyp Result	30
21	Empty und Fail	31
22	Success	31
23	Begleitobjekt des Result Traits	32
24	Die Hilfsfunktion forkIO erzeugt ein neues Runnable und weist dieses anschließend einem Threadpool zu.	32
25	Datenstruktur Future	33
26	Erzeugung eines neuen Futures	34
27	Lesen des Ergebnisses	34
28	Erzeugung eines gescheiterten Futures	35
29	Die Funktion onSuccess erlaubt die Ausführung eines Callbacks, falls das Future gelingt	35
30	WaitForSuccess versucht die Ausführung des Callbacks zu erzwingen und wartet deshalb bis das Future ein Resultat besitzt	36
31	FollowedBy wird eingesetzt um Folgeaufgaben asynchron durchzuführen und das Ergebnis in einem Future zu speichern	37
32	Combine gestattet das Zusammenfassen von zwei Ergebnissen	38
33	Die Funktion orAlt erlaubt die Angabe eines alternativen Futures	38
34	When gestattet die Übergabe eines Guards	39
35	Funktion zur Ermittlung des ersten, erfolgreich ausgeführten Futures	40

36	Die Promise Klasse und das dazugehörige Begleitobjekt	41
37	Funktion zur Erfüllung eines Promise	41
38	Funktion zur Erfüllung von zwei Promises	42
39	Funktion zur Erfüllung mehrerer Promises	43
40	ForceSuccess erzwingt die Erfüllung des Promises	44
41	Verwendung des Resultats eines Futures, um das Promise zu erfüllen . .	44
42	Verwendung eines validen Resultats eines Futures, um das Promise zu erfüllen	45
43	Das Promise Trait	47
44	Await blockt den Hauptthread und entspricht der get-Funktion	48
45	Verschachtlung von Funktionen höherer Ordnung (nativen Futures) . . .	49
46	Schema eines Benchmark-Objekts	54
47	Nebenläufiges Lesen eines Resultats	55
48	Performance beim mehrfachen, sequentiellen Zugriff auf ein Ergebnis. . .	56
49	CPU Sampling der eigenen Implementierung	57
50	CPU Sampling der nativen Implementierung	57
51	Nebenläufiges Lesen eines Resultats mit Threads	58
52	Nebenläufiges Lesen mit Threads	59
53	Nebenläufiges Lesen unter Verwendung des ForkJoinPools	60
54	Nebenläufiges Lesen unter Verwendung des ThreadPoolExecutors	60
55	CPU Zeiten der eigenen Implementierung	61
56	CPU Zeiten der nativen Implementierung	61
57	Verschachtelung mehrere unabhängiger Tasks	62
58	Das Ergebnis des Benchmarks beim dem mehrere Operationen verschach- telt werden	63
59	Verschaltung von Alternativen	64
60	Das Ergebnis des orAlt-Benchmarks	64
61	CPU Sampling der orAlt-Funktion	65
62	CPU Sampling der transform-Methode	65
63	Thread CPU Sampling der orAlt-Funktion	66
64	Thread CPU Sampling der transform-Methode	66
65	Benchmark der first-Funktion	67
66	Ergebnisse des Benchmarks der first-Funktion (mit sleep)	68
67	Die Funktion busyWait	69
68	Ergebnisse des Benchmarks der first-Funktion (mit busy waiting)	70
69	Genauigkeit der first-Funktion in Abhängigkeit von der Anzahl der aus- zuführenden Futures	71
70	Sleeping Barber Implementierung	71
71	Messergebnisse des Sleeping Barber Benchmarks	72
72	Santa Clause Implementierung	73
73	Hilfsfunktion collectHelper	74
74	Ergebnis des Santa Clause Benchmarks	74
75	Ergebnis des Santa Clause Benchmarks (ohne sleep)	75

76	Philosophers mit Promises und MultiTrySuccess	76
77	Die Funktion philosopher, die für die Erzeugung eines Philosophen zuständig ist	77
78	Philosophers mit Locks	77
79	Philosophers mit STM	78
80	Ergebnis des Dining Philosophers Benchmarks	79
81	Verhalten der Implementierung vor der Optimierung	81
82	Die angepasste forkIO-Funktion, die nun ein neues Future erzeugt, um die Berechnung asynchron auszuführen	81
83	Die Klasse Future und dessen Begleitobjekt nach der Optimierung	82
84	Die Funktion complete wird angewendet, um ein Future abzuschließen	83
85	Funktion, welche für die Ausführung der Callbacks verantwortlich ist	84
86	Mit addOrExecuteCallback können Callbacks hinzugefügt oder ausgeführt werden	84
87	Optimierte Version der followedBy-Funktion	85
88	Verhalten der Implementierung nach der Optimierung	86
89	Benchmark-Ergebnis der andThen-Funktion nach der Optimierung	86
90	Benchmark-Ergebnis von orAlt/fallbackTo nach der Optimierung	87
91	Benchmark-Ergebnis der first-Funktion nach der Optimierung	87
92	Genauigkeit der first-Funktion nach der Optimierung	88
93	Benchmark der onSuccess-Funktion	89
94	Benchmark-Ergebnis der Callback-Funktion	90

Tabellenverzeichnis

1	Einsatz der einzelnen Mechanismen sortiert nach Häufigkeit	26
2	Kombiniertes Auftreten der Mechanismen	27
3	Lines of Code der nativen Futures und Promises	51
4	Lines of Code der eigenen, funktionalen Futures und Promises	51
5	Vergleich beider Implementierungen mit gleicher Funktionalität	51

1 Einleitung

Moderne Prozessoren verfügen heutzutage über mehrere physische und logische Kerne. Jeder Kern kann dabei mindestens einen Thread gleichzeitig ausführen[1], wodurch die parallele Abarbeitung von Aufgaben möglich ist.

Prinzipiell ist dabei eine Unterscheidung zwischen Aufgaben- und Datenparallelität möglich [2, S. 85 ff.]. Bei der Aufgabenparallelität arbeiten alle Threads unabhängig voneinander und besitzen in der Regel dedizierte Aufgaben. Nutzen diese Threads dabei gemeinsame Ressourcen, müssen zusätzliche Mechanismen wie beispielsweise Sperren (Locks), Semaphoren oder Transaktionen zur Synchronisation eingesetzt werden. Die Datenparallelität wiederum kommt bei Szenarien zum Einsatz, bei denen eine gemeinsame Datenstruktur, wie beispielsweise ein Array oder eine Matrix, vorliegt und alle Threads dieselbe Aktion auf unterschiedlichen Teilen dieser Datenstruktur durchführen. Dabei ist es wichtig, dass die Elemente entweder völlig unabhängig voneinander sind oder nur eine bedingte Abhängigkeit besteht. Eine bedingte Abhängigkeit besteht zum Beispiel dann, wenn alle Elemente einer Reihe, Spalte oder die Nachbarelemente benötigt werden um ein Resultat zu berechnen.

Die parallele Ausführung von Programmcode kann dabei zu besserer Performance, einer besseren Ausnutzung von vorhandenen Ressourcen sowie einer höheren Skalierung von Anwendungen führen. Den Vorteilen gegenüber stehen die damit verbundenen Kosten. Diese sind zum einen Leistungseinbußen, die bei einem Kontextwechsel von Threads entstehen und zum anderen die Aufgabe sicherzustellen, dass gemeinsam genutzte Speicherbereiche nicht quasi gleichzeitig verändert werden, da dies sonst zum Verlust der Datenintegrität führen kann.

Um sicherzustellen, dass es zu keinem Verlust der Integrität kommt, werden klassisch pessimistische Verfahren verwendet, wobei davon ausgegangen wird, dass es häufig zu Konflikten kommt. Bei diesen Ansätzen findet der Zugriff auf die gemeinsam genutzten Daten nur innerhalb kritischer Bereiche statt. Diese Abschnitte werden dabei oft von Konstrukten wie binären Semaphoren oder Monitoren geschützt und ermöglichen einen exklusiven Zugriff auf die gemeinsamen Ressourcen. Diese Art der Synchronisation ermöglicht das Auftreten verschiedenster Fehler, wie beispielsweise Starvation, Deadlocks, Livelocks, Race Conditions oder einer Prioritätsinversion, welche bei der Entwicklung einer Anwendung bedacht und bestenfalls ausgeschlossen werden sollen.

Bei optimistischen Ansätzen hingegen werden nur wenige Konflikte erwartet, weshalb kein

kritischer Bereich notwendig ist. Erst wenn es zur Änderungen eines Datums kommt, wird geprüft, ob ein Konflikt aufgetreten ist. Sollte dies der Fall sein, werden fehlerkorrigierende Maßnahmen eingeleitet, um das Problem zu beseitigen. Die Konfliktbereinigung selbst ist für den Anwendungsentwickler nicht sichtbar und wird automatisch im Hintergrund ausgeführt.

Bereits 1977 beschrieb Lomet [3] ein Konzept für ein optimistisches Verfahren, bei dem Transaktionen verwendet werden, um nebenläufige Anwendungen zu synchronisieren. Transaktionen, die sich bereits im Datenbankbereich bewährt haben, abstrahieren den Zugriff auf die gemeinsamen Ressourcen. Die Synchronisation selbst findet dabei im Hintergrund statt und geschieht ohne Zutun der Nutzer, wodurch die Entwicklung paralleler Anwendungen vereinfacht wird.

Datenbanktransaktionen unterscheiden sich jedoch von Software Transaktionen hinsichtlich der Zugriffszeiten auf das Speichermedium und dem Kontext, in dem sie verwendet werden [4, S. 12 f.]. Während Datenbanken Informationen auf nichtflüchtigen Medien, mit Zugriffszeiten im Bereich von 5-10 Millisekunden, speichern, benötigen Software Transaktionen nur einige hundert Instruktionen, um auf ein Datum im Hauptspeicher zuzugreifen. Durch den deutlich schnelleren Zugriff auf den Speicher ist eine effiziente Konflikterkennung und -bereinigung deutlich wichtiger als bei Datenbanktransaktionen. Zudem werden Datenbanktransaktionen in einem geschlossenen System ausgeführt, welches eine feste Konfiguration besitzt, wohingegen Software Transaktionen mit einer Vielzahl unterschiedlicher Paradigmen, Bibliotheken und Frameworks interagieren müssen.

Aufgrund der Anforderungen bezüglich Performance und Integration von Software Transaktionen stand die Entwicklung effizienter Algorithmen und Frameworks während der letzten 20 Jahre im Fokus wissenschaftlicher Arbeiten und brachte eine Vielzahl unterschiedlicher Lösungsmöglichkeiten für nahezu jede populäre Sprache wie C/C++, Java, Haskell, Scala etc. hervor. Die Implementierungen unterschieden sich dabei hauptsächlich dadurch, dass Konflikte zu unterschiedlichen Zeitpunkten erkannt und behoben werden. Weitere Unterscheidungsmerkmale sind die Art und Weise wie Transaktionen ausgeführt werden und die interne Datenverwaltung.

Auch die Art der Bereitstellung von Transaktionen unterscheidet sich. Einige Implementierungen brachten Software Bibliotheken hervor, die über ein einfaches Interface aufgerufen werden und alle Implementierungsdetails abstrahieren. Andere Umsetzungen wiederum erweitern die verwendete Programmiersprache oder verwenden Techniken wie das Byte-Code Rewriting, um die Funktionalität bereitzustellen.

1.1 Motivation

Scala ist eine relativ junge Sprache, welche 2004 veröffentlicht wurde und die beiden Paradigmen der funktionalen und objektorientierten Programmierung unterstützt. Der erzeugte Code selbst wird dabei von einem Scala Compiler in Java Bytecode umgewandelt und in der virtuellen Maschine von Java ausgeführt. Durch die Verwendung der JVM zur Ausführung des Codes ist auch die Interaktionen mit Java selbst möglich. Das Aufrufen von Methoden, Erstellen von Java Objekten sowie Vererbung und die Implementierung von Java Interfaces wird dabei nativ unterstützt.

Aktuell gibt es mehrere STM Implementierungen für Scala. Diese verfolgen unterschiedliche Ansätze, um Software Transaktionen zur Verfügung zu stellen. Die prominentesten Vertreter sind ScalaSTM [5] und MUTS [6]. ScalaSTM ist eine Software Bibliothek, die die Verwendung von Software Transaktionen über ein einfaches Interface ermöglicht. MUTS hingegen ist eine Kombination aus einer minimalen Compilererweiterung und Nutzung eines Java-Agenten. Die Erweiterung umfasst dabei nur eine Modifikation des Parsers, die die Verwendung neuer Schlüsselworte ermöglicht. Der Agent wiederum basiert auf der im Java Framework DEUCE [7] verwendeten Agentenkomponente und gestattet es Änderungen an den geladenen Klassen vorzunehmen. Auf diese Weise können auch externe Bibliotheken ohne Bedenken innerhalb von Transaktionen verwendet werden.

In dieser Arbeit soll evaluiert werden, wie häufig STM in reinen Scala Projekten eingesetzt wird, um Concurrency-Probleme zu lösen. Zudem soll gezeigt werden, wie STM eingesetzt werden kann, um eine Implementierung der beiden Concurrency Abstraktionen Futures und Promises zu realisieren. Anschließend soll die eigene Umsetzung mit der nativen Implementierung der beiden Mechanismen verglichen werden, um Gemeinsamkeiten und Unterschiede aufzuzeigen. Zudem soll die Performance der Implementierungen analysiert werden, um Vor- und Nachteile beider Alternativen zu veranschaulichen.

1.2 Ziel der Arbeit

Das Ziel dieser Arbeit kann in drei Teilaufgaben untergliedert werden. Zuerst soll STM selbst vorgestellt werden. Dies umfasst zum einen eine Übersicht über die von STM zur Verfügung gestellten Features und zum anderen eine Beschreibung der Vor- und Nachteile von STM gegenüber pessimistischen Ansätzen wie Sperren. Im Anschluss werden in Scala verfügbare STM Implementierungen vorgestellt. Abschließend soll ermittelt wer-

den, wie häufig STM eingesetzt wird, um Concurrency Probleme zu lösen.

Die zweite Teilaufgabe beschäftigt sich mit der Implementierung funktionaler Futures und Promises unter Einsatz von STM. Ziel ist es dabei zu zeigen, wie eine solche Implementierung aussehen und wie diese verwendet werden kann. Hierunter fällt die Beschreibung der einzelnen Funktionen, die auf ein Future oder Promise angewendet werden können sowie deren Funktionsweise.

Die letzte Teilaufgabe befasst sich mit dem Vergleich der Implementierungen dieser Abstraktionen. Die Gegenüberstellung enthält dabei quantitative und qualitative Aspekte. Bei der quantitative Analyse wird verglichen, wie viele Lines of Code die einzelnen Umsetzungen verwenden, um die Funktionalität bereitzustellen. Der qualitative Vergleich befasst sich mit dem Benchmarking spezieller Anwendungsfälle. Abschließend erfolgt eine Bewertung der Ergebnisse.

1.3 Vorgehensweise

Der Aufbau der weiteren Arbeit stellt sich wie folgt dar. Das zweite Kapitel befasst sich mit der Beschreibung von STM im Scala Kontext. Dies umfasst die Vorstellung aller Features, deren Funktionsweise anhand von Beispielen verdeutlicht wird. Zusätzlich wird in diesem Kapitel erläutert, welche Vor- und Nachteile STM gegenüber Sperren und ähnlichen pessimistischen Mechanismen besitzt. Zum Abschluss des zweiten Kapitels wird die Evaluation, welche im Rahmen dieser Arbeit erstellt worden ist, vorgestellt. Diese Evaluation beschäftigt sich mit der Frage, welche Concurrency Mechanismen in Scala eingesetzt werden. Hierdurch soll gezeigt werden, welche dieser Mechanismen für die Lösung von Nebenläufigkeitsproblemen eingesetzt werden und welchen Stellenwert STM dabei besitzt.

Im dritten Kapitel wird die Implementierung der funktionalen Futures und Promises, die im Rahmen dieser Arbeit erstellt werden, vorgestellt. Der Fokus liegt hierbei auf der Beschreibung der verwendeten Datenstrukturen, dem Aufbau der einzelnen Funktionen sowie der Beantwortung der Frage, welche Rolle STM dabei spielt.

Das vierte Kapitel widmet sich dem quantitativen und qualitativen Vergleich der eigenen und nativen Implementierung der Futures und Promises.

Das fünfte Kapitel befasst sich mit der Optimierung der Futures. Diese basieren auf Erkenntnissen, die beim Benchmarking erlangt werden konnten.

Im letzten Kapitel dieser Arbeit befindet sich das Fazit. Dort werden die ausgeführten Tätigkeiten nochmals kritisch betrachtet und abschließend bewertet.

2 Software Transactional Memory

Software Transactional Memory basiert, wie der Name bereits antizipieren lässt, auf Transaktionen. Hierbei wird das Transaktionskonzept verwendet, um den Zugriff mehrerer Prozesse auf gemeinsam genutzte Ressourcen zu abstrahieren und das Entwickeln von nebenläufigen Anwendungen zu vereinfachen. Der Vorteil gegenüber anderen Synchronisationsverfahren wie beispielsweise Semaphoren oder Locks ist, dass das STM die Komplexität verbirgt und in der Lage ist gängige Fehler wie Deadlocks oder die Prioritätsinversion zu verhindern.

Zur Erläuterung was sich hinter dem Begriff Software Transactional Memory verbirgt, soll zuallererst das Leitmotiv von STM vorgestellt werden (Abschnitt 2.1). Dabei soll hervorgehoben werden, welchen Mehrwert STM liefert und welche Einschränkungen existieren. Daraufgehend werden STM Features, die in Scala eingesetzt werden können, vorgestellt und deren Funktionsweise erläutert (Abschnitt 2.2). Die beiden Abschnitte 2.3 und 2.4 widmen sich den Vor- und Nachteilen die STMs generell besitzen. Im vorletzten Abschnitt (2.5) werden die STMs, die in Scala verfügbar sind, vorgestellt. Zum Abschluss dieses Kapitels wird eine Evaluierung, die im Rahmen dieser Arbeit erstellt wurde, vorgestellt (Abschnitt 2.6). Diese Evaluierung befasst sich mit der Frage, welche Concurrency Mechanismen in Scala eingesetzt werden.

2.1 Leitmotiv von STM

Die Idee Transaktionen als Mechanismus zur Sicherstellung der Konsistenz von gemeinsam genutzten Daten im Hauptspeicher zu verwenden ist alt und wurde bereits 1977 von Lomet [3] beschrieben. Aufgrund fehlender Einsatzgebiete wurde in den folgenden Jahren keine weitere Forschung in diesem Bereich betrieben. Durch den Einsatz von Multithreading und der Entwicklung von Mehrkernprozessoren, gewann das Thema in den letzten Jahren jedoch wieder an Bedeutung und rückte in den Vordergrund. Hierdurch sind eine Vielzahl unterschiedlicher Konzepte und Implementierungen entstanden, die in modernen STMs zu finden sind.

Der Grundgedanke von STM ist der Einsatz der Transaktionsabstraktion zur Koordination von simultanen Lese- und Schreibvorgängen, die auf gemeinsam genutzte Ressourcen zugreifen [4, S. 6 ff.]. Die Verwendung einer höheren Abstraktionsebene soll dabei

die Entwicklung nebenläufiger Anwendungen vereinfachen. Zudem soll Entwicklern die Bürde abgenommen werden, sich jedes Mal erneut Gedanken über die Threadsynchronisation machen zu müssen, da auch moderne Sprachen wie Java, Go oder Scala bis heute nur Low-Level Mechanismen wie Locks, Semaphoren oder Monitore hierfür bereitstellen. Low-Level Mechanismen haben zwei entscheidende Nachteile. Zum einen stellen sie eine Fehlerquelle dar, weshalb deren Einsatz gerade für unerfahrene Programmierer problematisch ist. Zum anderen sind Funktionskompositionen nicht möglich, da diese Mechanismen nicht garantieren können, dass die ausführenden Prozesse Sperren in richtiger Reihenfolge erwerben und freigeben. Geschieht dies nicht in der richtigen Ordnung kommt es zu Deadlocks. Da gerade funktionale Programmiersprachen Funktionen als Objekte erster Klasse behandeln und die Komposition dieser ein wichtiger Bestandteil solcher Sprachen ist, erscheint die Verwendung von diesen Techniken unangebracht. STMs hingegen übernehmen die Synchronisation und verstecken die Komplexität dieser hinter simplen Schnittstellen oder Sprachkonstrukten. Zudem garantieren solche Systeme die Korrektheit der Ausführung des Programmcodes und erlauben eine beliebige Komposition von Transaktionen. Ein weiterer Vorteil ist, dass die Anzahl potentieller Fehler, die von Entwicklern verursacht werden können, verringert werden. Dennoch sind Transaktionen kein Allheilmittel und nicht im Stande alle Fehlerquellen zu eliminieren. So ist beispielsweise die Verwendung von I/O-Operationen oder die Nutzung externer Bibliotheken in Kombination mit Transaktionen problematisch, da bereits ausgeführte Aktionen nicht rückgängig gemacht werden können und somit das Transaktionsprinzip ausgehebelt wird.

2.2 STM Features in Scala

Nach der Vorstellung des Leitmotives, soll der Einsatz von STM in Scala näher betrachtet werden. Dieser Abschnitt widmet sich daher den von STMs bereitgestellten Features sowie einer näheren Beschreibung dieser. Da die verschiedenen STMs in Scala eine teilweise unterschiedliche Namensgebung besitzen, werden die Features mit Hilfe von Pseudocode beschrieben. Dieser Code ähnelt dem der Implementierungen, enthält der Vereinfachung halber jedoch nicht alle implementierungsspezifischen Details. Eine vollständige Beschreibung der Syntax sowie der darunterliegenden Mechanismen der verfügbaren STMs erfolgt in Abschnitt [2.5](#).

2.2.1 Atomare Blöcke

Das erste zu betrachtende Feature sind atomare Blöcke. Diese Blöcke werden mit dem Schlüsselwort *atomic* eingeleitet und kapseln den Zugriff auf die geschützten Ressourcen. Abbildung 1 illustriert, wie eine sonst nicht threadsichere Datenstruktur wie eine Queue, durch Verwendung von STM threadsicher gemacht wird und von mehreren Threads gleichzeitig aufgerufen werden kann. Die beiden Operationen *put* und *pop* ermöglichen dabei das nebenläufige Hinzufügen und Entnehmen eines Elements. Durch die Ausführung innerhalb atomarer Blöcke wird sichergestellt, dass das STM den Zugriff überwacht und auftretende Konflikte behandelt. Konflikte können bei diesem Beispiel bei jeder Operation entstehen, da sowohl das Lesen als auch das Schreiben die Warteschlange verändern. STM stellt dabei sicher, dass **eine** der ausgeführten Transaktion immer erfolgreich durchgeführt wird. Alle anderen Transaktionen scheitern, werden zurückgerollt und anschließend erneut ausgeführt, bis auch diese erfolgreich abgeschlossen werden.

```
def put(value: A): Unit = {      def pop(): A = {
  atomic {                       atomic {
    queue enqueue value          queue dequeue
  }                             }
}
```

Abbildung 1: Atomare Blöcke

2.2.2 Retry

Ein weiteres Feature, welches von STMs bereitgestellt werden kann, ist *retry*. Diese Feature wurde erstmals von Harris et al. [8] beschrieben und erweitert die Steuerung des Kontrollflusses. Wird beim Ausführen einer Transaktion festgestellt, dass das Gelingen dieser unter den gegenwärtigen Umständen nicht möglich ist, kann die *retry* Anweisung dazu verwendet werden, einen Abbruch des aktuellen Versuchs zu erzwingen. Im Zuge dessen werden alle bisherigen Änderungen zurückgerollt, die Transaktion pausiert und zu einem späteren Zeitpunkt erneut ausgeführt.

Um zu verhindern, dass ein erneuter Versuch ohne Aussicht auf Erfolg gestartet wird, kommt ein bedingter Wartemechanismus zum Einsatz. Dieser verzögert den Start eines neuen Versuchs so lange, bis sich mindestens eine der bei der Transaktion verwendeten Variablen ändert. Die Modifikation wird automatisch erkannt und leitet die Ausführung eines neuen Versuchs selbstständig ein. Dieser Ansatz ähnelt dabei stark dem

wait/notify-Mechanismus, ist jedoch mächtiger, da hierbei Transaktionen, bzw. Threads explizit angesteuert und nicht zufällig gewählt werden.

Abbildung 2 illustriert den Einsatz von `retry` in Kombination mit der Entnahme eines Elements aus einer Warteschlange. Im Gegensatz zum ersten Beispiel, wird hier geprüft, ob sich Elemente in der Queue befinden. Ist dies nicht der Fall, wird ein `retry` ausgeführt. Dies führt in diesem Fall dazu, dass die Transaktion solange zurückgestellt wird, bis Elemente hinzugefügt worden sind und eine Entnahme möglich ist.

```
def pop(): Int = atomic {  
  if (queue isEmpty) retry  
  queue dequeue  
}
```

Abbildung 2: Einsatz von `retry`, um die Transaktion erneut auszuführen

2.2.3 orElse

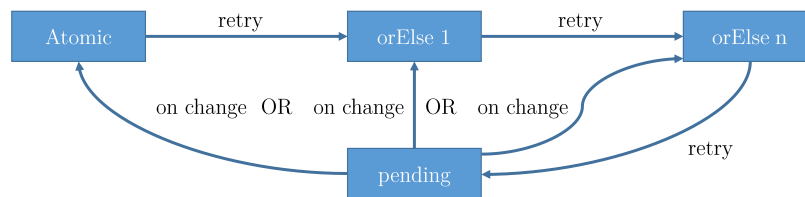
Das letzte zu betrachtende Feature von STMs ist *orElse*. Dieses Konstrukt basiert ebenfalls auf einer Idee von Harris et al. [8] und versetzt den Anwender in die Lage alternative Aktionen zu definieren. Alternative Berechnungspfade werden durch das Schlüsselwort `orElse` gekennzeichnet und ähneln syntaktisch einer `if`-Kaskade, da eine beliebige Anzahl an Alternativen definiert werden kann. Das Beispiel aus Abbildung 3 veranschaulicht den Einsatz von `orElse`. Die dort zu sehende Funktion `popOrElseDefault` besitzt einen zweiten Berechnungspfad, welcher zählt, wie oft kein Element in der Warteschlange vorhanden war und der Standardwert zurückgegeben worden ist.

`Retry` kommt hierbei ebenfalls zum Einsatz, erfüllt in Kombination mit `orElse` jedoch eine andere Funktionalität. Wird die Bedingung, die zum `retry` führt, erfüllt, dann wird die aktuelle Berechnung ebenfalls abgebrochen und alle Änderungen rückgängig gemacht.

```
def popOrElseDefault(): Int = atomic {  
  if (queue isEmpty) retry  
  queue dequeue  
} orElse {  
  popMiss = popMiss + 1  
  Int.MinValue  
}
```

Abbildung 3: Angabe alternativer Berechnungspfade (`orElse`)

Anders als bei der Verwendung ohne `orElse` wird im Anschluss die Berechnung der Alternative ausgelöst, anstatt zu warten, bis sich eine der verwendeten Variablen ändert. Die Ausführung der Alternativen findet in einer festgelegten Reihenfolge statt, beginnend mit dem `atomic`-Block. Sollte die Berechnung aller Möglichkeiten scheitern, wird der Prozess in einen wartenden Zustand versetzt. Wie in Abbildung 4 zu sehen ist, verweilt der Prozess solange in diesem Zustand, bis die Bedingung der `retry`-Anweisung einer Alternative erfüllt ist und eine erneute Berechnung durchgeführt werden kann. Dabei gilt, dass jeder Zweig gleichberechtigt ist und die festgeschriebene Reihenfolge aufgehoben wird.

Abbildung 4: Funktionsweise von `orElse`

2.3 Vorteile STM

Die Vorteile von STM gegenüber pessimistischen Concurrency Mechanismen wie Locks, Semaphoren oder Monitoren lassen sich am besten anhand eines konkreten Beispiels aufzeigen. Die zu bewältigende Aufgabe besteht darin, Geld von einem Bankkonto auf ein Anderes zu überweisen [9]. Ein Konto, wie in Abbildung 5 zu sehen, besteht aus dem aktuellen Guthaben sowie den beiden Methoden *withdraw* und *deposit*, die zum Ein- und Auszahlen eines bestimmten Betrags verwendet werden. Beide Methoden werden zusätzlich synchronisiert, um zu verhindern, dass diese von mehreren Threads gleichzeitig ausgeführt werden können.

Die Transferfunktion (Abbildung 6) selbst besteht aus den beiden Konten, welche von

```

class Account {
  var balance: Long = 0
  def withdraw(amount: Long) = synchronized { balance - amount }
  def deposit(amount: Long) = synchronized { balance + amount }
}

```

Abbildung 5: Withdraw und Deposit mit Locks

der Überweisung betroffen sind sowie dem zu übermittelnden Betrag. Bevor der Transfer durchgeführt werden kann, muss geprüft werden, ob das Geberkonto (from) über genügend finanzielle Mittel verfügt. Ist dies gegeben, kann der Transfer durchgeführt werden. Dazu findet eine Abbuchung auf Seiten des Gebers und eine Einzahlung auf dem Nehmerkonto statt. Während sich diese Funktion bei einem sequentiellen Programm fehlerlos ausführen lässt, kommt es bei einer nebenläufigen Anwendung zu Dateninkonsistenzen. Die Synchronisation der Methoden verhindert zwar den mehrfachen Zugriff auf ein Konto, schützen jedoch nicht davor, dass der Zwischenstand des Transfers, bei dem der Betrag bereits abgebucht aber noch nicht eingezahlt ist, kurzzeitig für andere Threads sichtbar ist. Dies kann zur Folge haben, dass Änderungen eines Threads verloren gehen, da diese von einem Anderen überschrieben werden. Eine Komposition der synchronisierten Methoden ist ebenfalls nicht möglich, da es sich um zwei unterschiedliche Objekte handelt, die nicht denselben Monitor besitzen. Zudem werden die beiden Konten nicht gleichzeitig, sondern nacheinander ge- und entsperrt.

```
def transfer (from: Account, to: Account, amount: Int) = {  
  if (from.balance() >= amount) {  
    from.withdraw(amount)  
    to.deposit(amount)  
  }  
}
```

Abbildung 6: Funktion zur Durchführung eines Geldtransfers

Beim ersten naiven Ansatz zur Lösung des Konsistenzproblems werden Sperren eingesetzt, um den kritischen Bereich vor mehrfachem Zugriff zu schützen. Wie im Codebeispiel 7 abgebildet, wird die Sperre bereits vor der Prüfung der Kreditwürdigkeit des Geberkontos erworben. Hierdurch wird sichergestellt, dass kein weiterer Thread die Möglichkeit erhält Änderungen durchzuführen. Zusätzlich wird ein try-finally Statement genutzt, damit die Sperre auch im Fehlerfall wieder freigegeben wird. Wird darauf verzichtet, so ist es möglich, dass die Sperre permanent aufrechterhalten wird und ein Deadlock entsteht. Dies wiederum hat zur Folge, dass keine weiteren Transfers ausgeführt werden können. Ein zusätzliches Problem solcher grobgranularen Sperrungen ist, dass die Ausführung der Operation nur noch sequentiell geschieht und alle Transferversuche warten müssen bis diese ausgeführt werden können, selbst wenn die Transfers disjunkte Konten betreffen. Der dadurch entstehende Flaschenhals beeinflusst die Skalierbarkeit einer nebenläufigen Applikation maßgeblich, da die Threads solange warten müssen, bis sie an der Reihe sind. Dies wiederum führt zu einer Verschwendung von

```
def transfer (from: Account, to: Account, amount: Int) = {  
  lock.lock()  
  try {  
    if (from.balance() >= amount) {  
      from.withdraw(amount)  
      to.deposit(amount)  
    }  
  } finally {  
    lock.unlock()  
  }  
}
```

Abbildung 7: Geldtransfer mit grobgranularen Locks

CPU Ressourcen und einer schlechteren Performance von Applikationen.

Die Einführung feinerer Sperrungen, wie in Abbildung 8 veranschaulicht, löst die eben beschriebenen Probleme. Bei diesem Ansatz besitzt jedes Konto eine eigene Sperre. Dies verhindert Inkonsistenzen, da dasselbe Konto immer nur an einem Transfer beteiligt sein kann, erlaubt jedoch die gleichzeitige Ausführung nebenläufiger Transfers auf disjunkten Konten. Da die Transfers nun parallel ausführbar sind, wird die Performance und Skalierbarkeit der Applikation gesteigert. Dennoch ist auch diese Implementierung fehleranfällig. Angenommen die beiden Überweisungen `transfer(A, B, 100)` und `transfer(B, A, 100)` werden gleichzeitig ausgeführt, so führt dies zu einer zyklischen Abhängigkeit. Der erste Transfer versucht hierbei zuerst Konto A und daraufhin B zu sperren. Der zweite Transfer wiederum versucht zuerst Konto B und danach A zu sperren. Bei dieser Konstruktion kann es passieren, dass beide Transfers versuchen die Sperren gleichzeitig zu erwerben, was zu einem Deadlock führt.

```
def transfer (from: Account, to: Account, amount: Int) = {  
  if (from.balance() >= amount) {  
    try {  
      from.lock()  
      to.lock()  
      from.withdraw(amount)  
      to.deposit(amount)  
    } finally {  
      to.unlock()  
      from.unlock()  
    }  
  }  
}
```

Abbildung 8: Geldtransfer mit Locks für jedes Konto

Wie in Abbildung 9 skizziert, bedarf die Implementierung einer deadlockfreien Transferoperation neben feingranularer Sperren eine zusätzliche Funktion, mit deren Hilfe ermittelt wird in welcher Reihenfolge die Sperren erworben und freigegeben werden müssen. Bereits die Definition einer solchen Ordnungsfunktion ist eine nicht triviale Aufgabe und zeigt deutlich, welche Überlegungen gemacht werden müssen, wenn Low-Level Mechanismen wie Sperren eingesetzt werden. Neben der Komplexität, zeigt dieses Beispiel auch, dass eine nicht unerhebliche Menge an Codezeilen benötigt wird, wodurch der Code schwerer lesbar wird und die Fehleranfälligkeit steigt.

```
def transfer (from: Account, to: Account, amount: Int) = {  
  if (from.balance() >= amount) {  
    if ( /** determine lock order */ ) {  
      from.lock()  
      to.lock()  
      from.withdraw(amount)  
      to.deposit(amount)  
      to.unlock()  
      from.unlock()  
    } else {  
      // reverse order of lock acquisition and release  
    }  
  }  
}
```

Abbildung 9: Geldtransfer mit Locks für jedes Konto und Ordnungsfunktion

Als Nächstes soll die Lösung des gegebenen Problems, unter Verwendung von STM, betrachtet werden. Die einzige Veränderung des Kontos umfasst den Austausch des Schlüsselwortes `synchronized` durch `atomic`.

```
class Account {  
  var balance: Long = 0  
  def withdraw(amount: Long) = atomic { balance - amount }  
  def deposit(amount: Long) = atomic { balance + amount }  
}
```

Abbildung 10: Withdraw und Deposit mit STM

Diese Änderung ermöglicht nun die Komposition der beiden Methoden. Wie in Abbildung 11 dargestellt, wird lediglich ein weiterer atomarer Block benötigt, welcher die beiden Anweisungen umschließt. Da auch Software Transaktionen die Eigenschaft der Atomarität besitzen, kann der gesamte Transfer nur dann erfolgreich abgeschlossen werden, wenn beide Subtransaktionen erfolgreich sind. Die Korrektheit der Ausführung

selbst, wird dadurch garantiert, dass STM alle Transaktionen überwacht. Hierdurch können Konflikte, die auftreten wenn mehrere Threads versuchen dasselbe Konto zu verwenden, automatisch erkannt und behoben werden. Neben der Garantie der korrekten Ausführung der Operation, gewährleistet STM zudem Deadlockfreiheit, wodurch die Implementierung einer Ordnungsfunktion sowie komplexer Sperrmechanismen entfällt.

```
def transfer(from: Account, to: Account, amount: Int) = {  
  atomic {  
    if (from.balance() >= amount) {  
      from.withdraw(amount)  
      to.deposit(amount)  
    }  
  }  
}
```

Abbildung 11: Geldtransfer mit STM

2.4 Nachteile STM

Trotz der Einfachheit der Nutzung und der Reduzierung potentieller Fehlerquellen ist STM kein Allheilmittel. Optimistische Mechanismen, wie STM, funktionieren am besten, wenn die Anzahl an Leseoperationen die Anzahl der Schreiboperationen übersteigt, da es allen Threads möglich ist gleichzeitig zu lesen. Überwiegt die Anzahl der Schreiboperationen führt dies dazu, dass es vermehrt zu Konflikten kommt. Daraus resultiert, dass Transaktionen erneut ausgeführt werden müssen und die Performance aufgrund dessen sinkt.

Transaktionen mit langen Laufzeiten stellen ebenfalls ein Problem dar, da die Ausführung dieser immer wieder von kurzen Transaktionen unterbrochen werden kann. Einige STM Implementierungen, wie beispielsweise SwissTM[10], adressieren dieses Problem explizit und besitzen Mechanismen, die dafür sorgen, dass lange Transaktionen erfolgreich abgeschlossen werden können.

Ebenfalls problematisch ist der Einsatz von STM im imperativen Kontext. Das Problem das hierbei besteht ist, dass nur transaktionale Variablen vom STM verwaltet werden können. Aufrufe einer Software Bibliothek, legacy Code oder I/O-Operationen, wie das Senden von Informationen über das Netzwerk oder das Schreiben auf die Konsole können beim Scheitern der Transaktion nicht wieder rückgängig gemacht werden. Wie in Abbildung 12 zu sehen, ist die Inkrementierung von `x` im Fehlerfall umkehrbar, die Übertragung dieses Wertes an einen Client jedoch nicht, da der Wert bereits gesendet wurde.

Dies schränkt den Einsatz von STM ein und kann sogar dazu führen, dass der Einsatz hierdurch verhindert wird.

Einige STMs, wie beispielsweise DEUCE [7], lösen dieses Problem teilweise und unterstützen den Einsatz von legacy Code und Software Bibliotheken. Hierfür findet beim Laden aller Klassen ein Bytecode Rewriting statt, welches dafür sorgt, dass eine Kopie jeder Methode generiert wird. Diese speziellen Methoden werden immer dann verwendet, wenn ein transaktionaler Kontext vorhanden ist. Die Verwaltung aller genutzten Variablen obliegt dabei dem STM, wodurch ein Rollback sämtlicher Änderungen möglich ist.

```
def doSomething() = {  
  atomic {  
    x = x + 1    // rollback possible  
    sendToClient(x) // rollback not possible  
  }  
}
```

Abbildung 12: Transaktion mit Seiteneffekten

Die Implementierung effizienter Algorithmen hat sich ebenfalls als schwierig herausgestellt. Aus Anwendersicht erscheint dieses Problem erst einmal nicht relevant, da es nicht die Nutzung des Mechanismus betrifft. Da die Effizienz der Algorithmen jedoch direkten Einfluss auf die Performance und den Speicherverbrauch einer Anwendung hat, stellt sich spätestens vor dem operativen Einsatz der Software die Frage, welche Technik eingesetzt werden soll, um gegebene Anforderungen zu erfüllen.

Zur Überwindung der Performancenachteile und der Verringerung des entstehenden Overheads, welcher durch die Verwaltung der Ressourcen entsteht, wird seit über 20 Jahren intensiv geforscht. Daraus resultiert eine Vielzahl unterschiedlicher Ansätze, welche sich hinsichtlich der Versionsverwaltung sowie dem Zeitpunkt der Konflikterkennung und -behebung unterscheiden.

Die Verwaltung der Version einer Ressource, die maßgeblich zur Erkennung von Konflikten beiträgt, kann dabei entweder durch lokale Zeitstempel, globale Zeitstempel oder globalen Metadaten erfolgen [4, S. 108 ff.]. Die Versionsverwaltung ist zudem für die Aktualisierung der Variablen verantwortlich. Hierbei wird zwischen einer eifrigen (eager) und faulen (lazy) Versionsverwaltung unterschieden. Die eifrige Verwaltung führt die Änderungen sofort aus und erzeugt Backups der betroffenen Variablen, die im Fehlerfall zurückgeschrieben werden. Bei der faulen Versionsverwaltung hingegen werden Änderungen in privaten Locks gespeichert, die beim Scheitern verworfen werden. Die

Aktualisierung der Variablen wird bis zum Commit der Transaktion zurückgehalten. Die Konflikterkennung sowie das Updaten der Variablen können zu unterschiedlichen Zeitpunkten stattfinden [4, S. 19 ff.]. Bei der eifrigen Erkennung werden potentielle Konflikte bereits beim Ressourcenzugriff ermittelt. Ebenfalls möglich ist der Einsatz von zusätzlichen Validierungsschritten. Die dritte Möglichkeit, die auch als lazy bezeichnet wird, findet direkt vor dem Commit statt.

Es hat sich herausgestellt, dass bestimmte Kombinationen der unterschiedlichen Versionsverwaltungen und Konflikterkennungen in unterschiedlichen Situationen Vorteile gegenüber anderen Kombinationen besitzen. Hybride Ansätze, wie SwissTM [10], setzten deshalb unterschiedliche Techniken in bestimmten Situationen ein, um die Performance zu steigern. Schreib/Schreib-Konflikte werden hierbei früh erkannt, um zu verhindern, dass CPU Ressourcen unnötigerweise in Anspruch genommen werden, da eine der beiden Transaktionen sowieso scheitern wird. Bei Schreib/Lese-Konflikten kommt eine späte Konflikterkennung zum Einsatz. Hierbei ist es möglich, dass der Lesevorgang vor dem Schreiben abgeschlossen wird und eine Wiederholung des Lesens daher nicht notwendig ist.

2.5 STM Implementierungen in Scala

Nach der Vorstellung der Verwendung von STM sowie der Aufzählung von Vor- und Nachteilen, behandelt dieser Abschnitt verfügbare STM Implementierungen in Scala. Eine Auflistung der STMs kann der folgenden Wikiseite [11] entnommen werden. Von der Beschreibung ausgeschlossen werden AkkaSTM und LucreSTM. Dies wird dadurch begründet, dass AkkaSTM seit der Version 2.4 deprecated ist und nicht weiter unterstützt wird. LucreSTM wird nicht näher betrachtet, da dieses STM lediglich ScalaSTM um eine Persistenzschicht erweitert.

2.5.1 RadonSTM

RadonSTM [12] ist eine zeitstempelbasierte STM Implementierung in Scala. Dieses Projekt basiert auf der initialen Idee von Spiewak [13][14]. In diesen Blogeinträgen beschreibt Spiewak grundlegende Eigenschaften von STM und stellt eine rudimentäre Implementierung dieser bereit. Die beiden Features `retry` und `orElse` (siehe Abschnitt 2.2) werden dabei jedoch nicht erwähnt. Dennoch unterstützt RadonSTM `retry`, wohingegen `orElse` nicht verfügbar ist. Transaktionen werden bei RadonSTM durch das Schlüsselwort *tran-*

sactional eingeleitet. Variablen die vom STM verwaltet werden sollen, müssen bei dieser Implementierung von Typen `Ref` sein. Abbildung 13 zeigt den Aufbau einer Transaktion. Die verwaltete Ressource `ref` wird in diesem Beispiel mit dem Wert 100 initialisiert. Hierbei wird der Typ automatisch erkannt. Innerhalb des `transactional` Blocks ist der Zugriff auf diese Variable möglich. Updates können dabei entweder durch die Verwendung der `set`-Methode oder durch den Einsatz des `:=` Operators durchgeführt werden. Der lesende Zugriff erfolgt mittels der `get`-Methode oder dem `unarray` Operator.

```
val ref = new Ref(100)

transactional {
  ref.set(200) or ref := 200
  println(ref.get) or println(!ref)
}
```

Abbildung 13: RadonSTM Syntax

Neben der eben vorgestellten Steuerung von Transaktionen erlaubt dieses STM die explizite Zerstörung von Referenzen und besitzt zudem eine transaktionale Schleife (Abbildung 14). Diese führt eine Transaktion solange aus, bis die Abbruchbedingung erfüllt ist.

```
transactionalWhile(buffer.nonEmpty) {
  buffer.pop
}
```

Abbildung 14: Transaktionale Schleife

Zudem unterstützt RadonSTM die asynchrone Ausführung von Transaktion. Hierdurch wird vermieden, dass der Hauptthread geblockt wird und dafür gesorgt, dass alle Änderungen im Hintergrund vollzogen werden.

Ein weiteres Alleinstellungsmerkmal ist die Unterstützung von Propagationen. Inspiriert von EJB Propagationen, erlaubt dieses Feature die Kennzeichnung von Transaktionen. Dies gestattet eine explizite Steuerung der Transaktionen. Wie im Beispiel 15 illustriert, kann die Propagation eingesetzt werden, um Transaktionen zu verschachteln. Dabei gilt, dass die gesamte Transaktion nur gelingt, falls die eigentliche Transaktion und alle Subtransaktionen erfolgreich ausgeführt werden.

Ebenfalls in diesem Beispiel zu sehen ist die Propagation *requiresNew*. Diese sorgt dafür, dass Änderungen in einer neuen, von der äußeren unabhängigen, Transaktion ausgeführt

werden. Dabei wird die äußere Transaktion solange pausiert, bis die Ausführung der Inneren abgeschlossen ist. Scheitert die innere Transaktion, so hat dies keinerlei Einfluss auf die Äußere. Eine Aufzählung der EJB Propagationen und deren Bedeutung ist hier zu finden [15].

```
val ref = new Ref(100)

transactional {
  transactional(nested) {
    ref := 200
  }
  transactional(requiresNew) {
    ref := !ref + 100
  }
  println(!ref)
}
```

Abbildung 15: Verschachtelte Transaktionen

2.5.2 MUTS

Ein weiteres, in Scala verfügbares, STM ist MUTS (Manchester University Transactions for Scala)[6]. Die Funktionalität wird hierbei durch die Kombination von zwei Techniken erbracht. Zum einen wird der Parser des Scala Compilers erweitert. Zum anderen findet ein Bytecode Rewriting beim Laden einer Klasse statt.

Die Modifikation des Scala Parsers besteht darin, dass die drei Schlüsselworte *atomic*, *retry* und *orElse* akzeptiert und als Blöcke interpretiert werden. Bevor die Blöcke im Abstract Syntax Tree (AST) gespeichert werden, wird der Quellcode um zusätzliche Instruktionen erweitert. Da das Parsen zu Beginn des Kompiliervorgangs geschieht, ist die Verwendung der Scala Syntax noch möglich.

Das Snippet 16 veranschaulicht, welche Instruktionen bei der Erweiterung des atomic Blocks hinzugefügt werden. Wie dem Snippet entnommen werden kann, wird ein Kontext sowie ein Backup für alle Variablen hinzugefügt. Der Kontext wird eingesetzt, um alle transaktionsbezogenen Informationen zu verwalten. Diese Informationen werden wiederum benötigt, um Konflikte zu erkennen oder einen Commit durchzuführen.

Die Ausführung der gegebenen Funktion (Body) geschieht innerhalb eines try-catch-Blocks, welcher drei Catch-Anweisungen besitzt. Diese erfüllen folgende Funktionalität:

1. Die erste Exception **TransactionArea** ist ein Marker. Dieser kennzeichnet einen transaktionalen Bereich und wird beim Bytecode Rewriting verwendet, um diese Regionen zu identifizieren.
2. **TransactionException** fängt alle Exceptions, welche durch den Transaktionsmechanismus selbst geworfen werden. Diese Exception wird verwendet, um gescheiterte Transaktionen abubrechen und eine Neuausführung einzuleiten.
3. Die letzte Catch-Anweisung dient dazu alle Exceptions zu behandeln, die durch den Anwender geworfen werden. Sollte die Transaktion dennoch gelingen, wird die Exception propagiert. Andernfalls wird ein Rollback durchgeführt.

```

    val context$TM = eu.teraflux.uniman.transactions.TM.beginTransaction();
    var transaction_Variables_Backup = transaction_Variables

    try {
      Body
      if(!eu.teraflux.uniman.transactions.TM.commit(context$TM))
        transaction_Variables = transaction_Variables_Backup
    } catch {
      case (e @ ( : eu.teraflux.uniman.transactions.TransactionArea)) => ()
      case (e @ ( : org.deuce.transaction.TransactionException)) => ()
      case (e @ ) =>
        if(eu.teraflux.uniman.transactions.TM.commit(context$TM))
          throw e
        else
          transaction_Variables = transaction_Variables_Backup
    }
  }
}

```

Abbildung 16: MUTS Transformation eines atomic-Blocks

Durch die Parsererweiterung werden sowohl die Kontrollstrukturen als auch das Handling der Methodenvariablen hinzugefügt. Dies stellt einen Großteil der benötigten Funktionalität dar.

Die Handhabung des Zugriffs auf die Klassenvariablen, wird bei diesem STM durch das Bytecode Rewriting ergänzt. Hierfür werden beim Laden einer Klasse alle Methoden dupliziert. Die Duplikate besitzen einen zusätzlichen Parameter, welcher die Übergabe des Kontextes ermöglicht. Die duplizierten Methoden werden zudem dahingehen angepasst, dass jeglicher Zugriff auf die Klassenvariablen durch den Aufruf des Kontextes ersetzt wird, damit dieser jeden Schreib- und Lesezugriff aufzeichnen kann. Zuletzt werden die Methodenaufrufe angepasst, die innerhalb der duplizierten Methode stattfinden. Dabei wird der Kontext als zusätzlicher Parameter übergeben. Die duplizierten Methoden werden immer dann aufgerufen, wenn eine Transaktion ausgeführt wird. Nichttransaktionaler Code wiederum verwendet die unangepassten Methoden.

Eine modifizierte Version von DEUCE[7] ist für das Bytecode Rewriting zuständig. Dies hat den Vorteil, dass diese Komponente bereits gut getestet ist und eine Vielzahl unterschiedlicher STM Algorithmen unterstützt.

2.5.3 ScalaSTM

Inspiziert von den STMs in Haskell[8] und Clojure[16] ist ScalaSTM eine leichtgewichtige, in sich geschlossene STM Bibliothek, welche von der Scala STM Expert Group entwickelt und bereitgestellt wird. Dieses STM ist seit Scala 2.9 als STM Standardimplementierung für Scala anerkannt [17].

Der Aufbau von ScalaSTM ist klar strukturiert und untergliedert sich in drei Komponenten (Abbildung 17), die auch von Goodman et al. [18] beschrieben werden.

1. Das Frontend ist jene Komponenten, welche die nutzbaren Features für den Anwender bereitstellt (Abschnitt 2.5.3.1).
2. Das boundary Interface fungiert als Zwischenschicht, welche vom Backend implementiert und vom Frontend aufgerufen wird. Der Nutzen dieser Schicht besteht darin, dass dadurch die Bereitstellung mehrere Backends ermöglicht wird (Abschnitt 2.5.3.2).
3. Das Backend ist für die Verwaltung und Steuerung der Transaktionen verantwortlich. Dies umfasst die Protokollierung aller Schreib- und Leseaktionen, die Konflikterkennung sowie deren Beseitigung und das Ausführen eines Commits (Abschnitt 2.5.3.3).

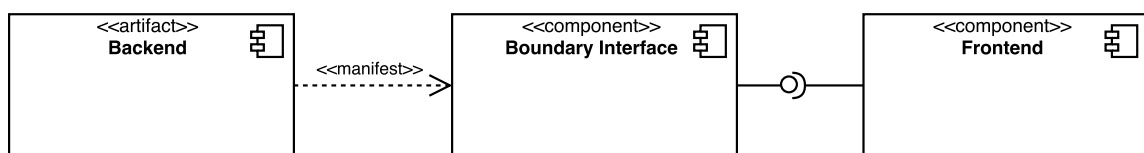


Abbildung 17: Aufbau von ScalaSTM

2.5.3.1 ScalaSTM Frontend

Das ScalaSTM Frontend unterstützt alle in Abschnitt 2.2 vorgestellten Features. Diese sind als Closures umgesetzt. Bevor die Funktionsweise der Features näher betrachtet wird, muss zuerst geklärt werden, wie die Deklaration der vom STM verwalteten Variablen geschieht. Wie im Beispiel 18 abzulesen ist, genügt der Aufruf des Ref Objekts und die Übergabe eines initialen Wertes. ScalaSTM erkennt den Typen dabei automatisch und instanziiert die Variable.

Der Zugriff auf diese Variablen erfolgt innerhalb atomarer Blöcke die vom Typen `Txn => Z` sind. Die transaktionale Ausführung dieser Blöcke wird wiederum durch den Aufruf der `atomic` Methode des `package`-Objekts¹ eingeleitet. Diese Methode besitzt zwei Parameter, welche zum einen die auszuführende Funktion und zum anderen ein Kontextobjekt sind. Während die auszuführende Funktion explizit übergeben wird, erfolgt die Übergabe des Kontextes (*txn*) implizit. Die Prüfung, ob ein impliziter Parameter übergeben worden ist, erfolgt bereits beim Kompilieren. Somit wird der Kontext statisch an den Ref-Aufruf gebunden, wodurch es zu Performancevorteilen gegenüber eines dynamischen Kontextlookups kommt.

```
val number = Ref(-10)

def absoluteValue(): Int = {
  atomic { implicit txn =>
    if (number() < 0) retry
    number()
  } orAtomic { implicit txn =>
    -number()
  }
}
```

Abbildung 18: ScalaSTM Syntax

2.5.3.2 ScalaSTM Boundary Interface

Das boundary Interface von ScalaSTM wird durch das Trait `TnxExecutor` repräsentiert. Implementierungen dieses Traits sind für die transaktionale Ausführung der atomaren Blöcke verantwortlich. Dieses Trait gibt vor, welche Features implementiert werden müssen und welche Operation verwendet wird, um eine Referenz zu aktualisieren. Zudem

¹Package Objekte werden in Scala verwendet, um häufig verwendete Funktionen und Methoden zu deklarieren. Diese können innerhalb des Packages ohne Import oder Namensqualifikation verwendet werden.

schreibt das Trait die Implementierung verschiedener Handler vor, die für die Konfliktbehebung verantwortlich sind.

Der Aufruf dieses Traits ist lediglich dem Frontend gestattet.

2.5.3.3 ScalaSTM Backend

Das Backend von ScalaSTM besteht aus der Referenzimplementierung CCSTM[5], welches im Rahmen der Dissertation von Bronson entwickelt wurde. CCSTM ist ein referenzbasiertes STM, welches ausschließlich in der Programmiersprache Scala geschrieben ist.

CCSTM unterstützt sowohl die starke als auch schwache Isolation der Referenzen. Während alle Zugriffe innerhalb atomarer Blöcke einer starken Isolation unterliegen, Änderungen also erst nach dem Commit nach außen sichtbar werden, ist ein Zugriff außerhalb dieser Blöcke durch die gebundene View Ref.View möglich. Zudem bietet CCSTM zusätzliche Möglichkeiten Aktionen mit Nebeneffekten innerhalb von Transaktionen zu handhaben. Hierfür besitzt dieses Framework eine umfangreiche Palette an Lifecycle Callbacks, die die Implementierung unterschiedlicher Strategien gestatten.

Die Versionsverwaltung und Konflikterkennung von CCSTM basieren auf SwissTM[10]. Das Versionsmanagement ist lazy. Die Schreibberechtigungen werden eager erworben. Zusätzlich verwendet dieses STM einen globalen Timestamp der 51 Bits lang ist, weshalb ein Counteroverflow theoretisch ausgeschlossen werden kann.

Der Einsatz eines Contention Management Schemas verhindert, dass langlaufende Transaktionen verhungern (Starvation). Dabei wird jeder Transaktion eine zufällige Priorität zugewiesen, wodurch die Behebung von Schreib/Schreib-Konflikten vereinfacht wird. Ein zusätzlicher Mechanismus, der eingesetzt wird um das Starvation Problem zu lösen, versetzt mehrfach gescheiterte Transaktionen in einen „Barging“ Modus. Diese Transaktionen erlangen dadurch frühzeitig alle Schreibberechtigungen, wodurch sichergestellt wird, dass diese abgeschlossen werden.

Ein weiteres Feature von CCSTM sind die sogenannten Wakeup Channels. Diese sorgen dafür, dass ein „busy waiting“ vermieden wird. Dieser Mechanismus wird verwendet, wenn eine Transaktion auf eine Schreibberechtigung warten muss oder das retry-Primitiv verwendet wird. Das STM erkennt dabei, dass die aktuelle Transaktion nicht weiter bearbeitet werden kann und pausiert diese. Gleichzeitig merkt sich CCSTM welche Bedingung zur Pausierung geführt hat und markiert diese. Wird eine Änderung der Bedingung bemerkt, führt dies wiederum dazu, dass eine Wiederausführung der entsprechenden Transaktionen eingeleitet wird.

2.6 Verwendete Concurrency Mechanismen in Scala

Zum Abschluss dieses Kapitels wird die Evaluierung, die im Rahmen dieser Arbeit durchgeführt worden ist, vorgestellt. Die Evaluierung soll Aufschluss darüber geben, welche Concurrency Mechanismen allgemein und STM im speziellen in Scala Projekten eingesetzt werden. Die Basis für diese quantitative Analyse bilden Scala Projekte, die auf Github² veröffentlicht sind. Das für diesen Zweck entwickelte Tool ist öffentlich zugänglich und kann hier [19] heruntergeladen werden.

Github selbst besitzt eine RESTful API³, welche eine Volltextsuche nach Repositories sowie innerhalb dieser ermöglicht. Neben der Eingabe simpler Suchbegriffe, bietet die API weitere Funktionen, wie das Filtern nach Programmiersprachen oder das Einschränken der Suche auf einen bestimmten Zeitraum. Der Zugriff auf die API erfolgt durch den Einsatz der Java Bibliothek Github API[20]. Diese ist instand, Suchanfragen zu erzeugen, die JSON-Antwort zu parsen und in Java Objekte umzuwandeln.

Da die verwendete API keine Kombination der Suche nach Repositories und dem Code innerhalb dieser gestattet, muss die Analyse in drei Teilschritten untergliedert werden. Der erste Schritt umfasst die Suche nach allen Repositories, in denen Scala eingesetzt wird (Abschnitt 2.6.1). Darauf folgend wird untersucht, welche Concurrency Mechanismen in den einzelnen Repositories zum Einsatz kommen (Abschnitt 2.6.2). Zum Abschluss dieses Abschnitts wird erläutert, welche Schritte notwendig waren, um die gesammelten Daten zu aggregieren. Zudem wird das Ergebnis der Evaluation präsentiert.

2.6.1 Suche nach Scala Repositories

Zum Zeitpunkt der Evaluation gab es über 50.000 öffentliche Repositories, bei denen die Programmiersprache Scala eingesetzt wird. Aufgrund der Restriktion, dass ein Suchergebnis nur bis zu 1000 Resultate pro Suchanfrage liefert, sind zusätzliche Kriterien nötig, um jedes einzelne Repository zu finden. Wie bereits erwähnt, erlaubt die Github API den Einsatz von Filtern, die eine Eingrenzung des Suchraums erlauben. Nach der Analyse der angebotenen Filter, kommt nur eine Beschränkung auf ein vorgegebenes Zeitintervall in Frage. Prinzipiell kommen hierfür zwei Parameter in Frage. *Pushed* ermöglicht die Eingrenzung der Ergebnisse basierend auf dem letzten Update, wohingegen *created* das Erstellungsdatum berücksichtigt. Da das Erstellungsdatum einen fixen Zeitpunkt repräsentiert und die Ausführung der Analyse mehrfach möglich sein soll, ohne dass sich die Zwischenergebnisse verändern, wird dieser Parameter verwendet.

²www.github.com

³<https://api.github.com/>

Nach Festlegung auf den zu verwendenden Parameter, muss eine Intervallgröße bestimmt werden. Die Ausführung mehrerer Tests hat ergeben, dass eine Intervallgröße von sieben Tagen optimal ist. Ist das Intervall kleiner, werden mehr Anfragen an den RESTful Service gesendet, was wiederum zu einer längeren Laufzeit des Programms führt. Wird ein zu großes Intervall gewählt, kann es dazu kommen, dass das Resultat der Anfrage über 1000 Ergebnisse besitzt und daher unvollständig ist.

Das letzte Kriterium, welches bei der Repository Suche zum Einsatz kommt, ist der Zeitraum innerhalb dessen die einzelnen Intervalle liegen können. Da das erste Scala Repository am 02.04.2008 erstellt worden ist, erstreckt sich der gewählte Zeitraum vom 01.04.2008 und bis zum 31.09.2015.

Aus diesen Kriterien ergibt sich folgendes Schema für die Aufruf URLs:

<https://api.github.com/search/repositories?q=language:Scala+created:FROM..TO>

Wie bereits erwähnt, übernimmt die eingesetzte Bibliothek die Erzeugung der URLs sowie das Parsen der JSON-Antwort.

Das Ergebnis dieser Suche wird in einer XML Datei gespeichert. Dies gestattet die Verwendung der Resultate in allen darauffolgenden Analyseschritten. Die gespeicherten Informationen umfassen dabei den Namen des Repository, dessen URL sowie das Erstellungsdatum.

2.6.2 Durchsuchen der Repositories

Das Durchsuchen der einzelnen Repositories nach Concurrency Mechanismen erfolgt ebenfalls unter Einsatz der Java Bibliothek Github API. In diesem Schritt wird über das Ergebnis des ersten Teilschritts iteriert, um an die Namen der Repositories zu gelangen. Dieser wird benötigt, um die Inhaltssuche auf ein Repository einzuschränken. Eine zusätzliche Einschränkung besteht darin, dass nur Scala Klassen durchsucht werden. Das letzte Kriterium ist der Suchbegriff selbst. Aus diesen Kriterien ergibt sich folgendes Schema für die Aufruf URLs:

[https://api.github.com/search/code?q=\"searchTerm\"+extension:SCALA+repo:NAME](https://api.github.com/search/code?q=\)

Die Auswahl eines geeigneten Suchbegriffs erweist sich als schwierig, da hierbei mehrere Dinge zu beachten sind. Wird nur der Name des jeweiligen Features als Suchbegriff gewählt, ist es bei Futures beispielsweise nicht möglich zu unterscheiden, ob es sich um das Java oder Scala Derivat handelt. Wird der vollqualifizierte Namen verwendet, ist eine Unterscheidung nach Programmiersprache möglich, sorgt jedoch dafür, dass die Anzahl der Suchanfragen steigt. Ebenfalls problematisch hierbei sind Scalas Multiimport-

Anweisungen⁴ und Aliase⁵. Diese Mechanismen führen dazu, dass die Suche nach dem vollqualifizierten Namen keine Ergebnisse liefert, da die Importanweisung nicht mehr zusammenhängend ist.

Da der Packagename eindeutig ist und weder vom Multiimport noch von Aliasen beeinflusst wird, erscheint die Verwendung des Packages als Suchbegriff am geeignetsten. Der Einsatz dieses Suchbegriffs führt dabei zur Verringerung der benötigten Suchanfragen und einer gleichzeitigen Erhöhung der gefundenen Mechanismen.

Aber auch dieser Ansatz weist Schwächen auf. Da die Suchergebnisse bei dieser Vorgehensweise nicht mehr eindeutig sind, werden zusätzliche Bearbeitungsschritte benötigt, um die Resultate zu zerlegen und die einzelnen Mechanismen herauszukristallisieren. Hierzu werden alle Klassen, die von der API zurückgeliefert werden, heruntergeladen und anschließend untersucht. Reguläre Ausdrücke helfen dabei, alle Zeilen zu matchen, in denen der Packagename vorkommt. Es sei angemerkt, dass Wildcard-Imports ignoriert werden, da hierbei nicht bestimmt werden kann, welches Feature eingesetzt wird. Das Ergebnis dieses Teilschritts wird ebenfalls in einer XML-Datei gespeichert. Das XML weist dabei folgende Struktur auf:

- Das Rootelement besitzt eine Liste von Repositories, in denen ein oder mehrere Concurrency Mechanismen verwendet werden.
- Jedes Repository-Element besitzt eine List von Klassen, in denen diese Mechanismen eingesetzt werden.
- Jede Klasse besitzt eine Liste von Textzeilen, die auf die Verwendung eines Mechanismus hindeuten.

⁴Scala erlaubt den Import mehrerer Klassen durch eine Importanweisung. Hierzu werden zuerst der Packagename und darauffolgend alle zu ladenden Klasse innerhalb geschweifter Klammern angegeben, z.B. `import scala.util.{Try, Success, Failure}`

⁵Ein Alias erlaubt die Umbenennung eines Klassennamens, der daraufhin anstatt des eigentlichen Namens verwendet werden kann, z.B. `import scala.concurrent.{Future => ScalaFuture}`

2.6.3 Ergebnis

Abschließend sollen die Ergebnisse der Evaluation vorgestellt werden. Zuvor muss jedoch erläutert werden, wie die geparsten Textzeilen, die das Resultat des zweiten Teilschrittes darstellen, normalisiert werden.

Eine Normalisierung ist notwendig, weil die geparsten Zeilen nicht ohne weitere Aufbereitung aggregiert werden können, da diese Multiimports, Alias und Kommentare enthalten können. Deshalb müssen zuerst alle Anführungszeichen, Kommentare, Sonderzeichen, Schlüsselwörter und Leerzeichen entfernt werden. Zudem müssen Multiimports und Alias gesondert behandelt werden. Die Normalisierung dieser beiden Importstatements umfasst zwei Schritte. Zuerst wird das Packagepräfix extrahiert und zwischengespeichert. Danach wird jeder Klassenname und jede Aliasanweisung innerhalb der geschweiften Klammern ausgelesen und mit dem Präfix konkateniert. Dies führt zu einer Erhöhung der Anzahl der Einträge, die für die Aggregation verwendet werden.

Nach der Normalisierung der Daten werden diverse Matcher eingesetzt. Diese prüfen, ob eine der geparsten Zeilen zu einem der vorgegebenen Concurrency Mechanismen passt. Im Falle eines Treffers wird der gefundene Mechanismus in einem Set zwischengespeichert. Der Grund hierfür ist, dass das Auftreten eines Mechanismus nur einfach gezählt wird und daher nur einmal pro Repository auftreten darf. Nach der Berechnung der Daten für jedes Repository, werden die gesammelten Werte aggregiert und darauffolgend gespeichert. Der Pseudocode, den Abbildung 19 zeigt, illustriert den eben beschriebenen Prozess.

```
for ( Repository repository : Repositories ) {  
    Set<String> repositoryMatches = getOtherMatches();  
    for ( String line : repository.getLines ) {  
        String filtered = _filterChain.doFilter(line);  
        if ( null != filtered ) {  
            List<String> matched = _matcherChain.doMatch(filtered);  
            if( null != matched ) {  
                repositoryMatches.addAll(matched);  
            }  
        }  
    }  
    addRepositoryMatchesToResult(repository, repositoryMatches);  
}
```

Abbildung 19: Aggregation der Ergebnis

Das Ergebnis der Evaluierung kann der Tabelle 1 entnommen werden. Die Evaluation umfasst dabei 54267 Scala Repositories, die auf Verwendung von Concurrency Mechanismen untersucht worden sind. Hierbei kann es vorkommen, dass mehrere Mechanismen in demselben Projekt verwendet werden. Der am häufigsten eingesetzte Mechanismus ist das Aktorenmodell, welches durch die Akka Bibliothek bereitgestellt und in circa 13,6% aller Projekte eingesetzt wird. Scalas Futures werden ebenfalls häufig verwendet und sind in circa 11,9% aller Projekte vorzufinden. Am dritthäufigsten werden Monitore eingesetzt. Diese werden, wie auch in Java, mit dem Schlüsselwort `synchronized` eingeleitet. In circa 1% aller Repositories sind Scalas Promises sowie Javas Futures, Semaphoren und Locks vorzufinden. Mit 0,35% liegt STM weit abgeschlagen auf dem vorletzten Platz, wobei nur Scalas Lockimplementierung (0,16%) seltener verwendet wird.

Mechanismus	Anzahl Einsatz in Repos	Prozentualer Einsatz
Aktoren	7387	13,61%
Future (Scala)	6470	11,92%
Monitor	3416	6,29%
Promise (Scala)	693	1,28%
Locks (Java)	544	1,00%
Future (Java)	489	0,90%
Semaphore (Java)	301	0,55%
ScalaSTM	193	0,36%
Locks (Scala)	90	0,17%
Repositories Gesamt	54267	100%

Tabelle 1: Einsatz der einzelnen Mechanismen sortiert nach Häufigkeit

Aus dem ersten Ergebnis geht nicht hervor, inwieweit die dort aufgeführten Mechanismen kombiniert eingesetzt werden. Aufgrund dessen findet eine zweite Evaluierung statt. Diese hat das Ziel zu ergründen, in welchen Kombinationen die vorgestellten Features auftreten. Hierfür können die vorliegenden Daten verwendet werden, wobei keine Anreicherung dieser vonnöten ist.

Da die Ergebnismenge deutlich höher ist als bei der letzten Evaluation, wird die Ergebnisliste gekürzt. In der Liste enthalten sind nur noch Kombinationen, die in über 100 Repositories vorzufinden sind. Hiervon ausgenommen sind alle Kombinationen, in denen STM vorkommt. Die Untergrenze hierfür beträgt 10.

Die Repräsentation der Ergebnisse erfolgt auch hier tabellarisch (Tabelle 2). Wie zu

erwarten, besteht die am häufigsten eingesetzte Kombination aus Aktoren und Scalas Futures (4,19%). Dies ist deshalb nicht überraschend, da die beiden Mechanismen bereits einzeln sehr häufig vorkommen. Am zweithäufigsten werden Aktoren, Monitore und Scalas Futures miteinander kombiniert. Die beste Kombination in der auch STM vorkommt, besteht aus den Mechanismen Aktoren, Monitoren, Scala's Futures und Promises. Diese Kombination wird gerade einmal in 0,05% aller Fälle verwendet. Ebenfalls kombiniert wird STM mit Aktoren (0,03%), Scala's Futures (0,03%) und Monitoren (0,02%).

Die beiden Analysen zeigen, dass Aktoren der beliebteste Concurrency Mechanismus in Scala sind. Zudem ist auffällig, dass Aktoren mit jedem anderen Mechanismus kombiniert werden können. STM hingegen ist kaum von Bedeutung und wird nur äußerst selten verwendet.

Mechanismus	Anzahl Einsatz in Repos	Prozentualer Einsatz
Aktoren, Future (Scala)	2273	4,19%
Aktoren, Monitor, Future (Scala)	372	0,69%
Monitor, Future (Scala)	299	0,55%
Actor, Monitor	220	0,41%
Actor, Future (Scala), Promise (Scala)	174	0,32%
Future (Scala), Promise (Scala)	122	0,22%
Monitor, Locks (Java)	105	0,19%
Aktoren, Monitor, ScalaSTM, Future (Scala), Promise (Scala)	29	0,05%
Aktoren, Monitor, ScalaSTM, Future (Scala)	25	0,05%
Actor, ScalaSTM	17	0,03%
ScalaSTM, Future (Scala)	14	0,03%
Monitor, ScalaSTM	12	0,02%
Repositories Gesamt	54267	100%

Tabelle 2: Kombiniertes Auftreten der Mechanismen

3 Futures mit STM

Im dritten Kapitel wird die Implementierung der Futures und Promises vorgestellt. Das Resultat eines Futures wird dabei von STM verwaltet. Zudem wird bei der Realisierung das funktionale Programmierparadigma verwendet.

Da in Scala mehrere STM Implementierungen zur Verfügung stehen, wird zuerst erläutert, welche Implementierung eingesetzt werden soll, um die Futures und Promises zu realisieren (Abschnitt 3.1). Nach der Auswahl des STMs wird die eigentliche Implementierung der beiden Mechanismen beschrieben (Abschnitt 3.2). Dies umfasst zum einen die Datenstrukturen und Hilfsfunktionen, die hierfür benötigt werden, und zum anderen alle anwendbaren Funktionen.

3.1 Auswahl des STMs

Da Scala, wie bereits in Abschnitt 2.5 dargelegt, über mehrere STMs verfügt, muss eines dieser STMs vor Beginn der Umsetzung ausgewählt werden.

Als erstes soll RadonSTM betrachtet werden. Diese Implementierung verfügt über eine dürftige Beschreibung, da es weder wissenschaftliche Veröffentlichungen noch eine ausführliche Dokumentation besitzt. Zudem unterstützt dieses STM nicht alle Features. Zwar besitzt RadonSTM eine retry-Funktion, jedoch wird nirgends beschrieben wie diese angewendet wird. OrElse wird nicht unterstützt.

Ein weiteres Defizit von RadonSTM sind die dort eingesetzten Propagationen. Diese erschweren sowohl die Lesbarkeit des Codes als auch die Verwendung des STM, da die Semantik jeder Propagation bekannt sein muss. Ebenfalls problematisch erscheint, dass es keinerlei Benchmarks gibt, die die Performance dieses STM belegen.

Das zweite STM, welches betrachtet werden soll, ist MUTS. Diese Implementierung basiert auf dem gut getesteten Framework DEUCE. Durch das Bytecode Rewriting ist es möglich externe Bibliotheken bedenkenlos einzusetzen. Ebenfalls positiv zu bewerten ist das Vorhandensein eines wissenschaftlichen Artikels, der erläutert welche Designentscheidungen getroffen worden sind, welche Features eingesetzt werden können und wie diese implementiert sind. Zudem ermöglicht dieses STM den Gebrauch aller vorgestellten Features.

Die Nachteile von MUTS sind, dass keinerlei Benchmarks zur Verfügung stehen und daher kein Vergleich mit anderen Concurrency Mechanismen oder STMs möglich ist. Hierbei wird argumentiert, dass DEUCE selbst über eine Vielzahl von Benchmarks verfügt. Das Problem dabei ist, dass diese ausschließlich in Java geschrieben sind und daher keine Aussagen über die Performance in Scala gemacht werden kann. Ein weiterer Nachteil von MUTS ist, dass keinerlei Weiterentwicklung stattfindet und die bereitgestellte Compilererweiterung nur in Kombination mit Scala 2.7 eingesetzt werden kann. Da zum Erstellungszeitpunkt dieser Arbeit Scala 2.11 verfügbar ist, erscheint die Verwendung einer veralteten Version nicht sinnvoll, da etliche Features hinzugefügt und entfernt worden sind.

ScalaSTM ist die letzte Implementierung, die für die Umsetzung der Aufgabe in Betracht gezogen wird. Dieses STM verfügt über eine umfangreiche Dokumentation, die aus einer wissenschaftlichen Publikation sowie Tutorials und diversen Anwendungsbeispielen besteht. Zudem unterstützt dieses STM alle vorgestellten Features. Ein weiterer Vorteil ist die einfache Handhabung dieses STMs. Ebenfalls vorteilhaft ist, dass ScalaSTM über Benchmarks verfügt, wodurch die Leistungsfähigkeit der Implementierung untermauert wird. Zuletzt wird angemerkt, dass diese Bibliothek regelmäßig gewartet wird, wodurch die Kompatibilität mit dem neuesten Scala Release garantiert ist.

Da ScalaSTM das umfangreichste aller STMs ist, eine gute Dokumentation besitzt und zudem alle STM Features unterstützt, wird dieses STM für die Implementierung der Futures und Promises verwendet.

3.2 Implementierung der Futures und Promises

Nach der Auswahl des STMs, welches für die Verwaltung des Ergebnisses eingesetzt wird, erfolgt die Präsentation der Futures und Promises.

Hierzu wird zunächst erläutert, welche zusätzlichen Datenstrukturen und Hilfsfunktionen eingesetzt werden. Im darauffolgenden Abschnitt erfolgt die Beschreibung der Futures und Promises sowie der dazugehörigen Funktionen. Die Umsetzung selbst erfolgt in einem Package Objekt, wodurch der Zugriff auf die Funktionen vereinfacht wird. Der Quellcode ist auf Github veröffentlicht und kann hier [\[21\]](#) eingesehen und heruntergeladen werden.

3.2.1 Datenstrukturen und Hilfsfunktionen

Um das Ergebnis der Berechnung beziehungsweise den Zustand des Futures adäquat abzubilden, wird ein neuer Datentyp benötigt. Die Erzeugung eines solchen algebraischen Datentypen (ADT) erfolgt unter Einsatz von Traits sowie Case Klassen und Objekten. Das Trait `Result` (Abbildung 20) und dessen Subklassen repräsentieren das Ergebnis eines Futures. Der sealed Modifikator wird hierbei eingesetzt, um auszuschließen, dass dieses Interface von Dritten erweitert wird. Die Besonderheit des sealed Modifikators ist, dass anders als bei final Erweiterungen des Traits möglich sind. Dies ist allerdings nur innerhalb derselben Scala Klasse gestattet.

Da das Resultat eines Futures einen beliebigen Typen haben kann, ist das Trait zudem generisch. Durch das Plus vor der Typendeklaration wird festgelegt, dass der zugewiesene Typ kovariant sein muss. Ist der Typ invariant, kann keine Zuweisung eines konkreten Typen erfolgen. Weshalb dies vonnöten ist, wird im weiteren Verlauf dieses Abschnitts erörtert.

Das Trait selbst definiert drei Methoden, die von den Subklassen implementiert werden müssen. Die Methode `get` wird verwendet, um das gewrappte Ergebnis zu lesen. Die anderen beiden Methoden `isEmpty` und `isFail` gestatten die Prüfung, ob ein Ergebnis leer oder die Ausführung des Futures fehlgeschlagen ist. Ein leeres Ergebnis liegt immer dann vor, wenn die Berechnung noch nicht abgeschlossen ist.

```
sealed trait Result[+A] {  
  def get: A  
  def isEmpty: Boolean  
  def isFail: Boolean  
}
```

Abbildung 20: Algebraischer Datentyp Result

`Result` besitzt drei Subklassen, die zur Repräsentation des konkreten Zustands eines Futures genutzt werden. Zuerst werden die beiden Typen `Empty` und `Fail` vorgestellt. Beide Typen besitzen eine ähnliche Struktur, weshalb `Empty` stellvertretend für beide illustriert wird (Abbildung 21). Die Unterschiede werden textuell erörtert.

Der Case Modifikator wird bei beiden Typen verwendet, damit ein späteres Pattern Matching durchgeführt werden kann. Zudem verkörpern beide Typen ein leeres Ergebnis, weshalb deren Typ nicht generisch sondern `Nothing` ist. `Nothing` ist der Subtyp jedes Scala Objekts und signalisiert, dass kein normales Ergebnis zurückgeliefert wird, sondern mit einem Fehler gerechnet werden muss. Dieser Fehler ist im Falle von `Empty`

und Fail eine `NoSuchElementException`, die geworfen wird, wenn `get` aufgerufen wird. Der Grund weshalb `Result` kovariant sein muss, besteht darin, dass diese beiden Objekte das `Trait` erweitern und keinen generischen Typen besitzen sondern vom Typ `Nothing` sind. Wäre `Result` invariant, dann könnten `Empty` und `Fail` keiner Variable vom Typen `Result` zugewiesen werden, da der Compiler einen Typenfehler bemängeln würde. Neben den gleichartigen Merkmalen, besitzen `Fail` und `Empty` zwei Unterschiede. Zum einen ist die Fehlermeldung, die beim Aufruf von `get` ausgegeben wird unterschiedlich. `Empty` gibt die Fehlermeldung „`Empty.get`“ aus. Analog dazu wird die Meldung „`Fail.get`“ zurückgeliefert, wenn die `get` Methode des `Fail` Objekts aufgerufen wird. Zum anderen liefern beide Objekte bei der entsprechenden `isEmpty`, bzw. `isFail` Methode `true` zurück.

```
case object Empty extends Result[Nothing] {  
  def get = throw new NoSuchElementException("Empty.get")  
  override def isEmpty: Boolean = true  
  override def isFail: Boolean = false  
}
```

Abbildung 21: Empty und Fail

Die Klasse `Success` repräsentiert ein erfolgreich berechnetes Ergebnis. Wie die Illustration 22 zeigt, handelt es sich hierbei um eine finale Klasse. Auch hier wurde der `case` Modifikator verwendet, um ein späteres Pattern Matching zu ermöglichen. Zur Instanziierung eines `Success` Objektes muss das Resultat der Berechnung als Argument übergeben werden, welches wiederum durch Verwendung der `get` Methode abgefragt werden kann. Zudem wird beim Aufrufen von `isEmpty` und `isFail` `false` zurückgegeben.

```
final case class Success[T](value: T) extends Result[T] {  
  def get = value  
  override def isEmpty: Boolean = false  
  override def isFail: Boolean = false  
}
```

Abbildung 22: Success

Neben den drei vorgestellten Subklassen des Traits `Result` besitzt die Scala Klasse ein zusätzliches Begleitobjekt (companion object), welches denselben Namen wie das `Trait` besitzt (Abbildung 23). Dieses Objekt erfüllt mehrere Funktionen. Die `apply`-Funktion vereinfacht die Erzeugung eines Resultats. Falls die Berechnung fehlschlägt und der

Rückgabewert null ist, liefert diese Funktion Fail zurück. Anderenfalls wird Success zurückgegeben.

Die *empty*-Funktion wird für die Initialisierung des STM Referenzobjekts benötigt, weil die direkte Zuweisung des Empty Objektes zu einem Fehler beim Kompilieren führt, da ein falscher Typ erkannt wird. Um den Fehler zu beheben muss das Empty Objekt gecastet werden. Da der Einsatz einer Funktion eleganter ist als ein explizites Casten, wird diese fortan verwendet.

```
object Result {  
  def apply[A](x: A): Result[A] = if (x == null) Fail else Success(x)  
  def empty[A]: Result[A] = Empty  
}
```

Abbildung 23: Begleitobjekt des Result Traits

Zuletzt soll die Hilfsfunktion *forkIO* (Abbildung 24) präsentiert werden. Diese simple Funktion nimmt eine anonyme Funktion als Parameter entgegen und führt diese im Hintergrund aus. Hierzu wird ein neues Runnable erzeugt, welches die übergebene Funktion ausführt sobald die *run*-Methode aufgerufen wird. Um die exzessive Erzeugung von Threads zu vermeiden, wird ein Threadpool für die Ausführung der Runnables eingesetzt. Dies hat den Vorteil, dass Threads wiederverwendet werden können, wodurch die kostspielige Erzeugung neuer Threads entfällt. Der Threadpool selbst ist ein leicht modifizierter ForkJoinPool.

```
private def forkIO[A](body: => Unit) = {  
  val runnable = new Runnable {  
    override def run(): Unit = body  
  }  
  executor.prepare().execute(runnable)  
}
```

Abbildung 24: Die Hilfsfunktion *forkIO* erzeugt ein neues Runnable und weist dieses anschließend einem Threadpool zu.

3.2.2 Future

Ein Future ist eine simple Abstraktion eines Platzhalters, der das Ergebnis einer noch unbekannten oder unfertigen Berechnung darstellt. Bei diesen Berechnungen kann es sich beispielsweise um langwierige numerische Operationen oder Netzwerkaufrufe handeln. Die Berechnung oder das Warten auf das Ergebnis geschieht dabei asynchron, also im Hintergrund. Dies sorgt dafür, dass der Hauptthread nicht geblockt wird und seine eigentliche Arbeit fortsetzen kann.

Wird das Ergebnis der Kalkulation benötigt, kann geprüft werden, ob ein Resultat vorliegt. Ist die Berechnung abgeschlossen, kann das Ergebnis sofort gelesen werden. Ist die Kalkulation noch im Gange, besteht die Möglichkeit den Hauptthread solange zu blocken, bis das Resultat vorliegt. Zudem gilt, dass das Ergebnis nach Abschluss der Berechnung beliebig oft gelesen werden kann.

Ein Future wird bei dieser Implementierung durch die Klasse Future repräsentiert (Abbildung 25). Der Zugriffsmodifikator des Futures ist public. Der Grund hierfür ist, dass das Future zur Deklaration von Funktionsparameter oder Variablen eingesetzt werden kann. Zudem ist die Klasse final, wodurch keine Erweiterung dieser gestattet wird. Der bereits mehrfach vorgestellte Case Modifikator bewirkt, dass ein Matchen von Futures möglich ist. Zudem ist das Future generisch, da das Resultat prinzipiell jeden Typen annehmen kann.

Die Klasse selbst besitzt exakt ein Attribut, welches ein ScalaSTM Referenzobjekt ist und für die Verwaltung des Resultats eingesetzt wird. Bei der Erzeugung eines neuen Futures wird der Wert initial auf Empty gesetzt. Hierdurch wird signalisiert, dass die Kalkulation noch nicht beendet ist und kein Ergebnis vorliegt.

```
final case class Future[A]() {  
  val result: Ref[Result[A]] = Ref(Result.empty)  
}
```

Abbildung 25: Datenstruktur Future

Die Erzeugung eines neuen Futures geschieht unter Verwendung der *apply*-Funktion (Abbildung 26). Als Parameter wird eine Funktion mit generischem Rückgabewert erwartet. Der Typ des Rückgabewertes wiederum bestimmt den Typ des neuen Futures. Nach der Erzeugung eines neuen Future-Objektes wird *forkIO* aufgerufen, um die Kalkulation der übergebenen Funktion *func* von einem Hintergrundthread ausführen zu lassen. Anschließend wird das neue Future zurückgegeben.

Nach Beendigung der Berechnung, wird das bis dahin leere Referenzobjekt durch das Resultat der Kalkulation überschrieben. Hierzu wird die im letzten Abschnitt vorgestellte apply-Funktion des Result-Objektes verwendet. Sollte func null zurückliefern, wird das Resultat auf Fail gesetzt. Andernfalls wird das Ergebnis in einem Success gewrapppt. Nach

```
object Future {  
  def apply[A](func: => A): Future[A] = {  
    val future = new Future[A]  
    forkIO {  
      atomic { implicit txn => future.result() = Result(func) }  
    }  
    future  
  }  
}
```

Abbildung 26: Erzeugung eines neuen Futures

der Vorstellung des Datentyps Future, werden die Funktionen vorgestellt, die auf ein oder eine Menge von Futures angewendet werden können. Die erste dieser Funktionen ist *get*. Diese Funktion wird eingesetzt, um das Ergebnis eines Futures zu erlangen. Sofern das Resultat bereits vorliegt, wird dieses sofort zurückgegeben. Ist die Berechnung jedoch noch im Gang, blockiert die Methode den Hauptthread solange, bis die Operation beendet ist und das Ergebnis zurückgeliefert werden kann.

Get erwartet ein Future als Parameter und gibt ein Option zurück. Wie im Codebeispiel 27 abgebildet, wird Pattern Matching verwendet, um den Zustand des Ergebnisses zu überprüfen. Dies geschieht innerhalb eines atomaren Blocks, da das Lesen der Referenz sonst nicht möglich ist.

Ist das Resultat Empty, bedeutet dies, dass die Berechnung noch im Gang ist. Deshalb wird in diesem Fall ein retry ausgeführt, wodurch das STM automatisch wartet bis der Zustand des Resultats sich verändert und eine erneute Prüfung durchgeführt wird. Ist

```
def get[A](future: Future[A]): Option[A] =  
  atomic { implicit txn =>  
    future.result() match {  
      case Empty => retry  
      case Fail => return None  
      case Success(x) => return Some(x)  
    }  
  }
```

Abbildung 27: Lesen des Ergebnisses

das Resultat vom Typ Fail, wird None zurückgegeben. Hierdurch wird signalisiert, dass die Berechnung fehlgeschlagen und kein Resultat vorhanden ist. Im Falle einer erfolgreichen Kalkulation wird Some zurückgegeben. Wie auch Success ist dies ein Wrapper, welcher das eigentliche Resultat nochmals kapselt.

Die nächste Funktion die betrachtet wird ist *fail*. Diese Funktion ist parameterlos und gibt ein Future zurück. Hierzu wird zuerst ein neues Future erzeugt. Anschließend wird das Resultat auf Fail gesetzt und das Future zurückgegeben. Diese Funktion findet dann Anwendung, wenn bereits vor der geplanten Kalkulation klar ist, dass diese niemals erfolgreich sein kann.

Angenommen man möchte eine Serveranfrage asynchron durchführen, um das Ergebnis später zu verwenden. Vor dem Senden der eigentlichen Anfrage, prüft man ob der Server erreichbar ist. Wird dabei festgestellt, dass dies nicht der Fall ist, kann fail genutzt werden ohne zusätzliche Aktionen auszuführen.

```
def fail[A]() : Future[A] = {  
  val future = new Future[A]  
  atomic { implicit tnx => future.result() = Fail }  
  future  
}
```

Abbildung 28: Erzeugung eines gescheiterten Futures

Die Funktion *onSuccess*, welche in Abbildung 29 zu sehen ist, ist ebenfalls asynchron und blockt den Hauptthread nicht. Diese Funktion kann immer dann verwendet werden, wenn das Resultat des Futures nur einmalig benötigt wird. Dies ist beispielsweise dann der Fall, wenn das Resultat lediglich in eine Datenbank geschrieben werden soll.

OnSuccess besitzt zwei Parameter und gibt nichts zurück. Da die Ausführung des Callbacks vom Resultat des Futures abhängt, wird der Hintergrundthread solange geblockt, bis ein Ergebnis vorliegt. Besitzt das Future kein Ergebnis, geschieht nichts und der Thread terminiert. Ist dieses Ergebnis wiederum valide, besitzt also ein Resultat, kann

```
def onSuccess[A](future: Future[A], callback: (A) => Unit) = forkIO {  
  get(future) match {  
    case None => ()  
    case Some(x) => callback(x)  
  }  
}
```

Abbildung 29: Die Funktion onSuccess erlaubt die Ausführung eines Callbacks, falls das Future gelingt

das übergebene Callback ausgeführt werden.

Das Gegenstück zu `onSuccess` ist `onFailure`. Da die beiden Funktionen nahezu identisch sind, wird auf die Visualisierung von `onFailure` verzichtet und lediglich beschrieben welche Unterschiede bestehen. Wie auch `onSuccess` erwartet `onFailure` einen Callback als Parameter. Dieser Callback ist jedoch anders wie bei `onSuccess` parameterlos und wird im Falle eines Scheiterns des Futures ausgeführt.

`WaitForSuccess` ähnelt `onSuccess`, führt jedoch ein `retry` aus anstatt zu terminieren, wenn die Ausführung des Futures fehlschlägt. Dabei wird keine Garantie gegeben, dass der Thread jemals terminiert. Diese Funktion kann immer dann eingesetzt werden, wenn die Berechnung des Futures solange ausgeführt wird bis es zum Erfolg kommt oder ein anderer Thread in der Lage ist das Resultat direkt zu überschreiben.

```
def waitForSuccess[A](future: Future[A], callback: (A) => Unit) = {
  forkIO {
    atomic { implicit txn =>
      future.value() match {
        case Success(x) => callback(x)
        case _ => retry
      }
    }
  }
}
```

Abbildung 30: `WaitForSuccess` versucht die Ausführung des Callbacks zu erzwingen und wartet deshalb bis das Future ein Resultat besitzt

`FollowedBy` kann eingesetzt werden, um Folgeaktionen zu definieren. Dabei wird eine Funktion erwartet, die auf das Ergebnis des übergebenen Futures angewendet wird. `FollowedBy` gibt ein neues Future zurück. Das Resultat des neuen Futures ist das Ergebnis der Funktion. Es ist zudem prinzipiell möglich mehrere voneinander abhängige Aufgaben nacheinander auszuführen. Ein Fehlschlag in der Kalkulationskette führt jedoch dazu, dass die gesamte Ausführung fehlschlägt und kein Resultat zurückgeliefert wird.

Betrachtet man die Funktion selbst (Abbildung 31), sieht man die eben beschriebenen Parameter. Da sich der Typ des Resultats in Laufe der Ausführung ändern kann, muss sichergestellt werden, dass zwei generische Typen vorhanden sind. Zudem muss die übergebene Funktion von Typen `A => B` sein. Bevor die übergebene Funktion ausgeführt werden kann, muss das Ergebnis des Futures durch Verwendung der `get`-Funktion abgerufen werden. Daraufhin wird geprüft, ob das Resultat valide ist. Fällt die Prüfung positiv aus, wird die übergebene Funktion ausgeführt. Besitzt das Future kein Ergebnis,

wird null zurückgegeben. Scala unterscheidet prinzipiell, ob ein Objekt null sein kann oder nicht. Alle Klassen die von AnyVal erben sind nicht nullable. Hierzu gehören sowohl Zahlen wie z.B. Int, Long oder Float als auch die Klasse Boolean. Da im Falle eines Fehlschlags null zurückgeliefert werden muss, aber nicht jedes Objekt null sein kann, wird eine vom Typen abhängige Repräsentation des null-Wertes benötigt. Abhilfe schafft hierbei die *asInstanceOf[Typ]*-Methode, die auf das null-Objekt angewendet werden kann. Abhängig von der Situation ersetzt der Compiler null mit dem Standardwert des jeweiligen Typen oder gibt, wie in diesem Fall, wirklich null zurück. Wird auf die Verwendung der asInstanceOf-Methode verzichtet, so kommt es zum Fehler beim Kompilieren.

```
def followedBy[A, B](future: Future[A], function: (A) => B):  
  Future[B] = {  
    val following = Future {  
      val resultFirstFuture = get(future)  
      resultFirstFuture match {  
        case Some(x) => function(x)  
        case None => null.asInstanceOf[B]  
      }  
    }  
    following  
  }
```

Abbildung 31: FollowedBy wird eingesetzt um Folgeaufgaben asynchron durchzuführen und das Ergebnis in einem Future zu speichern

Der Kombinator *combine* gestattet die Komposition zweier Futures, wobei deren Ergebnis vereint, ein neues Future erzeugt und zurückgegeben wird. Auf diese Weise lassen sich zwei noch ausstehende, voneinander unabhängige Ergebnisse, die die Grundlage für eine weitere Kalkulation darstellen, kombinieren. Die Berechnung selbst findet auch hier im Hintergrund statt, da ein neues Future zurückgegeben wird.

Die Implementierung von combine kann [Abbildung 32](#) entnommen werden. Combine besitzt drei generische Typen, da sowohl die beiden zu übergebenden Futures als auch das Resultat unterschiedliche Typen besitzen können. Selbiges gilt für die verwendete Funktion, die eingesetzt wird, um die beiden Resultate zu vereinen.

Bevor die Vereinigung durchgeführt werden kann, müssen die beiden Ergebnisse der gegebenen Futures erlangt werden. Hierzu wird die blockende get-Funktion verwendet. Da dies jedoch in einem Hintergrundthread geschieht, wird nur dieser geblockt, was keinerlei Problem darstellt. Nachdem beide Ergebnisse vorliegen, werden die Werte in einem Tupel zusammengefasst. Anschließend wird durch den Einsatz von Pattern Matching

```
def combine[A, B, C](first: Future[A], second: Future[B],  
  function: (A, B) => C): Future[C] = {  
  val combinedFuture = Future{  
    val tuple = (get(first), get(second))  
    tuple match {  
      case (Some(v1), Some(v2)) => function(v1, v2)  
      case _ => null.asInstanceOf[C]  
    }  
  }  
  combinedFuture  
}
```

Abbildung 32: Combine gestattet das Zusammenfassen von zwei Ergebnissen

überprüft, ob die beiden Rückgabewerte valide sind und ein Ergebnis vom Typen `Some` vorliegt. Ist dies gegeben, wird die Funktion angewendet, um das neue Resultat zu berechnen. In allen anderen Fällen wird auch bei dieser Funktion `null` zurückgegeben, wobei auch hierbei `asInstanceOf` eingesetzt werden muss, damit der Code kompiliert. Der nächste Kombinator der betrachtet werden soll ist *orAlt* (Abbildung 33). Diese Funktion nimmt zwei Futures gleichen Typs entgegen und gibt ein neues Future desselben Typs zurück. Das neue Future liefert anschließend das Ergebnis einer der beiden übergebenen Futures zurück, wobei das Erste bevorzugt wird. Dieser Kombinator ähnelt einer `orElse`-Funktion, welche eine ähnliche Semantik besitzt. Die Einsatzmöglichkeit dieser Funktion ist dabei selbsterklärend. Um die Bevorzugung des ersten Futures zu realisieren, wird dieses zuerst evaluiert. Anschließend wird geprüft, ob ein valides Ergebnis vorliegt. Ist dies gegeben wird dieses zurückgegeben. Sollte dies nicht der Fall sein, wird das Ergebnis des zweiten Futures gelesen und ebenfalls überprüft. Besitzt auch

```
def orAlt[A](future: Future[A], other: Future[A]): Future[A] = {  
  val result = Future {  
    get(future) match {  
      case Some(x) => x  
      case None => get(other) match {  
        case Some(x) => x  
        case None => null.asInstanceOf[A]  
      }  
    }  
  }  
  result  
}
```

Abbildung 33: Die Funktion *orAlt* erlaubt die Angabe eines alternativen Futures

das zweite Future kein Resultat wird null zurückgegeben und das neue Future scheitert. Anderenfalls wird das Resultat des zweiten Futures zurückgegeben.

Als nächstes wird die Funktion, welche in Abbildung 34 illustriert ist, betrachtet. *When* kann als Filter angesehen werden, der durch Einsatz eines Prädikats verhindert, dass ungewollte Ergebnisse zurückgeliefert werden. Somit lässt sich beispielsweise verhindern, dass bei einer numerischen Kalkulation negative Werte zurückgeliefert werden.

Das Prädikat ist eine Funktion, die das Ergebnis des Futures als Eingabeparameter erwartet und ein booleschen Wert zurückliefert. Dieser Wert wird beim Pattern Matching als Guard eingesetzt. Hierbei gilt, dass der Case nur gemached wird, wenn die Bedingung des Guards erfüllt wird. Anderenfalls wird, wie in anderen Beispielen zuvor gesehen, null zurückgeliefert, wodurch signalisiert wird, dass die Berechnung fehlgeschlagen ist.

```
def when[A](future: Future[A], condition: (A) => Boolean):
  Future[A] = {
    val result = Future {
      val result = get(future)
      result match {
        case Some(x) if condition(x) => x
        case _ => null.asInstanceOf[A]
      }
    }
    result
  }
```

Abbildung 34: When gestattet die Übergabe eines Guards

Abschließend werden die beiden Funktionen *first* und *firstSuccessful* präsentiert. Da beide Funktionen einen ähnlichen Aufbau besitzen, wird lediglich *firstSuccessful* illustriert (Abbildung 35). Beide Funktionen verwenden die Promise Abstraktion, welche im nächsten Abschnitt vorgestellt wird.

Die beiden Funktionen besitzen exakt einen Parameter. Dieser ist eine Liste von Futures gleichen Typs. Der Rückgabewert beider Funktionen ist ein neues Future. Das Resultat dieses Futures entspricht dabei dem Resultat des ersten, fertigen Futures. Jedes in der Liste vorhandene Future ist dabei ein potentieller Kandidat, welcher das Promise erfüllen kann.

Zu Beginn wird ein neues Promise erzeugt. Anschließend wird über die Liste der Futures iteriert und *trySuccessfulCompleteWith*, respektive *tryCompleteWith*, aufgerufen. Die Futures versucht dabei das Promise zu erfüllen. Da ein Promise nur einmal erfüllt werden kann, ist dies nur dem ersten, fertigen Future gestattet. Während *firstSuccessful* zwischen Erfolg und Misserfolg der Kalkulation unterscheidet, spielt dies bei der *first*-

Funktion keine Rolle.

Da ein Promise nur einmal erfüllt werden kann, ist sichergestellt, dass nur das erste Future das Resultat zu schreiben vermag. Sollten mehrere Futures quasi gleichzeitig versuchen das Ergebnis zu ändern, wird dies vom STM erkannt und der entstandene Konflikt automatisch behoben.

```
def firstSuccessful[A](futures: List[Future[A]]):  
    Future[A] = {  
    val promise = Promise[A]()  
    futures.foreach(trySuccessfulCompleteWith(promise, _))  
    promise.future  
}
```

Abbildung 35: Funktion zur Ermittlung des ersten, erfolgreich ausgeführten Futures

3.2.3 Promise

Futures sind eine Concurrency Abstraktion, die eingesetzt werden kann, um Berechnungen asynchron durchzuführen. Sie sind jedoch read-only und verfügen über keinerlei Möglichkeit, dass Ergebnis zu einem spezifischen Zeitpunkt zu erfüllen. Da dies jedoch in einigen Situationen, wie zum Beispiel bei Verwendung des Producer-Consumer Patterns, notwendig ist, wird eine weitere Abstraktion benötigt, die die Erfüllung zu einem beliebigen Zeitpunkt gestattet.

Promises, denen dieser Abschnitt gewidmet ist, stellen eine solche Abstraktion dar. Dieses Konstrukt, ist ein write-once Datencontainer, der initial leer ist und genau einmal erfüllt werden kann. Das Ergebnis wird dabei in einem Future gespeichert und kann darüber erlangt werden.

Zuerst soll das Promise selbst betrachtet werden (Abbildung 36). Dieses ist ebenfalls als Klasse realisiert und besitzt nur ein Attribut. Dieses ist ein Future, welches bei der Erzeugung des Promise initialisiert wird und somit leer ist. Der Klassenmodifikator ist public, da ein Promise zur Deklaration von Parametern und Variablen einsetzbar sein soll. Zudem ist diese Klasse generisch, wobei der Typ des Promises dem des Futures entspricht.

Zur Erzeugung eines neuen Promise-Objekts wird ebenfalls die apply-Funktion des Begleitobjekts verwendet. Hierbei wird lediglich der new Operator verwendet, um ein neues Objekt zu erzeugen. Es ist möglich auf das Begleitobjekt zu verzichten. Da dies jedoch zur Uneinheitlichkeit bei der Objekterzeugung führen würde, wird dies unterlassen.

```
final class Promise[A] {  
  val future: Future[A] = new Future[A]  
}  
  
object Promise {  
  def apply[A]() = new Promise[A]  
}
```

Abbildung 36: Die Promise Klasse und das dazugehörige Begleitobjekt

Als Nächstes sollen die Funktionen, die auf ein Promise angewendet werden können, vorgestellt und beschrieben werden. Die erste dieser Funktionen ist *trySuccess* (Abbildung 37). Diese Funktion wird eingesetzt, um ein Promise zu erfüllen und gibt einen booleschen Wert zurück, welcher darüber Auskunft gibt, ob die Operation erfolgreich war.

Neben dem Promise, welches erfüllt werden soll, wird ein Resultat als Parameter erwartet. Das Resultat besitzt denselben Typen wie das Promise. Da ein Promise genau einmal erfüllt werden kann, muss der Zustand des Futures vor dem Erfüllen überprüft werden. Hierzu wird das Pattern Matching eingesetzt. Ist das Future leer, dann kann das Resultat geschrieben und ein Erfolg der Operation signalisiert werden. In allen anderen Fällen geschieht nichts und die Funktion gibt false zurück.

Das Gegenstück zu trySuccess ist tryFail. Da beide Funktionen nahezu identisch sind, wird lediglich erläutert, welche Unterschiede vorliegen. Das erste Unterscheidungsmerkmal der beiden Funktionen ist, dass tryFail neben dem Promise keinen weiteren Parameter besitzt, da dort kein Resultat benötigt wird. Der zweite Unterschied besteht darin, dass beim Schreiben des Resultats Fail anstatt Success verwendet wird, sofern das Future leer ist.

```
def trySuccess[A](promise: Promise[A], result: A): Boolean = {  
  atomic { implicit txn =>  
    val v = promise.future.result()  
    v match {  
      case Empty => promise.future.result() = Success(result)  
        true  
      case _ => false  
    }  
  }  
}
```

Abbildung 37: Funktion zur Erfüllung eines Promise

Möchte man mehrere Promises gleichzeitig erfüllen, so kann *multiTrySuccess* verwendet werden. Diese Funktion besitzt dieselbe Semantik wie *trySuccess*, nimmt jedoch Paare, die aus einem Promise und dem dazugehörigen Resultat bestehen, entgegen. Eine Erfüllung der Promises erfolgt hierbei nur dann, wenn alle Promises leer sind.

Abbildung 38 zeigt die Implementierung dieser Funktion. Da die Funktion zwei voneinander unabhängige Promises erfüllt, können die übergebenen Tupel unterschiedliche Typen besitzen. Vor der Erfüllung der Promises muss geprüft werden, ob beide leer sind. Der Zugriff auf die Elemente der Tupel erfolgt dabei unter Verwendung der „Unterstrich-Syntax“. Sind beide Promises noch unerfüllt, werden die Ergebnisse geschrieben und anschließend *true* zurückgeliefert. In allen anderen Fällen gibt die Funktion *false* zurück und erfüllt keines der Promises.

STM garantiert hierbei die Korrektheit der Funktion. Durch die Atomarität wird gewährleistet, dass beide Promises leer sind, wenn das Ergebnis geschrieben wird. Kommt es dazu, dass eines der Promises von einem anderen Thread erfüllt wird, erkennt STM den Konflikt und behebt diesen. Dies kann dazu führen, dass bei der erneuten Ausführung festgestellt wird, dass nur noch ein Promise leer ist, woraufhin keines der Promises erfüllt wird.

```
def multiTrySuccess[A, B](first: (Promise[A], A),
    second: (Promise[B], B)): Boolean = {
  atomic { implicit txn =>
    val tuple = (first._1.future.result(), second._1.future.result())
    tuple match {
      case (Empty, Empty) =>
        first._1.future.result() = Success(first._2)
        second._1.future.result() = Success(second._2)
        true
      case _ => false
    }
  }
}
```

Abbildung 38: Funktion zur Erfüllung von zwei Promises

Eine alternative Implementierung von *multiTrySuccess* kann dem Snippet 39 entnommen werden. Hierbei wird anstatt zweier Tupel eine Liste von Tupeln übergeben. Es wird darauf hingewiesen, dass alle Promises denselben Typen besitzen müssen, da diese invariant sind. Werden Promises unterschiedlichen Typs übergeben, kommt es zu Fehlern beim Kompilieren.

Bevor die Promises erfüllt werden können, muss geprüft werden, ob jedes Promise unbeschrieben ist. Hierzu wird die `forall`-Funktion verwendet. `forall` erwartet ein Prädikat als Parameter und liefert einen booleschen Wert zurück, welcher darüber Auskunft gibt, ob das Prädikat von jedem Element erfüllt worden ist oder nicht. Im nächsten Schritt wird geprüft, welches Ergebnis `forall` zurückgeliefert hat. Sind alle Promises leer, wird die `foreach`-Funktion verwendet, um jedes Promise zu erfüllen und anschließend `true` zurückzugeben. Hat mindestens ein Promise das Prädikat nicht erfüllt, liefert die Funktion `false` zurück und endet.

Auch bei dieser Variante gilt, dass die Korrektheit durch das STM garantiert wird. Ändert sich nur ein Promise in der Liste, muss der gesamte Block erneut ausgeführt werden.

```
def multiTrySuccess[A](promises: List[(Promise[A], A)]): Boolean = {
  atomic { implicit txn =>
    val allEmpty = promises.forall(promise =>
      promise._1.future.result() == Empty)
    allEmpty match {
      case true => promises.foreach(promise =>
        promise._1.future.result() = Success(promise._2))
        true
      case _ => false
    }
  }
}
```

Abbildung 39: Funktion zur Erfüllung mehrerer Promises

Die nächste Funktion, die präsentiert wird, ist *forceSuccess*. Diese weicht die write-once Eigenschaft des Promises auf und gestattet das Überschreiben eines Ergebnisses. Da STM eingesetzt wird, um den Zugriff auf das Resultat zu verwalten, ist garantiert, dass alle Threads die das Future gerade lesen über die Änderung in Kenntnis gesetzt werden. Im Zuge dessen wird der Lesevorgang abgebrochen und erneut ausgeführt, um das neue Resultat zu erhalten. STM schützt dabei jedoch nicht davor, dass der Wert bereits vorher gelesen und als Basis für weitere Berechnungen verwendet worden ist. Wird die Berechnung nach der Änderung des Futures erneut ausgeführt, ist ein abweichendes Resultat wahrscheinlich. Aus diesem Grund sollte diese Funktion immer mit Vorsicht verwendet werden. Nichtsdestotrotz kann der Einsatz dieser Funktion sinnvoll sein, um Korrekturen durchzuführen.

Die Implementierung von `forceSuccess` kann [Abbildung 40](#) entnommen werden. `ForceSuccess` wird auf ein Promise angewendet und besitzt keinen Rückgabewert. Dies wird

```
def forceSuccess[A](promise: Promise[A], value: A) = {  
  atomic { implicit txn =>  
    promise.future.result() = Success(value)  
  }  
}
```

Abbildung 40: ForceSuccess erzwingt die Erfüllung des Promises

damit begründet, dass hierbei kein Fehlschlag möglich ist. Das Schreiben des Resultats geschieht ohne Prüfung, ob das Future leer ist.

Eine weitere Funktion, welche auf ein Promise angewendet werden kann, ist *tryCompleteWith* (Abbildung 41). Bei dieser Funktion wird das Resultat eines Futures verwendet, um das Promise zu erfüllen. Da nicht klar ist, ob das Future zum Zeitpunkt des Aufrufs abgeschlossen ist, wird die eigentliche Tätigkeit asynchron ausgeführt.

TryCompleteWith besitzt zwei Parameter. Der erste Parameter ist ein Promise, welches erfüllt werden soll. Der zweite Parameter ist ein Future, dessen Ergebnis zur Erfüllung des Promise verwendet wird. Um das Blocken des Hauptthreads zu vermeiden, wird *forkIO* eingesetzt, um die Aufgabe im Hintergrund auszuführen. Da das Resultat des Futures obligatorisch ist, wird *get* eingesetzt, um auf den Abschluss der Berechnung zu warten oder das Ergebnis direkt zu lesen. Als Nächstes wird abermals Pattern Matching eingesetzt, um zu bestimmen, was für eine Art Ergebnis vorliegt. Abhängig davon wird entweder *trySuccess* oder *tryFail* verwendet, um das Promise zu erfüllen.

```
def tryCompleteWith[A](promise: Promise[A], future: Future[A]) = {  
  forkIO {  
    atomic { implicit txn =>  
      get(future) match {  
        case Some(v) => trySuccess(promise, v)  
        case None => tryFail(promise)  
      }  
    }  
  }  
}
```

Abbildung 41: Verwendung des Resultats eines Futures, um das Promise zu erfüllen

Die Funktion `trySuccessfulCompleteWith`, welche in Abbildung 42 zu sehen ist, ähnelt der eben vorgestellten Funktion. Hierbei werden ebenfalls ein Promise sowie ein Future erwartet. Sobald das Resultat des Futures vorliegt, wird das Promise erfüllt, falls das Ergebnis valide ist. Ist die Berechnung des Futures fehlgeschlagen, so bleibt das Promise unangetastet und ist damit weiterhin erfüllbar.

```
def trySuccessfulCompleteWith[A](promise: Promise[A],
    future: Future[A]) = {
  forkIO {
    atomic { implicit txn =>
      get(future) match {
        case Some(v) => trySuccess(promise, v)
        case None =>
      }
    }
  }
}
```

Abbildung 42: Verwendung eines validen Resultats eines Futures, um das Promise zu erfüllen

4 Vergleich mit der nativen Implementierung

Nach der Vorstellung der funktionalen Futures und Promises, die im Rahmen dieser Arbeit entwickelt worden sind, soll die Implementierung mit den nativen Mechanismen, die in Scala verfügbar sind, verglichen werden. Hierzu wird betrachtet, welche Gemeinsamkeiten die beiden Implementierungen besitzen und welche Unterschiede bestehen (Abschnitt 4.1). Zudem soll verglichen werden, wie viele Lines of Code bei der jeweiligen Implementierung benötigt werden, um die Funktionalität bereitzustellen (Abschnitt 4.2). Abschließend erfolgt ein Vergleich der Performance und Skalierbarkeit beider Alternativen. Hierzu werden Mikrobenchmarks entworfen und ausgeführt. Das Ergebnis wird anschließend visualisiert und analysiert (Abschnitt 4.3).

4.1 Vergleich des Aufbaus

Während die nativen Futures und Promises vorwiegend objektorientiert sind, jedoch auch funktionale Aspekte besitzen, ist die in dieser Arbeit vorgestellte Implementierung völlig funktional.

Zuerst werden die Gemeinsamkeiten betrachtet. Beide Implementierungen besitzen alle üblichen Funktionen beziehungsweise Methoden, die zu der jeweiligen Abstraktion gehören. Hierbei kann sich die Namensgebung der jeweiligen Features unterscheiden. Zu den gemeinsamen Funktionen zählen `onSuccess` sowie `onFailure`, die verwendet werden, um ein Callback im entsprechenden Fall auszuführen. `Fail` respektive `failed` kann genutzt werden, um ein fehlgeschlagenes Future zu erzeugen. Die vorgestellte `followedBy`-Funktion ähnelt den drei Methoden `transform`, `map` sowie `andThen` und genehmigt die Ausführung weiterer Berechnungen, aufbauend auf dem Ergebnis eines anderen Futures. Die Methode `filter` des nativen Futures entspricht der `when`-Funktion und gestattet die Übergabe eines zu erfüllenden Prädikats. Die Methode `fallbackTo` entspricht der `orAlt`-Funktion und ermöglicht die Übergabe einer Alternative, die im Falle eines Scheiterns zurückgegeben wird, sofern diese ein valides Ergebnis besitzt. Zudem besitzen beide Implementierungen eine `first`- beziehungsweise `firstCompletedOf`-Funktion, welche das Ergebnis des erste abgeschlossenen Futures in einem neuen Future speichert und zurückgibt.

Beide Promise Implementierungen besitzen zudem die drei Funktionen `trySuccess`, `tryFail` und `tryCompleteWith`, welche eingesetzt werden, um das Promise zu erfüllen.

Neben den aufgeführten Gemeinsamkeiten, besitzen die Implementierungen Unterschiede, die zum einen auf die Verwendung unterschiedlicher Programmierparadigmen und zum anderen auf Designentscheidungen bei der Entwicklung zurückzuführen sind. Während die in dieser Arbeit vorgestellten Futures und Promises in einem einzigen Packageobjekt untergebracht sind, werden bei der nativen Realisierung sechs Scala Klassen verwendet.

Die eigentliche Funktionalität der nativen Futures und Promises ist in partiell implementierten Traits vorzufinden. Diese Traits besitzen den jeweiligen Namen der Abstraktion und befinden sich im Package `scala.concurrent`. Lediglich die Methode, die den Zugriff auf das Resultat des Futures ermöglicht, ist abstrakt und muss von einer konkreten Klasse, die das Future Trait erweitert, implementiert werden.

Diese beiden Traits werden wiederum durch ein weiteres Trait erweitert. Dieses heißt ebenfalls Promise und befindet sich im Package `scala.concurrent.impl`. Wie in Abbildung 43 zu sehen, besitzt dieses Trait eine Membervariable, die den Name `future` trägt. Hierbei fällt auf, dass die Variable das Trait selbst referenziert. Dadurch wird erreicht, dass das Future und Promise durch dasselbe Objekt repräsentiert werden. Infolgedessen kann auf die Implementierung einer konkreten Futureklasse verzichtet werden. Der Grund hierfür ist, dass jegliche Funktionalität von den partiell implementierten Traits bereitgestellt wird und der noch fehlende Zugriff auf das Resultat in einer Klasse umgesetzt werden kann.

```
package scala.concurrent.impl

private[concurrent] trait Promise[T] extends
  scala.concurrent.Promise[T] with scala.concurrent.Future[T] {
  def future: this.type = this
}
```

Abbildung 43: Das Promise Trait

Das zweite vorgestellte Promise Trait besitzt zwei Realisierungen. Wie auch das Trait selbst, ist die Sichtbarkeit der Realisierungen auf das Package `scala.concurrent` eingeschränkt. Hierdurch ist die Objekterzeugung nur innerhalb dieses Packages möglich.

Die erste Implementierung des Promise Traits ist das *DefaultPromise*. Dieses wird, wie der Name bereits impliziert, standardmäßig verwendet, um Berechnungen asynchron durchzuführen, kann aber ebenfalls zu einem selbstbestimmten Zeitpunkt erfüllt werden.

Eine Instanz dieses Promises wird immer erzeugt, wenn die Kalkulation eines Ergebnisses aussteht.

Das Gegenstück hierzu ist das *KeptPromise*. Dieses Promise wird bereits bei der Initialisierung erfüllt und erwartet das Resultat als Konstruktorargument. Die Verwendung dieses Promise ist immer dann sinnvoll, wenn das zu verwaltende Ergebnis bereits zum Zeitpunkt der Initialisierung des neuen Objekts vorliegt. Hierdurch wird ein anschließender Methodenaufruf, um das Promise zu erfüllen, obsolet.

Die partielle Implementierung von Traits gestattet zwar die Trennung von Funktionalität und Datenhaltung, kann jedoch, wie in diesem Fall, dafür sorgen, dass die Lesbarkeit des Codes erschwert wird und dieser schwerer zu verstehen ist. Die in dieser Arbeit vorgestellte funktionale Implementierung der beiden Mechanismen erscheint klarer. Auch hier wird Funktionalität und Datenhaltung voneinander getrennt. Das Future sowie das Promise sind hierbei einfache Datencontainer, welche durch unterschiedliche Objekte repräsentiert werden und exakt ein Attribut besitzen, um das jeweilige Ergebnis zu verwalten. Die Funktionalität wiederum wird durch die etwaigen Funktionen, die auf die Typen angewendet werden, erbracht.

Ein weiteres Unterscheidungsmerkmal zwischen den nativen Promises und Futures und der eigenen Implementierung ist, dass die native Realisierung über keinerlei blockende „get“-Methode verfügt. Stattdessen muss das Await-Objekt verwendet werden, um den aktuellen Thread zu blocken, auf die Beendigung der Kalkulation zu warten und anschließend das Ergebnis zu erlangen. Das folgende Beispiel illustriert den Einsatz von Await (Abbildung 44). Die result-Methode nimmt zwei Parameter entgegen. Der erste Parameter ist vom Typ Awaitable, welches vom Future Trait erweitert wird und ein ausstehendes Ergebnis, auf welches gewartet werden kann, repräsentiert. Zusätzlich wird eine Zeitangabe erwartet, die vorgibt wie lange der aktuelle Thread geblockt werden soll. Kommt es zu einer Überschreitung der vorgegeben Zeit, wird eine TimeoutException geworfen, der Wartevorgang abgebrochen und die Ausführung des geblockten Threads fortgesetzt.

Abschließend sollen die Funktionen und Methoden betrachtet werden, welche nur in einer der beiden Implementierungen vorhanden sind. Das native Future unterstützt eine Vielzahl von Funktionen höherer Ordnung, welche bereits von anderen Datentypen wie

```
val f = Future(/* execute calculation */)  
val result = Await.result(f, Duration.Inf)
```

Abbildung 44: Await blockt den Hauptthread und entspricht der get-Funktion

Listen bekannt sind. Map, flatmap, foreach, collect und zip können verwendet werden, um weitere Funktion auf das Resultat des aktuellen Futures anzuwenden. Der Grund für die Bereitstellung dieser Funktionen ist darauf zurückzuführen, dass das Resultat selbst von einem Try-Objekt gewrappt wird und der Funktionsaufruf lediglich delegiert werden muss, sobald das Ergebnis verfügbar ist. Dies hat den Vorteil, dass eine Verschachtelung der Berechnungsschritte ermöglicht wird. Zudem gestattet dies die Deklaration mehrerer Bearbeitungsschritte, lässt eine Fehlerkorrektur zu und macht das Warten auf Zwischenergebnisse obsolet. Das Beispiel aus Abbildung 45 verdeutlicht das eben beschriebene nochmals. Wie dort abgebildet, wird zuerst ein Future erzeugt, dessen Ergebnis ein zufälliger Wert zwischen 0 und 999 ist. Darauffolgend wird die map- Funktion eingesetzt, um das Ergebnis in einen String umzuwandeln. Anschließend wird ein Filter verwendet, welcher prüft, ob der erzeugte String aus mehr als zwei Zeichen besteht. Wird die Bedingung nicht erfüllt, wird eine NoSuchElementException geworfen. Die recover-Funktion wird bei diesem Beispiel dazu verwendet, geworfene Fehler zu behandeln und in ein valides Ergebnis umzuwandeln.

```
val f = Future[Int](Random.nextInt(1000)) map {  
  int => int.toString  
} filter {  
  x => x.length > 2  
} recover { case x: Throwable => "-1" }
```

Abbildung 45: Verschachtelung von Funktionen höherer Ordnung (nativen Futures)

Neben den Funktionen höherer Ordnung, die direkt auf einem Future aufgerufen werden können, besitzt die native Future Implementierung weitere Funktionen, welche auf eine Menge von Futures angewendet werden und im Begleitobjekt des Futures verankert sind. Der Einsatz dieser Funktionen ist beispielsweise dann sinnvoll, wenn eine Menge an Ergebnissen zusammengefasst werden soll (sequence, fold, reduce, traverse) oder ein spezifisches Future, welches ein bestimmtes Kriterium erfüllt, innerhalb einer Menge von Futures gesucht wird (find).

Die Funktionen, die nur in der eigenen Future Implementierung bereitgestellt werden, sind waitForSuccess, combine und firstSuccessful. WaitForSuccess deckt einen eher ungewöhnlichen Anwendungsfall ab. Hierbei wird davon ausgegangen, dass das Resultat des Futures nach Beendigung der Kalkulation manuell überschrieben werden kann. Diese Funktion besitzt eher exemplarischen Charakter und soll zeigen, wie ein solcher Fall gehandhabt werden kann. Die Kombination zweier Ergebnisse wiederum löst ein allgemeines Problem, welches immer wieder auftritt. Durch die Komposition mehrerer combine

Aufrufe ist es ebenfalls möglich eine unbestimmte Menge an Ergebnisse zu kombinieren. `FirstSuccessful` ähnelt der `first`-Funktion, erfüllt das Promise jedoch nur, wenn ein valides Ergebnis vorliegt.

Betrachtet man die nativen Promises, so fällt auf, dass diese keine weitere Funktionalität bereitstellen. Die Implementierung besitzt lediglich redundante Methoden für jede Operation. Diese Methoden haben keinen `try`-Präfix und delegieren den Aufruf lediglich an die entsprechende Methode mit diesem Präfix.

Die eigene Implementierung der Promises besitzt mehrere Funktionen, welche zusätzliche Funktionalität hinzufügen oder die Eigenschaften bestehender Funktionen weiter konkretisieren. `ForceSuccess`, welches eher exemplarischen Charakter besitzt, entfernt die `write-once` Eigenschaft eines Promise und erlaubt das ungeprüfte Schreiben des Resultats. `MultiTrySuccess` gestattet die Erfüllung mehrere Promises gleichzeitig. Die „Erweiterung“ der `tryCompleteWith` Funktion `trySuccessfulCompleteWith` gestattet die Erfüllung eines Promise nur im Falle einer erfolgreichen Ergebnisberechnung und ignoriert Fehlschläge.

4.2 Vergleich der Lines of Code

Neben dem Vergleich des Aufbaus der Futures und Promises soll betrachtet werden, wie viele Zeilen Code beide Implementierungen besitzen, um die vorhandene Funktionalität bereitzustellen. Da ein Vergleich nicht ohne Weiteres durchgeführt werden kann, müssen Kriterien definiert werden, die von beiden Implementierungen zu erfüllen sind. Diese Kriterien sollen dafür sorgen, dass kein verzerrtes Bild entsteht und bestehen aus den folgenden Punkten:

1. Alle Kommentare im Quellcode sind zu entfernen.
2. Importanweisungen dürfen immer nur aus einer Klasse beziehungsweise einem Objekt bestehen.
3. Alle Zeilenabstände innerhalb von Funktionen und Methoden müssen, sofern vorhanden, entfernt werden.
4. Zeilenabstände zwischen Klassen und Methoden betragen exakt eine Zeile.
5. Die für den Vergleich genutzte Scala Version ist 2.11.7.

Unter Beachtung der gegebenen Kriterien wird der Quellcode manuell bereinigt und die Anzahl der Zeilen bestimmt. Neben den eigentlichen Klassen, wird auch das Packageobjekt eingerechnet, da dieses zusätzliche Funktionalität bereitstellt. Die Tabelle 3

illustriert das Ergebnis der nativen Implementierung. Wie dort abzulesen ist, besteht die Implementierung in Summe aus 582 Codezeilen.

Klassen	Codezeilen
scala.concurrent.Future	238
scala.concurrent.impl.Future	25
scala.concurrent.Promise	46
scala.concurrent.impl.AbstractPromise	29
scala.concurrent.impl.Promise	207
scala.concurrent (Packageobjekt)	37
Summe	582

Tabelle 3: Lines of Code der nativen Futures und Promises

Unter Anwendung derselben Kriterien wird auch der eigene Sourcecode analysiert. Da die Implementierung lediglich aus einem Packageobjekt besteht, wird eine logische Aufteilung eingeführt, um zu zeigen, wie viele Codezeilen für die jeweilige Abstraktion benötigt werden. Wie der Tabelle 4 entnommen werden kann, besitzt die in dieser Arbeit vorgestellte Implementierung weniger als die Hälfte an Codezeilen (257).

Klassen	Codezeilen
Future	126
Promise	84
Result	39
Summe	249

Tabelle 4: Lines of Code der eigenen, funktionalen Futures und Promises

Da, wie bereits im vorherigen Abschnitt (4.1) erläutert, beide Implementierungen unterschiedliche Funktionen unterstützen, soll zudem betrachtet werden, wie viele Codezeilen zur Erbringung derselben Funktionalität benötigt werden. Hierzu wird ein weiteres Kriterium benötigt, welches besagt, dass nur Features, die in beiden Implementierungen vorhanden sind, verglichen werden sollen. Nach Entfernung der ausgeschlossenen Funktionen ergibt sich ein ähnliches Bild wie zuvor. Wieder besitzt die eigene Implementierung weniger als die Hälfte an Zeilen und ist daher deutlich kompakter als die native Umsetzung. Das Ergebnis des zweiten Vergleichs kann der folgenden Tabelle entnommen werden.

native Futures & Promises	vorgestellte Futures & Promises
466	143

Tabelle 5: Vergleich beider Implementierungen mit gleicher Funktionalität

4.3 Benchmarking der Implementierungen

Nach der Beschreibung der strukturellen Unterschiede zwischen der nativen Umsetzung und der eigenen Implementierungen sowie dem Vergleich der Lines of Code, soll das Laufzeitverhalten untersucht werden. Um dies zu bewerkstelligen, werden verschiedene Benchmarks definiert und ausgeführt. Anschließend werden die Ergebnisse vorgestellt und analysiert.

Gerade bei Sprachen, die eine virtuelle Maschine zur Ausführung des Codes besitzen, ist das Bechmarken eine nicht triviale Aufgabe. Die JVM, welche in Scala verwendet wird, verfügt über eine Vielzahl an Mechanismen, die das Laufzeitverhalten gravierend beeinflussen. Diese sind dem Anwender zwar bekannt, können von diesem jedoch kaum oder gar nicht kontrolliert werden[22].

Zu diesen Mechanismen zählen die Just in Time Compilation (JIT), das Classloading sowie das automatische Speichermanagement (Garbage Collection).

Bei der Just in Time Compilation wird häufig ausgeführter Code kontinuierlich analysiert und in Maschinencode umgewandelt. Dies kann jederzeit geschehen und jeden Codeabschnitt betreffen. Zudem wird der Code periodisch neu kompiliert, was dazu führen kann das bei bestimmten Abschnitten Optimierung wieder entfernt werden. Das eben beschriebene Verhalten des JIT führt dazu, dass die Laufzeitcharakteristik eines beliebigen Abschnitts während der Laufzeit verändert werden kann und somit Einfluss auf das Ergebnis des Bechmarks nimmt.

Das Classloading beeinflusst das Verhalten eines Benchmarks insofern, als dass die JVM imstande ist nicht-lokale Optimierungen, basierend auf globalen Programminformationen, durchzuführen. Ein Beispiel hierfür ist das „Inline Caching“. Hierbei optimiert die JVM polymorphe Methodenaufrufe automatisch und ersetzt den generischen Typen durch einen Konkreten. Da bei Benchmarks in der Regel nicht alle Klassen geladen werden, werden viele solcher Methodenaufrufe optimiert. Dies sorgt wiederum dafür, dass die Messergebnisse verfälschten werden. Dies bedeutet im Umkehrschluss, dass scheinbar unwichtiger Code große Auswirkungen auf die Ergebnisse eines Benchmarks haben kann.

Das automatische Speichermanagement kann ebenfalls einen nicht unerheblichen Einfluss auf die Messergebnisse haben. In der Regel werden beim Benchmarken bestimmte Codeabschnitte mehrfach ausgeführt und das Ergebnis nach Abschluss aller Iterationen gemittelt. Da sich der Zustand des Heaps vor jedem Durchlauf unterscheiden kann, ist es prinzipiell möglich, dass eine Speicherbereinigung nur in einigen Durchläufen statt-

findet. Dies wiederum beeinflusst die Laufzeitcharakteristik dieser Durchläufe und kann das Ergebnis somit verfälschen.

Um die eben beschriebenen Nebeneffekt möglichst gering zu halten, wird die Benchmarksuite ScalaMeter eingesetzt. ScalaMeter ist ein hochkonfigurierbares Testframework, welches mehrere konfigurierbare JVM Benchmarkingmethodolgien unterstützt und ein DSL zur Erzeugung von Eingabedaten bereitstellt. Um sicherzustellen, dass der JIT Compiler alle Optimierungen ordnungsgemäß durchgeführt hat und alle Klassen geladen worden sind, besitzt das Framework eine Aufwärmphase, die solange ausgeführt wird, bis eine gewisse Konvergenz erreicht wird und die Ergebnisse stabil sind. Erst daraufhin beginnt die eigentliche Messung der Resultate der einzelnen Durchläufe. Zudem besitzt ScalaMeter eine Ausreißererkennung und ignoriert Durchläufe, in denen eine Garbage Collection durchgeführt worden ist. Ein weiterer Vorteil, der für die Verwendung von ScalaMeter spricht ist, dass die Generierung von Reports automatisch erfolgt und eine manuelle Erzeugung entfällt.

Bevor die eigentlich Benchmarks selbst vorgestellt werden, wird erläutert, wie diese aufgebaut und konfiguriert sind (Abschnitt 4.3.1). Abschnitt 4.3.2 befasst sich mit den ersten Benchmarks, welche das Verhalten beim Zugriff auf das Resultat eines Future betrachten. Hiernach folgen mehrere Benchmarks, die zeigen, wie die beiden Implementierungen sich Verhalten, wenn eine Vielzahl an Kombinatoren eingesetzt wird, um ein finales Ergebnis zu berechnen (Abschnitt 4.3.3). Im folgenden Abschnitt (4.3.4) wird eine alternative Implementierung des Sleeping Barber Problems mit Promises vorgestellt und analysiert. Abschnitt 4.3.5 zeigt eine Implementierung des Santa Clause Problems. Hierbei werden ebenfalls Promises eingesetzt. Zuletzt wird auch das Dining Philosophers Problem mit Promises realisiert und anschließend mit Implementierungen, die Sperren und STM verwenden, verglichen (Abschnitt 4.3.6).

4.3.1 Aufbau der Benchmarks

Um zu gewährleisten, dass die durchgeführten Benchmarks vergleichbar sind, wird jeder Test auf demselben System ausgeführt. Das verwendete System besitzt 128 GB Arbeitsspeicher und verfügt über 8 Intel(R) Xeon(R) E7-4820 Prozessoren, wodurch 64 Kerne mit jeweils 2 GHz zur Verfügung stehen. Der Code selbst wird mit Scala 2.11.7 kompiliert und in einer Java 8 (build 1.8.0_64) JVM ausgeführt. Die verwendete Version von ScalaMeter ist 0.7.

Die Benchmarks selbst folgen demselben Schema, welches in Abbildung 46 illustriert ist. Jeder Test wird als Scala-Objekt realisiert. Jedes Objekt erweitert dabei die Klasse

Bench.OfflineReport. Hierdurch erbt jeder Test die folgenden elementaren Eigenschaften:

- Jeder Test wird insgesamt 30-mal ausgeführt. Dabei werden 9 Samples erzeugt, die jeweils in einer neuen JVM ausgeführt werden.
- Jedes Sample besitzt ein Warmup, sodass der JIT Compiler imstande ist den Code zu kompilieren und alle Klassen ordnungsgemäß geladen werden können.
- Das Warmup ist beendet, sobald die Zeitabweichung der letzten Versuche klein genug ist.
- Ausreißer werden erkannt und entfernt. Dies betrifft vorwiegend Durchläufe in denen ein GC erkannt wird.
- Daten aller Samples werden automatisch aggregiert. Hierzu wird das arithmetische Mittel verwendet.
- Nach der Aggregation werden die Daten automatisch gespeichert und eine Grafik erzeugt.

Darüber hinaus kann ein Benchmark Importanweisungen besitzen, falls weitere Klassen benötigt werden. Nach den Importanweisungen kommen Anweisungen, die verwendet werden um Testdaten zu erzeugen. Hierbei hilft ein Generator, der von ScalaMeter bereitgestellt wird.

Die Benchmarks selbst befinden sich innerhalb des Blocks, der mit dem Schlüsselwort `performance` eingeleitet wird. Jeder dieser Blöcke benötigt einen eindeutigen Namen, um eine spätere Speicherung der Rohdaten zu ermöglichen. Innerhalb des `performance`-Blocks sind wiederum Blöcke vorzufinden, die mit dem Schlüsselwort `measure` eingeleitet werden und den auszuführenden Code umschließen. Auch hierbei wird ein eindeutiger Name verlangt. Da die jeweiligen Benchmarks unabhängig voneinander sind, wird der Name „own“ für die eigene Implementierung und „native“ für die native Umsetzung verwendet.

```
object Benchmark extends Bench.OfflineReport {  
  // imports & generation of the input data  
  performance of "Test case" in {  
    measure method "functional" in { /* exeuction steps */ }  
    measure method "native" in { /* exeuction steps */ }  
  }  
}
```

Abbildung 46: Schema eines Benchmark-Objekts

Für jeden zu testenden Anwendungsfall wird ein Benchmark geschrieben, um die durchschnittliche Ausführungszeit der beiden Future Varianten zu bestimmen. Da die Messung von Ausführungszeiten der häufigste Gegenstand solcher Analysen ist, werden die Benchmarks mit der Standardkonfiguration ausgeführt und benötigen vorerst keine Anpassung.

4.3.2 Zugriff auf das Ergebnis eines Futures

Die ersten Benchmarks, die vorgestellt werden, beschäftigen sich mit der Frage, wie die beiden Implementierungen sich beim Zugriff auf das Resultat eines Futures verhalten. Hierbei werden zwei Anwendungsfälle betrachtet:

1. Das Ergebnis eines Future wird berechnet. Daraufhin greift ein Thread mehrfach auf das Ergebnis zu.
2. Das Ergebnis eines Future wird berechnet. Anschließend greifen n Threads nebenläufig auf das Ergebnis zu.

Zuerst soll der sequentielle Zugriff auf das Resultat betrachtet werden. Der Benchmark, der hierzu verwendet wird ist in Abbildung 47 visualisiert. Die Variable `gets` wird durch Verwendung des bereitgestellten Generators erzeugt und gibt vor, wie oft das Resultat gelesen wird. Der Generator erzeugt dabei eine Liste von Eingabeparameter für den Test. Der erste Parameter der `range`-Methode des Generators ist ein String, welcher bei der Visualisierung für die Beschriftung der x-Achse verwendet wird. Zudem erwartet diese Methode drei weitere Parameter, die den Start-, Endwert und die Schrittweite vorgeben.

```
val gets = Gen.range("get")(100, 1000, 100)

measure method "own sequential" in {
  using(gets) in { g =>
    val f = Future {
      Thread.sleep(1)
      42
    }
    for (i <- 1 to g) {
      get(f)
    }
  }
}
```

Abbildung 47: Nebenläufiges Lesen eines Resultats

Durch Verwendung der `using`-Methode wird über die generierten Daten iteriert und der aktuelle Wert als Parameter für den gegenwärtigen Durchlauf genutzt.

Jeder Durchgang beginnt damit, dass ein neues Future erzeugt wird. Das Schreiben des Resultats wird um etwa eine Millisekunde verzögert. Währenddessen beginnt die Schleife den ersten Durchlauf, der jedoch solange verzögert wird, bis das Future ein Resultat liefert. Anschließend können die weiteren Runden ohne Verzögerung auf das Resultat zugreifen.

Bei der nativen Version dieses Tests, wird das `Await`-Objekt anstelle der `get`-Funktion verwendet, um das Resultat zu lesen.

Jeder Testdurchlauf wird 1000-mal ausgeführt. Anschließend wird der Median ermittelt. Das erste Testergebnis kann Abbildung 48 entnommen werden. Betrachtet man zuerst die native Variante (grün), fällt auf dass diese nahezu konstant ist. Um 100 Aufrufe auszuführen, werden etwas mehr als 1,1 Millisekunden benötigt. 1000 Aufrufe benötigen dagegen circa 1,15 Millisekunden, was eine Steigerung von 4,5% darstellt.

Die STM-basierte Variante benötigt für die Ausführung von 100 Aufrufen ebenfalls etwas mehr als 1,1 Millisekunden. Wie der Grafik jedoch entnommen werden kann, werden 1,27 Millisekunden benötigt, um 1000 Iterationen durchzuführen. Dies stellt eine Steigerung von 15% dar.

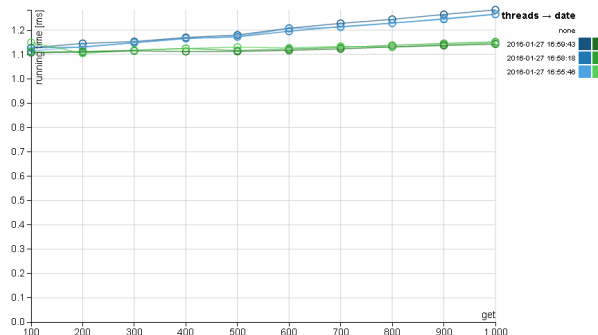
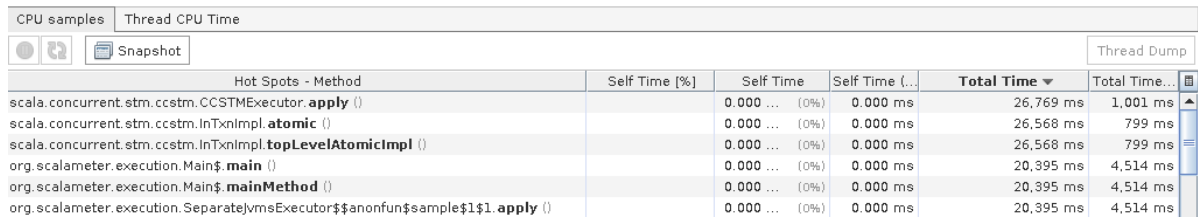


Abbildung 48: Performance beim mehrfachen, sequentiellen Zugriff auf ein Ergebnis.

Die erhöhte Laufzeit der eigenen Implementierung ist dabei auf STM selbst zurückzuführen. Um das Ergebnis zu lesen, muss auf die geschützte Ressource zugegriffen werden. Hierzu benötigt STM zusätzliche Objekte, wie einen Executor und den Kontext. Zudem muss das STM sicherstellen, dass keinerlei Konflikte vorliegen. Beides führt dazu, dass bei jeder Abfrage Overhead entsteht, der sich wiederum auf die Laufzeit auswirkt. Um diese Vermutung zu bestätigen, wird geprüft, wie lange die einzelnen Methoden ausgeführt werden. Hierzu wird das Analysetool JVisualVM eingesetzt. Wie die Analyse

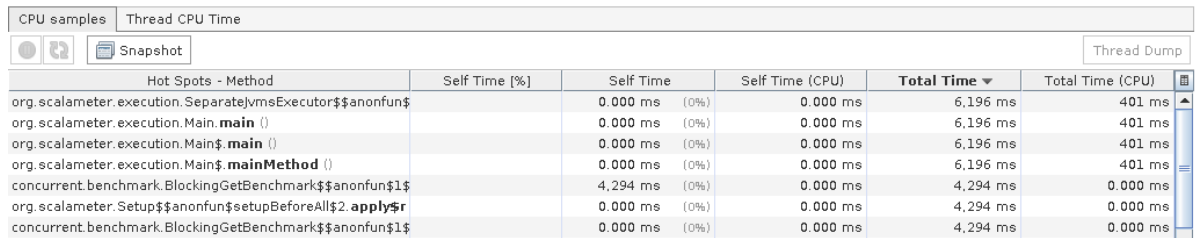


Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
scala.concurrent.stm.ccstm.CCSTMExecutor.apply ()	0.000 ... (0%)	0.000 ms	0.000 ms	26,769 ms	1,001 ms
scala.concurrent.stm.ccstm.InTxnImpl.atomic ()	0.000 ... (0%)	0.000 ms	0.000 ms	26,568 ms	799 ms
scala.concurrent.stm.ccstm.InTxnImpl.topLevelAtomicImpl ()	0.000 ... (0%)	0.000 ms	0.000 ms	26,568 ms	799 ms
org.scalameter.execution.Main\$.main ()	0.000 ... (0%)	0.000 ms	0.000 ms	20,395 ms	4,514 ms
org.scalameter.execution.Main\$.mainMethod ()	0.000 ... (0%)	0.000 ms	0.000 ms	20,395 ms	4,514 ms
org.scalameter.execution.SeparatelyVmsExecutor\$\$anonfun\$sample\$1\$1.apply ()	0.000 ... (0%)	0.000 ms	0.000 ms	20,395 ms	4,514 ms

Abbildung 49: CPU Sampling der eigenen Implementierung

zeigt, sind die drei am längsten ausgeführten Methoden STM-spezifisch (Abbildung 49). Somit ist die zuvor aufgestellte Vermutung belegt.

Ein anderes Bild ergibt sich bei der Betrachtung des Benchmarks, bei dem das native Future eingesetzt wird (Abbildung 50). Unter den am längsten ausgeführten Methoden befinden sich ausschließlich Methoden, die für die Ausführung des Benchmarks verantwortlich sind.



Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
org.scalameter.execution.SeparatelyVmsExecutor\$\$anonfun\$sample\$1\$1.apply ()	0.000 ms (0%)	0.000 ms	0.000 ms	6,196 ms	401 ms
org.scalameter.execution.Main\$.main ()	0.000 ms (0%)	0.000 ms	0.000 ms	6,196 ms	401 ms
org.scalameter.execution.Main\$.mainMethod ()	0.000 ms (0%)	0.000 ms	0.000 ms	6,196 ms	401 ms
org.scalameter.execution.Main\$.mainMethod ()	0.000 ms (0%)	0.000 ms	0.000 ms	6,196 ms	401 ms
concurrent.benchmark.BlockingGetBenchmark\$\$anonfun\$1\$1.run ()	4,294 ms (0%)	0.000 ms	0.000 ms	4,294 ms	0.000 ms
org.scalameter.Setup\$\$anonfun\$setUpBeforeAll\$2\$.apply ()	0.000 ms (0%)	0.000 ms	0.000 ms	4,294 ms	0.000 ms
concurrent.benchmark.BlockingGetBenchmark\$\$anonfun\$1\$1.run ()	0.000 ms (0%)	0.000 ms	0.000 ms	4,294 ms	0.000 ms

Abbildung 50: CPU Sampling der nativen Implementierung

Als Nächstes soll der nebenläufige Zugriff auf das Resultat eines Futures betrachtet werden. Der erste Test verwendet Java Threads, die innerhalb einer Schleife erzeugt und anschließend gestartet werden (Abbildung 51). Hierbei wird die run-Methode des Threads überschrieben und die entsprechende blockende Methode eingesetzt, um das Resultat zu erlangen. Zudem wird eine Barriere benötigt, um die Ausführung des Tests solange hinauszuzögern, bis jeder Thread das Ergebnis gelesen hat. Die Barriere wird durch eine CountdownLatch realisiert. Bei diesem Benchmark werden zwischen 10 und 60 Threads erzeugt, die das Ergebnis parallel lesen sollen.

```
val threads = Gen.range("threads")(10, 60, 5)

measure method "own concurrent with Threads" in {
  using(threads) in { threads =>
    val f = Future {
      Thread.sleep(1)
      42
    }
    val latch = new CountDownLatch(threads)
    for (i <- 1 to threads) {
      val t = new Thread() {
        override def run(): Unit = {
          get(f)
          latch.countDown()
        }
      }.start()
    }
    latch.await()
  }
}
```

Abbildung 51: Nebenläufiges Lesen eines Resultats mit Threads

Das Ergebnis dieses Benchmarks, welches der Grafik [52](#) entnommen werden kann, zeigt, dass beide Implementierungen nahezu identische Ausführungszeiten besitzen. Bei der Ausführung von 10 Threads benötigten beide Tests im Mittel 1,2 Millisekunden. Da das Future etwa 1 Millisekunde benötigt, um das Ergebnis bereitzustellen, ergibt sich eine reine Berechnungszeit von 0,2 Millisekunden.

Entgegen aller Erwartung, ist die Ausführungszeit bei diesem Test nicht konstant sondern linear. Bei beiden Experimenten vergehen fast 4 Millisekunden bis 60 Threads das Ergebnis gelesen und das Latch dekrementiert haben. Dies ist eine Vervierfachung gegenüber dem Anfangswert.

Es liegt nahe, dass die Erzeugung neuer Threads die Ursache für das unerwartet Verhalten ist. Der dadurch entstehende Overhead kann direkten Einfluss auf das Ergebnis haben. Um den Verdacht zu bestätigen, wird JVisualVM erneut eingesetzt. Hierbei konnte festgestellt werden, dass beide Benchmarks jeweils etwa 500.000 Threads erzeugen. Zudem hat sich gezeigt, dass der CPU bei der Ausführung des einminütigen Benchmarks, etwa 30 Sekunden lang mit der Erzeugung neuer Threads beschäftigt ist.

Um zu verhindern, dass das Erschaffen neuer Threads die Messresultate beeinflusst, wird ein weiterer Benchmark ausgeführt. Bei diesem Test wird der ForkJoinPool verwendet. Dieser nimmt Runnables entgegen und speichert diese in einer Queue, falls alle Threads mit anderen Aufgaben beschäftigt sind. Sind Threads verfügbar erfolgt eine Zuweisung

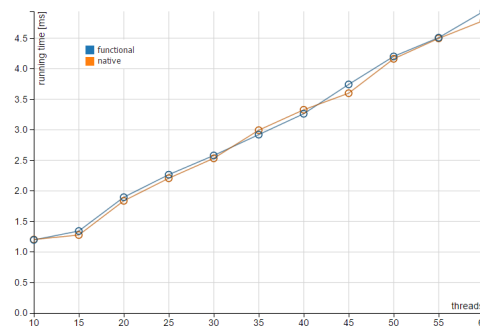


Abbildung 52: Nebenläufiges Lesen mit Threads

des Tasks. Anschließend wird der Thread gestartet und führt die Aufgabe aus.

Da der Pool, wie bereits erwähnt, Runnables und keine Threads entgegennimmt, muss die Implementierung dahingehend angepasst werden, dass innerhalb der Schleife Runnables erzeugt werden. Zudem wird das Starten des Threads durch die Übergabe an den Threadpool ersetzt.

Nach der Durchführung der neuen Benchmarks ist eine Reduzierung der Ausführungszeiten sichtbar. Zudem hat sich die Anzahl der erzeugten Threads auf 80 reduziert. Die Verwaltung der Threads obliegt dabei dem Threadpool. Die Grafik 53 veranschaulicht das Ergebnis von drei voneinander unabhängigen Tests, bei denen jeweils 1000 Messungen pro Anzahl an Tasks durchgeführt werden. Anschließend wird der Median ermittelt. Die native Variante (lila) ist auch hierbei etwas schneller als die Eigene (orange). Wie zu erkennen ist, unterliegen die Resultate beider Varianten teils starken Schwankungen. Die eigene Variante weist hierbei Abweichungen von bis zu 0,2 ms (50 Threads) auf. Bei der nativen Variante sind sogar Abweichungen von bis zu 0,3 ms (60 Threads) feststellbar. Die Ursache für diese Schwankungen lässt sich nur schwer ermitteln, da viele, nicht beeinflussbare Faktoren, dafür verantwortlich sein können. Prinzipiell gilt, dass STM ein retry ausführt, wenn das Ergebnis nicht vorliegt. Dabei werden die Threads vorübergehend pausiert, um ein „busy waiting“ zu vermeiden. Zwar gibt es keine anderen Tasks, die zwischenzeitlich ausgeführt werden, jedoch kann keine konkrete Aussage darüber gemacht werden, wie der Pool sich hierbei verhält.

Die native Implementierung wiederum setzt Callbacks und ein Latch ein, um auf das Ergebnis zu warten. Dabei wird das Latch freigegeben, sobald das Ergebnis vorliegt. Da der Callback ebenfalls asynchron ist und in einem Thread ausgeführt werden muss, kann es passieren, dass die Freigabe des Latch verzögert wird, da die auf das Ergebnis wartenden Threads, die Ausführung der Callbacks blockieren.

Beim letzten Test, der diesbezüglich durchgeführt wird, soll das Verhalten der beiden Va-

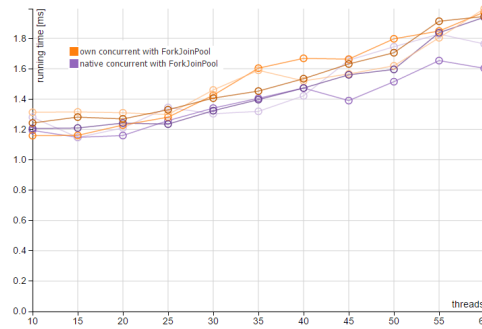


Abbildung 53: Nebenläufiges Lesen unter Verwendung des ForkJoinPools

rianten nochmals analysiert werden. Hierbei kommt ein anderer Threadpool zu Einsatz. Dies soll Aufschluss darüber geben, ob der Threadpool selbst Einfluss auf die Resultate hat.

Bei diesen Benchmarks wird Java's `ThreadPoolExecutor` eingesetzt, um die `Runnables` auszuführen. Da dieser Threadpool, anders als der `ForkJoinPool`, nicht automatisch terminiert, sobald die Applikation beendet wird, muss sichergestellt werden, dass der shutdown explizit aufgerufen wird, nachdem alle Tasks fertig sind.

Auch bei diesen Benchmarks sind Schwankungen zu vermerken. Der Grund für diese kann ebenfalls auf die zuvor erläuterten Charakteristiken zurückgeführt werden. Es ist zudem auffällig, dass die Schwankungen geringer als zuvor sind. Darüber hinaus sieht man, dass die Performance beider Implementierungen bereits zu Beginn etwas langsamer ist als zuvor. Ebenfalls erkennbar ist, dass die Ausführungszeiten konstanter geworden sind und die Anwendungen besser skalieren. Der Höchstwert bei der Verwendung von 60 Threads beträgt etwa 1,6 Millisekunden. Bei gleichen Parametern dauert die Ausführung mit dem `ForkJoinPool` etwa 1,9 Millisekunden.

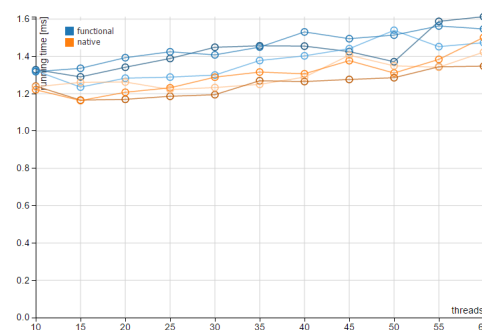
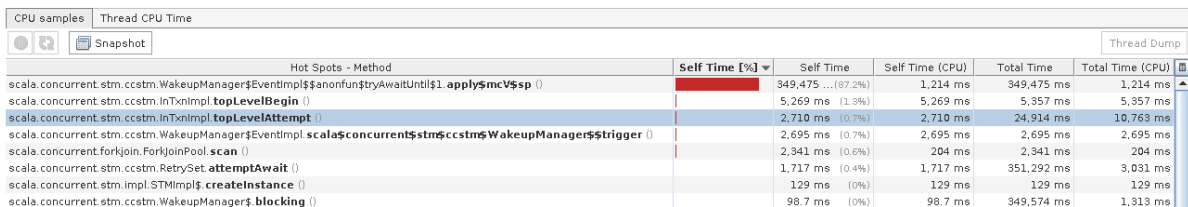


Abbildung 54: Nebenläufiges Lesen unter Verwendung des ThreadPoolExecutors

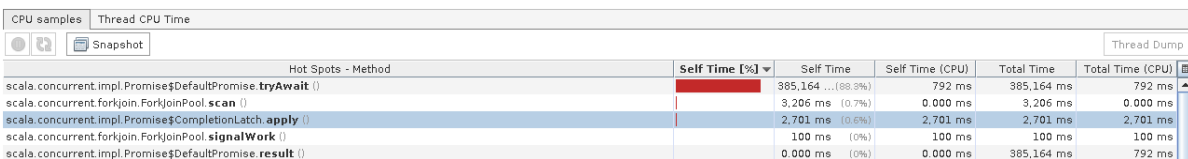
Betrachtet man die Resultate des CPU Samplings (Abbildung 55, 56), ist zu erkennen, dass beide Implementierungen in etwa dieselbe Zeit warten. Die einzige, zusätzliche Aktion, die bei der nativen Version ausgeführt wird, ist die Freigabe des CompletionLatch. Bei der STM Variante hingegen werden deutlich mehr Berechnungen durchgeführt. Dies umfasst ein explizites blocken und wecken durch den WakeupManager, die Instanziierung neuer STM Instanzen sowie die Ausführung der atomic Blöcke.

Die Untersuchung verdeutlicht die Unterschiede der beiden Implementierungen. Der durch STM entstehende Overhead sorgt dafür, dass die eigene Implementierung etwas langsamer als die Native ist, weißt aber sonst keine Auffälligkeiten auf. Da die Schwankungen im Mikrosekundenbereich und daher sehr gering sind und beim Sampling nichts Außergewöhnliches festgestellt werden konnte, erhärtet sich die Vermutung, dass die Schwankungen durch das Threading selbst verursacht werden.



Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
scala.concurrent.stm.ccstm.WakeupManager\$EventImpl\$\$anonfun\$tryAwaitUntil\$1.apply\$mcV\$sp ()	87.2%	349,475 ms	1,214 ms	349,475 ms	1,214 ms
scala.concurrent.stm.ccstm.InTxImpl.topLevelBegin ()	1.3%	5,269 ms	5,269 ms	5,357 ms	5,357 ms
scala.concurrent.stm.ccstm.InTxImpl.topLevelAttempt ()	0.7%	2,710 ms	2,710 ms	24,914 ms	10,763 ms
scala.concurrent.stm.ccstm.WakeupManager\$EventImpl.scala\$concurrent\$stm\$ccstm\$WakeupManager\$trigger ()	0.7%	2,695 ms	2,695 ms	2,695 ms	2,695 ms
scala.concurrent.forkjoin.ForkJoinPool.scan ()	0.6%	2,341 ms	204 ms	2,341 ms	204 ms
scala.concurrent.stm.ccstm.RetrySet.attemptAwait ()	0.4%	1,717 ms	1,717 ms	351,292 ms	3,031 ms
scala.concurrent.stm.impl.STMImpl\$.createInstance ()	0%	129 ms	129 ms	129 ms	129 ms
scala.concurrent.stm.ccstm.WakeupManager\$.blocking ()	0%	98.7 ms	98.7 ms	349,574 ms	1,313 ms

Abbildung 55: CPU Zeiten der eigenen Implementierung



Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
scala.concurrent.impl.Promise\$DefaultPromise.tryAwait ()	88.3%	385,164 ms	792 ms	385,164 ms	792 ms
scala.concurrent.forkjoin.ForkJoinPool.scan ()	0.7%	3,206 ms	0.000 ms	3,206 ms	0.000 ms
scala.concurrent.impl.Promise\$CompletionLatch.apply ()	0.6%	2,701 ms	2,701 ms	2,701 ms	2,701 ms
scala.concurrent.forkjoin.ForkJoinPool.signalWork ()	0%	100 ms	100 ms	100 ms	100 ms
scala.concurrent.impl.Promise\$DefaultPromise.result ()	0%	0.000 ms	0.000 ms	385,164 ms	792 ms

Abbildung 56: CPU Zeiten der nativen Implementierung

4.3.3 Verschachtelung der Kombinatoren

Nach der Analyse der Zugriffszeit auf das Resultat des Futures, wird untersucht wie sich beide Implementierungen verhalten, wenn mehrere Kombinatoren verschachtelt werden. Zuerst betrachtet werden der followedBy-Kombinator der eigenen Implementierung beziehungsweise das native Gegenstück transform. Insgesamt wird jeder Test 1000-mal ausgeführt. Dabei werden die Funktionen 10 bis 100 mal aufgerufen, um die entsprechende Verschachtelungstiefe zu erreichen.

Die Implementierung des Benchmarks ist simpel und folgt dem bekannten Schema (Abbildung 57). Zu Beginn jedes Durchlaufs wird ein neues Future instanziiert und der

Variable `f` zugewiesen. Das Future selbst erzeugt eine Liste mit 10000 Elementen und liefert diese zurück, sobald die Operation abgeschlossen ist. Es ist anzumerken, dass die Liste nicht zu klein sein darf, da die Berechnungszeiten der darauf angewendeten Operationen sonst zu gering ausfallen. Dies hätte den Effekt, dass die gemessene Zeit maßgeblich durch den Overhead, der durch den Einsatz von Threads entsteht, beeinflusst wird. Dies wiederum führt zu verfälschten Ergebnissen.

Nach der Erzeugung des Future-Objekts wird eine `for`-Schleife verwendet, um die vorgegebene Verschachtelungstiefe zu erlangen. Innerhalb dieser Schleife wird die `followedBy`-Funktion respektive die `transform`-Methode verwendet und das Future als Parameter übergeben. Anschließend wird das neue Future der Variable `f` zugewiesen, damit dieses in der nächsten Iteration als Parameter übergeben werden kann. Neben dem Future erwartet `followedBy` eine Funktion als weiteren Parameter. Diese Funktion nimmt in diesem konkreten Fall eine Liste entgegen und erzeugt eine neue Liste desselben Typen. Um dies zu bewerkstelligen wird die `map`-Funktion auf den Eingabeparameter angewendet und jedes Listenelement inkrementiert.

Nach dem Durchlaufen der Schleife wird die `get`-Funktion beziehungsweise die `result`-Methode eingesetzt, um auf das Resultat des äußersten Futures zu warten.

```
val combinations = Gen.range("combinations")(10, 100, 10)

performance of "Combinator Nesting" in {
  measure method "functional" in {
    using(combinations) in { combination =>
      var f = Future(List.fill(10000)(0))
      for (i <- 1 to combination) {
        f = followedBy(f, (list: List[Int]) => list.map(_ + 1))
      }
      get(f)
    }
  }
}
```

Abbildung 57: Verschachtelung mehrere unabhängiger Tasks

Betrachtet man das Messergebnis (Grafik 58), fällt auf, dass beide Lösungen linear sind. Lediglich bei einer Verschachtelungstiefe von 20 gibt es kleinere Abweichungen, die auf Messungenauigkeiten zurückzuführen sind. Zudem zeigt die Grafik, dass die native Lösung schneller ist. Ferner ist zu erkennen, dass der Abstand zwischen den Resultaten größer wird. Während bei 10 Kombinatoren ein Abstand von ~ 1 ms gemessen wird, beträgt der Abstand bei 100 Kombinatoren ~ 5 ms.

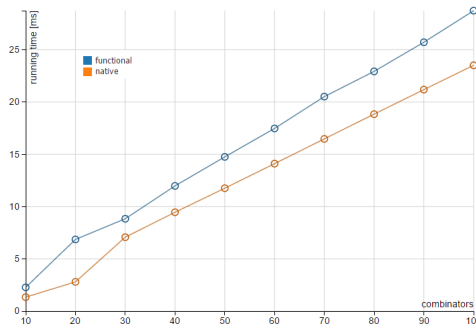


Abbildung 58: Das Ergebnis des Benchmarks beim dem mehrere Operationen verschachtelt werden

Die schlechtere Skalierung der eigenen Implementierung kann dadurch erklärt werden, dass die Ausführung der Futures voneinander abhängig ist. Dadurch muss jedes Future darauf warten, bis das Ergebnis des Vorherigen zur Verfügung steht. Erst hiernach ist die Ausführung der Funktion möglich. Dies sorgt dafür, dass lediglich ein Thread gleichzeitig arbeiten kann, während die Restlichen warten. Zudem kann davon ausgegangen werden, dass das Wecken des folgenden Futures zusätzliche Zeit in Anspruch nimmt.

Die nativen Futures hingegen verwenden Callbacks, welche ausgeführt werden, sobald die Berechnung eines Resultats beendet ist. Zudem besitzt die Standardimplementierung einen Mechanismus, welcher in der Lage ist eine Verschachtelungen zu erkennen. Dieser Mechanismus verlinkt die Futures automatisch und ist darüber hinaus in der Lage die auszuführenden Funktionen zur Wurzel zu transferieren, wodurch eine zentrale Ausführung der Funktionen ermöglicht wird.

Die nächsten Tests befassen sich mit der Untersuchung des Laufzeitverhaltens der eigenen `orAlt`- und nativen `fallbackTo`-Funktion. Hierbei werden zwei Extremfälle betrachtet. Beim ersten Extremfall ist die Ausführung des ersten Futures erfolgreich, wodurch das Ergebnis bereits von Anfang an feststeht. Der zweite Extremfall ist das genaue Gegenteil des ersten Szenarios. Dort scheitert jedes Future, bis auf das Letzte.

Bei diesen Benchmarks wird derselbe Generator wie zuvor eingesetzt (Abbildung 59). Dies hat zur Folge, dass auch hierbei 10 Messpunkte vorliegen und die Funktionen 10 bis 100 mal eingesetzt werden, um die entsprechende Verschachtelungstiefe zu erhalten. Auch bei diesen Tests wird eine Schleife eingesetzt, um die Futures zu verschachteln. Der erste Fall ist einfach zu lösen. Als Erstes wird ein normales Future erzeugt und der Variable `f` zugewiesen. Da alle weiteren Futures nicht relevant sind und kein Ergebnis besitzen müssen, wird die `fail`-Funktion verwendet, um diese zu erzeugen.

Beim zweiten Fall verhält es sich andersherum. Hierbei wird `fail` verwendet, um die

```

val combinations = Gen.range("combinations")(10, 100, 10)

measure method "own orAlt, first succeed" in {
  using(combinations) in { toCombine =>
    var f = Future(42)
    for (i <- 1 to toCombine) {
      f = orAlt(f, fail())
    }
    get(f)
  }
}

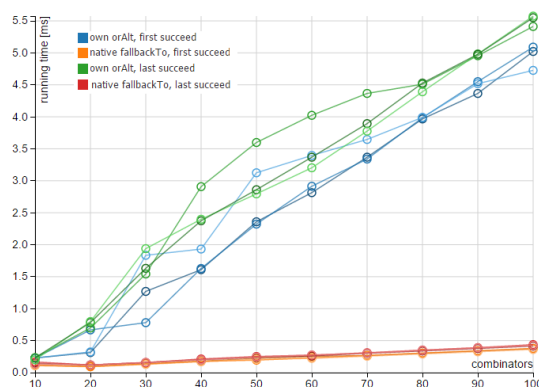
```

Abbildung 59: Verschaltung von Alternativen

Variable `f` zu initialisieren. Zudem wird eine `if`-Anweisung benötigt, um festzustellen, welches `Future` gerade erzeugt wird. Solange der Zähler `i` kleiner als die Anzahl der zu erstellenden `Futures` ist, wird `fail` für die Erzeugung der `Futures` eingesetzt. Lediglich wenn `i` dem Parameter `toCombine` gleicht, wird ein reguläres `Future` kreiert. Auf die Visualisierung eines Beispiels wird verzichtet, da die beiden Fälle sich kaum voneinander unterscheiden.

Die Resultate dieser Benchmarks können der Grafik 60 entnommen werden. Es ist zu erkennen, dass die eigene Implementierung deutlich schlechter abschneidet als die Native. Zudem sieht man, dass das native `Future` konstante Ergebnisse liefert. Weiterhin fällt auf, dass der zweite Extremfall nur unbedeutend langsamer als der Erste ist.

Betrachtet man die Ergebnisse der eigenen Implementierung, erkennt man die schlechtere Skalierung. Dabei ist ein deutlicher Leistungsabfall bei zunehmender Verschachtelungstiefe zu erkennen. Zudem schwankt das Ergebnis teils stark. Ebenfalls erkennbar sind die Unterschiede bei der Ausführung der beiden Extremfälle.

Abbildung 60: Das Ergebnis des `orAlt`-Benchmarks

Es liegt nah, dass das Handling der Futures für die schlechtere Performance verantwortlich ist. Auch bei diesem Test muss jedes Future auf das Resultat des vorherigen Futures warten, bevor weitere Aktionen durchgeführt werden können. Diesbezüglich triggert jedes Future das retry des STM, was wiederum dazu führt, dass der ausführende Thread pausiert und zu einem späteren Zeitpunkt reaktiviert werden muss. Das Pausieren des Threads und die Wiederaufnahme der Arbeit stellt dabei zusätzlichen Overhead dar, welcher zu erhöhten Ausführungszeiten führt.

Die gute Performance der nativen Variante hingegen, ist auf den bereits vorgestellten Callback-Mechanismus zurückzuführen. Da die auszuführenden Tätigkeiten zur Wurzel transferiert werden, kann die Abarbeitung der Aufgaben zentral erfolgen. Hierdurch wird unnötiger Overhead, der durch das Threading entsteht, vermieden. Zudem ist die sequentielle Abarbeitung unproblematisch, da die Futures voneinander abhängig sind.

Um die aufgestellten Hypothesen zu bestätigen, wird abermals ein Profiling der JVM durchgeführt. Wie Abbildung 61 entnommen werden kann, bestätigt das CPU Sampling die Annahme. Die Applikation befindet sich einen Großteil der Zeit im Wartezustand. Dies ist daran zu erkennen, dass der Pool oft inaktiv ist (idle) und der WakeupManager eine hohe CPU Zeit besitzt. Daraus kann geschlussfolgert werden, dass das Warten auf die Teilergebnisse für die schlechte Performance verantwortlich ist.

CPU samples

Thread CPU Time

Snapshot

Thread Dump

Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
scala.concurrent.forkjoin.ForkJoinPool. scan ()	<div></div>	1,073.54... (42.3%)	32,419 ms	1,413,613 ms	41,233 ms
scala.concurrent.stm.ccstm.WakeupManager\$EventImpl\$\$anonfun\$tryAwaitUntil\$1. apply	<div></div>	1,060.43... (41.8%)	11,068 ms	1,060,430 ms	11,068 ms
scala.concurrent.forkjoin.ForkJoinPool. idleAwaitWork ()	<div></div>	334,640 ... (13.2%)	3,383 ms	334,640 ms	3,383 ms
scala.concurrent.forkjoin.ForkJoinPool. tryCompensate ()	<div></div>	14,692 ... (0.6%)	14,692 ms	14,692 ms	14,692 ms
scala.concurrent.forkjoin.ForkJoinPool. signalWork ()	<div></div>	14,667 ... (0.6%)	14,667 ms	15,502 ms	15,502 ms
scala.concurrent.stm.ccstm.InTxnImpl. atomic ()	<div></div>	14,646 ... (0.6%)	14,646 ms	1,119,336 ms	69,589 ms

Abbildung 61: CPU Sampling der orAlt-Funktion

Bei der Betrachtung der Samplingergebnisse der nativen Realisierung (Abbildung 62) sind keinerlei Anzeichen vorzufinden, die auf ein Warten hindeuten.

CPU samplesThread CPU TimeSnapshotThread Dump

Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
concurrent.benchmark.CombinatorNestingBenchmark\$\$\$anonfun\$1\$\$\$anonfun\$apply\$mcV\$	2,996 ms (38.9%)	0.000 ms	2,996 ms	0.000 ms	
concurrent.benchmark.CombinatorNestingBenchmark\$\$\$anonfun\$1\$\$\$anonfun\$apply\$mcV\$	2,133 ms (27.7%)	2,133 ms	2,133 ms	2,133 ms	
scala.runtime.BoxesRunTime. boxToDouble ()	698 ms (9.1%)	698 ms	698 ms	698 ms	
scala.collection.LinearSeqOptimized\$class. foldLeft ()	399 ms (5.2%)	399 ms	1,098 ms	1,098 ms	

Abbildung 62: CPU Sampling der transform-Methode

Die bisherigen Samplingresultate geben Aufschluss darüber, womit die beiden Anwendungen beschäftigt sind und weshalb die eigene Implementierung schlechtere Ergebnisse liefert. Dies lässt jedoch keine Rückschlüsse darüber zu, wie viele Threads eingesetzt werden und womit diese beschäftigt sind. Aufgrund dessen wird eine zusätzliche Analyse der Thread CPU Zeit durchgeführt.

Betrachtet man zuerst das Resultat der eigenen Implementierung, so fällt auf, dass insgesamt 110 Threads mit der Abarbeitung der Aufgaben beschäftigt sind. Ein Großteil der erschaffenen Threads sind dabei Worker des eingesetzten Pools und für die Ausführung der Future verantwortlich. Wie die Grafik 63 zeigt, ist die Ausführungszeit der einzelnen Worker sehr gering. Es ist anzumerken, dass die Grafik nur ein Ausschnitt ist und nicht alle Worker zeigt.

Thread Name	Thread CPU Time [%]	Thread CPU Time [ms]	Thread CPU Time [ms]/sec
main		4,025.773 (0.0%)	62.703
RMI TCP Connection(1)-192.168.246.231		3,155.813 (0.0%)	155.699
ForkJoinPool-1-worker-13		2,096.833 (0.0%)	44.827
ForkJoinPool-1-worker-115		2,053.699 (0.0%)	42.26
ForkJoinPool-1-worker-41		2,039.59 (0.0%)	44.282
ForkJoinPool-1-worker-73		2,038.485 (0.0%)	37.777
ForkJoinPool-1-worker-23		2,037.753 (0.0%)	41.291
ForkJoinPool-1-worker-17		2,030.924 (0.0%)	42.926

Abbildung 63: Thread CPU Sampling der orAlt-Funktion

Im Gegensatz dazu werden bei der nativen Implementierung nur 12 Threads erzeugt. Unter diesen 12 Threads befindet sich exakt ein Worker, der für die Berechnung der Futures zuständig ist. Dies bestätigt die zweite Annahme, dass die gute Skalierung und Performance auf die Zusammenfassung der Arbeit und den Transfer zur Wurzel zurückzuführen sind.

Thread Name	Thread CPU Ti...	Thread CPU Time [...]	Thread CPU Time [...]
main		5,916.317 (0.0%)	962.953
RMI TCP Connection(1)-192.168.246.231		354.216 (0.0%)	13.901
ForkJoinPool-1-worker-43		197.39 (0.0%)	33.379
Attach Listener		144.77 (0.0%)	0

Abbildung 64: Thread CPU Sampling der transform-Methode

Zuletzt sollen die first-Funktionen beider Implementierungen miteinander verglichen werden. Wie auch zuvor wird zuerst der Benchmark selbst beschrieben. Anschließend erfolgt die Analyse der Resultate. Bei diesem Benchmark werden 10 bis 100 Futures erzeugt, in

einer Liste gespeichert und anschließend als Parameter beim Aufruf der first-Funktion verwendet.

Das Schreiben des Resultats eines Futures und der damit verbundene Abschluss der Kalkulation wird bei diesen Tests, um eine gewisse Zeit verzögert. Die abzuwartende Zeit beträgt mindestens 500 Mikrosekunden und erhöht sich bei jeder Schleifeniteration um eine weitere Mikrosekunde. Der auf den Basiswert addierte Wert ist dabei von der Schleifenvariable *i* und daher auch von der Initialisierungsreihenfolge der Futures abhängig. Hieraus resultiert eine nach Wartezeit aufsteigend sortierte Liste von Futures.

Um die Verzögerung zu realisieren, wird eine der beiden sleep-Methoden der Klasse Thread eingesetzt. Die erste Methode gestattet das Pausieren eines Threads für die angegebene Anzahl von Millisekunden. Die zweite Methode besitzt einen weiteren Parameter, der die Angabe von Nanosekunden erlaubt. Diese werden auf die angegebenen Millisekunden addiert. Wie dem Codeschnipsel 65 entnommen werden kann, beträgt die zu wartende Zeit bei diesem Test 0 Millisekunden und 500000 + *x* Nanosekunden.

Wie auch beim Benchmark zuvor, werden auch hier zwei Fälle betrachtet. Der erste Fall befasst sich mit dem eben beschriebenen Szenario. Dabei wird eine Liste von Futures erzeugt. Diese sind der Wartezeit nach absteigend sortiert. Der zweite Testfall ist das exakte Gegenteil des Ersten. Dort besteht die Liste aus aufsteigenden sortierten Futures, wodurch die Kalkulation des ersten Futures als letztes abgeschlossen ist.

```
val concurrentTasks = Gen.range("concurrentTasks")(10, 100, 10)

measure method "own first, first fastest" in {
  using(concurrentTasks) in { tasks =>
    var futures = ListBuffer.empty[Future[Int]]
    for (i <- 0 until tasks) {
      futures += Future {
        Thread.sleep(0, 500000 + (i * 1000))
        i
      }
    }
    get(first(futures))
  }
}
```

Abbildung 65: Benchmark der first-Funktion

Bei näherer Betrachtung der Ergebnisse (Grafik 66) fällt auf, dass die Ergebnisse auch bei mehrfacher Ausführung der Tests konstant bleiben und keiner starken Schwankung unterliegen. Bei einer geringen Anzahl von Futures (10 bis 30) liegen die Ergebnisse der beiden Varianten dicht beieinander. Mit zunehmender Anzahl von Futures ist jedoch

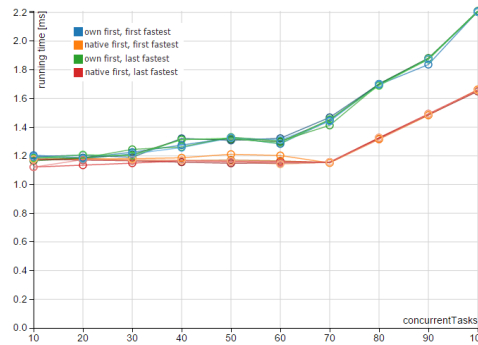


Abbildung 66: Ergebnisse des Benchmarks der first-Funktion (mit sleep)

ein Leistungsabfall bei der eigenen Variante erkennbar. Ab etwa 70 Futures bricht die Leistung beider Varianten ein.

Der erste Leistungsabfall, der nur bei der eigenen Implementierung auftritt, ist dadurch zu erklären, dass die parallele Ausführung aller Tasks nicht mehr möglich ist, sobald mehr als 32 Futures eingesetzt werden. Hierfür verantwortlich ist die Tatsache, dass für jedes Future zwei Runnables erzeugt und vom Threadpool ausgeführt werden. Der erste Task wird benötigt, um das Future selbst auszuführen. Der zweite Task ist mit der Erfüllung des Promise beschäftigt. Dabei wartet dieser solange bis das Ergebnis berechnet ist. Da STM diese Threads automatisch pausiert, um ein busy waiting zu vermeiden, kann es zu Verzögerungen bei der Wiederaufnahme der Arbeit kommen.

Der zweite Leistungseinbruch tritt bei beiden Implementierungen ab einer Anzahl von 70 Futures auf. Hierfür verantwortlich ist ebenfalls der Overhead, der beim Threading entsteht. Hierbei kommt es bei beiden Implementierungen zu vermehrten Kontextwechseln, da nun bei beiden Varianten die Anzahl der Tasks die Anzahl der CPUs übersteigt. Ein zusätzliches Problem, besteht darin, dass das Ergebnis nicht den Erwartungen entspricht. Alle Testergebnisse liegen über einer Millisekunde, wobei die einzelnen Threads nur etwas mehr als 500 Mikrosekunden pausiert sein sollten. Nach der Analyse der sleep-Methode wird klar, dass der Parameter der für die Bestimmung der Nanosekunden gedacht ist quasi ignoriert wird. Die Methode prüft dabei lediglich, ob die angegebenen Nanosekunden größer als 500.000 sind. Ist dies der Fall, werden lediglich die ebenfalls übergebenen Millisekunden inkrementiert. Andernfalls wird der Parameter komplett ignoriert.

Weil eine Steigerung der Verzögerung um eine Millisekunde pro Iteration nicht der gewünschten Granularität entspricht, muss ein anderer Weg gefunden werden, um die Fertigstellung des Futures zu verzögern. Die wohl präziseste Methode, um dies zu be-

werkstelligen ist ein busy waiting[23]. Die Implementierung eines solchen Mechanismus ist recht trivial und kann der Abbildung 67 entnommen werden. Die Funktion nimmt einen Long als Parameter entgegen. Dieser repräsentiert die Zeit in Mikrosekunden und gibt an wie lange gewartet werden soll. Zu Beginn der Funktion wird der Endzeitpunkt berechnet. Daraufhin wird eine Schleife gestartet, die keinerlei Instruktionen besitzt. Ist die Bedingung erfüllt und der Endzeitpunkt überschritten, wird die Schleife beendet und die Funktion verlassen.

Die inline Annotation besagt, dass der Compiler besonders hartnäckig versuchen soll den Methodenaufruf durch den Funktionscode zu ersetzen, um zusätzlichen Overhead durch den Funktionsaufruf zu vermeiden.

Messungen dieser Funktion haben ergeben, dass die Genauigkeit dieser Funktion nahezu 100% beträgt. Zwischen zwei Aufrufen von nanoTime vergehen circa 40 Nanosekunden. Weiterhin ist anzumerken, dass diese Funktion nur für die Benchmarks eingesetzt werden sollte, da sonst Prozessorressourcen verschwendet werden.

```
@inline
def busyWait(micros: Long) = {
    val waitUntil = System.nanoTime() + (micros * 1000)
    while (waitUntil > System.nanoTime()) {}
}
```

Abbildung 67: Die Funktion busyWait

Nach der Ersetzung der sleep-Methode durch die busyWait-Funktion und der Reduzierung der nebenläufigen Tasks von 100 auf 60, wird der Benchmark erneut durchgeführt. Die Ergebnisse des Tests sind in Abbildung 68 visualisiert.

Die Ergebnisse aller Tests weisen Schwankungen auf, wobei diese bei der nativen Variante deutlich geringer ausfallen und gegen Ende kaum noch vorhanden sind. Es kann zudem festgestellt werden, dass die native Variante auch bei diesen Benchmarks etwas besser abschneidet. Eine mehrfache Ausführung des Tests zeigt zudem, dass die Ergebnisse zwar leicht variieren, jedoch immer zwischen 0,5 und 0,7 Millisekunden liegen. Auf die Visualisierung der weiteren Testdurchläufe wird verzichtet, da die Grafik sonst unübersichtlich wird.

Ferner fällt auf, dass sich die Leistung der eigenen Implementierung, ab 40 Futures, verschlechtert. Ein ähnliches Verhalten bei der nativen Umsetzung ist nicht erkennbar. Der Grund dafür ist, dass das native Future selbst für die Erfüllung des Promise verantwortlich ist. Bei der eigenen Implementierung wiederum, obliegt diese Aufgabe einem anderen Thread. Dieser wird vom STM in Kenntnis gesetzt, dass die Berechnung des Futures ab-

geschlossen ist. Hierbei ist nicht transparent, wie oft dies geschieht und ob mehrere Threads gleichzeitig informiert werden. Eine weitere Unbekannte ist der Threadpool, da auch hier nicht transparent ist, wie dieser das Warten der einzelnen Tasks handhabt und wie der nächste auszuführenden Task bestimmt wird.

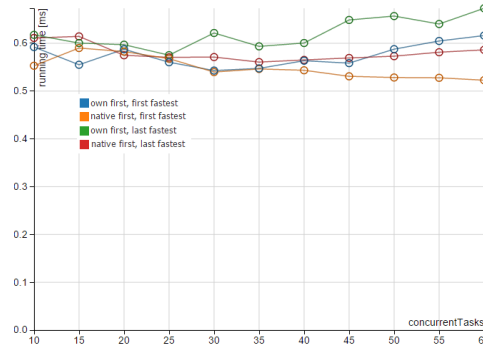


Abbildung 68: Ergebnisse des Benchmarks der first-Funktion (mit busy waiting)

Um die Resultate des first-Benchmarks besser zu verstehen, wird ein weiterer Test benötigt. Dabei wird analysiert, wie hoch die Genauigkeit der beiden Implementierungen ist. Dazu wird die first-Funktion 1000-mal ausgeführt und das Ergebnis jedes Durchlaufs gespeichert. Ein spezieller Mesurer zählt dabei das Auftreten der verschiedenen Antworten und speichert diese in einer Map. Tritt ein Wert mehrfach auf, so wird der entsprechende Wert in der Map inkrementiert. Am Ende wird das Ergebnis ausgegeben. Das Resultat dieser Messung (Abbildung 69) zeigt, dass die Genauigkeit der nativen Implementierung $\sim 90\%$ beträgt, wenn die Liste aufsteigend sortiert ist. Die Präzision der eigenen Implementierung liegt bei etwa $50\% - 80\%$ und fällt bei 60 Futures sogar auf 25% . Hierfür verantwortlich ist die bereits beschriebene Tatsache, dass die Erfüllung des Promises von einem anderen Task durchgeführt wird. Es besteht eine hohe Wahrscheinlichkeit, dass dieser Task noch keinem Thread zugewiesen ist. Dies wiederum macht die Erfüllung des Promises unmöglich.

Die Genauigkeit beider Implementierungen, bei einer absteigend sortierten Liste, ist nahe null. Hierfür verantwortlich ist der geringe Unterschied der Wartezeiten. Dies sorgt dafür, dass früher gestartete Futures bereits abgeschlossen sein können und das Promise erfüllt haben. Die eigene Implementierung besitzt in diesem Fall einen weiteren Nachteil. Dort werden die Runnables, die für die Erfüllung des Promises verantwortlich sind, in umgekehrter Reihenfolge an den Threadpool übergeben. Somit kann es vorkommen, dass das schnellste Future fertig ist, das Promise jedoch nicht erfüllt werden kann, da das zweite Runnable noch nicht gestartet worden ist.

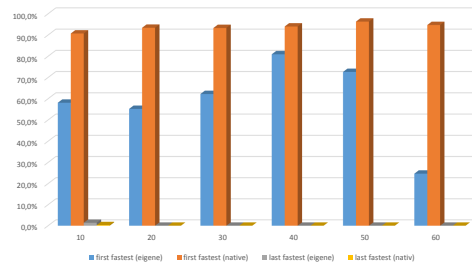


Abbildung 69: Genauigkeit der first-Funktion in Abhängigkeit von der Anzahl der auszuführenden Futures

4.3.4 Sleeping Barber mit Promises

Beim nächsten Benchmark werden Futures und Promises eingesetzt, um das Sleeping Barber Problem zu lösen. Hierbei gibt es einen Barbier, der sich in seinem Laden aufhält und auf Kunden wartet, um diesen die Haare zu schneiden. Treffen mehrere Kunden gleichzeitig ein, wählt der Barbier einen zufälligen Kunden aus und schneidet dessen Haare.

Sämtliche Funktionalität, die für die Lösung dieses Problems benötigt wird, ist in der Funktion `serve` untergebracht (Abbildung 70). Die Funktion besitzt einen Parameter, mit dem sich bestimmen lässt, wie viele Kunden einen Haarschnitt bekommen sollen. Bevor der Barbier damit beginnt den Kunden die Haare zu schneiden, wird eine Liste erstellt. Die Größe dieser entspricht der vorgegeben Anzahl von Kunden. Daraufhin wird eine while-Schleife ausgeführt. Die Ausführung dieser endet, sobald alle Kunden abge-

```
def serve(customersToServe: Int): Unit = {
  var customers = (for (customer <- 1 to customersToServe) yield
    customer).toList
  while (customers.nonEmpty) {
    val barber = Promise[Int]()
    val done = Promise[Int]()
    def cutHair(customer: Int) = if (trySuccess(done, customer)) {
      customers = customers.filterNot(i => i == customer)
    }
    onSuccess(barber.future, cutHair)
    for (customer <- customers) Future { Thread.sleep(1)
      trySuccess(barber, customer)
    }
    get(done.future)
  }
}
```

Abbildung 70: Sleeping Barber Implementierung

arbeitet worden sind.

In jeder Iteration werden zwei neue Promises erzeugt. Das erste Promise repräsentiert den Barbier. Das Zweite wird verwendet, um die Fertigstellung des Haarschneidens zu signalisieren. Der Vorgang des Haarschneidens selbst wird mit Hilfe eines Closure realisiert. Innerhalb des Closures wird die `trySuccess`-Funktion auf das Promise `done` angewendet. Gelingt dies, ist der Kunde fertig und kann aus der Warteschlange entfernt werden.

Das Closure wird als Callback für die `onSuccess`-Funktion verwendet. Dieses Callback wird ausgeführt, sobald das Barber Promise erfüllt ist. Die Erfüllung dieses Promise obliegt einem in der Warteschlange stehenden Kunden. Jeder Kunde besitzt dabei dieselbe Chance den Haarschnitt als Nächstes zu bekommen. Dies wird dadurch realisiert, dass jeder Kunde eine Millisekunde wartet und anschließend versucht das Barber Promise zu erfüllen. Eine Iteration ist beendet, sobald das Promise `done` erfüllt ist.

Die Implementierung mit nativen Futures unterscheidet sich hierbei ein wenig von der vorgestellten Version. Die dortige `onSuccess`-Methode erwartet anstelle eines Callbacks eine partiellen Funktion als Parameter. Diesbezüglich wird das Closure entfernt und die Funktionalität in der partiellen Funktion ausgeführt.

Wie die Grafik 71 zeigt, weisen beide Implementierungen bei diesem Benchmark nahezu identische Laufzeiten auf. Der Grund hierfür ist der Einsatz der `sleep`-Methode, um die Ausführung von `trySuccess` zu verzögern und jedem Kunden dieselbe Chance zu gewähren seinen Haarschnitt als Nächstes zu bekommen. Die Abweichung der beiden Geraden lässt sich auf Schwankungen bei der Messung zurückführen.

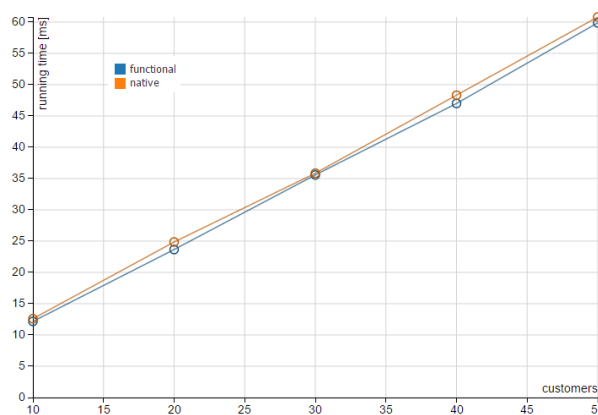


Abbildung 71: Messergebnisse des Sleeping Barber Benchmarks

4.3.5 Santa Clause mit Promises

Das Santa Clause Problem ist das zweite bekannte Concurrency Problem, welches betrachtet und mit Promises gelöst werden soll. Santa verfällt dabei regelmäßig in Schlaf und muss von seinen Helfern geweckt werden. Hierzu in der Lage sind alle 9 Rentiere oder eine Gruppe von 3 Elfen. Wecken die Rentiere Santa, begibt dieser sich zu seinem Schlitten und liefert die Geschenke aus. Wird er von der Gruppe Elfen geweckt, dann muss er diese unterweisen und inspirieren. Abweichend von der originalen Aufgabenstellung, besitzt die hier vorgestellte Umsetzung keine Priorisierung der Rentiere, falls beide Gruppen gleichzeitig vor Santas Tür stehen.

Die Implementierung selbst ist wie folgt aufgebaut. Die Hauptfunktionalität ist in der Funktion `santaLoop` untergebracht (Abbildung 72). Diese besitzt einen Parameter. Mit diesem wird definiert, wie oft Santa geweckt wird, bevor das Programm terminiert. Bevor der eigentliche Durchlauf beginnt, werden die beiden Gruppen initialisiert. Hierzu wird die Funktion `collectHelper` verwendet. Diese wird im späteren Verlauf dieses Abschnitts vorgestellt.

Nach der Initialisierung der beiden Gruppen beginnt die Ausführung der Schleife. Diese endet, sobald die vorgegebene Anzahl Runden erreicht ist. Die `first`-Funktion wird eingesetzt, um zu bestimmen, welche Gruppe zuerst vollständig ist. Zudem wird die Funktion

```
def santaLoop(rounds: Int) = {
  var elves = collectHelper(Future(Elf), 3)
  var reindeers = collectHelper(Future(Reindeer), 9)
  for (a <- 0 until rounds) {
    val group = when(first(Promise[Helper](), elves :: reindeers ::
      Nil), isSantaSleeping)
    get(group) match {
      case Some(Elf) =>
        santaSleeps = false
        Thread.sleep(1)
        santaSleeps = true
        elves = collectHelper(Future(Elf), 3)
      case Some(Reindeer) =>
        santaSleeps = false
        Thread.sleep(1)
        santaSleeps = true
        reindeers = collectHelper(Future(Reindeer), 9)
    }
  }
}
```

Abbildung 72: Santa Clause Implementierung

```
def collectHelper[T <: Helper](helper: Future[T], toCollect: Int):
  Future[T] = {
    val promise = Promise[T]()
    tryCompleteWith(promise, collectN(toCollect, helper))
    promise.future
  }
```

Abbildung 73: Hilfsfunktion collectHelper

when eingesetzt, um sicherzustellen das Santa schläft.

Durch den Einsatz von Pattern Matching lässt sich bestimmen, welche Gruppe zuerst fertig ist. Beide Gruppen wecken Santa zuerst, warten danach 1 Millisekunde, um deren Aufgabe zu erfüllen und verlassen anschließend das Büro, da Santa wieder eingeschlafen ist. Nach dem Verlassen des Büros, beginnt die Gruppe sich wieder zu sammeln.

Die Hilfsfunktion collectHelper wird eingesetzt, um Helfer zu gruppieren (Abbildung 73). Diese Funktion besitzt zwei Parameter und gibt ein Future zurück, welches eine Gruppe von Helfern repräsentiert. Der erste Parameter bestimmt dabei, um welchen Typ Helfer es sich handelt. Der zweite Parameter gibt vor, wie viele Helfer für eine vollständige Gruppe benötigt werden.

Im Rumpf der Funktion wird ein neues Promise erzeugt. Der Typ des Promises ist dabei abhängig vom Typen des Helfers. Daraufhin wird die tryCompleteWith-Funktion verwendet, um dieses Promise zu erfüllen. Hierbei kommt die collectN-Funktion zum Einsatz. Diese rekursive Funktion wird solange angewendet, bis genügend Helfer gesammelt worden sind. Abschließend wird das noch nicht erfüllte Future des Promises zurückgegeben.

Bei der Betrachtung der Zeitmessung (Abbildung 74) wird deutlich, dass beide Benchmarks nahezu identische Laufzeiten besitzt. Dies ist darauf zurückzuführen, dass die

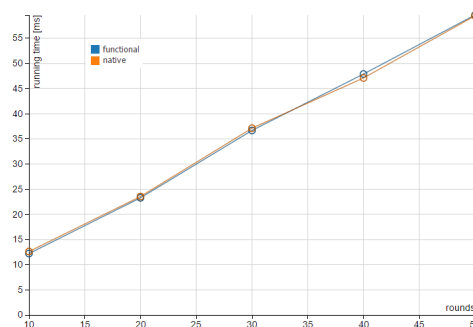


Abbildung 74: Ergebnis des Santa Clause Benchmarks

sleep-Funktion eingesetzt wird. Hierdurch wird eine untere Grenze für die Ausführungszeit geschaffen. Das Warte sorgt dafür, dass die inaktive Gruppe weiterhin gesammelt wird und diese das Promise in der nächsten Runde sofort erfüllt.

Während der Einsatz der sleep-Methode beim Sleeping Barber Problem adäquat ist, da jeder Kunde dieselbe Chance besitzen soll als Nächstes an die Reihe zu kommen, ist dies bei diesem Problem nicht der Fall. Deshalb wird ein zweiter Benchmark durchgeführt. Hierbei wird auf das Pausieren der Threads verzichtet. Gleichzeitig wird die Anzahl der Runden erhöht.

Im Vergleich zum ersten Benchmark hat sich die Ausführungszeit trotz einer Verzehnfachung der Runden deutlich reduziert. Während auch hierbei Schwankungen erkennbar sind, ist festzuhalten, dass die eigene Implementierung bei diesem Problem besser skaliert und immer schneller ist als das native Gegenstück.

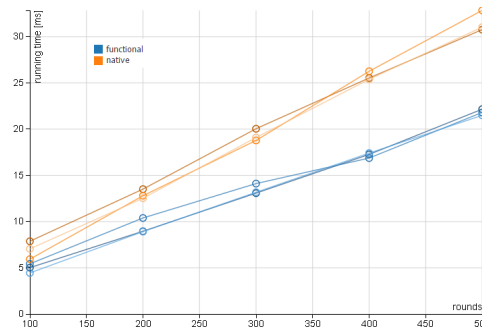


Abbildung 75: Ergebnis des Santa Clause Benchmarks (ohne sleep)

4.3.6 Dining Philosophers mit Promises

Der letzte Test, der betrachtet werden soll, befasst sich mit dem Dining Philosophers Problem. Das Problem wird hierbei unter Einsatz von 3 unterschiedlichen Mechanismen gelöst. Anschließend werden die Ergebnisse miteinander verglichen. Die 3 Mechanismen sind Sperren, STM sowie die eigene Promise Implementierung. Der Einsatz der nativen Promises kommt hierbei nicht infrage, da diese keine Funktion besitzen, welche es erlaubt mehrere Promises gleichzeitig zu erfüllen. Da die Implementierung einer solchen Funktion nur unter Verwendung von Sperren möglich ist und eine nicht triviale Aufgabe darstellt, wird darauf verzichtet.

Anders als bei der bekannten Lösung des Problems, wird die Aufgabestellung dahingehend angepasst, dass mehrere voneinander unabhängige Runden ausgeführt werden. In jeder Runde versuchen die Philosophen das Essbesteck zu greifen und zu essen. Dabei

besitzt jeder Philosoph die gleiche Chance.

Die Hauptfunktion `philosopherRace` (Abbildung 76) wird bei jedem Benchmark 100-mal ausgeführt. Die Anzahl der teilnehmenden Philosophen ist dabei abhängig vom übergebenen Parameter. Bevor eine Runde endet, muss jeder Philosoph versucht haben das Essbesteck zu erlangen und zu essen. Da das Ergebnis dabei nicht in einem Promise gespeichert wird, wird eine Barriere benötigt. Diese wird durch ein `CountDownLatch` realisiert.

Das Essbesteck wird durch Promises repräsentiert. Diese werden bei der Aufnahme durch einen Philosophen erfüllt. Zudem muss eine entsprechende Anzahl Philosophen erzeugt werden. Dies wird durch den Aufruf der Funktion `philosopher` bewerkstelligt. Diese Funktion liefert ein Future zurück, welches das Gelingen oder Scheitern des Vorgangs wiedergibt.

Nach der Erzeugung der Philosophen werden die beiden Callback-Funktionen `onSuccess` und `onFailure` eingesetzt, um das Latch zu dekrementieren.

```
def philosopherRace(philos: Int): Unit = {  
  for (a <- 0 until 100) {  
    val latch = new CountDownLatch(philos)  
    val forks = (for (i <- 1 to philos) yield Promise[Int]()).toList  
    val philosophers = for (i <- 0 until philos) yield  
      philosopher(i + 1, forks(i), forks((i + 1) % forks.size))  
    for (philosopher <- philosophers) {  
      onSuccess(philosopher, (philosopher: Philosopher) =>  
        latch.countDown())  
      onFailure(philosopher, () => latch.countDown())  
    }  
    latch.await()  
  }  
}
```

Abbildung 76: Philosophers mit Promises und MultiTrySuccess

Die Funktion, die für die Erzeugung der Philosophen verwendet wird, kann Abbildung 77 entnommen werden. Diese besitzt drei Parameter. Die Id kennzeichnet einen Philosophen und wird für die Erfüllung der Promises benötigt. Die beiden Promises repräsentieren das Essbesteck und sollen erfüllt werden. Der Rückgabewert dieser Funktion ist ein Future, welches angibt, ob der Philosoph essen konnte oder nachdenken musste.

Im Rumpf der Funktion erfolgt lediglich die Erzeugung eines neuen Futures, welches sofort zurückgegeben wird. Innerhalb des Futures wird die `multiTrySuccess`-Funktion eingesetzt, um die beiden Promises zu erfüllen. Da `multiTrySuccess` einen Booleschen Rückgabewert besitzt, kann geprüft werden, ob die Ausführung gelungen ist. Konnte

```
def philosopher(id: Int, left: Promise[Int], right: Promise[Int]):
  Future[Philosopher] = {
    Future { multiTrySuccess((left, id) :: (right, id) :: Nil) match {
      case false => null
      case _ => Philosopher(id)
    }
  }
}
```

Abbildung 77: Die Funktion `philosopher`, die für die Erzeugung eines Philosophen zuständig ist

der Philosoph beide Promises erfüllen, wird ein neues Philosophen-Objekt zurückgegeben. Andernfalls wird `null` zurückgeliefert, um das Scheitern zu signalisieren. STM stellt hierbei sicher, dass die Operation korrekt ausgeführt wird. Versuchen zwei Philosophen dasselbe Essbesteck zu nehmen, erkennt STM dies und versucht den Konflikt zu lösen. Dies kann dazu führen, dass `multiTrySuccess` mehrfach ausgeführt wird, bis ein Philosoph kein oder beide Essbestecke genommen hat.

Für die Implementierung der anderen beiden Alternativen muss die Funktion `philosopherRace` geringfügig modifiziert werden. Zum einen wird das Essbesteck durch den entsprechenden Concurrency Mechanismus ersetzt. Dies sind `ReentrantLocks` bei Sperren respektive `Ref-Objekte` bei der STM Variante. Zum anderen werden die Philosophen in beiden Alternativen als Threads realisiert. Aufgrund dessen entfällt der Aufruf der Callback-Funktionen. Anstelle dessen werden die einzelnen Philosophen einem Thread-pool zugewiesen, welcher für die Ausführung dieser verantwortlich ist.

```
private case class Philosopher(id: Int, left: Lock,
  right: Lock, latch: CountdownLatch) extends Runnable {
  override def run() {
    try {
      if (left.tryLock(5, TimeUnit.NANOSECONDS)) {
        if (right.tryLock(5, TimeUnit.NANOSECONDS)) {
          right.unlock()
        }
        left.unlock()
      }
      latch.countDown()
    } catch { case e: Exception => e.printStackTrace() }
  }
}
```

Abbildung 78: Philosophers mit Locks

Die Philosophen-Klasse der sperrenbasierten Alternative ist im Codeschnipsel 78 abgebildet. Diese Klasse ähnelt der `philosopher`-Funktion strukturell und erwartet sowohl eine `Id` als auch das linke und rechte Essbesteck als Konstruktorargument. Zusätzlich muss das `CountDownLatch` als Parameter übergeben werden, da das Latch nicht mehr innerhalb eines Callbacks dekrementiert werden kann.

In der `run`-Methode ist jegliche Funktionalität untergebracht. Dabei wird zuerst versucht die linke und daraufhin die rechte Sperre zu erwerben. Um das Verhalten des ersten Ansatzes zu imitieren, wird der Versuch die Sperren zu erwerben auf 5 Nanosekunden begrenzt. Warten die Philosophen länger, können zu viele essen, da die anderen Philosophen das Essbesteck zurücklegen, nachdem sie gegessen haben.

Unabhängig davon, ob ein Philosoph essen konnte oder nicht wird das Latch abschließend dekrementiert.

Zuletzt soll die STM-basierte Variante der Klasse `Philosopher` vorgestellt werden (Abbildung 79). Die ebenfalls als Threads realisierten Philosophen besitzen dieselben Parameter wie zuvor. Lediglich der Typ des Essbestecks muss hierbei durch Ref-Objekte vom Typen `Option[Int]` ersetzt werden.

Die `run`-Methode muss ebenfalls angepasst werden. Bei diesem Ansatz wird ein `atomic`-Block verwendet, um auf die Referenzobjekte zugreifen zu können. Jeder Philosoph prüft dabei zuerst, ob das linke und rechte Essbesteck bereits von einem anderen aufgenommen worden ist. Sollten beide frei sein, nimmt der Philosoph das Besteck auf und isst. Wie zuvor wird das Latch, unabhängig davon ob der Philosoph eine Mahlzeit zu sich nehmen konnte, dekrementiert.

```
private case class Philosopher(id: Int, left: Ref[Option[Int]],
    right: Ref[Option[Int]], latch: CountDownLatch) extends Runnable {
  override def run() {
    atomic { implicit txn =>
      if (left().isEmpty && right().isEmpty) {
        left() = Some(id)
        right() = Some(id)
      }
    }
    latch.countDown()
  }
}
```

Abbildung 79: Philosophers mit STM

Die Zeitmessung zeigt, dass die Varianten mit Sperren und STM deutlich schneller sind als die Implementierung mit Promises. Kleinere Schwankungen der STM Realisierung sind auf Ausreißer zurückzuführen. Für die deutlich schlechtere Performance der Promi-sevariante ist die multiTrySuccess-Funktion verantwortlich. Diese sorgt dafür, dass es zu vielen retries kommt, da die Philosophen gleichzeitig versuchen das Essbesteck zu nehmen. Dies wiederum sorgt dafür, dass die Funktion mehrfach ausgeführt werden muss, bis einer der beiden Philosophen das Essbesteck nimmt. Da dies bei mehrere Philosophen vorkommen kann, muss das STM dafür Sorge tragen, dass jeder Konflikt gelöst ist, bevor ein Durchgang beendet werden kann.

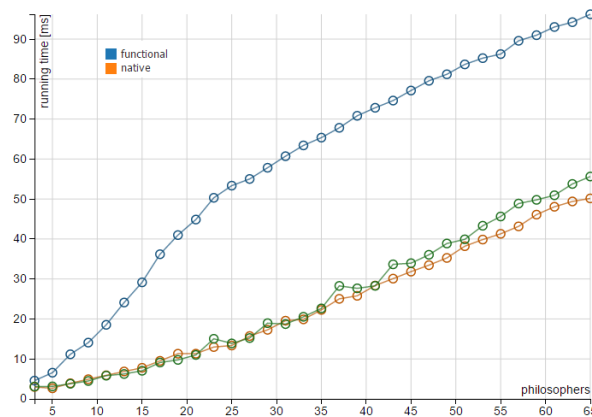


Abbildung 80: Ergebnis des Dining Philosophers Benchmarks

5 Optimierung der Implementierung

Die Analyse der Benchmarks hat gezeigt, dass die eigene, funktionale Implementierung teils signifikant schlechtere Resultate als das native Gegenstück liefert.

Um das identifizierte Problem zu beseitigen, muss zuerst untersucht werden, wodurch die schlechtere Performance und Skalierbarkeit verursacht wird. Aufgrund dessen erfolgt eine Analyse der Schwachstellen der eigenen Implementierung (Abschnitt 5.1). Anschließend werden die vorgenommenen Optimierungen beschrieben (Abschnitt 5.2).

5.1 Analyse der Schwachstellen

Die in dieser Arbeit vorgestellte Future Implementierung ist hochgradig asynchron und besitzt nur eine blockende Funktion. Diese ist `get` und wird eingesetzt, um das Resultat eines Futures zu lesen. Falls dieses zum Zeitpunkt des Aufrufs nicht vorliegt, wird der aktuelle Thread pausiert. Anschließend wird solange gewartet bis das Ergebnis verfügbar ist. Um die Asynchronität zu realisieren, werden bei jedem Funktionsaufruf `Runnables` erzeugt und an einen Threadpool übergeben. Der Threadpool führt die `Runnables` in einem Thread aus. Die Ausführung mehrerer `Runnables` ist bei einer geringen Verschachtelungstiefe unproblematisch und kaum spürbar, führt jedoch bereits hierbei dazu, dass Threads unnötigerweise gestartet, pausiert und geweckt werden müssen.

Die Verschachtelung von Futures erzeugt eine transitive Abhängigkeit zwischen diesen, da das Ergebnis eines Futures von allen vorherigen Resultaten abhängig ist. Dieser Abhängigkeit geschuldet, muss jedes Future auf das Ergebnis des Vorherigen warten. Dies geschieht durch das Blocken des jeweiligen Threads (Abbildung 81). STM pausiert diese Threads automatisch und weckt sie sobald das Resultat bereitsteht. Hierdurch wird vermieden, dass CPU Ressourcen verschwendet werden. Jedoch erzeugt dies gleichzeitig zusätzlichen Overhead, da das STM einerseits alle pausierten Threads kennen muss und andererseits das Wecken und Wiederausführen Zeit in Anspruch nimmt. Der Overhead sorgt wiederum dafür, dass die Ausführung der Futures etwas mehr Zeit benötigt als nötig.

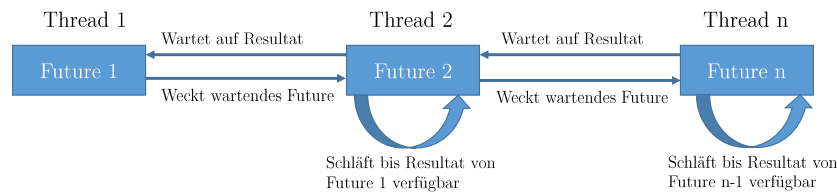


Abbildung 81: Verhalten der Implementierung vor der Optimierung

5.2 Optimierung

Nach der Analyse der Schwachstellen, wird erläutert, welche Anpassungen vorgenommen werden könnten, um die Implementierung zu optimieren. Beim ersten Optimierungsversuch werden native Futures eingesetzt, um die Kalkulation im Hintergrund auszuführen (Abschnitt 5.2.1). Der zweite Optimierungsversuch sieht vor Callbacks einzusetzen, um die Berechnung von Folgeaufgaben solange zu verzögern, bis alle Voraussetzungen erfüllt sind (Abschnitt 5.2.2). Abschließend wird betrachtet, welche Auswirkungen die Optimierungen auf die Leistungsfähigkeit der Implementierung haben (Abschnitt 5.2.3).

5.2.1 Erster Optimierungsversuch

Der erste Optimierungsversuch besteht daraus, dass die Ausführung der Tasks überarbeitet wird. Hierbei wird die Erzeugung des Runnables und die anschließende Übergabe an den Threadpool durch den Aufruf des nativen Futures ersetzt. Es soll getestet werden, ob die ausgeklügelten Mechanismen des nativen Futures zur Verbesserung der Laufzeit der eigenen Implementierung führen.

Die einzige Änderung die hierfür vonnöten ist, besteht darin, dass die `forkIO`-Funktion angepasst werden muss. Dort wird nun anstelle der Erzeugung des Runnables und dem Aufruf der `execute`-Methode des Threadpools ein neues Future erzeugt (Abbildung 82). Das Future übernimmt nun die asynchrone Ausführung des Tasks.

```
private def forkIO(body: => Unit) = {
  scala.concurrent.Future(body)
}
```

Abbildung 82: Die angepasste `forkIO`-Funktion, die nun ein neues Future erzeugt, um die Berechnung asynchron auszuführen

Nach der Änderung, werden die Benchmarks erneut ausgeführt. Da die eigene Implementierung lediglich beim Einsatz von mehreren Kombinatoren schlechter abgeschnitten hat, werden nur diese Tests betrachtet.

Die erneute Ausführung der Tests zeigt, dass die Anpassung zu keinerlei Verbesserung geführt hat. Da die Resultate den Ergebnissen, die in Abschnitt 4.3.3 vorgestellt worden sind, gleichen, wird auf die Visualisierung verzichtet.

5.2.2 Zweiter Optimierungsversuch

Weil der erste Optimierungsversuch zu keinerlei Verbesserung geführt hat, wird erneut versucht die Performance der eigenen Implementierung zu verbessern. Diesmal werden die Erkenntnisse, die bei der Analyse der Schwachstellen erlangt werden konnten, eingesetzt, um das Problem zu lösen.

Wie die Analyse gezeigt hat, kann die Ausführung einer Folgetätigkeit solange verzögert werden, bis alle Voraussetzungen erfüllt sind. Inspiriert von der nativen Implementierung sollen Callbacks eingesetzt werden, um dies zu bewerkstelligen.

Dazu sind einige Anpassungen der aktuellen Implementierung erforderlich. Dies umfasst die Fertigstellung eines Futures, die Hinzunahme einer Callbackverwaltung sowie die Ausführung der Callbacks selbst.

Zuerst werden die Änderungen an der Klasse Future und dem dazugehörige Begleitobjekt betrachtet. Wie dem Codeschnipsel 83 entnommen werden kann, hat sich die Signatur des Futures nicht verändert. Die einzige Änderung der Klasse besteht darin, dass das Attribut callbacks hinzugefügt wird. Dieses Attribut ist eine ConcurrentLinkedQueue und gestattet das Speichern einer beliebigen Anzahl von Runnables. Es ist anzumerken,

```
final case class Future[A]() {  
  val callbacks = new ConcurrentLinkedQueue[Runnable]()  
  val result: Ref[Result[A]] = Ref(Result.empty)  
}  
  
object Future {  
  def apply[A](func: => A): Future[A] = {  
    val future = new Future[A]  
    execute(createRunnable(complete(future, func)))  
    future  
  }  
}
```

Abbildung 83: Die Klasse Future und dessen Begleitobjekt nach der Optimierung

dass ein Java Klasse verwendet werden muss, da die einzig verfügbare Alternative Scalas `SynchronizedQueue` ist. Diese ist jedoch deprecated und sollte daher nicht eingesetzt werden. Weshalb die Datenstruktur synchronisiert sein muss, wird im späteren Verlauf dieses Abschnitts deutlich.

Die `apply`-Funktion des Begleitobjekts muss ebenfalls modifiziert werden. Während die Instanziierung und die Rückgabe des Future gleich bleiben, muss die Ausführung der Funktion angepasst werden. Durch Kombination von `execute` und `createRunnable` wird die Berechnung in den Hintergrund geschoben. Wie sich erahnen lässt, sind diese Funktionen das Produkt der Zerlegung der ehemals verwendeten `forkIO`-Funktion.

Zur Berechnung des Ergebnisses werden `execute`, `createRunnable`, `complete` sowie die anonyme Funktion *func* komponiert. Da bereits bekannt ist, welchen Zweck die ersten beiden Funktionen erfüllen, bleibt nur noch zu erläutern, wann das Lambda evaluiert und wozu `complete` benötigt wird.

Die Evaluierung der anonymen Funktion *func* erfolgt vor dem Aufruf der `complete` Funktion. Erst wenn das Resultat feststeht wird das Ergebnis als Funktionsparameter übergeben.

Die Implementierung von `complete` ist in Abbildung 84 visualisiert. Diese private Hilfsfunktion wird für das Schreiben des Resultats eingesetzt. Wie zu erkennen ist, besitzt diese Funktion zwei Parameter. Dies sind zum einen das Future, welches abgeschlossen werden soll und zum anderen das Resultat des Lambdas, welches bereits vor dem Aufruf der Funktion evaluiert wird.

Die Funktion selbst ist recht simpel, da lediglich zwei Aktionen durchgeführt werden müssen. In der ersten Codezeile wird das Ergebnis des Futures geschrieben.

```
private def complete[A](future: Future[A], value: A) = {  
  atomic { implicit tnx => future.result() = Result(value) }  
  afterCompletion(future)  
}
```

Abbildung 84: Die Funktion `complete` wird angewendet, um ein Future abzuschließen

Die zweite Zeile zeigt den Aufruf von `afterCompletion`. Diese Funktion hat die Aufgabe alle gespeicherten Callbacks auszuführen (Snippet 85). Dazu wird eine Schleife durchlaufen, solange die Queue nicht leer ist. Innerhalb der Schleife wird das jeweils erste Runnable entnommen und ausgeführt. Vor der Ausführung muss dabei geprüft werden, ob das Runnable ungleich null ist, da es vorkommen kann, dass diese Funktion von mehreren Threads aufgerufen wird. Dies kann zu einer Konstellation führen, bei der mehrere

```
private def afterCompletion[A](future: Future[A]): Unit = {
  while (!future.callbacks.isEmpty()) {
    val task = future.callbacks.poll()
    if (task != null) execute(task)
  }
}
```

Abbildung 85: Funktion, welche für die Ausführung der Callbacks verantwortlich ist

Threads eine nicht leere Queue vorfinden und versuchen einen Task zu entnehmen. Die Queue ist hierbei der Synchronisationspunkt und sorgt dafür, dass Tasks nur einmal entnommen werden können. Dies führt wiederum dazu, dass null zurückgegeben wird, nachdem der letzte Task entnommen worden ist. Eine Alternative hierzu ist die Synchronisation der Funktion selbst. Hiergegen spricht jedoch, dass ein null check sehr günstig ist und eine Synchronisation dafür sorgt, dass der aufrufende Thread geblockt werden kann.

Mit `addOrExecuteCallback` können Callbacks an das entsprechende Future geheftet werden (Abbildung 86). Hierbei muss zwischen zwei Fällen unterschieden werden. Ist die Kalkulation des Futures zum Zeitpunkt des Hinzufügens nicht abgeschlossen, wird keine weitere Aktion durchgeführt. Der Grund hierfür ist, dass das Future `afterCompletion` aufruft, sobald die Kalkulation abgeschlossen ist.

Anders verhält es sich, wenn das Resultat bereits vorliegt. Da das Future somit nicht mehr berechnet wird, ist es nicht mehr imstande die Callbacks zu starten. Deshalb muss der Aufrufer die Ausführung der Callbacks selbst triggern. Zudem muss beachtet werden, dass der Aufruf von `afterCompletion` innerhalb eines `atomic` Blocks stattfindet. Wird das Future abgeschlossen während der Zustand dessen abgefragt wird, nimmt das

```
private def addOrExecuteCallback[A](future: Future[A], runnable:
  Runnable) = {
  future.callbacks.add(runnable)
  atomic { implicit txn =>
    future.result() match {
      case Empty => // do nothing here
      case _ => afterCompletion(future)
    }
  }
}
```

Abbildung 86: Mit `addOrExecuteCallback` können Callbacks hinzugefügt oder ausgeführt werden

STM davon Kenntnis und führt die Zustandsprüfung erneut aus. Dies führt wiederum dazu, dass sowohl das Future selbst als auch der aufrufende Thread `afterCompletion` aufrufen.

Der mögliche `retry` ist ebenfalls der Grund, weshalb das `Runnable` vor dem `atomic` Block hinzugefügt wird. Würde der Aufruf innerhalb des Blocks geschehen, müsste geprüft werden, ob das `Runnable` bereits in der Queue ist. Erst danach könnte dieses hinzugefügt werden.

Zum Schluss wird anhand der `followedBy`-Funktion illustriert, wie die Funktionen nach der Optimierung strukturiert sind (Abbildung 87). Innerhalb jeder Funktion wird ein neues Future-Objekt erzeugt. Hierzu wird jedoch nicht die `apply`-Funktion des Begleitobjekts, sondern der `new` Operator verwendet. Hierdurch entfällt das Starten eines neuen Threads. Anschließend wird das Callback erzeugt. Dazu wird die Funktion `createRunnable` benutzt. Der Body des neuen `Runnable`s beherbergt die eigentliche, auszuführende Funktionalität. In diesem Beispiel wird das Resultat des übergebenen Futures gelesen. Anschließend wird das Pattern Matching eingesetzt, um zu bestimmen wie weiter verfahren werden soll. In beiden Fällen wird `complete` eingesetzt, um das neu erzeugte Future abzuschließen. Besitzt das übergebene Future ein valides Resultat, wird die Funktion ausgeführt und das Ergebnis verwendet, um das neue Future abzuschließen. Andernfalls wird `null` benutzt. Der Verwendung von `complete` stellt sicher, dass alle später hinzugefügten Callbacks ausgeführt werden.

```
def followedBy[A, B](future: Future[A], function: (A) => B):  
  Future[B] = {  
    val result = new Future[B]  
    val callback = createRunnable {  
      get(future) match {  
        case Some(x) => complete(result, function(x))  
        case None => complete(result, null.asInstanceOf[B])  
      }  
    }  
    addOrExecuteCallback(future, callback)  
    result  
  }
```

Abbildung 87: Optimierte Version der `followedBy`-Funktion

Vor der Rückgabe des neuen Futures wird `addOrExecuteCallback` aufgerufen. Es gilt zu beachten, dass das Callback an das als Parameter übergebene Future angehängt werden muss. Sollte anstelle dessen das neue Future `result` verwendet werden, so wird der Callback niemals ausgeführt und das neue Future niemals abgeschlossen.

Nach der Optimierung der Futures, wird das unnötige Starten und anschließende Pausieren von Threads vermieden. Die Ausführung der Callbacks obliegt dabei dem Thread der das Ergebnis des Futures kalkuliert oder dem Thread der das Callback hinzufügen. Das eben beschriebene Verhalten ist in Abbildung 88 visualisiert.

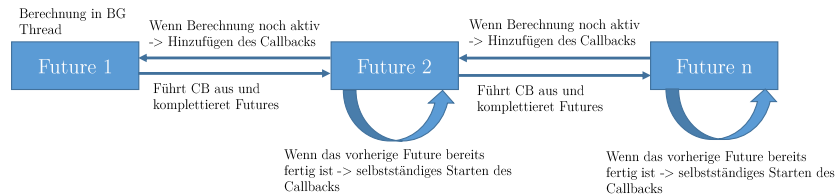


Abbildung 88: Verhalten der Implementierung nach der Optimierung

5.2.3 Validierung der Optimierung

Nach der Optimierung der Futures, werden die Benchmarks erneut ausgeführt. Es soll ermittelt werden, welche Auswirkungen die zweite Optimierung hat. Da die eigene Implementierung lediglich beim Einsatz von mehreren Kombinatoren schlechter abgeschnitten hat, werden nur diese Tests erneut ausgeführt.

Als Erstes wird die Funktion `andThen` erneut betrachtet. Wie das Diagramm 89 illustriert, ist die optimierte Implementierung etwas schneller als zuvor, kommt jedoch nicht an die Performance der nativen Implementierung heran. Die optimierte Version ist etwa zwei Millisekunden langsamer als das native Future. Ausschlaggebend für die etwas schlechtere Performance ist hierbei die Tatsache, dass zusätzlicher Overhead durch den Einsatz von STM entsteht. Dieser lässt sich jedoch nicht weiter reduzieren, da hierauf kein Einfluss genommen werden kann.

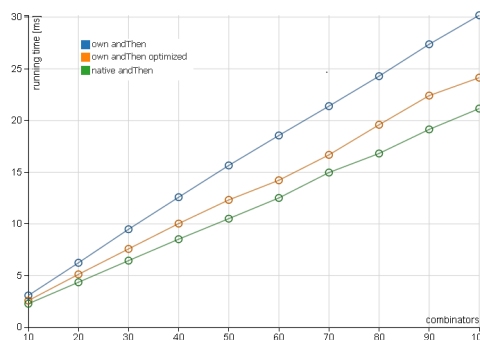


Abbildung 89: Benchmark-Ergebnis der `andThen`-Funktion nach der Optimierung

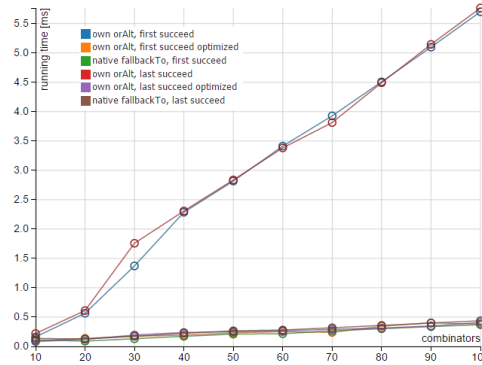


Abbildung 90: Benchmark-Ergebnis von orAlt/fallbackTo nach der Optimierung

Als Nächstes werden die Tests wiederholt, bei denen orAlt mit fallbackTo verglichen wird. Das Ergebnis (Grafik 90) verdeutlicht, dass die schlechte Skalierung der vorherigen Version vorwiegend durch die hohe Anzahl gestarteter, aber untätiger Threads verursacht wird. Nach der Optimierung kann eine deutliche Steigerung der Performance der eigenen Implementierung festgestellt werden.

Die Analyse der Laufzeitumgebung zeigt, dass der Threadpool auch bei der optimierten Version mehrere Threads erzeugt. Da die Tasks jedoch sofort ausgeführt und nicht noch einmal pausiert werden müssen, hat dies keinerlei negative Auswirkung auf die Ausführungsgeschwindigkeit.

Die Tests der first-Funktionen werden ebenfalls erneut betrachtet. Wie die Testergebnisse zeigen (Grafik 91), schwanken die Ergebnisse zu Beginn. Mit steigender Anzahl von Futures stabilisieren sich die Werte jedoch und liefern ab 30 Futures nahezu konstante Resultate.

Anders als beim ersten Benchmark bricht die Performance der eigenen Implementierung nicht ein, sobald mehr als 30 Futures eingesetzt werden. Durch das optimierte Hand-

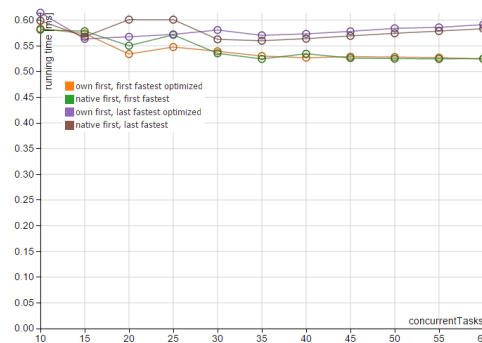


Abbildung 91: Benchmark-Ergebnis der first-Funktion nach der Optimierung

ling der Aufgaben triggert das Future die Erfüllung des Promises selbst. Hierdurch wird erreicht, dass die Anzahl aktiver Threads niemals die Anzahl der verfügbaren Kerne übersteigt.

Neben der Performance soll zudem analysiert werden, ob die Optimierung zu einer erhöhten Präzision führt. Wie dem Resultat dieses Tests (Grafik 92) entnommen werden kann, ist die Genauigkeit beider Varianten in etwa gleich, wobei die native Implementierung immer noch leichte Vorteile besitzt. Bei mehrfacher Ausführung der Tests, zeigt sich zudem, dass die Ergebnisse leicht schwanken. Ein Grund für dieses Abschneiden ist, dass die Berechnung aller Futures beinahe gleichzeitig abgeschlossen ist. Der Abstand liegt dabei zwischen 10 und 60 Mikrosekunden. Kommt es hierbei zu einer kurzen Verzögerung beim Starten des Folgetasks, führt dies dazu, dass ein anderes Future das Promise erfüllt.

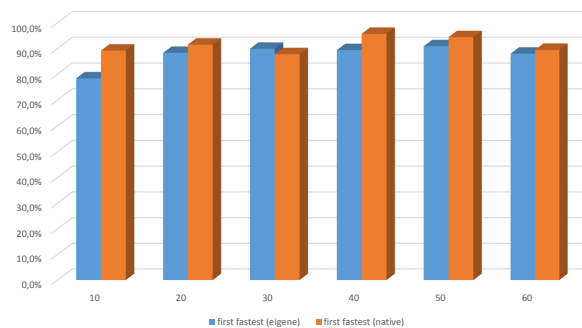


Abbildung 92: Genauigkeit der first-Funktion nach der Optimierung

Aufgrund der neu hinzugekommenen Funktionalität, soll abschließend analysiert werden, wie die Implementierungen sich verhalten, wenn einem Future eine Vielzahl von Callbacks zugewiesen werden.

Hierbei werden erneut zwei Anwendungsfälle betrachtet. Beim ersten Anwendungsfall wird ein Future erzeugt, welches nahezu sofort abgeschlossen wird. Nach der Initialisierung des Futures erfolgt die Zuweisung von 10 bis 100 Callbacks. Hierzu wird eine Schleife verwendet. Um die vorzeitige Beendigung des Durchlaufs zu verhindern, wird zudem eine Barriere benötigt. Das hierfür verwendete `CountDownLatch` wird zu Beginn jedes Durchlaufs initialisiert. Anschließend wird es von jedem Callback dekrementiert, bis der interne Zähler auf 0 fällt und der aktuelle Durchlauf beendet wird.

Der zweite Anwendungsfall besitzt die gleiche Struktur wie der Erste. Der einzige Unterschied hierbei ist, dass die Fertigstellung des Futures um eine Millisekunde verzögert wird. Hierdurch können alle Callbacks hinzugefügt werden, bevor die Kalkulation beendet ist. Somit wird erreicht, dass die Callbacks vom Future selbst gestartet werden.

```
val callbacks = Gen.range("callbacks")(10, 100, 10)

measure method "optimized own no wait" in {
  using(callbacks) in { max =>
    val latch = new CountdownLatch(max)
    val f = Future {
      42
    }
    for (i <- 1 to max) {
      onSuccess(f, (input: Int) => latch.countDown())
    }
    latch.await()
  }
}
```

Abbildung 93: Benchmark der onSuccess-Funktion

Das Ergebnis der Benchmarks kann Abbildung 94 entnommen werden. Es sei angemerkt, dass die Grafik Ergebnisse von drei unterschiedlichen und voneinander unabhängigen Messungen zeigt.

Betrachtet man zuerst den ersten Anwendungsfall (untere Ergebnisse), fällt auf, dass die native Implementierung bei bis zu 50 Callbacks nahezu sofort fertig ist. Ab 60 Callbacks erhöht sich die benötigte Zeit, um alle Callbacks auszuführen. Zudem treten ab diesem Zeitpunkt Schwankungen auf. Dies ist daran zu erkennen, dass die drei grünen Geraden teils deutlich auseinander liegen.

Die beiden Tests, bei denen die eigene Implementierung verwendet wird, sind beinahe gleich schnell. Hierbei kommen die Vorteile, die durch die Optimierung entstanden sind nicht zur Geltung. Grund hierfür ist, dass das Future bereits nach kurzer Zeit fertig ist und der Hauptthread daher alle Callbacks starten muss.

Ein anderes Bild ergibt sich, wenn die Callbacks hinzugefügt werden, bevor das Future abgeschlossen ist. Auch hierbei ist die native Implementierung am performantesten. Zudem sind dessen Ausführungszeiten konstant. Wie der Grafik ebenfalls entnommen werden kann, bleibt das Ergebnis auch bei mehrfacher Ausführung des Tests gleich.

Die optimierte Version der eigenen Implementierung besitzt dieselben Eigenschaften wie die native Variante, ist jedoch etwa 0,1 Millisekunden langsamer. Erwartungsgemäß wirkt sich das Starten aller Folgetasks, durch einen Thread, positiv auf die Performance aus. Die etwas schlechtere Performance gegenüber dem nativen Future, ist darauf zurückzuführen, dass STM etwas mehr Overhead erzeugt und die Tasks bei der nativen Umsetzung zusammengefasst und von einem einzigen Thread ausgeführt werden.

Die schlechteste Leistung wird von der nicht optimierten Variante erbracht. Diese ist an-

fangs linear und wird gegen Ende hin konstanter. Dieses Verhalten ist den in Abschnitt 5.1 vorgestellten Schwachstellen zuzuschreiben.

Dieser Test zeigt, dass die optimierte Version auch bei diesem Anwendungsfall mit der nativen Implementierung mithalten kann. Zwar ist die Performance in beiden Fällen etwas schlechter, weist jedoch ähnlich Charakteristiken auf. Die Unterschiede von unter einer Millisekunde, sind zudem sehr gering und kaum von Relevanz.

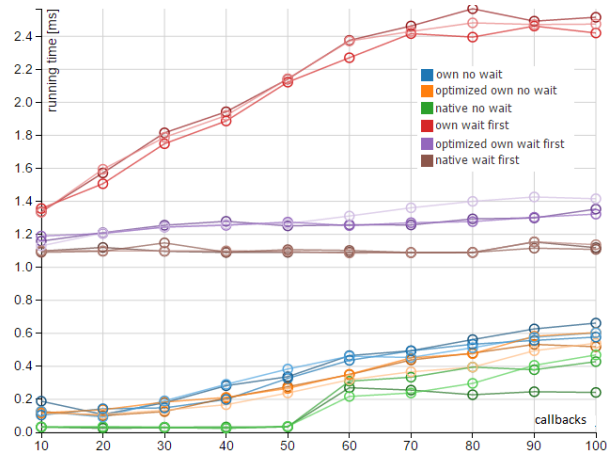


Abbildung 94: Benchmark-Ergebnis der Callback-Funktion

6 Fazit

Im letzten Kapitel dieser Arbeit wird rekapituliert, inwieweit die in der Einleitung definierten Ziele erreicht worden sind. Hierzu wird zuerst erläutert, welche Tätigkeiten durchgeführt worden sind, um zur vorliegenden Lösung zu gelangen (Abschnitt 6.1). Ebenfalls in diesen Abschnitt enthalten, ist die Bewertung des Ergebnisses. Abschließend erfolgt ein Ausblick (Abschnitt 6.2).

6.1 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung funktionaler Futures und Promises in Scala. Für die Verwaltung des Resultats eines Futures sollte STM eingesetzt werden.

Grundlegend hierfür war die Vorstellung von STM, den dazugehörigen Features sowie der verfügbaren Implementierungen in Scala.

Nach der Vorstellung von STM wurde analysiert, wie oft dieser Mechanismus im Scalakontext eingesetzt wird, um Concurrency Probleme zu lösen. Zur Durchführung der Analyse ist ein Tool [19] entwickelt worden. Mit Hilfe dieses Tools kann auf die Github API zugegriffen werden, um Metainformationen über die dort publizierten Repositories zu erlangen. Diese Daten werden anschließend aggregiert und ausgewertet. Die Analyse selbst hat ergeben, dass STM nur in einem Bruchteil aller analysierten Projekte verwendet wird. Zudem konnte festgestellt werden, dass die am häufigsten eingesetzten Mechanismen Aktoren und Futures sind.

Anschließend wird die Implementierung der Futures und Promises präsentiert. Die Umsetzung ist auf Github veröffentlicht [21]. Beide Abstraktionen verfügen über alle charakteristischen Funktionen. Dies umfasst den Zugriff auf das Ergebnis, das Hinzufügen von Callbacks, das Verschachteln von Futures sowie die Definition von Folgeoperationen, die nach Abschluss der Kalkulation ausgeführt werden. Die Verwendung von STM bringt mehrere Vorteile mit sich. Zum einen ist die Realisierung der blockenden get-Funktion trivial. Liegt zum Zeitpunkt des Aufrufs kein Ergebnis vor, wird retry verwendet, um den aufrufenden Thread zu pausieren. Dieser wird automatisch geweckt, sobald das Ergebnis vorliegt. Zum anderen ist die Implementierung komplexer Funktionen möglich. Ein Beispiel hierfür ist die Funktion multiTrySuccess. Diese kann mehrere Promises gleichzeitig erfüllen. Durch die Atomarität von STM wird gewährleistet, dass die Funktion korrekt

ausgeführt wird. Die Umsetzung derselben Funktionalität ohne STM ist deutlich komplexer. Verwendet man beispielsweise Sperren, so wird eine Ordnungsfunktion benötigt, die festlegt in welcher Reihenfolge die Sperren erlangt und wieder freigegeben werden müssen. Dies wiederum stellt eine nicht triviale Funktion dar.

Beim anschließenden Vergleich mit der nativen Implementierung hat sich gezeigt, dass die eigene Umsetzung etwa die Hälfte an Codezeilen benötigt, um die gleiche Funktionalität bereitzustellen. Zudem wurden Benchmarks erstellt, um die Performance beider Implementierungen miteinander zu vergleichen. Beide Alternativen liefern beim sequenziellen und nebenläufigen Zugriff auf das Resultat eines Futures ähnliche Ergebnis, wobei die native Implementierung etwas schneller ist. Hierfür verantwortlich ist der Overhead, der durch den Einsatz von STM entsteht.

Bei weiteren Tests wurde untersucht, wie die Implementierungen sich verhalten, wenn eine Vielzahl von Kombinatoren verschachtelt werden. Hierbei schneidet die eigene Implementierung deutlich schlechter ab als das native Gegenstück. Analysen der Laufzeitumgebung haben ergeben, dass die Ursachen hierfür auf die transitive Abhängigkeit der Futures sowie den Overhead beim Threading zurückzuführen sind. Die eigene Implementierung erzeugt dabei sehr viele Threads, welche anschließend sofort pausiert und zu einem späteren Zeitpunkt geweckt werden.

Die bei der Analyse erworbenen Kenntnisse konnten genutzt werden, um die Implementierung zu optimieren. Dabei wird die Ausführung aller Folgeaktivitäten verzögert, wodurch das Starten und sofortige Pausieren von Threads verhindert wird. Die erneute Ausführung der Benchmarks hat belegt, dass die zuvor bestehenden Leistungsdefizite behoben werden konnten und die optimierte Version nahezu identische Ergebnisse, wie die native Variante, hervorbringt.

Insgesamt ist das Resultat der Arbeit positiv zu bewerten. Die optimierte Implementierung weißt ähnlich gute Performance und Skalierbarkeit wie das native Gegenstück auf, besitzt jedoch darüber hinaus die Fähigkeit eine Menge von Futures und Promises gleichzeitig zu bearbeiten. Zudem sind beide Abstraktionen einfacher erweiterbar als die nativen Varianten. Anstelle der Erweiterung des entsprechenden Traits und aller abhängigen Klassen, müssen hierzu lediglich neue Funktionen definiert werden.

6.2 Ausblick

Die in dieser Arbeit vorgestellte Implementierung der Futures und Promises kann als abgeschlossen betrachtet werden. Dennoch verbleiben verschiedene Tätigkeiten, bei denen eine tiefer gehende wissenschaftliche Analyse durchgeführt werden könnte.

Ähnlich wie bei der nativen Implementierung, könnte ein Mechanismus implementiert werden, der alle Callbacks zusammenfasst, sodass diese nur noch von einem Thread ausgeführt werden. Dies würde sich positiv auf die Auslastung der CPU Ressourcen auswirken und dazu führen, dass die Ausführung auch auf Systemen mit einer geringen Anzahl von Kernen effizienter wäre.

Zudem könnten die anderen, in dieser Arbeit vorgestellten STMs eingesetzt werden, um das Resultat des Futures zu verwalten. Ein anschließender Vergleich könnte Aufschluss darüber geben, ob dies Auswirkungen auf die Performance der Implementierung hat.

Ebenfalls vorstellbar ist ein Vergleich mit anderen Programmiersprachen, die über eine adäquate STM Implementierung verfügen. Hierfür würde sich beispielsweise Haskell anbieten. Diese Sprache besitzt ein genauso mächtiges STM und unterstützt das funktionale Programmierparadigma. Zudem wäre die Portierung der Implementierung relativ simpel.

Literatur

- [1] William Mangro, Paul Petersen und Shah Sanjiv. »Hyper-Threading Technology: Impact on Compute-Intensive Workloads«. In: (2002). URL: http://meseec.ce.rit.edu/eec722-fall2006/papers/smt/9/vol6iss1_art06.pdf (besucht am 24.02.2016).
- [2] Urs Gleim und Tobias Schüle. *Multicore-Software: Grundlagen, Architektur und Implementierung in C/C++, Java und C#*. 1. Aufl. Heidelberg, Neckar: dpunkt, 2012. ISBN: 3898647587.
- [3] D. B. Lomet. »Process Structuring, Synchronization, and Recovery Using Atomic Actions«. In: *SIGOPS Oper. Syst. Rev.* 11.2 (März 1977), S. 128–137. ISSN: 0163-5980. DOI: [10.1145/390018.808319](https://doi.org/10.1145/390018.808319). URL: <http://doi.acm.org/10.1145/390018.808319>.
- [4] Tim Harris, James R. Larus und Ravi Rajwar. *Transactional Memory*. 2nd. ed. Bd. 11. Synthesis lectures on computer architecture. [San Rafael, Calif.]: Morgan & Claypool, 2010. ISBN: 1608452352.
- [5] Nathan Bronson. *ScalaSTM — Library-Based Software Transactional Memory for Scala*. 2010. URL: <https://nbronson.github.io/scala-stm/> (besucht am 28.08.2015).
- [6] Daniel Goodman u. a. »MUTS: Native Scala Constructs for Software Transactional Memory«. In: *Scala Days Workshop, Stanford*. 2011. URL: <http://apt.cs.manchester.ac.uk/people/dgoodman/papers/MUTS-final.pdf> (besucht am 01.09.2015).
- [7] Guy Korland, Nir Shavit und Pascal Felber. »Noninvasive concurrency with Java STM«. In: *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*. 2010.
- [8] Tim Harris u. a. »Composable Memory Transactions«. In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '05. Chicago, IL, USA: ACM, 2005, S. 48–60. ISBN: 1-59593-080-9. DOI: [10.1145/1065944.1065952](https://doi.org/10.1145/1065944.1065952). URL: <http://doi.acm.org/10.1145/1065944.1065952>.

- [9] Simon Peyton Jones. »Beautiful concurrency«. In: (2007). URL: <http://research.microsoft.com/pubs/74063/beautiful.pdf> (besucht am 15. 11. 2015).
- [10] Aleksandar Dragojević, Rachid Guerraoui und Michal Kapalka. »Stretching Transactional Memory«. In: *SIGPLAN Not.* 44.6 (Juni 2009), S. 155–165. ISSN: 0362-1340. DOI: [10.1145/1543135.1542494](https://doi.org/10.1145/1543135.1542494). URL: <http://doi.acm.org/10.1145/1543135.1542494>.
- [11] Simon Ochsenreither. *Tools and Libraries - Scala Wiki - Scala Wiki*. 2015-09-27. URL: <https://wiki.scala-lang.org/display/SW/Tools+and+Libraries#ToolsandLibraries-Benchmarking> (besucht am 27. 10. 2015).
- [12] Flavio W. Brasil. *RadonSTM: fwbrasil/radon-stm*. 2014-09-20. URL: <https://github.com/fwbrasil/radon-stm> (besucht am 29. 10. 2015).
- [13] Daniel Spiewak. *Software Transactional Memory in Scala - Code Commit*. 2008-10-06. URL: <http://www.codecommit.com/blog/scala/software-transactional-memory-in-scala> (besucht am 29. 10. 2015).
- [14] Daniel Spiewak. *Improving the STM: Multi-Version Concurrency Control - Code Commit*. 2008-11-10. URL: <http://www.codecommit.com/blog/scala/improving-the-stm-multi-version-concurrency-control> (besucht am 29. 10. 2015).
- [15] TutorialsPoint.com. *EJB Transactions*. n.A. URL: http://www.tutorialspoint.com/ejb/ejb_transactions.htm (besucht am 24. 02. 2016).
- [16] Rich Hickey. *Clojure: Concurrent Programming*. n.A. URL: http://clojure.org/concurrent_programming (besucht am 11. 09. 2015).
- [17] Phil Bagwell. *STM Comes to Scala | The Scala Programming Language*. 2012-11-06. URL: <http://www.scala-lang.org/old/node/8359> (besucht am 21. 10. 2019).
- [18] Daniel Goodman u. a. »Software transactional memories for Scala«. In: *Journal of Parallel and Distributed Computing* 73.2 (2013), S. 150–163. URL: <http://apt.cs.manchester.ac.uk/people/dgoodman/papers/jpdc2012.pdf>.
- [19] Andreas Schwarz. *GitHub Analyzer Quellcode*. 2016-02-09. URL: <https://github.com/anschwar/GithubAnalyzer> (besucht am 09. 02. 2016).

-
- [20] Kohsuke Kawaguchi. *GitHub API for Java*. 2015-08-15. URL: <http://github-api.kohsuke.org/> (besucht am 27.11.2015).
 - [21] Andreas Schwarz. *A functional implementation of futures and promises which uses STM to manage the underlying data*. 2016-02-09. URL: <https://github.com/anschwar/FutureSTM> (besucht am 09.02.2016).
 - [22] n.A. *Introduction / ScalaMeter*. 2015-10-10. URL: <http://scalameter.github.io/home/gettingstarted/0.7/index.html> (besucht am 11.01.2016).
 - [23] Daniel Shaya. *Rational Java: Let's pause for a Microsecond*. 2016-01-29. URL: <http://www.rationaljava.com/2015/10/measuring-microsecond-in-java.html> (besucht am 01.02.2016).