Karlsruher Institut für Technologie

Fakultät für Informatik

Werkzeugunterstützte Ortung von Parallelisierungspotenzial

Diplomarbeit
von

Alexander Bieleš

Verantwortlicher Betreuer:	Prof. Dr. Walter F. Tichy
Betreuender Mitarbeiter:	Dipl.-Inform. Korbinian Molitorisz
	Dipl.-Inform. Thomas Karcher

Bearbeitungszeit: 01. Januar 2013 – 31. März 2013

Karlsruher Institut für Technologie · Institut für Programmstrukturen und Datenorganisation
Geb. 50.34 · Am Fasanengarten 5 · 76131 Karlsruhe

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 28. März 2013

Alexander Bieleš

Danksagung

Ein herzliches Dankeschön geht an alle, die mich bei der Erstellung meiner Diplomarbeit unterstützt haben. Insbesondere danke ich meinen Betreuern Thomas Karcher und Korbinian Molitorisz für die anregenden Diskussionen, die fachliche Unterstützung und die „Unit Tests“, die die Qualität der Arbeit deutlich verbessert haben. Zudem möchte ich Herrn Prof. Tichy und Dr. Padberg für die thematische Orientierungshilfe danken, die zur Schärfung des Themas beigetragen hat. Abschließend geht auch ein großer Dank an meine Freunde und Kommilitonen für den gewinnbringenden Meinungsaustausch, die technologische Unterstützung und die ausgefeilten Korrekturhilfen.

Inhaltsverzeichnis

1 Einleitung

Performanzverbesserungen können heutzutage nicht mehr durch Taktsteigerung von Prozessoren erreicht werden. Stattdessen werden Mehrkernprozessoren angeboten, welche mehr Rechenleistung durch mehr Kerne liefern. Sequenzielle Programme sind inhärent nicht in der Lage, Hardwareparallelität zu nutzen, da sie für Einkernprozessoren programmiert wurden. Die Erforschung automatischer Parallelisierungstechniken verspricht Abhilfe. Die Ortung von Parallelisierungsmöglichkeiten wird in der Informatik auf verschiedenen Abstraktionsebenen untersucht. Zumeist werden dabei die Abhängigkeiten im Kontrollfluss analysiert, um Anweisungen zu identifizieren, die parallel ausgeführt werden können. Abstraktere Ebenen wie Klassendiagramme, aber auch feingranularere Laufzeitaspekte wie die Anordnung der Daten im Speicher sind Untersuchungsgegenstand bei der Suche nach Parallelisierungspotenzial.

In objektorientierten Programmen sind Datenstrukturen wiederkehrende Bausteine, deren Funktionalität hinter einer Schnittstelle gekapselt ist. Algorithmen verwenden diese Container durch Zugriffe und hinterlassen dabei beschreibbare Verhaltensmuster. Es stellt sich daher die Frage, inwiefern aus Laufzeit-Zugriffsprofilen von Datenstrukturen Parallelisierungspotenzial gewonnen werden kann.

Ziel dieser Diplomarbeit ist die automatische Lokalisierung von Parallelisierungspotenzial anhand des zur Laufzeit beobachteten Datenstrukturzugriffsverhaltens in sequenziellen objektorientierten Programmen. Es wird gezeigt, dass aus der Analyse von Zugriffshistorien sogenannte Fingerzeige abgeleitet werden können, deren Umsetzung zu einer Laufzeitverbesserung führen. Beschleunigungen bis zu einem Faktor von 3 werden in den untersuchten Testprogrammen erreicht.

Das entstandene Werkzeug erzeugt selbst keinen parallelisierten Code. Stattdessen sind die Ausgaben Quelltextstellen mit begründetem Parallelisierungspotenzial. Diese werden anschließend vom Nutzer bewertet und umgesetzt. Schlussendlich bedeutet dies, dass ein Programmierer durch diese Hinweise schneller an die parallelisierungsrelevanten Stellen herangeführt wird als durch manuelle Durchsicht des gesamten Programms.

2 Grundlagen

Dieses Kapitel geht auf die Grundlagen der vorliegenden Arbeit ein. Zunächst werden dazu Begriffe erläutert, die für den Rahmen der Arbeit definiert wurden. Danach wird grundlegendes Wissen über die verwendeten Technologien eingeführt. Darunter zählt das Rahmenwerk Roslyn, welches in dieser Diplomarbeit als Werkzeug verwendet wurde, und der Aufbau von C#-Projekten des *Visual Studio*.

2.1 Begriffe

Die vorliegende Arbeit verwendet teils selbst definierte Fachbegriffe. Diese werden im Folgenden eingeführt.

Zugriffsart

Zugriffe sind die Aufrufe von Schnittstellenmethoden einer Datenstruktur. Dabei gibt es Methoden, deren Zweck sehr ähnlich ist. Die Zugriffsart beschreibt die Natur eines Zugriffs auf einer hohen Abstraktionsebene. Lesen, Schreiben, Einfügen oder Löschen sind beispielsweise primitive Zugriffsarten. Sortieren oder Finden sind komplexere Zugriffsarten.

Zugriffsereignis

Ein Zugriffsereignis ist eine Notiz über das Auftreten eines Zugriffes (Aufruf einer Schnittstellenmethode) einer Datenstruktur. Praktisch gesehen ist ein Zugriffsereignis ein Zusammenschluss von Informationen zu einem solchen Zugriff, wie zum Beispiel die Zugriffsart, die Zeit oder die Position an der der Zugriff stattfand.

Instanziierungsort

Der Instanziierungsort ist die Stelle im Code, an der eine Datenstruktur instanziiert wurde. Klar zu unterscheiden ist dies von einer Variablendefinition, da einer Variablen nacheinander zwei Instanzen zugeordnet werden können, wobei die zweite Zuweisung die alte Instanz ersetzt. Die beiden Quelltextstellen, die die beiden Instanzen erzeugen und der Variablen zuweisen, entsprechen zwei Instanziierungsorten.

Zugriffshistorie

Eine Zugriffshistorie ist eine zeitlich geordnete Menge von Zugriffsereignissen auf eine konkrete Datenstrukturinstanz und ist somit einem Instanziierungsort zugeordnet.

Phasencharakteristikum

Ein Phasencharakteristikum beschreibt ein Metakonzept für ein schablonenartiges Muster. Es beinhaltet Bedingungen, die in einer Teilmenge von Zugriffsereignissen gelten müssen, damit diese Menge von Ereignissen als zusammengehörig gelten. Die Zeit der Zugriffsereignisse und die Position der Zugriffe werden dabei berücksichtigt.

Zum Beispiel beschreibt das Phasencharakteristikum `Lineares-Lesen-Vorwärts` folgende Bedingungen: Zugriffe sind zeitlich benachbart, ihre Zugriffspositionen sind nach zeitlicher Sortierung aufsteigend und es handelt sich um Leseoperationen (Zugriffsart ist Lesen).

Phasen

Treffen die Bedingungen eines Phasencharakteristikums auf eine Menge von Zugriffsereignissen zu, dann stellt diese Menge eine Phase (bezüglich des Charakteristikums) dar. Eine Phase ist somit eine Ausprägung eines Phasencharakteristikums. Phasen sind maximal groß. Das bedeutet, dass es keine zwei zeitlich aneinandergrenzende Phasen gibt, deren Vereinigung ebenfalls das Phasencharakteristikum erfüllt.

Die Phasen eines Phasencharakteristikums sind unabhängig von den Phasen eines anderen Charakteristikums. Das bedeutet einerseits, dass ein Zugriffsereignis zu Phasen unterschiedlicher Phasencharakteristika zugeordnet sein kann. Andererseits kann ein Zugriffsereignis auch in gar keiner Phase enthalten sein. Bezüglich eines Charakteristikums wird ein Zugriffsereignis aber maximal einer Phase zugeordnet.

Zugriffsprotokoll

Ein Zugriffsprotokoll ist eine um Phaseninformationen angereicherte Zugriffshistorie. Es umfasst dementsprechend eine Historie und die Abbildung von Zugriffsereignissen zu Phasen bezüglich aller Phasencharakteristika.

Protokollsammlung

Eine Protokollsammlung ist der Zusammenschluss mehrerer Zugriffsprotokolle.

Fingerzeig

Ein Fingerzeig ist ein Hinweis zu einer Stelle im Quelltext, wo Optimierungsschritte durch den Nutzer/Programmierer zu einer Laufzeitverbesserung führen können. Sie sind begründete Anregungen, die dem Benutzer zeigen, wo im Code Parallelisierungspotenzial enthalten sein kann.

2.2 Technologien und Werkzeuge

2.2.1 Roslyn

Roslyn ist ein Forschungsprojekt von Microsoft, welches das Ziel verfolgt, die Blackbox des C#-Kompilierers für Programmierer zugänglich zu machen. Das Rahmenwerk „Roslyn June 2012 CTP“ stellt dafür eine API zur Verfügung. Die mit der API erzeugten Objektmodelle dienen dem Programmierer zur Erzeugung, Analyse und Refaktorisierung von Quelltext. Dabei können nicht nur statische und strukturelle Informationen wie abstrakte Syntaxbäume aus bestehendem Code gewonnen werden, sondern auch semantische Zusammenhänge wie Typinformationen von Variablen. Diese Arbeit bedient sich dieses Werkzeuges, um Quelltext den semantischen Kontexten entsprechend zu transformieren.

2.2.2 C#-Projekte

Projekte, die mit dem *Visual Studio* in C# programmiert sind, werden in einer sogenannten Arbeitsmappe (engl.: Solution) gespeichert. Darin befinden sich eine Beschreibungsdatei für die Referenzen der Teilprojekte und weitere Unterordner. Die Unterordner enthalten den Quelltext der Teilprojekte, aus denen die Arbeitsmappe zusammengesetzt ist. Jedes Teilprojekt enthält eine eigene Projektkonfigurationsdatei.

3 Verwandte Arbeiten

In diesem Kapitel werden verwandte Forschungsarbeiten vorgestellt, welche ebenfalls die Benutzung von Datenstrukturen im Programmlauf untersuchen oder Parallelisierung auf Ebene der Datenstrukturen anstreben. Die Publikation „Brainy: Effective Selection of Data Structures“ [JR+11], welche in Kapitel 3.4 beleuchtet wird, liegt der vorliegenden Diplomarbeit am nächsten und wird dementsprechend ausführlich betrachtet. Eine tabellarische Gegenüberstellung der relevanten Arbeiten fasst dieses Kapitel am Ende zusammen.

3.1 Parallelisierungsansätze

Parallelisierung ist auf unterschiedlichen Abstraktionsebenen und Methoden möglich. STAPL [TB+11] beispielsweise ist eine Bibliothek von parallel nutzbaren Datenstrukturen und parallelisierten Algorithmen. Bei der Verwendung solchen Bibliotheken muss der Programmierer nicht nur parallel programmieren, sondern auch Parallelisierungspotenzial selbstständig erkennen. PetaBricks [AC+09] ist eine Spracherweiterung und unterstützt den Programmierer ein Stück weiter, sodass dieser sich nicht mehr um Performanzabschätzungen verschiedener Parallelisierungsumsetzungen kümmern muss. Sprachkonstrukte ermöglichen es, verschiedene Algorithmenalternativen anzugeben, sodass die Algorithmenwahl zur Laufzeit von der Laufzeitumgebung getroffen werden kann. MapReduce [LH+11] stellt ein Rahmenwerk zur Verfügung, welches dem Programmierer hilft, schneller und fehlerfrei zu parallelisieren. Parallelisierungen muss er dabei immer noch selbstständig erkennen, dafür wird er bei der Implementierung unterstützt. In allen Fällen liegt es somit am Programmierer, Parallelisierungsmöglichkeiten zu identifizieren. Mit existierendem Quelltext können die genannten Ansätze nicht arbeiten. An dieser Stelle setzen Werkzeuge an, die Parallelisierungspotenzial erkennen. MAPS [CC+11] ist zum Beispiel ein System, welches erkennt, wo und wie parallelisiert werden soll. Die dabei eingesetzte Instrumentierung speichert Laufzeitdaten, wie Ausführungsanzahl von Codeblöcken oder Zeigerveränderungen, welche im Nachhinein automatisiert untersucht werden. Noch einen Schritt weiter gehen die Ansätze zur automatischen Parallelisierung. Die Studienarbeit [H11] ist dabei ein Beispiel, wie paralleler Code aus der statischen Analyse eines

sequenziellen Quelltextes abgeleitet werden kann. Die Arbeit [TF10] verwendet hingegen dynamische Laufzeitdaten in Form eines Abhängigkeitsgraphen, um automatisch Quelltext zu parallelisieren.

Die vorliegende Arbeit beschränkt sich auf die Extraktion von Parallelisierungspotenzial und möchte damit dem Programmierer helfen, der ein bestehendes sequenzielles Programm parallelisiert. Dabei werden auf Ebene der Datenstrukturinstanzen Beobachtungen dynamischen Laufzeitverhaltens ausgewertet und Parallelisierungshinweise auf algorithmischer Ebene ausgegeben. Parallelisieren muss der Entwickler zwar immer noch selbstständig, abgenommen wird ihm aber das aufwendige Analysieren seines Programms.

3.2 Zhang – Optimizing Data Layouts for Parallel Computation on Multicores

[ZD+11] versuchen das Datenlayout im Speicher zu optimieren, sodass der Cache effektiver genutzt wird – insbesondere in Bezug auf parallele Programme. Die dabei angewandte Technik wird Array-Restrukturierung genannt. Sie versuchen die im Speicher liegenden Arrayzellen in Bezug auf Cache-Hierarchien optimaler im Speicher zu verteilen. Dazu wird das Zugriffsverhalten der Threads auf Arrayindizes überwacht. Mit Hilfe von Hyperebenen findet eine räumliche oder zeitliche Aufteilung der Arrayelemente statt.

Der Unterschied zur vorliegenden Arbeit besteht darin, dass für Zhang nur die Bereiche der Datenstruktur, auf die ein Thread zugreift, von Interesse sind. Die vorliegende Arbeit untersucht zusätzlich auch den zeitlichen Verlauf der unterschiedlichen Operationen und nutzt die gesammelten Informationen für Verbesserungen am Quelltext.

3.3 Rane – Performance optimization of data structures using memory access characterization

In [RB11] legen die Autoren Historien von Cache-Zugriffszeiten an, um Probleme wie zum Beispiel Flaschenhälse zu erkennen. Dabei definieren sie verschiedene Metriken, die zur Laufzeit gesammelt werden, wie zum Beispiel Elementzugriffszeit, Cache-Hit-Rate oder Rate wiederholender Zugriffe. Die Instrumentierung zur Ermittlung dieser Zahlenmesswerte ermöglicht eine Zuordnung der Messungen zum Quelltext. Der Nutzer bekommt dadurch die Messungen den Datenstrukturen zugeordnet zu sehen. Weitere Abschätzungen, ob Parallelisierungspotenzial vorhanden ist, liegt in seiner Hand.

Der Unterschied zum Ansatz dieser Arbeit ist zum einen, dass die Autoren Aggregate oder Historien über Messwerten einsetzen. Hingegen werden hier Historien über Zugriffsereignisse von Datenstrukturinstanzen ausgewertet. Zum anderen werden in dieser Arbeit aus den Messdaten bereits automatisiert Handlungsempfehlungen abgeleitet, sodass dem Nutzer die Interpretation der Daten abgenommen wird.

3.4 Jung – Brainy: effective selection of data structures

Jung et al. veröffentlichten in [JR+11] ein System namens Brainy, welches Vorhersagen zum Einsatz optimaler Datenstrukturen machen soll. Dabei werden das Programmverhalten, die Eingabedaten und die Zielarchitektur in der Analyse mit einbezogen. Grundlage sind diverse Performanzmessungen der Datenstrukturen auf der Zielplattform mit unterschiedlichsten Eingabeparametern. Diese Messungen trainieren ein Maschinenlernverfahren, welches dann für das reale Zugriffsverhalten einer konkreten Datenstrukturinstanz eine angemessenere Alternativdatenstruktur schätzt.

Brainy stellt letztlich ein Modell dar, welches Performanzmessdaten eines Programms auf Datenstrukturimplementierungen abbildet. Diese Abbildung wird von einem Klassifikator erlernt, der aus sechs künstlichen neuronalen Netzwerken besteht – jedes für eine der ausgewählten Ausgangsdatenstrukturen der Standardbibliothek. Wenn eine bestimmte Datenstruktur eines Programms optimiert werden soll, wird das zum Datentyp korrespondierende neuronale Netz dazu befragt, welche alternative Datenstruktur am besten geeignet sei. Die Trainingsdaten sind dabei Tupel, bestehend aus den charakterisierenden Messdaten der Zugriffe zur Laufzeit und der dafür besten Datenstruktur als Klassifizierungsziel. Um dem Klassifizierer genügend solcher Trainingsdaten zur Verfügung zu stellen, wurde ein Programmgenerator entworfen. Dieser Generator erstellt Programme, die unterschiedliche Zugriffsverhalten auf eine (noch unbestimmte) Datenstruktur realisieren. Das Verhalten wird zufällig generiert und umfasst die folgenden Parameter:

- Anzahl Schnittstellenaufrufe
- Größe der Datenelemente
- Anzahl einzufügender Elemente
- Anzahl zu löschender Elemente
- Anzahl zu findender Elemente
- Anzahl zu traversierender Elemente

Abbildung 1 aus [JR+11] zeigt die tabellarische Übersicht der Arbeit über die Austauschbarkeit von Datenstrukturen. Beispielsweise kann eine `map` nur durch eine `avl_map` oder `hash_map` ersetzt werden, während `vector` von `list`, `deque` oder drei `set`-Derivaten ersetzbar ist.

DS	Alternate DS	Benefit	Limitation
vector	list	Fast insertion	None
	deque	Fast insertion	None
	set (map)	Fast search	Order-oblivious
	avl_set (avl_map)	Fast search	Order-oblivious
	hash_set (hash_map)	Fast insertion & search	Order-oblivious
list	vector	Fast iteration	None
	deque	Fast iteration	None
	set (map)	Fast search	Order-oblivious
	avl_set (avl_map)	Fast search	Order-oblivious
	hash_set (hash_map)	Fast search	Order-oblivious
set	avl_set	Fast search	None
	vector	Fast iteration	Order-oblivious
	list	Fast insertion & deletion	Order-oblivious
	hash_set	Fast insertion & search	Order-oblivious
map	avl_map	Fast search	None
	hash_map	Fast insertion & search	Order-oblivious

Abbildung 1: Austauschbarkeit von Datenstrukturen aus [JR+11]

In der ersten Phase der Trainingsdatenerstellung werden für alle Ausgangsdatenstrukturen viele dieser generierten Programme in Kombination mit allen Alternativdatenstrukturen vermessen. Die Datenstruktur mit der kürzesten Laufzeit für ein gegebenes Programm wird dabei protokolliert. Diese wird in der zweiten Phase instrumentiert und nochmals ausgeführt. Die Instrumentierungen sorgen dafür, dass jetzt die Performanzdaten gemessen werden. Damit können dann die anfangs beschriebenen Tupel für den Klassifizierer erzeugt werden. Jedes neuronale Netz erhält eine einstellbare Mindestanzahl an Tupeln zum Erlernen. Für jede Ausgangsdatenstruktur wurde zuvor ermittelt, welche Kennzahlen aus den Performanzdaten am relevantesten für die Klassifizierung sind. Die Trainingstupel werden daher für jedes neuronale Netz auf die fünf ausschlaggebendsten Eigenschaften reduziert. Der sogenannte *Back-Propagation*-Algorithmus wird zum Einlernen verwendet, wobei die Autoren nicht sagen, wie die neuronalen Netze im Detail aufgebaut sind.

Zur Validierung der Vorhersagekraft des klassifizierenden Kollektivs wurde jedes neuronale Netz mit 1000 generierten Programmen getestet. Die vorhergesagten Austauschvorschläge wurden jeweils mit der Datenstruktur verglichen, die real am schnellsten war. Zwei Hardwaresysteme wurden zur Evaluierung verwendet – ein Desktop-Rechner und ein Laptop. Die Korrektheit schwankte zwischen 80 % und 90 % bzw. zwischen 70 % und 80 % je nach Plattform. Zudem wurden vier weitere Fallbeispiele erläutert, bei denen mit Brainy Optimierungen vorgenommen werden konnten. Gezeigt wird auch, wie die optimale Alternativdatenstruktur je nach Eingabeparameter und Zielplattform variieren kann. Die Ergebnisse zeigen, dass durchschnittlich ein ca. 1,2-facher Geschwindigkeitsvorteil durch Datenstrukturaustausch erzielt werden konnte.

Brainy versucht Datenstrukturen durch den Einsatz geeigneterer Alternativen zu optimieren. Die Analyse ist auf zufallsbasierten Eingabedaten zur Geschwindig-

keitsmessung gegebener Datenstrukturen gestützt. In der vorliegenden Arbeit stehen die realen Zugriffsmuster im Zentrum des Interesses. Weiterhin benötigt Brainy Zugriff auf hardwareinterne Maßzahlen, wie Anzahl von Cache-Verdrängungen, wohingegen Messungen hier auf solche Kenngrößen nicht zurückgreifen. Zudem setzt diese Diplomarbeit Historien nicht für die Optimierung von Datenstrukturen ein, sondern für die Parallelisierung. Dabei wird sogar die Ebene der Datenstrukturen verlassen und Rückschlüsse auf darüber liegende Algorithmen gezogen.

3.5 Software-Visualisierung

Software-Visualisierung (engl. *software visualization*) beschäftigt sich mit der Darstellung von Programmen, Algorithmen und Datenstrukturen. Unterschiedliche Aspekte werden dabei visualisiert:

- Quelltext: Zusammenhang zwischen Codeabschnitten
- Laufzeitverhalten:
    - Zeigerbeziehungen zwischen Instanzen und deren Umordnung
    - Häufigkeitsverläufe und Historien verschiedener Kennzahlen und Metriken

Solche Arbeiten beschäftigen sich ausschließlich mit der Darstellung von gemessenen Kennzahlen oder protokollierten Ereignissen. Das gewonnene Wissen wird nicht weiter verwendet und beispielsweise für Parallelisierung genutzt. In [MS93] wird auf Ebene der Klassen beobachtet, wie viele Objekte instanziiert werden und welche Methoden anderer Objekte im Laufe der Ausführung aufgerufen werden. Ein Aufrufgraph angereichert mit Instanziierungszählern kann dann im zeitlichen Verlauf Schritt für Schritt betrachtet werden. Die Arbeit [WM98] intensiviert den Einsatz von Animationen zur Veranschaulichung von Laufzeitdaten. Durch manuelle Instrumentierungen werden Animationen erzeugt, die dem Nutzer zu einem tieferen Verständnis über die Algorithmen verhelfen. Die Instrumentierungen sind dabei Anweisungen, wie Laufzeitattribute auf Parameter der Visualisierungsobjekte abgebildet werden. Zum Beispiel kann eine Arrayposition Auswirkungen auf die Höhe eines Balkens oder den Radius eines Kreises nehmen.

Diese Diplomarbeit visualisiert hingegen nicht nur die Historien über Zugriffsereignissen (ohne Animationen), sondern extrahiert zudem Parallelisierungsmöglichkeiten, die es dem Nutzer ermöglichen, schneller das Zielprogramm zu parallelisieren. Ein Anwender dieses Werkzeuges braucht das zu untersuchende Programm nicht in vollem Umfang zu verstehen. Die nachträgliche Verarbeitung der Laufzeitdaten ermittelt, welche Programmteile von Bedeutung sind, sodass der Nutzer diese nicht selbst angeben muss.

3.6 Zusammenfassung

	SW-Visualisierung [WM98], [MS93]	Speicherbereichsanordnungsoptimierung [ZD+11]	Speicherzugriffszeitanalyse [RB11]	Datenstrukturoptimierung [JR+11]	diese Arbeit
Zeitl. Verlauf von skalaren Messgrößen	+	∘	+	−	∘
Speichern des zeitl. Verlaufs der Zugriffe	∘	+	−	−	+
Deduktion von Optimierungspotenzial im Quelltext	−	−	−	+	+

Tabelle 1: Gegenüberstellung der verwandten Arbeiten

Tabelle 1 stellt zusammenfassend einzelne Aspekte der verwandten Arbeiten gegenüber. Man kann sehen, dass diese Arbeit Untersuchungsmethoden ähnlich der vorhandenen Literatur verwendet, um das Ziel der Identifikation von Parallelisierungspotenzial zu erreichen. Historien über Zugriffsereignisse anstatt skalarer Messgrößen kommen hier zum Einsatz. Letztlich werden dadurch zeitliche Verläufe von Zugriffen auf Datenstrukturen gespeichert, welche nach Parallelisierungspotenzial analysiert werden. Die ausgegebenen Hinweise intendieren algorithmische Parallelisierungen am Quelltext. Durch die zusätzliche grafische Repräsentation der Zugriffshistorien kann der Benutzer die angebotenen Parallelisierungspotenziale sehr gut nachvollziehen.

4 Voruntersuchungen

Wie bereits eingeführt, werden in dieser Arbeit lediglich Datenstrukturzugriffe zur Identifikation von Parallelisierungspotenzial herangezogen. Zielorientiert wurden dazu im Vorfeld explorative Fragen gestellt, deren Beantwortung nötig war, um eine adäquate Konzeption zu erstellen. Grundlegende Fragen waren beispielsweise: Welche Datenstrukturen werden von Programmierern verwendet, müssen also untersucht werden? Wie lassen sich Zugriffshistorien geschickt darstellen? Treten Regelmäßigkeiten in Zugriffshistorien auf? Lassen sich Fingerzeige aus den Historien ableiten? Die Antworten auf diese Fragen, welche in den folgenden Unterkapiteln gegeben werden, nehmen Einfluss auf die Konzeption.

4.1 Welche Datenstrukturen werden verwendet?

Verschiedenartige Datenstrukturen stehen dem Programmierer zur Verfügung, aber nicht alle werden im Programm verwendet. Lineare Datenstrukturen (wie zum Beispiel Liste, Stapel, Schlange, Array, etc.) werden im Allgemeinen von Programmierern sehr häufig eingesetzt, da sie einfach zu bedienen sind und in den meisten Sprachen nativ zur Verfügung stehen. Einige Entscheidungen des Programmierers darüber, dass bestimmte Datenstrukturen und Algorithmen zum Einsatz kommen, sind suboptimal (beispielsweise durch eine sequenzielle Denkweise) – bieten somit höchstwahrscheinlich Parallelisierungspotenzial. Daher stellt sich die Frage, welche Datenstrukturen Programmierer oft verwenden, wo dementsprechend die meiste Beschleunigung zu erwarten ist. Eine Vorstudie über den Einsatz von Datenstrukturen zeigt, dass einige der Datenstrukturen aus der Standardbibliothek deutlich bevorzugter von Programmierern eingesetzt werden als andere.

Programm	Klassenanzahl	Domäne
7zip	11	Kompression
Arcanum	14	Simulation
BeHappy	55	Bürosoftware
borys-MeshRouting	42	Simulation
clipper	25	Bürosoftware

cognitionmaster	200	Bildanalyse
compgeo	12	Bibliothek
Contentfinder	27	Suche
csparser	200	Parser
‚dsa‘	20	Bibliothek
dddpds (SmartCA)	250	Bürosoftware
dotqcf	350	Simulation
dotspatial	1500	Bibliothek
evo	40	Simulation
fire	24	Simulation
gpdotnet	45	Simulation
graphsharp	150	Graphenbibliotheken
greatmaps	250	Bürosoftware
ittycoon.net	54	Spiele
ManicDigger2011	220	Spiele
metaclip	25	Bürosoftware
Net_With_UI	9	Simulation
netinfotrace	100	Bürosoftware
orazio1	65	Bibliothek
OsmExplorer	227	Bürosoftware
ProcessHacker	21	Bürosoftware
rrrsroguelike	12	Spiele
rushHour	25	Simulation
SequenceViz	100	Programmvisualisierung
sharpener	25	Programmoptimierung
starsystemsimulator	18	Simulation
TerraBIB	60	Bürosoftware
theAirline	200	Spiele
TreeLayoutHelper	18	Graphenbibliotheken
twodsphsim	15	Simulation
waveletstudio	60	Bürosoftware
zedgraph	100	Graphenbibliotheken

Tabelle 2: Untersuchte Programme

Es wurden 37 Open-Source Programme aus verschiedenen Anwendungsdomänen (Kompression, Simulation, Graphenbibliotheken, Spiele, Datei- und Textsuche, Datenstruktur- und Algorithmenbibliotheken sowie Bürosoftware) von SourceForge und CodePlex heruntergeladen und analysiert. Alle Programme sind in C# programmiert. Tabelle 2 zeigt die Programme, deren Klassenanzahl und die Anwendungsdomäne. Die Programme haben zwischen 300 (rrrsroguelike) und 460.000 (dotspatial) Zeilen Quelltext und weisen eine Klassenanzahl zwischen 9 (Net_With_UI) und 1500 (dotspatial) auf.

Die manuelle Inspektion der Programme erfolgte mit regulären Ausdrücken unter Verwendung des Programms Notepad++. Es wurde nach Variablendeklarationen gesucht, welche mit einer Datenstruktur aus der Standardbibliothek definiert sein mussten. Alle Datenstrukturen der Standardbibliothek wurden betrachtet.

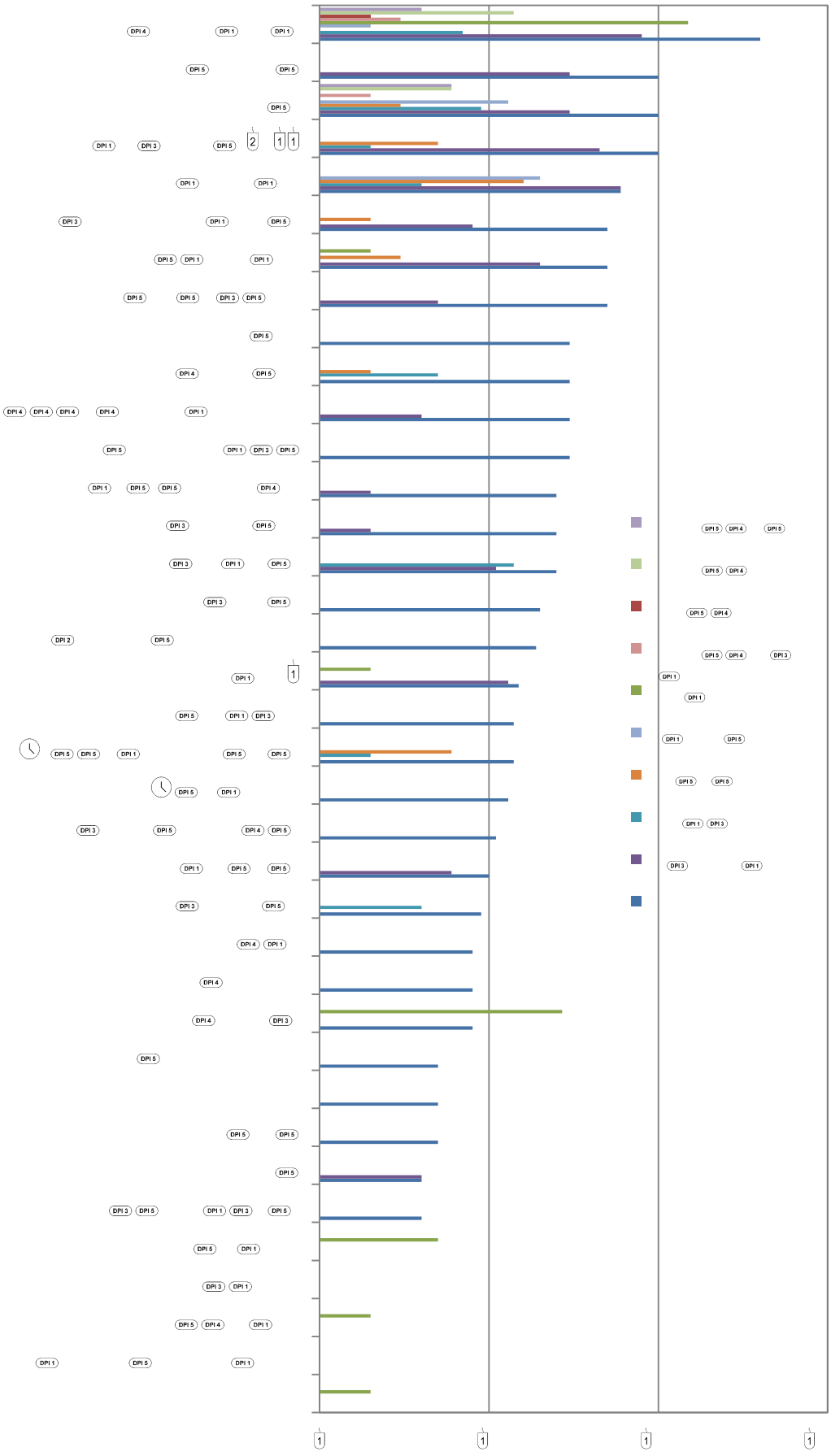

Abbildung 2: Anzahl von Datenstrukturen in Programmen

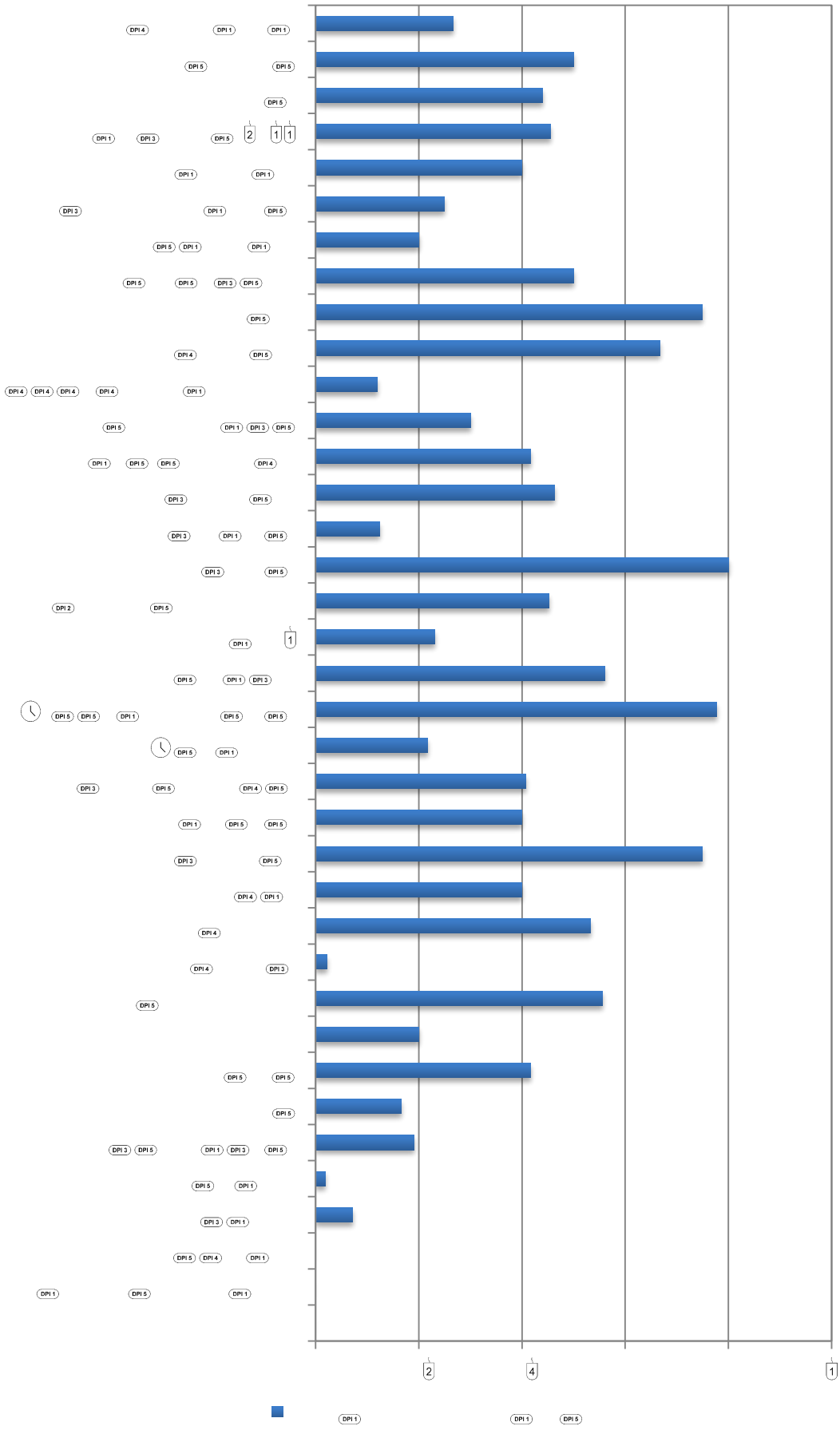

Abbildung 3: Anzahl Variablen vom Typ List pro Klasse

Tabelle 7 auf Seite 78 im Anhang zeigt eine detaillierte Ergebnisübersicht der Projekte und die jeweils enthaltenen Variablentypen. Abbildung 2 konstituiert das Ergebnis grafisch: für die jeweiligen Programme sind die Gesamtanzahl und die Häufigkeit der Variablendefinition von Datenstrukturen der Standardbibliothek ablesbar. Abbildung 3 zeigt zudem für jedes Programm, wie viele Variablen vom Typ `List` durchschnittlich in einer Klasse enthalten sind.

Die eingangs gestellte Frage, welche Datenstrukturen häufig zum Einsatz kommen, kann damit beantwortet werden und spiegelt sich wieder in Anforderung A3, sowie der Implementierung in Kapitel 6.2.2. Wie man erkennen kann, ist die `List` der mit Abstand meist benutzte Datentyp der Standardbibliothek von .NET. Mit fast 1300 Vorkommen wird sie vier Mal häufiger eingesetzt als die zweithäufigste Datenstruktur `Dictionary`. Durchschnittlich 0,35 Deklarationen von `List` kommen pro Klasse vor – sieben Mal mehr als das `Dictionary`. Im Schnitt enthält somit ca. jede dritte Klasse eine `List`. Das ergibt für die durchschnittlich 123 Klassen pro Programm eine Anzahl von 41 List Deklarationen je Open-Source-Projekt.

4.2 Wie kann man Zugriffshistorien visualisieren?

Eine geeignete Darstellung ermöglicht es, zu einem guten Verständnis über die Eigenschaften der Datenstrukturzugriffe zu gelangen. In diesem Kapitel wird darauf eingegangen, wie Zugriffshistorien in dieser Arbeit visualisiert sind.

Zugriffshistorien sind, wie in 2.1 beschrieben, die Menge von Zugriffsereignissen auf einer Datenstrukturinstanz. Sie haben, bedingt durch den Programmablauf, eine zeitliche Ordnung. Diese Ordnung ist die erste Dimension der Visualisierung. Weiterhin finden viele Zugriffe jeweils an einer bestimmten Stelle in der Datenstruktur statt (siehe indexbehaftete Zugriffe in Kapitel 5.2). Zusätzlich gibt es Zugriffe, die keinen Index brauchen oder zurückgeben (indexlos). Es liegt nahe, den Zugriffsindex und die Art des Zugriffs als zwei weitere orthogonale Parameter zur Visualisierung eines Zugriffsereignisses zu benutzen. Diese drei Dimensionen werden übersichtlich in einer Grafik veranschaulicht.

Realisiert wird dies durch ein zweidimensionales kartesisches Koordinatensystem, bei dem auf der x-Achse die Zeit und auf der y-Achse der Index eines Zugriffes abgetragen wird. Die dritte Dimension, also die Art eines Zugriffs, geht farbgebend in die Visualisierung ein. Da bei linearen Datenstrukturen die Länge eine interessante Rolle spielt und daher diese stets im Blick sein sollte, wird zu jedem Zugriff – zusätzlich zum Index – auch die aktuelle Länge abgetragen. Somit ist schnell einsichtig, ob beispielsweise im vorderen oder hinteren Teil eines Arrays zugegriffen wurde. Die indexlosen Zugriffe werden so behandelt, als hätten sie am letzten Element stattgefunden. Dies hat den Vorteil, dass bei einer Balkendarstellung diese Zugriffe ebenfalls gut erkennbar bleiben.

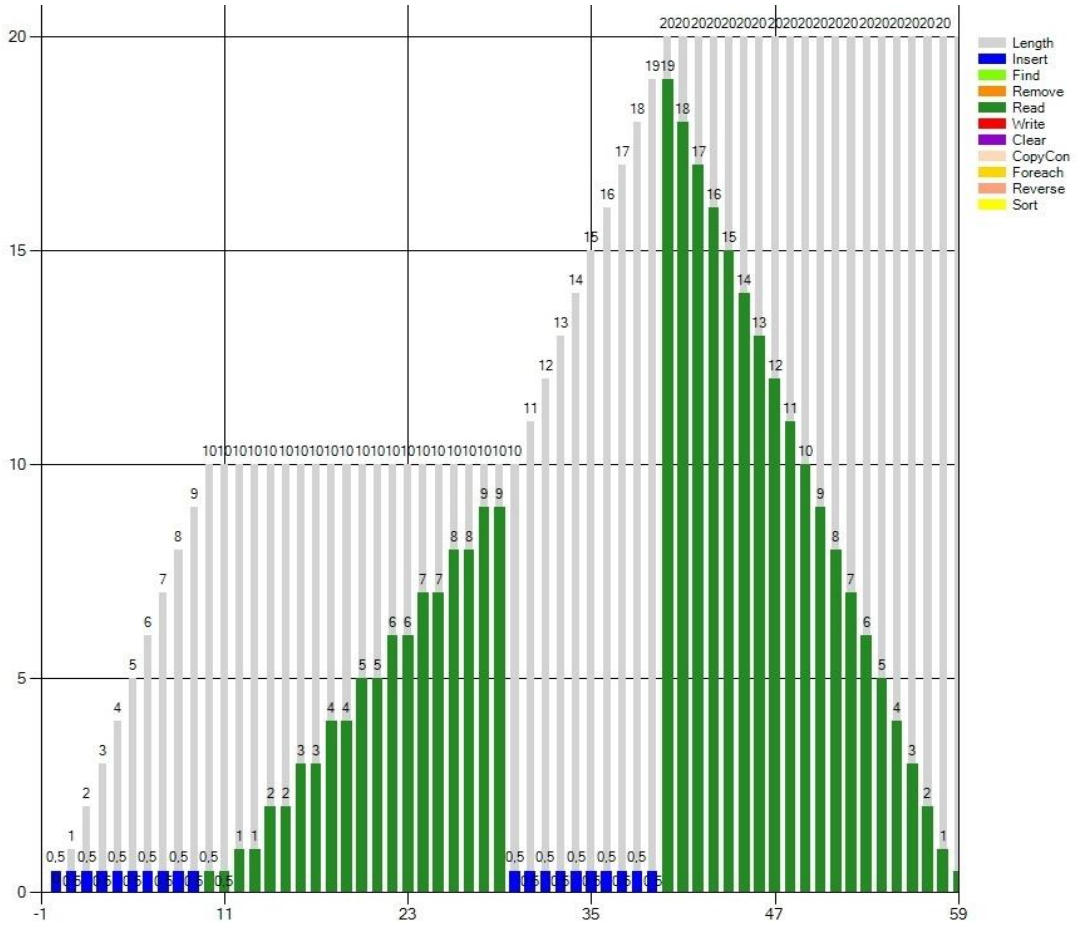

Abbildung 4: Einfache Historie mit Einfüge- und Leseoperationen

Abbildung 4 zeigt beispielhaft die Zugriffshistorie einer `List` mit der maximalen Länge von 20 Elementen. Die ersten zehn Operationen sind Einfügeoperationen von vorne. Danach wird einmal linear von vorn nach hinten jedes Element einmal gelesen. Dann wieder zehn Mal vorn eingefügt und im Anschluss umgekehrt (von hinten nach vorn) gelesen. Die Länge ist stets als grauer Balken sichtbar. Damit die Farbe und damit die Zugriffsart von Zugriffen auf den Index Null auch sichtbar werden, bekommen sie eine Höhe von 0,5.

Diskutabel ist weitergehend die Zeit der Zugriffe. Zwei Zeitskalen sind hier denkbar. Zum einen die reale Zeit oder aber eine Art virtuelle Zeit, die lediglich die Reihenfolge der Zugriffe durch Nummerierung wiedergibt, wie es bei Vektoruhren der Fall ist. Der Vorteil der realen Zeit ist, dass man genau sieht, in welchen Zeitabschnitten des Programmlaufs auf die Datenstruktur zugegriffen wurde und in welchen Zeitspannen nicht. Das reale zeitliche Verhalten wird hierdurch angemessen vertreten. Der Nachteil allerdings ist, dass dies unübersichtlich wird, wenn man sich für die reale Zeit gar nicht interessiert, sondern eher für die Abfolge der Zugriffsereignisse – unabhängig davon, wie viel Zeit zwischen zwei Zugriffen vergangen ist. Beide Skalen haben ihre Daseinsberechtigung, weshalb beide angeboten werden, sodass dem Benutzer alle Vorteile gewährt bleiben. Bei Abbildungen von Historien wird in der vorliegenden Arbeit jedoch immer die durchnummerierende Skala verwendet.

4.3 Treten Regelmäßigkeiten in Zugriffshistorien auf?

Will man nach Parallelisierungspotenzial suchen, braucht man Suchkriterien oder Suchregeln, die die Potenziale musterartig beschreiben. Es müssen dazu auf der einen Seite wiederkehrende Regelmäßigkeiten in Zugriffshistorien zu finden sein. Auf der anderen Seite müssen die Regelmäßigkeiten in ähnlicher Weise oft auftreten, da man es sonst mit programmspezifischen Strukturen zu tun hätte. Es stellt sich somit die Frage: Lassen sich Regelmäßigkeiten in den Zugriffshistorien finden und treten solche gehäuft auf?

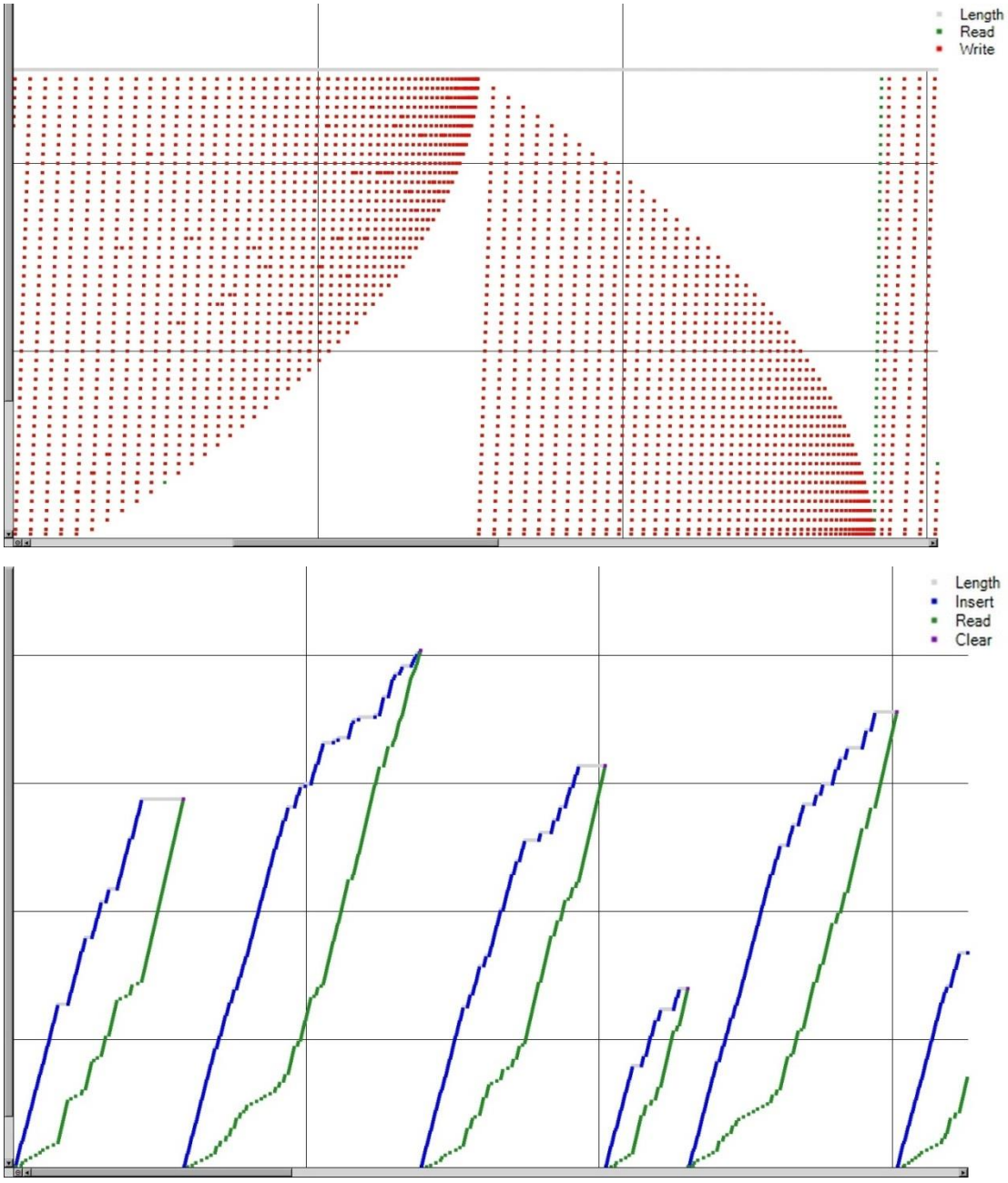

Abbildung 5: Beispielmuster verschiedener Programme

(oben: Linpack, unten: MeshRouting)

16 Programme in der Sprache C# wurden von SourceForge und CodePlex heruntergeladen. Deren Zugriffshistorien wurden einer manuellen Inspektion unterzogen. Dazu wurden die enthaltenen `List` und Array Variablen instrumentiert. Die Instrumentierung protokollierte die Zugriffe auf den Instanzen in einer komma-separierten Datei. Danach wurden händisch die Zugriffsdaten in Diagramme überführt. Diese prototypischen Zugriffshistorien wurden dann manuell bewertet und einer von drei Kategorien zugeordnet. Die erste Kategorie (erkennbare Regelmäßigkeiten) besagt, dass „neuartige“ Regelmäßigkeiten zu sehen sind. Die Kategorie „Redundante Regelmäßigkeiten“ enthält Historien, die keine „neuartigen“ Regelmäßigkeiten enthalten, sondern nur solche, die eine sehr starke Ähnlichkeit zu einer bereits gesehenen haben. Die dritte Kategorie enthält Historien, die keine Regelmäßigkeiten oder zu kleine Historien enthalten, die keine verwertbaren oder interessanten Informationen liefern.

Abbildung 5 zeigt beispielhaft zwei Zugriffshistorien, bei denen sich optisch Muster erkennen lassen. Beim oberen Beispiel wird ständig von vorn nach hinten geschrieben und später anstatt von ganz vorn, immer ein Index weiter hinten begonnen. Dadurch entsteht die erste weiße dreiecksförmige Fläche. Danach wird zwar wiederum immer wieder von vorn nach hinten geschrieben, allerdings immer früher aufgehört, sodass die zweite freie Fläche entsteht. Dann wird wieder stets komplett von vorn nach hinten geschrieben. Beim unteren Beispiel werden Elemente hinten angefügt, während zwischendrin Elemente aufsteigend gelesen werden. Irgendwann erreicht der Leseindex das letzte Element – das Einfügen wurde inzwischen eingestellt – woraufhin die Datenstruktur mit einem `Clear()` zurückgesetzt bzw. geleert wird. Dieser Prozess wiederholt sich immer wieder aufs Neue. Auf hoher Abstraktionsebene sieht man hier eine Implementierung einer Schlange.

Anzahl Historien … Programm	mit erkennbaren Regelmäßigkeiten	mit redundanten Regelmäßigkeiten	ohne verwertbare Regelmäßigkeiten
astrogrep		2	1
borys-MeshRouting	4	3	7
clipper	3	9	1
compgeo	2		
Contentfinder		2	
csparser	2	5	3
‚dsa‘		5	
dotqcf	4	2	
fire	1	1	1
ManicDigger2011	1	6	7
MidiSheetMusic	4	14	
Net_With_UI	3	11	3
netinfotrace	4	13	4
rrrsroguelike	1	1	
TerraBIB	2	1	1
TreeLayoutHelper		6	2
Σ	**31**	**81**	**30**

Tabelle 3: Anzahl von Regelmäßigkeiten in Zugriffshistorien

Tabelle 3 zeigt zu den Programmen jeweils, wie viele Historien den drei Kategorien zugeordnet sind. Identifiziert wurden 31 Historien mit Regelmäßigkeiten, die als „neuartig“ eingestuft wurden. 81 Historien enthielten Regelmäßigkeiten, die bereits in stark ähnlicher Form in der ersten Kategorie gezählt wurden.

Mit 31 als interessant verordneten Historien kann konstatiert werden, dass Regelmäßigkeiten in Zugriffshistorien auftreten. Weiterhin lässt sich aus 81 redundanten Historien schließen, dass Regelmäßigkeiten häufig auftreten. Dieses Ergebnis wird sich in Kapitel 5.4 als Teil des Konzeptes widerspiegeln.

4.4 Lassen sich Fingerzeige ableiten?

Die Regelmäßigkeiten in Zugriffshistorien werden durch Muster beschrieben. Diese Muster heißen Phasencharakteristika. Das Auftreten und die Länge der Phasen sollen Aufschluss darüber geben, ob eine Datenstruktur ein Parallelisierungspotenzial birgt. Es stellt sich somit die Frage, ob sich Fingerzeige ableiten lassen, die auf diese Parallelisierungspotenziale hinweisen.

Dazu wurde zunächst untersucht, ob sich anhand des Auftretens und der Eigenschaften der Phasen wiederkehrende Zugriffsverhalten erkennen lassen, Fingerzeige dementsprechend ableitbar sind. Verschiedene Programme aus unterschiedlichen Domänen wurden auf die Ableitbarkeit von Fingerzeigen untersucht. 48 Programme (ähnlich zu den in Kapitel 4.1 genannten) wurden dazu instrumentiert, sodass bei ihrer Ausführung Zugriffshistorien der Datenstrukturinstanzen protokolliert wurden. In den Zugriffshistorien wurde automatisiert nach den fünf Mustern gesucht, die den Fingerzeigen aus Kapitel 5.4 entsprechen.

Tabelle 8 im Anhang zeigt zu allen Programmen die Anzahl der dabei ausgegebenen Fingerzeige. Insgesamt ergeben sich 66 Hinweise, wobei 49 von langen Einfüge-Phasen und 10 von häufigen Linear-Lese-Phasen bedingt werden. Bei fast allen Programmen, bei denen keine Fingerzeige auftreten, sind keine instrumentierbaren Datenstrukturen enthalten.

4.5 Zusammenfassung

Die Studie „Welche Datenstrukturen werden verwendet?“ in Kapitel 4.1 zeigt, dass Programmierer sich entweder an einige wenige Datenstrukturen gewöhnt haben und diese phlegmatisch benutzen oder dass sie sich über die Existenz anderer Datenstrukturen nicht bewusst sind. Aus welchem Grund auch immer – die `List` ist die am häufigsten eingesetzte Datenstruktur in den untersuchten Programmen. Zweifellos ist das Array ebenfalls eine beliebte Datenstruktur. Lineare Datenstrukturen wie `List` und Array sind damit die meist benutzten Datencontainer. Deswegen schleichen sich hier auch die meisten Parallelisierungspotenziale ein, die im Rahmen dieser Arbeit aufgedeckt werden sollen. Aus diesem Grund konzentriert sich das Konzept auf diese beiden Datenstrukturen.

Die Studie „Treten Regelmäßigkeiten in Zugriffshistorien auf?“ in Kapitel 4.4 zeigt, dass regelmäßige Strukturen in Zugriffshistorien zu finden sind. Eine Beschreibungsmethodik zur Formulierung solcher Regelmäßigkeiten kann somit die Möglichkeit eröffnen, auf einer höheren Abstraktionsebene nach Verhaltensmustern zu suchen.

Die letzte Voruntersuchung in Kapitel 4.4 „Lassen sich Fingerzeige ableiten?“ zeigt, dass Fingerzeige durch Mustersuche in Zugriffshistorien abgeleitet werden können.

Die Art und Weise der Darstellung von Zugriffshistorien in dieser Arbeit wurde in Kapitel 4.2 beschrieben.

5 Eigener Ansatz

In diesem Kapitel wird das Konzept veranschaulicht. Dazu wird zunächst genauer auf die Zielstellung und die daraus abgeleiteten Schritte eingegangen. Zusätzlich werden Nebenbedingungen eingeführt, deren konzeptuelle Berücksichtigung ebenfalls beschrieben wird. In Kapitel 5.2 wird die Konzeption dieser Arbeit beleuchtet.

5.1 Ziel

Ein Programm wird maßgeblich von seinen Datenstrukturen getragen. Dennoch bleiben sie lediglich Datenbehälter, auf und mit denen algorithmisch versucht wird ein Problem zu lösen. Die Parallelisierung eines Programms stützt sich ebenfalls auf die enthaltenen Datenstrukturen. Auf der einen Seite parallelisieren Programmierer die Algorithmen, sodass sie laufzeiteffizienter auf den Datenstrukturen arbeiten. Auf der anderen Seite dienen sie hinterher als Informationsträger zum Datenaustausch zwischen parallelen Komponenten oder sie werden zum Informationsvermittler zwischen den Ausführungseinheiten in Form von Puffern und Warteschlangen. Aus dieser Vielfalt heraus stellt sich die Frage: Wie viele Informationen kann man dem Laufzeitverhalten einer Datenstruktur abgewinnen und für die Programmparallelisierung nutzbar machen?

Ziel dieser Diplomarbeit ist es daher, Parallelisierungspotenziale aus dem Laufzeitverhalten von Datenstrukturen in objektorientierten Programmen abzuleiten, wobei deren Orte, Begründungen und Handlungsempfehlungen den Programmierer zu den parallelisierungsrelevanten Datenstrukturen leiten.

Dafür sollen die Zugriffe auf Datenstrukturinstanzen analysiert werden. Die Voruntersuchung in Kapitel 4.3 zeigt, dass Zugriffshistorien durch enthaltene Regelmäßigkeiten greifbar gemacht werden können. Aus der Voruntersuchung in Kapitel 4.4 lässt sich zudem schlussfolgern, dass aus den Regelmäßigkeiten in Zugriffshistorien Verhaltensweisen einer Datenstrukturinstanz abgeleitet werden können. Diese Verhaltensweisen sollen Parallelisierungspotenzial aufdecken.

Aufbauend auf den Ergebnissen der Voruntersuchungen werden folgende Teilschritte aufgestellt:

- S1: Erstellung von Zugriffshistorien
- S2: Ableiten von Fingerzeigen

Im Folgenden werden Nebenbedingungen aufgezeigt, die bei der Durchführung der genannten Schritte berücksichtigt werden sollen. Die Lösungen, die diesen Anforderungen Rechenschaft tragen und das Konzept maßgeblich beeinflussen, werden jeweils dazu erläutert.

5.1.1 Nebenbedingungen zum Schritt S1 – Erstellung von Zugriffshistorien

A0 Unterstützung mehrfädiger Programme
Zugriffshistorien sollen nicht auf sequenzielle Programme beschränkt, sondern ebenfalls auf mehrfädige Programme anwendbar sein, weil auch in parallelisierten Programmen weitere Parallelisierungspotenziale enthalten sein können.

Lösungsansatz
Um dieser Nebenbedingung gerecht zu werden genügt es, zu einem Zugriffsereignis die Kennung des Fadens abzuspeichern, der den Zugriff ausgelöst hat. Dadurch kann hinterher unterschieden werden welche Fäden welche Zugriffe durchgeführt haben.

A1 Geringe Laufzeitbeeinflussung
Die Ermittlung der Zugriffshistorien soll möglichst wenig invasiv bezüglich der Laufzeit sein. Das Programm soll somit nach Möglichkeit genauso schnell arbeiten wie bisher.

Lösungsansatz
Zur Beobachtung eines Programms ist ein Eingriff unumgänglich. Dabei werden zur Laufzeit lediglich Informationen über Datenstrukturzugriffe gesammelt und schnellstmöglich zurückgelegt. Der Einsatz einer Protokollierung in einer Datei ist insofern unvorteilhaft, da die IO-Prozesse zu unerwünschten Wartezeiten führen. Das Halten der Protokollierungsinformationen im Speicher, beispielsweise durch eine hauptspeicherresistente Datenbank, ist ebenfalls ungünstig, da die Speichergröße zu einem limitierenden Faktor wird. Daher wird die Protokollierung über zwei Prozessräume realisiert. Im Prozessraum des zu untersuchenden Programms werden die Datenstrukturen beobachtet und gesammelte Zugriffsereignisse in einem Kanal zur Interprozesskommunikation mit einem zweiten Prozess abgelegt. Für das Empfangen ist eine weitere Komponente verantwortlich. Erst dort wird über die beobachteten Ereignisse Buch geführt und diese weiterverarbeitet. Alle weiteren Untersuchungsschritte erfolgen somit *offline*.

A2 Eingreifbarkeit des Nutzers in die Analyse
Ein Automatismus, der alle unterstützten Datenstrukturen beobachtbar macht, soll angeboten werden. Zusätzlich soll der Nutzer in analysebedingte Programmtrans-

formationen eingreifen und diese an seine Anforderungen anpassen können. Insbesondere soll er die zu beobachtenden Datenstrukturen bestimmen können, sodass beispielsweise nur ein paar ganz bestimmte Datenstrukturinstanzen untersucht werden. In diesem Fall muss es möglich sein, den Automatismus zu umgehen und manuelle Anpassungen vorzunehmen.

Lösungsansatz
Es gibt drei Möglichkeiten die Datenstrukturen beobachtbar zu machen: durch Instrumentierung des Quelltextes, der Zwischensprache (des IL-Codes) oder der kompilierten Binärdateien. Damit der Programmierer manuell eingreifen kann, ist erstere Option, dass der Quelltext des zu untersuchenden Programms instrumentiert wird, die angemessenste. Eine instrumentierende Komponente wird Anpassungen im Quellcode vornehmen, die eine Beobachtung zur Laufzeit ermöglichen. Der Nutzer hat dann immer noch die Möglichkeit, Instrumentierungen rückgängig zu machen bzw. neue hinzuzufügen oder aber den Automatismus ganz wegzulassen und selbst manuell zu instrumentieren.

A3 Datenstrukturkompatibilität
Aus den Ergebnissen der Voruntersuchung in Kapitel 4.1 wurde die Konsequenz gezogen, dass die `List` und das Array die Datenstrukturen sind, mit denen sich diese Arbeit auseinandersetzt, weshalb zunächst nur diese zur Beobachtung unterstützt werden. Dennoch müssen weitere Datenstrukturen später erweitert werden können. Es bedarf somit einer Möglichkeit verschiedenartige Datenstrukturen zu untersuchen.

Lösungsansatz
Um dieser Nebenbedingung gerecht zu werden, wird das Entwurfsmuster des Stellvertreters eingesetzt. Für eine unterstützte Datenstruktur muss lediglich ein Stellvertreter erstellt werden, der alle ursprünglichen Schnittstellen implementiert. Die stellvertretenden Methoden melden einen Zugriff an die protokollierende Stelle und rufen danach die jeweiligen Methoden der intern gehaltenen Datenstruktur auf. Eine Schnittstelle zur Benutzung der protokollierenden Komponente ermöglicht die einheitliche Verarbeitung der Zugriffsereignisse und vereinfacht damit die Erstellung neuer Stellvertreter, wodurch die Kompatibilität zu weiteren Datenstrukturen hergestellt ist.

A4 Unverändertes Nutzungsverhalten des zu untersuchenden Programms
Die Art und Weise der Ausführung und Benutzung des Programms soll nicht eingeschränkt werden, wenn deren Datenstrukturen unter Beobachtung stehen. Das bedeutet beispielsweise, dass ein eingesetztes Testwerkzeug weiterhin genauso benutzt werden können muss wie zuvor.

Lösungsansatz
Übereinstimmend mit dem Lösungsansatz von A2 wird die Verwendung des Quelltextes auch dieser Nebenbedingung gerecht. Dadurch, dass das Programm später kompiliert wird, können Instrumentierungsänderungen am Code vorge-

nommen werden, ohne dass davon die Art und Weise der Benutzung eingeschränkt wird.

5.1.2 Nebenbedingungen zum Schritt S2 – Ableiten von Fingerzeigen

A5 Verständlichkeit der Fingerzeige

Fingerzeige sollen nachvollziehbar sein und den Nutzer auf weitere Untersuchungsmöglichkeiten hinweisen. Das bedeutet, dass Fingerzeige dem Nutzer so präsentiert werden müssen, dass er sie versteht und Parallelisierungspotenziale umsetzen kann.

Lösungsansatz

Fingerzeige bestehen aus vier Teilen. Am wichtigsten ist die Quelltextstelle der Instanziierung einer Datenstruktur, welche ein Parallelisierungspotenzial offeriert. Zudem wird eine Handlungsempfehlung ausgegeben, die besagt was getan werden sollte, um ein Parallelisierungspotenzial auszunutzen. Als Drittes wird eine Begründung angegeben, die dem Nutzer sagt, warum das Verfahren die ausgegebenen Handlungsempfehlungen gemacht hat. Schließlich gibt es eine Kennzahl, die die Ordnung der Fingerzeige angibt, sodass vielversprechende Fingerzeige von weniger wichtigeren unterschieden werden können. Diese vier Bestandteile eines Fingerzeigs verschaffen dem Nutzer ausreichend Klarheit darüber, wo und wie er sein Programm parallelisieren kann.

Die Visualisierung der Historien trägt ebenfalls dazu bei, dass der Nutzer die Ausgaben besser versteht. Er kann dort eigenständig die Historie inspizieren und die angebotenen Fingerzeige bewerten. Ein einheitliches Farbschema unterstützt ihn dabei.

A6 Visualisierung großer Datenmengen

Die Visualisierung soll auch bei großen Datenmengen handhabbar sein, sodass große Historien vom Nutzer untersuchbar bleiben. Die Darstellung einer Historie muss dementsprechend die Möglichkeit bieten, einen Teilbereich so zu vergrößern, dass die Menge der Datenpunkte übersichtlich und damit begreifbar wird.

Lösungsansatz

Zur Visualisierung wird ein kartesisches Koordinatensystem mit Zeit- und Index-Achse eingesetzt. Allgemeine Bedienungsmöglichkeiten, insbesondere das Vergrößern von Ausschnitten oder die Navigation durch die Datenpunkte, sind verfügbar.

5.2 Konzept

Im Folgenden wird der Lösungsweg auf verschiedenen Abstraktionsebenen beschrieben. Die Darstellungen orientieren sich an der Top-down-Methode. Das heißt, dass von den grobgranularen Zusammenhängen ausgehend auf die feingranularen Details eingegangen wird. Ein Überblick über das Konzept setzt zunächst die Grundlagen weiterer Ausführungen. Die folgenden Unterkapitel gehen genauer auf die einzelnen Konzeptschritte (KS) ein und verorten deren Bedeutung zu den anfangs gesetzten Zielen.

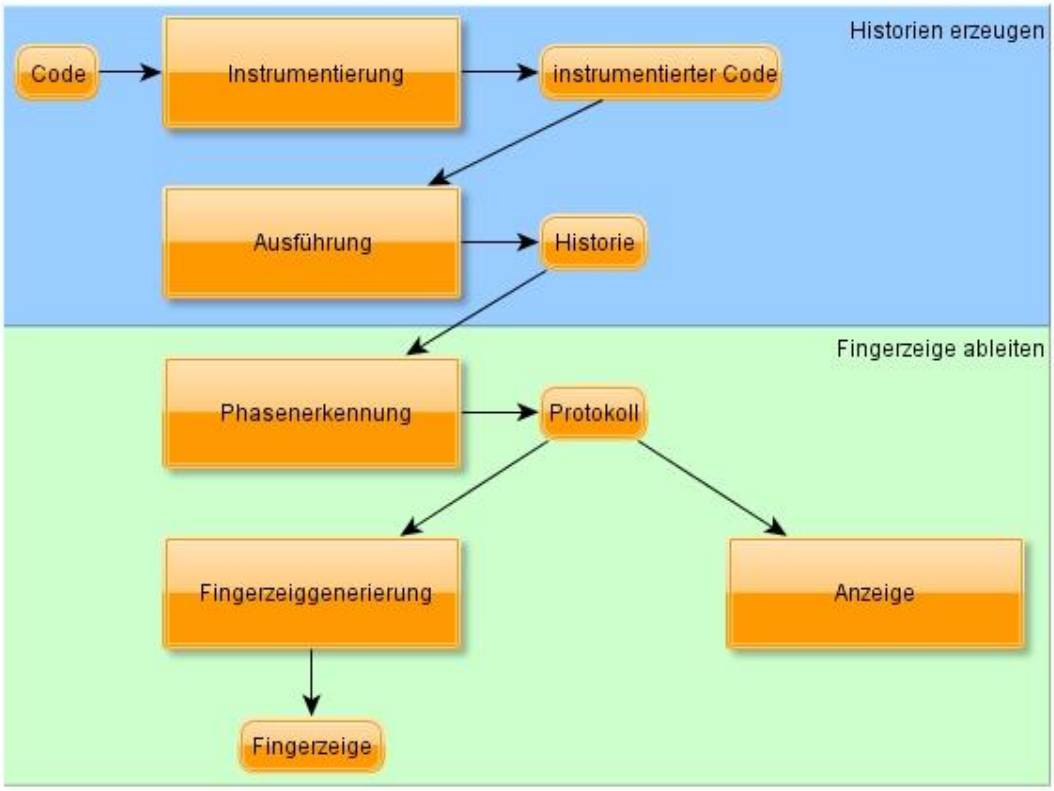

Abbildung 6: Lösungsweg als Prozess

Das Lösungskonzept ist prozessorientiert in Abbildung 6 gezeigt. Die eckigen, großen Kästchen repräsentieren Aktivitäten bzw. Module für bestimmte Aufgaben. Die runden, kleineren Kästchen stellen entstehende und ausgetauschte Daten dar.

Die erste Hälfte des Prozesses realisiert die Erstellung der Zugriffshistorien. Um die Zugriffe auf programminterne Datenstrukturen zu protokollieren, müssen Beobachtungsmechanismen in das Programm injiziert werden. Wie im Anforderungskapitel 5.1.1 motiviert, wird dazu der Quelltext herangezogen. Das erste Modul instrumentiert das Programm durch Änderungen am Code. Der Nutzer kompiliert im zweiten Schritt den instrumentierten Code und bringt ihn zur Ausführung. Die eingefügten Instrumentierungen sorgen dafür, dass beim Programmlauf die Zugriffe auf den Datenstrukturen protokolliert werden. Die Protokollierung wird in den Historien abgespeichert.

In der zweiten Hälfte des Prozesses geht es um die Analyse der Historien und die daraus entstehenden Fingerzeige. In einem ersten Schritt identifiziert ein Phasenerkenner die in den Historien enthaltenen Phasen. Die Phasen ermöglichen durch Gruppierung von Zugriffsereignissen, dass die Menge von beobachteten Zugriffen handhabbar wird. Historien, die durch enthaltene Phasen erweitert wurden, werden Protokolle genannt (siehe Begriffsdefinition in Kapitel 2.1). Auf der Abstraktionsebene der Phasen und gegebenenfalls unter Einbezug der Zugriffsereignisse sucht das Fingerzeigmodul im letzten Schritt nach Parallelisierungspotenzial-Mustern in den Protokollen. Die abgeleiteten Fingerzeige werden dem Nutzer anschließend ausgegeben.

Zusätzlich visualisiert eine grafische Oberfläche die Protokolle. Dem Nutzer werden dazu die Historien inklusive der erkannten Phasen angezeigt. Hier kann auch eingesehen werden, warum die Fingerzeige ausgegeben wurden – welche Muster also wo auftraten.

Zugriffsereignisse und Zugriffsarten

Nachdem der Begriff Zugriffsereignis bereits abstrakt definiert wurde, soll er an dieser Stelle konzeptionell erweitert werden. Ein Zugriffsereignis ist zunächst das Stattfinden eines Zugriffs auf eine Datenstruktur. Das Konzept Zugriffsereignis materialisiert nun dieses Vorkommnis im Zusammenschluss von Informationen, die mit einem solchen Zugriff in Verbindung stehen. Folgende Informationen ergeben die Dimensionen des Begriffs:

- Art des Zugriffs
- Realzeitstempel (-punkt)
- Position des Zugriffs
- Länge der Datenstruktur zum Zugriffszeitpunkt
- Kennung des Fadens, der den Zugriff angestoßen hat (im Kontext eines mehrfädigen Programms)

Die Art des Zugriffs entspricht nicht den Schnittstellenmethoden der Datenstruktur. Vielmehr werden die angebotenen Methoden den folgenden Zugriffsarten zugeordnet:

- Einfügen
- Suchen
- Löschen
- Lesen
- Schreiben
- Leeren*
- Kopieren/Konvertieren*
- Umdrehen*
- Sortieren*
- Auf-Alle-Anwenden-Operator* (`ForEach`)

Die ohne (*) gekennzeichneten Zugriffsarten brauchen einen Index als Parameter oder geben einen Index als Rückgabewert zurück. Sie werden daher als „indexbe-

haftet“ bezeichnet. Komplementär dazu sind die mit (*) gekennzeichneten „indexlose“ Arten.

Zu diskutieren ist die Abbildbarkeit indexloser Zugriffsarten durch indexbehaftete, da sie zumeist Algorithmen anstoßen und damit Operationen einer höheren Abstraktionsebene sind. Das Sortieren und das Umdrehen können beispielsweise jeweils durch eine Ansammlung von Lese- und Schreibzugriffen dargestellt werden. „Kopieren“ und „Leeren“ kann durch Lesen und Löschen abgebildet werden. Der Grund für das Zusammenfassen bzw. Zusammenlassen der Zugriffe ist der Informationsverlust. Zum Zeitpunkt des Zugriffs ist klar, welche Meta-Operation stattfindet – zum Beispiel das Sortieren. Würde man nun die Information des Sortierens durch Lese- und Schreiboperationen ersetzen, wäre sie verloren. Die Phasenerkennung versucht aber gerade solche Metainformationen aufzudecken. Aus diesem Grund erhalten die Zugriffe, die Operationen einer höheren Abstraktionsebene sind, eigene Zugriffsarten.

Im Gegensatz dazu wird mit kombinierten Operationen anders umgegangen, wenn sich keine genügend abstrakte Operation rechtfertigen lässt. Implementiert eine Schnittstellenmethode mehrere Operationen auf einmal, werden diese auch auf mehrere Zugriffe unterschiedlicher Arten abgebildet. Zum Beispiel das Löschen von allen Elementen, die eine bestimmte Eigenschaft erfüllen (`public int RemoveAll(Predicate<T> match)`), würde auf Such- und Löschoperationen abgebildet werden.

Das Konzept indexbehafteter und indexloser Zugriffe und die Auflistung der Zugriffsarten ergibt sich aus den Schnittstellenmethoden von `List` und Array. Da nur diese beiden Datenstrukturen berücksichtigt werden, wird sich auf die dafür nötigen Zugriffsarten beschränkt. Bei Bedarf können aber neue Zugriffsarten erweiternd definiert werden. Andere Indizierungsmethoden, wie sie beispielsweise bei Bäumen, Schlangen, Stapel oder assoziativen Datenfeldern nötig wären, werden dementsprechend ebenfalls nicht berücksichtigt.

5.3 KS1 – Erstellung von Zugriffshistorien

Instrumentierung

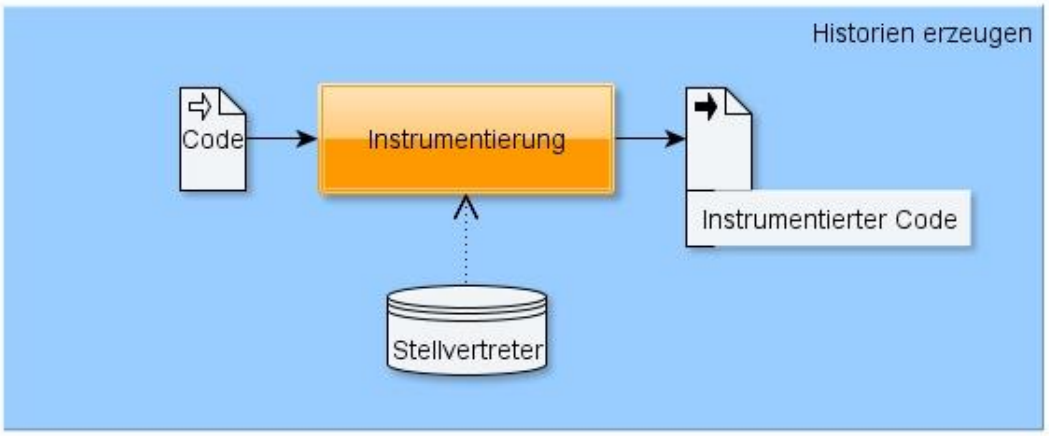

Abbildung 7: Instrumentierung als Teilprozess

Bei der Frage, wo die Beobachtung der Zugriffe auf Datenstrukturen instrumentiert werden soll, gibt es zwei mögliche Ansätze. Die eine Möglichkeit besteht darin vor jedem Methodenaufruf eine neue Zeile Code hinzuzufügen, die den Aufruf protokolliert. Die andere Möglichkeit ist, die Methode selbst so zu ändern, dass diese „von sich aus“ die Protokollierung anstößt. Da zwischen den Instanzen einer Datenstruktur unterschieden werden muss, müsste die aufgerufene Instanz mit übergeben werden. Zur Protokollierung müsste dann eine Buchhaltung von Instanzen stattfinden. Der Vorteil des zweiten Ansatzes ist an dieser Stelle, dass die Instanzen bereits die Buchführung darstellen und damit dieser Mehraufwand wegfällt. Der Nachteil ist, dass die Methoden von Datenstrukturen der Standardbibliothek nicht ohne Weiteres instrumentiert werden können. Dies wird aber durch das Entwurfsmuster „Stellvertreter“ umgangen. Eine Stellvertreter-Klasse kapselt dabei eine zu beobachtende Datenstruktur und bietet nach außen hin exakt die gleiche Schnittstelle an. Damit ist die ursprüngliche Datenstruktur problemlos austauschbar und gleichzeitig übernimmt der Stellvertreter die Protokollierung. Die resultierende Trennung von instrumentiertem Code und Protokollierungsmechanismus ermöglicht eine einfache Wartbarkeit.

Die Instrumentierung fügt somit effektiv keine neuen Anweisungen im Originalquelltext hinzu, sondern ersetzt die zu beobachtenden Datenstrukturen durch die entsprechenden Stellvertreter (Abbildung 7).

Ausführung

Da das Programm als Quelltext vorliegt und die Datenstrukturen durch Stellvertreter ausgetauscht werden, kann das Programm wie gewohnt ausgeführt werden. Dies ist unabhängig davon, ob der Nutzer bisher das Programm gestartet oder ein Testwerkzeug verwendet hat. Wie in Nebenbedingung A1 beschrieben, soll der Aufwand der Protokollierung in den Stellvertretern gering gehalten werden. Aus diesem Grund legen die Stellvertreter lediglich ihre Zugriffsereignisse in einem

Puffer ab. Nach der Ausführung wird der Puffer ausgelesen und die Zugriffsereignisse in die Zugriffshistorien einsortiert.

5.4 KS2 – Ableiten von Fingerzeigen

Die Suche von Fingerzeigen innerhalb der Zugriffshistorien ist in zwei Schritte untergliedert. Zu Beginn werden Phasen erzeugt, um die Zugriffsereignisse greifbar zu machen. Danach wird die Menge der Phasen nach fingerzeig-induzierenden Verhaltensmustern durchsucht. Die folgenden Kapitel gehen auf diese beiden Teilschritte ein.

5.4.1 Phasenerkennung

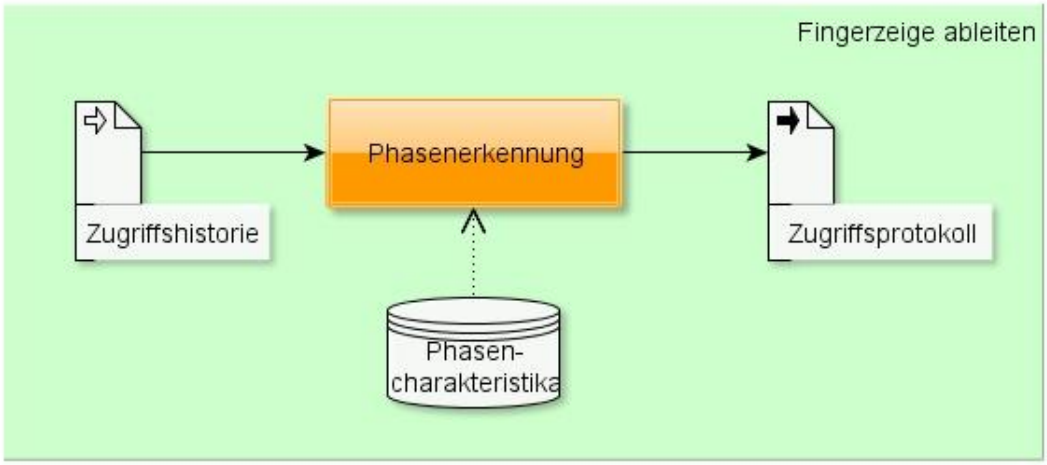

Abbildung 8: Phasenerkennung als Teilprozess

Die Phasenerkennung nimmt Zugriffshistorien entgegen und erzeugt daraus Zugriffsprotokolle, indem Phasen in den Historien erkannt und abgespeichert werden. Dabei wird auf die Menge der Phasencharakteristika zurückgegriffen (Abbildung 8).

Wie in Kapitel 2.1 beschrieben, besteht eine Phase aus einer Menge von Zugriffsereignissen, die gemeinsam in ähnlicher Weise häufig auftreten und damit Ausprägung eines Musters sind. Die Ereignisse dieser Menge werden jeweils einem bestimmten Phasencharakteristikum gerecht.

Das Konzept der Phasenidentifikation ist folgendermaßen zu verstehen: Drei benachbarte Zugriffsereignisse werden auf die Bedingungen eines Charakteristikums hin geprüft. Erfüllen sie diese, gehören sie in eine gemeinsame Phase des entsprechenden Phasencharakteristikums. Zwischen zwei getrennten Phasen muss mindestens ein bedingungswidriges[1] Zugriffsereignis liegen, ansonsten gehören die beiden vermeintlichen Phasen zur selben Phase. Phasencharakteristika partitionieren somit die Menge der Zugriffsereignisse in maximal große Partitionen, welche

[1] Es genügt, wenn eine der Bedingungen nicht zutrifft.

einem Charakteristikum gerecht werden, und in dazwischen liegende phasenfremde Abschnitte.

Folgende Charakteristika wurden durch manuelle Inspektion der Zugriffshistorien identifiziert und sind für diese Arbeit relevant:

- `Lineares-Lesen-Vorwärts`
  direkt benachbarte Lesezugriffe, deren Zugriffsindizes mit der Zeit (monoton) steigen
- `Lineares-Lesen-Rückwärts`
  direkt benachbarte Lesezugriffe, deren Zugriffsindizes mit der Zeit (monoton) fallen
- `Lineares-Schreiben-Vorwärts`
  direkt benachbarte Schreibzugriffe, deren Zugriffsindizes mit der Zeit (monoton) steigen
- `Lineares-Schreiben-Rückwärts`
  direkt benachbarte Schreibzugriffe, deren Zugriffsindizes mit der Zeit (monoton) fallen
- `Einfügen-Vorn`
  direkt benachbarte Einfügeoperationen, die stets vorn stattfinden
- `Einfügen-Hinten`
  direkt benachbarte Einfügeoperationen, die stets hinten stattfinden
- `Löschen-Vorn`
  direkt benachbarte Löschoperationen, die stets vorn stattfinden
- `Löschen-Hinten`
  direkt benachbarte Löschoperationen, die stets hinten stattfinden

Die Phasenerkennung findet für alle Charakteristika separat statt. Somit entstehen Phasen unterschiedlichen Typs, die sich evtl. überlappen. Zugriffsereignisse werden dementsprechend keiner, einer oder mehreren Phasen zugeordnet.

In Abbildung 4 kann man beispielsweise Phasen der beiden Phasencharakteristika `Einfügen-Vorn` und `Lineares-Lesen-Vorwärts` erkennen. Beide Charakteristika bedingen je zwei Phasen. Zum einen sind dies die beiden zusammenhängenden Abschnitte grüner Balken und zum anderen die beiden Abschnitte blauer Balken. Es liegen folglich insgesamt vier Phasen vor, die sich in diesem Fall nicht überlappen.

Bei einigen Zugriffsmustern ist es jedoch nicht nötig oder gar unerwünscht, dass Zugriffe tatsächlich direkt benachbart sein müssen. Wenn beispielsweise eine Schlange im parallelen Kontext zum Einsatz kommt, werden Elemente eingefügt und andere gleichzeitig bei der gegenüberliegenden Seite herausgenommen. Phasen würden dann zerrissen und nicht mehr als Ganzes betrachtet werden können. Aus diesem Grund gibt es zusätzlich weitere Phasencharakteristika, bei denen die enthaltenen Zugriffsereignisse nicht direkt benachbart sein müssen. Aus diesem Grund ist bei denen die Bedingung der direkten Nachbarschaft aufgeweicht. Folgende Phasencharakteristika wurden in Analogie zu den bereits genannten identifiziert:

- `Wenn-Lesen-Dann-Linear-Vorwärts`
- `Wenn-Lesen-Dann-Linear-Rückwärts`
- `Wenn-Schreiben-Dann-Linear-Vorwärts`
- `Wenn-Schreiben-Dann-Linear-Rückwärts`
- `Wenn-Einfügen-Dann-Vorn`
- `Wenn-Einfügen-Dann-Hinten`
- `Wenn-Löschen-Dann-Vorn`
- `Wenn-Löschen-Dann-Hinten`

Die Benennung mit Hilfe der „Wenn-Dann"-Kombination verdeutlicht, dass Zugriffsereignisse nicht mehr direkt benachbart sein müssen, um zur selben Phase zu gehören, sondern beliebig viele bedingungswidrige Zugriffsereignisse dazwischen liegen dürfen. Gleichzeitig folgt daraus folgende Konsequenz: Ein Zugriffsereignis, das einer der oben genannten Phasen zugeordnet wurde, ist auf jeden Fall auch der korrelierten unteren Phase zugehörig. Es kann beispielsweise nicht vorkommen, dass eine Leseoperation einer „Lineares-Lesen-Vorwärts"-Phase zugeordnet wird, aber keiner „Wenn-Lesen-Dann-Linear-Vorwärts"-Phase.

5.4.2 Fingerzeigerkennung

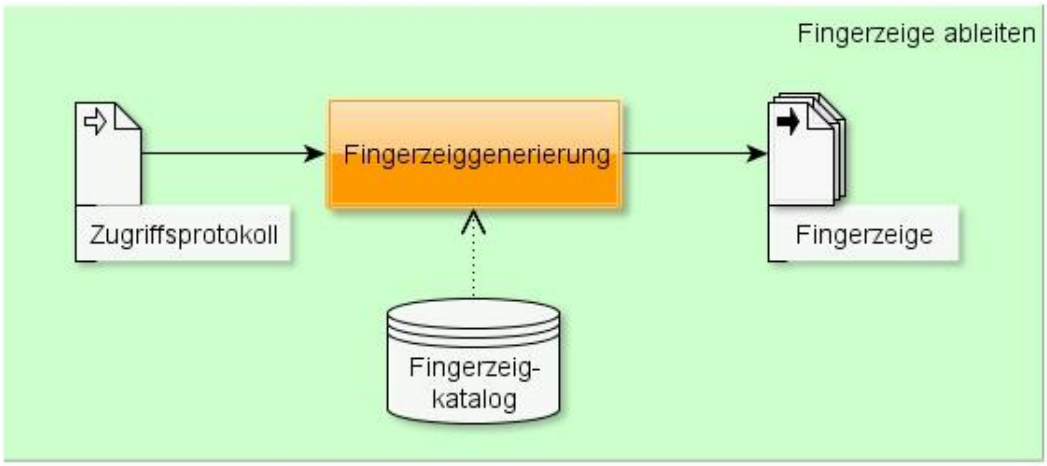

Abbildung 9: Fingerzeigerkennung als Teilprozess

Jedes Parallelisierungspotenzial ist einem parametrisierbaren Verhaltensmuster zugeordnet. Solche Muster beschreiben das Auftreten von Phasen und deren Eigenschaften. Wird ein Muster in einem Zugriffsprotokoll gefunden, wird ein Fingerzeig ausgegeben (Abbildung 9). Um irrelevante Datenstrukturen herauszufiltern, stehen zwei globale Schwellenwerte zur Verfügung. Einer gibt an, wie groß eine Datenstruktur mindestens gewesen sein musste, damit sie als laufzeitrelevant gilt. Kleine Datenstrukturen, die wenig Parallelisierungspotenzial aufweisen, da sie in den Cache passen, werden somit nicht weiter betrachtet. Der zweite Parameter spezifiziert die Anzahl der Zugriffe, die auf der Datenstruktur stattgefunden haben müssen, damit eine Parallelisierung Laufzeitvorteile erwarten lässt.

Fingerzeige werden sortiert ausgegeben, damit der Benutzer stichhaltige Hinweise zuerst bearbeiten kann. Folgende Ordnungsrelation sortiert Fingerzeige: Je mehr Zugriffe auf einer Datenstrukturinstanz stattgefunden haben, desto triftiger ist dort das Parallelisierungspotenzial. Die Reihenfolge bestimmende Kennzahl eines Fingerzeigs wird dementsprechend durch die Anzahl der Zugriffsereignisse in der betroffenen Historie berechnet und Ordnungskennzahl genannt.

5.4.3 Katalog der Fingerzeige

Im Folgenden werden die Parallelisierungspotenzial-Muster näher erläutert. Dazu wird geklärt, wie das jeweilige Muster aussieht und welcher Parallelisierungsvorschlag daraus resultiert. Unscharfe Zahlenangaben sind dabei stets als Schwellenwerte zu verstehen, deren programmatische Umsetzungen in Kapitel 6.6 aufgelistet sind. Die Fingerzeige sind Resultat einer manuellen Analyse der Protokolle. Die Auflistung der Fingerzeige beansprucht daher keine Vollständigkeit. Es können weitere Fingerzeige hinzugefügt werden. Die Phasen können dafür zu Hilfe genommen werden, welche aus diesem Grund ebenfalls erweiterbar sind.

LE – Langes Einfügen

Dieses Muster trifft zu, wenn es häufig Einfüge-Phasen gibt, die länger sind als ein gesetzter Schwellenwert. Dabei wird nicht unterschieden, ob stets vorn eingefügt wird oder hinten.

Die Handlungsempfehlung liegt hierbei darin, dass das Einfügen parallelisiert werden kann.

IS – Implementierung einer Schlange

Dieses Muster beschreibt die Verwendung einer `List` für die Realisierung einer Schlange. Eine Schlange ist eine lineare Datenstruktur, an deren einem Ende Elemente eingefügt und am anderen Ende entnommen werden. Dieses Muster wird entsprechend gefunden, wenn es ein großer Anteil schreibender Operationen an der einen Seite der Datenstruktur und ein großer Anteil löschender Operationen auf der anderen Seite auftreten.

Der Handlungsvorschlag lautet hier, dass die Datenstruktur „Schlange“ eingesetzt werden sollte, da der Programmierer vermutlich gerade dies beabsichtigte. Zudem sollte er prüfen, ob die Funktionseinheiten, die schreibend und lesend auf der Schlange arbeiten, parallel als Pipeline implementiert werden können.

SnE – Sortierung nach dem Einfügen

Wenn eine Liste nach einer langen Einfüge-Phase sortiert wird, spielt es offenbar keine Rolle, in welcher Reihenfolge die Elemente eingefügt werden. Eine Parallelisierung des Einfügens kann sich lohnen. Entsprechend wird dieses Muster gefunden, wenn eine Sortierung direkt nach einer langen Einfüge-Phase erkannt wird.

Es wird vorgeschlagen die Teile des Quelltextes zu parallelisieren, die für das Einfügen und das Sortieren verantwortlich sind.

HS – Häufiges Suchen

Ein häufiger Gebrauch von Suchoperationen deutet darauf hin, dass eine für die Suche optimierte Datenstruktur verwendet werden sollte. Bei Listen würde eine Suche immer wieder mit linearem Aufwand traversieren. Der Einsatz eines Suchbaumes oder der Binärsuche auf sortierten Listen bzw. Arrays stellen laufzeiteffiziente Alternativen dar. Dies amortisiert sich sehr schnell, wenn häufig Suchanfragen auf langen Datenstrukturen gestellt werden. In solchen Fällen wird das Muster gefunden.

Die Handlungsempfehlung ist die Parallelisierung der Suchoperation. Zusätzlich könnte der Algorithmus dahin gehend geändert werden, dass nicht mehr linear, sondern mit logarithmischem Aufwand gesucht werden kann.

HlL – Häufig langes Lesen

Analog zum vorangegangenen Muster, wird hier ebenfalls auf potenzielle Suchanfragen hingewiesen. Potenziell deswegen, weil in diesem Muster nicht mehr die Zugriffsart „Suchen“ auf der Datenstruktur auftritt, sondern lediglich häufiges lineares Lesen. Wenn es viele Phasen gibt, bei denen lineares Lesen über einen Großteil der Datenstruktur erfolgt, könnte es sich um einen ausprogrammierten Suchalgorithmus handeln. In diesem Fall greift dieselbe Argumentation wie beim Muster „Häufiges Suchen“.

Entsprechend lautet hier der Handlungsvorschlag zu überprüfen, ob die Leseoperationen von einer Schleife stammen, die bestimmte Elemente sucht. Der Suchalgorithmus sollte dann parallelisiert und die Datenstruktur durch eine Suchdatenstruktur ersetzt werden.

Fingerzeige zu Optimierungspotenzialen

Durch die Inspektion der Zugriffshistorien entstanden nicht nur Fingerzeige mit parallelisierendem Charakter. Folgende Fingerzeige orten sequenzielles Optimierungspotenzial und werden daher in dieser Arbeit nicht evaluiert.

Einfügen oder Löschen an vorderster Stelle

Einfüge- und Löschoperationen an vorderster Stelle bei der Datenstruktur `List` erzwingen das Kopieren aller Elemente in ein neues Objekt angepasster Größe. Das Muster trifft zu, wenn Einfüge- oder Löschoperationen gemeinsam „häufig“ auftreten.

Der Handlungsvorschlag ist eine verlinkte Liste zu verwenden, um das erzwungene Kopieren zu vermeiden.

Implementierung eines Stapels

In Analogie zum Muster der Schlangenimplementierung (siehe Kapitel 5.4.3) wird hier die Verwendung eines Stapels erkannt, wenn viele Einfüge- und Löschoperationen an derselben Seite der Datenstruktur stattfanden.

Entsprechend lautet der Handlungsvorschlag hier fürwahr, dass die Datenstruktur Stapel eingesetzt werden sollte, da dieser vermutlich vom Programmierer gemeint war.

Schreibzugriffe ohne folgende Leseoperation

Untersuchte Zugriffshistorien zeigen, dass oft Schreibphasen am Lebensende einer Datenstruktur auftreten und diese geschriebenen Elemente nicht mehr gelesen werden. Oftmals rührt diese Beobachtung aus Aufräummethoden her, die alle Elementeinträge auf NULL setzen. Durch die *Garbage Collection* wird dieses Aufräumen eigentlich überflüssig. Da die Garbage Collection trotzdem noch aufräumen muss, kann unter Umständen das Programm sogar verlangsamt werden. Dieses Muster wird dementsprechend gefunden, wenn viele Schreiboperationen an Stellen gefunden werden, die danach nicht mehr gelesen wurden.

Vorgeschlagen wird bei diesem Muster zu prüfen, ob die verantwortlichen schreibenden Zugriffe weggelassen werden können und diese gegebenenfalls zu entfernen.

5.5 Zusammenfassung

In diesem Kapitel wurde das Konzept beschrieben, mit dem Fingerzeige aus existierenden sequenziellen Programmen abgeleitet werden. Die Voruntersuchungen haben gezeigt, dass Regelmäßigkeiten in den Zugriffshistorien von Datenstrukturen wiederkehrend auftreten und sich damit Verhaltensmuster beschreiben lassen. Aus diesem Grund gliedert sich das Konzept in zwei Schritte. Als Erstes werden Zugriffshistorien erzeugt. Das bestehende Programm wird dazu instrumentiert und ausgeführt. Im zweiten Schritt wird in den gespeicherten Historien nach Mustern gesucht und daraus Fingerzeige abgeleitet. Dafür werden Zugriffsereignisse zunächst zu Phasen zusammengefasst. Ein Katalog von Fingerzeigen enthält Vorschriften, welche Phasen mit welchen Merkmalen vorkommen müssen, damit ein Fingerzeig ausgegeben werden kann. Fünf Fingerzeige wurden im Rahmen dieser Arbeit vorgeschlagen. Die genannten Ausprägungen der eingeführten Zugriffsarten, Phasencharakteristika und Fingerzeige sind erweiterbar.

6 Implementierung

Dieses Kapitel zeigt, wie das vorgeschlagene Konzept als Projekt in C# umgesetzt wurde. Kapitel 6.1 gibt zunächst einen Überblick über die Komponenten, die das Ziel der Erkennung von Parallelisierungspotenzial aus den Laufzeitverhalten objektorientierter Datenstrukturen umsetzten. Dabei werden die Pakete des Projektes, deren Beziehung untereinander und deren funktioneller Bezug zum Konzept beleuchtet. Danach folgt die Implementierung der konzeptionellen Schritte. Hierfür werden in den Kapiteln 6.2, 6.3 und 6.4 drei Implementierungsschritte (IS) definiert, die die Schritte aus den Kapiteln 0 und 5.4, sowie die Visualisierung aus Kapitel 4.2 technisch realisieren. Zur Veranschaulichung wird in Kapitel 6.2 ein Beispiel eingeführt, welches sich durch den in Kapitel 6.1 geschilderten Arbeitsablauf entwickelt und somit die Funktionsweise der Implementierung verdeutlicht.

6.1 Überblick

Zunächst wird ein Überblick über die Implementierung gegeben. Dazu zählen zum einen die Strukturierung der Pakete und zum anderen die Darstellung des Arbeitsablaufs auf derselben Granularitätsstufe.

Paketstruktur

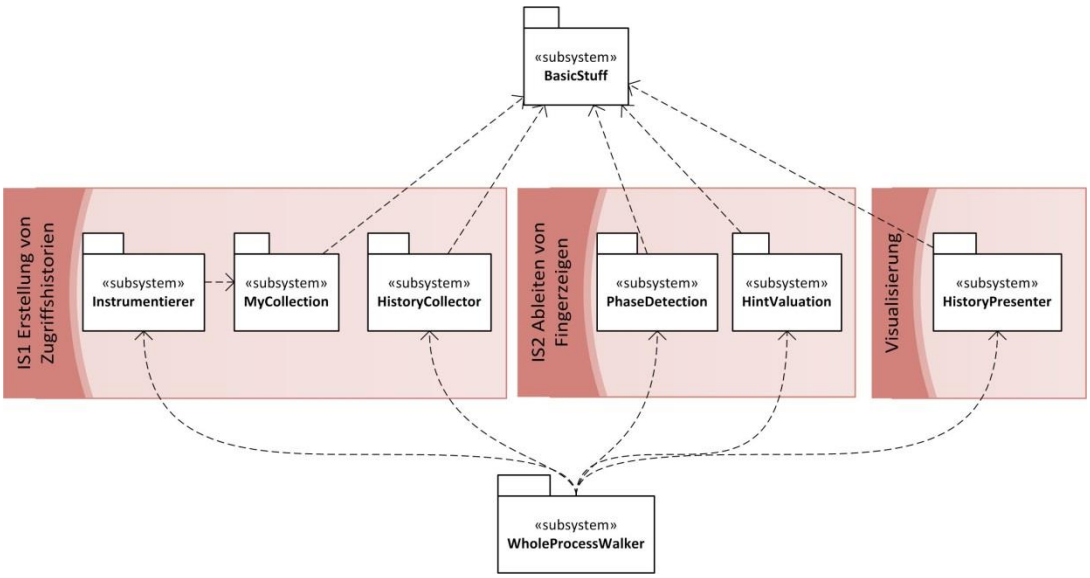

Abbildung 10: Abhängigkeiten der Pakete

Abbildung 10 zeigt die Paketabhängigkeiten aller relevanten Klassen der Implementierung. `BasicStuff` enthält die allgemein gebrauchten Klassen und Konstanten und dient damit als Grundlage für alle anderen Pakete. Folgende Klassen und Enumerationen werden dort definiert:

- Phase (Klasse: `Phase`)
- Protokollsammlung (Klasse: `Protokollsammlung`)
- Zugriffsereignis (Klasse: `Zugriffsereignis`)
- Zugriffsprotokoll (Klasse: `Zugriffsprotokoll`)
- Zugriffsart (Enumeration: `AccessType`)
- Phasencharakteristika (Enumeration: `Phasentyp`)

Die Definition von Konstanten, insbesondere der Interprozesskommunikation zur Mitteilung von Ereignissen an die Komponente, die die Zugriffe auswertet (siehe Nebenbedingung A1 und Kapitel 6.2.3), und einiger utilitärer Methoden vervollständigen den Inhalt dieses Pakets.

Das Paket `Instrumentierer` ist für die Instrumentierung zuständig, damit Zugriffe auf Datenstrukturen erfasst werden können (siehe Kapitel 5.3). Die Klassen im Paket `MyCollection` sind semantisch äquivalente Datenstrukturen zu denen der Standardbibliothek des .Net-Rahmenwerks. Wie in Kapitel 5.3 erläutert, wird das Entwurfsmuster „Stellvertreter“ eingesetzt. Die Stellvertreterklassen übernehmen die Protokollierung unter Zuhilfenahme der Klasse `HistorySender` (ebenfalls im Paket `MyCollection`), welche die Zugriffsereignisse aus dem Prozessraum sendet. Das Paket `HistoryCollector` ist das Gegenstück zum sendenden Teil des Protokollierungsmechanismus und stellt dementsprechend die Funktion des Empfangens der Zugriffsereignisse bereit. Zusammen setzen diese drei Pakete den Schritt S1 um.

Das Paket `HistoryPresenter` dient dem Anzeigen der Zugriffshistorien und realisiert somit A6.

Die Pakete `PhaseDetection` und `Fingerzeiggenerierung` implementieren die Phasenerkennung bzw. die Fingerzeigerkennung. Damit setzen sie Schritt S3 um.

Arbeitsablauf auf Paketebene

Das Paket `WholeProcessWalker` verknüpft nun die eingeführten Pakete und deren Funktionalität in einem Arbeitsablauf. Der Ablauf ist analog zu dem in Abbildung 6 dargestellten Prozess realisiert. Die einzelnen Schritte können aber auch manuell angestoßen und nachvollzogen werden. Zunächst wird dazu ein C#-Projekt durch den Instrumentierer instrumentiert. Es entsteht ein neues Projekt, welches die Stellvertreter aus MyCollection enthält. Die Ausführung dieses instrumentierten Projekts führt zu einem Senden der Zugriffe, welche von einem `HistoryCollector` empfangen werden. Dieser wird in einem neuen Prozess gestartet. Der `HistoryCollector` stellt die Zugriffshistorien zusammen und bündelt sie in einer Protokollsammlung. Die `PhaseDetection` ergänzt die Protokolle um die enthaltenen Phasen. Die Protokollsammlung wird anschließend an die `Fingerzeiggenerierung` weitergereicht. Als Ergebnis erhält man von dort eine Liste von Fingerzeigen, die ausgegeben werden können. Schließlich wird die Protokollsammlung an den `Presenter` übergeben, der ein neues Fenster mit der Visualisierung erzeugt.

Die folgenden Kapitel gehen genauer auf die Implementierung der genannten Funktionseinheiten ein und beleuchten deren Struktur und Kontrollfluss auf einer feineren Granularitätsstufe. Dabei wird der geschilderte Instrumentierungsprozess an einem veranschaulichenden Beispielquelltext durchlaufen.

6.2 IS1 Erstellung von Zugriffshistorien

Die Erzeugung von Zugriffshistorien beruht auf der Instrumentierung der Datenstrukturen im Quelltext mit den entsprechenden Stellvertretern und deren Protokollierungsmechanismen. Der Instrumentierungsvorgang wird in Kapitel 6.2.1 beschrieben. Kapitel 6.2.2 erläutert die Implementierung der dabei benötigten Stellvertreter. Der Protokollierungsmechanismus wird separat in Kapitel 6.2.3 beleuchtet. Das Zusammenspiel der bis dahin eingeführten Komponenten wird am Ende in 6.2.4 verdeutlicht.

6.2.1 Instrumentierung

Zur Instrumentierung wurde das Rahmenwerk Roslyn (Version: Juni 2012) benutzt. Es ermöglicht Modifikationen an Syntaxbäumen, wodurch strukturelles Arbeiten am Quelltext bewerkstelligt werden kann. Abbildung 17 im Anhang zeigt das Klassendiagramm des Instrumentierers. Die Schritte dieses Moduls werden im Folgenden chronologisch beschrieben, wobei unter „Projekt“ stets die zu untersuchende C#-Solution zu verstehen ist.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4
5  namespace BeispielNS {
6      class Program {
7
8          private List<int> l;
9
10         List<int> methode1(List<int> eingabe) {
11             List<string[]> liste = new List<string[]>();
12             for (int i = 0; i < 100; i++)
13                 liste.Add(new string[1]);
14             l = methode2(0).ToList();
15             while (liste.Count > 0) liste.RemoveAt(0);
16             return methode2(1).ToList();
17         }
18
19         int[] methode2(int x) {
20             int[] arr = { 1, 2, 3, x };
21             return new int[arr.Length];
22         }
23 }}
```

Quelltext 1: Zu instrumentierendes Beispielprogramm

Quelltext 1 zeigt das Beispielprogramm, an welchem die Transformationen schrittweise nachvollzogen werden. Zu jedem Schritt wird jeweils markiert, welche Änderungen diese im Quelltext bewirken. Das Programm besteht aus einer Klasse mit zwei Methoden, welche Eingabeparameter verlangen und Datenstrukturen als Ergebnis zurückliefern. Da das Beispiel veranschaulichender Natur ist, enthält es keinen sinnvollen Algorithmus.

Schritt 1: Projekt kopieren

In einem ersten Schritt kopiert der Instrumentierer zunächst die Dateien des zu überwachenden Projektes in einen temporären Ordner (siehe Kapitel 6.6). Alle Änderungen werden dort stattfinden, sodass das ursprüngliche Projekt unberührt bleibt.

Als Zweites macht er die Verweise/Referenzen zur Klassenbibliothek `MyCollection` dem Projekt bekannt, damit der Code kompilierbar bleibt und semantische Anfragen auf den Syntaxbäumen kommender Schritte zum Beispiel Typinformationen in diesen Ressourcen ermitteln können. Da diese in den XML-Dateien der Projektkonfigurationen eingetragen sind (siehe Kapitel 2.2.2), werden sie mithilfe eines XML-Prozessors eingefügt. Die hierfür verantwortliche Methode heißt `insertMyCollectionReferences`.

Die nächsten Schritte erfolgen auf einem durch Roslyn erzeugten Syntaxbaum mithilfe der darin zur Verfügung gestellten API. Das Konzept dazu baut auf dem Besucher-Muster aus der Softwaretechnik auf. Syntaxbäume sind nicht veränderbar. Transformationen eines Besuchers auf dem Syntaxbaum führen zur Generierung eines neuen Syntaxbaumes, welcher die gewünschten Änderungen enthält. Dies bedeutet, dass alle folgenden Schritte von Besucher-Klassen

durchgeführt werden, welche die abstrakte Klasse `SyntaxRewriter` implementieren. Die Erzeugung und das Starten der Besucher-Klassen erfolgen in den Methoden `TransformSolutionWith` und `TransformSyntaxRoot`.

Schritt 2: using-Direktiven einfügen

Der nächste Schritt besteht darin, die `using`-Anweisungen zu ergänzen, sodass der Namensraum `MyCollectionNS` aus dem Projekt `MyCollection` im Code verfügbar ist. Die Besucher-Klasse `MyCollectionUsingInjectionRewriter` durchläuft dazu alle `using`-Anweisungsblöcke und fügt die Direktive zur Benutzung des neuen Namensraums hinzu. Im Beispielquelltext sieht das Ergebnis wie folgt aus:

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using MyCollectionNS;
5
6  namespace BeispielNS {
   //...
```

Quelltext 2: Änderungen durch Schritt 2

Schritt 3: Arrays ersetzen

Der Besucher `ArrayDeclarationAndCreationFixingRewriter` normalisiert die kombinierten Deklarations- und Initialisierungsanweisungen von Arrays, sodass der Elementdatentyp auch auf der rechten Seite der Anweisung (im Array-Initialisierer) auftaucht. Beispielsweise entsteht aus der Anweisung

```
int[] arr={1,2,3};
```

die normalisierte Anweisung

```
int[] arr=new int[]{1,2,3};.
```

Diese Normalisierung ist nötig, damit syntaktische Korrektheit nach Ersetzung der Arrays gewährleistet ist, denn der implizite Array-Initialisierer (`{1,2,3}`) kann nicht einer Variablen vom Typ `MyArray` zugewiesen werden.

Dem Kapitel 5.3 entsprechend werden schließlich die Deklarationen von Arrays so geändert, dass der Datentyp der Variablen danach `MyArray<T>` lautet, wobei der generische Parameter dem alten Elementtyp entspricht. Aus

```
int[] arr;
```

wird beispielsweise

```
MyArray<int> arr;.
```

Die Initialisierungen bzw. die Wertzuweisungen müssen nicht mehr transformiert werden, da `MyArray` eine implizite Typkonvertierung enthält. Wenn im instrumentierten Quelltext ein Array einer Variablen vom Typ `MyArray` zugewiesen wird, dann ruft die Laufzeitumgebung die Konvertierungsoperation von `MyArray` auf.

Der Konvertierungsoperator erzeugt daraufhin eine `MyArray`-Instanz aus den Daten des zu konvertierenden Arrays. Dass dabei eine neue Instanz von `MyArray` erzeugt wird, ist deswegen unproblematisch, da durch die ursprüngliche Anweisung ebenfalls ein neues Array zugewiesen worden wäre. Damit spiegeln die unterschiedlichen Zugriffshistorien hinterher die Verwendung neuer Arrays wider.

```
1  //...
2          private List<int> l;
3
4          List<int> methode1(List<int> eingabe) {
5              List<MyArray<string>> liste = new List<MyArray<string>>();
6              for (int i = 0; i < 100; i++)
7                  liste.Add(new string[1]);
8              l = methode2(0).ToList();
9              while (liste.Count > 0) liste.RemoveAt(0);
10             return methode2(1).ToList();
11         }
12
13         MyArray<int> methode2(int x) {
14             MyArray<int> arr = new int[]{ 1, 2, 3, x };
15             return new int[arr.Length];
16         }
17 }}
```

Quelltext 3: Änderungen durch Schritt 3

In Quelltext 3 sieht man (gelb hervorgehoben), wie alle Typisierungen von Arrays zu `MyArray` geändert werden. Zudem wird in Zeile 12 die Array-Instanziierung normalisiert. Die Zuweisungen von Arrays zu Speicherbereichen vom Typ `MyArray` in den Zeilen 6, 12 und 13 werden beibehalten. An diesen Stellen wird zur Laufzeit die implizite Konvertierungsoperation aufgerufen.

Schritt 4: Listen ersetzen

Der Besucher `GenericsChangeRewriter` kümmert sich um die Transformation der Datenstruktur `List`. Dazu ersetzt er im Syntaxbaum alle generischen Typknoten mit dem Namen `List`, durch einen `MyList`-Typknoten. Deklarationen, Definitionen, Parametertypen und Rückgabetypen von Methoden werden dabei gleichermaßen verarbeitet. Es wird dabei geprüft, ob es sich um die Datenstruktur der Standardbibliothek handelt und nicht etwa um eine selbst programmierte Klasse, die unglücklicherweise ebenfalls „List“ heißt.

Quelltext 4 zeigt, wie alle Typisierungen mit `List` durch den Typ `MyList` ersetzt werden. Die Variablendefinitionen in den Zeilen 2 und 5, der Parametertyp in Zeile 4 und die Methodensignatur in Zeile 4 werden dabei gleichermaßen abgearbeitet.

```
1   //...
2       private MyList<int> l;
3
4       MyList<int> methode1(MyList<int> eingabe) {
5           MyList<MyArray<string>> liste = new MyList<MyArray<string>>();
6           for (int i = 0; i < 100; i++)
7               liste.Add(new string[1]);
8           l = methode2(0).ToList();
9           while (liste.Count > 0) liste.RemoveAt(0);
10          return methode2(1).ToList();
11      }
12
13      MyArray<int> methode2(int x) {
14          MyArray<int> arr = new int[]{ 1, 2, 3, x };
15          return new int[arr.Length];
16      }
17  }}
```

Quelltext 4: Änderungen durch Schritt 4

Schritt 5: Zuweisungen von Rückgabewerten nicht instrumentierbarer Methoden

Der `SystemMethodTypeConvertRewriter` übernimmt folgenden Sonderfall: Programmierer haben die Möglichkeit, einer Variablen den Rückgabewert einer Methode als eine neue Instanz zuzuordnen. Da die Rückgabetypen von eigenen Methoden bereits geändert wurden, besteht zunächst kein Konflikt zwischen Rückgabetyp und Variablentyp. Allerdings stellt die Laufzeitumgebung von C# Methoden zur Verfügung, welche `List` zurückgeben. Diese können nicht instrumentiert werden, da der Quelltext nicht verfügbar ist. In diesem Fall kann die Typsicherheit der Zuweisung nicht über Methodenanpassung erfolgen. Aus diesem Grund wird der Rückgabewert einer solchen Methode dem Konstruktor einer neuen `MyList`-Instanz als Eingabeparameter übergeben. Beispielsweise wird aus

```
List<int> foo=bar.ToList();
```

die Anweisung

```
MyList<int> foo=new MyList<int>(bar.ToList());.
```

Auf diese Art werden alle Variablen vom Typ `List` zu `MyList` transformiert und dabei sichergestellt, dass Variablenzuweisungen syntaktisch korrekt bleiben.

Es stellt sich an dieser Stelle die Frage, warum die Typkonvertierung bei Variablenzuweisungen nicht ebenfalls implizit stattfinden kann, wie es bei den Arrays praktiziert wird. Aufgrund der Vererbungsbeziehung ist dies für den Datentyp `List` nicht möglich. Der Grund sind die Schnittstellen. `List` implementiert eine Reihe von Schnittstellen und abstrakter Klassen, die `MyList` ebenfalls implementieren muss. Um diese zusätzlichen Methoden anbieten zu können, erbt `MyList` selbst von `List`. Diese Vererbungsbeziehung verhindert, dass es eine implizite Konvertierung von `List` nach `MyList` geben kann.

```
1  //...
2      private MyList<int> l;
3
4      MyList<int> methode1(MyList<int> eingabe) {
5          MyList<MyArray<string>> liste = new MyList<MyArray<string>>();
6          for (int i = 0; i < 100; i++)
7              liste.Add(new string[1]);
8          l = new MyList<int>(methode2(0).ToList());
9          while (liste.Count > 0) liste.RemoveAt(0);
10         return methode2(1).ToList();
11     }
12
13     MyArray<int> methode2(int x) {
14         MyArray<int> arr = new int[]{ 1, 2, 3, x };
15         return new int[arr.Length];
16     }
17 }}
```

Quelltext 5: Änderungen durch Schritt 5

Die Zuweisung in Zeile 7 im Quelltext 5 enthält einen Methodenaufruf, der nicht instrumentiert werden kann. Aus diesem Grund wird dort die Erzeugung einer Instanz vom Typ `MyList` injiziert, um Typsicherheit herzustellen.

Schritt 6: Rückgabetypen anpassen

Die Rückgabetypen von Methoden wurden zwar bereits durch die oben genannten Besucher angepasst, allerdings bedarf es einer gesonderten Behandlung der Rücksprunganweisung am Methodenende (`return`). Analog zum zweiten Teil des vorangegangenen Abschnitts „Listen ersetzen“, sind Rücksprunganweisungen mit Methodenaufrufen erst problematisch, wenn nicht beide Methoden den gleichen Typ zurückgeben. Oftmals werden allerdings Methoden aufgerufen, die `List` zurückgeben und nicht instrumentiert werden können. In solchen Fällen transformiert der Instrumentierer beispielsweise

```
return bar.ToList();
```

in die typsichere Anweisung

```
return new MyList<int>(bar.ToList());.
```

Zeile 8 in Quelltext 6 enthält eine Rücksprunganweisung mit dem Rückgabewert einer Methode, die nicht instrumentierbar ist. Aus diesem Grund wird dort ebenfalls die Erzeugung einer Instanz vom Typ `MyList` eingefügt.

```
1   //...
2       private MyList<int> l;
3
4       MyList<int> methode1(MyList<int> eingabe) {
5           MyList<MyArray<string>> liste = new MyList<MyArray<string>>();
6           for (int i = 0; i < 100; i++)
7               liste.Add(new string[1]);
8           l = new MyList<int>(methode2(0).ToList());
9           while (liste.Count > 0) liste.RemoveAt(0);
10          return new MyList<int>(methode2(1).ToList());
11      }
12
13      MyArray<int> methode2(int x) {
14          MyArray<int> arr = new int[]{ 1, 2, 3, x };
15          return new int[arr.Length];
16      }
17  }}
```

Quelltext 6: Letzte Änderungen durch Schritt 6

6.2.2 Stellvertreter für List und Array

Das Paket `MyCollection` stellt im Namensraum `MyCollectionNS` die Stellvertreter für `List` und Arrays bereit. Diese heißen `MyList` und `MyArray`. Sie verwenden die statische Klasse `HistorySender`, welche im selben Paket zu finden ist, um Aufrufe ihrer Methoden zu vermerken. Sich auf diese beiden Datentypen zu beschränken ist sinnvoll, wie die Voruntersuchung in Kapitel 4.1 nahelegt. Im Folgenden werden die beiden Stellvertreter näher beleuchtet. Der `HistorySender` wird in 6.2.3 gesondert beschrieben.

Die beiden Stellvertreter haben gemeinsame Grundfunktionalitäten. Jede Instanz speichert beispielsweise ihre eigene Kennung. Die Kennung erhält sie vom `HistorySender` bei der Initialisierung, wie am Beispiel von `MyList` in Abbildung 11 dargestellt. Die Konstruktoren rufen dazu ihre Initialisierungsmethode `initMyStuff` auf. Diese Methode ermittelt mit Hilfe von *reflection*, welche Codezeile in welcher Methode des Ursprungsprogramms die Initialisierung ausgelöst hat. Mit diesen Informationen meldet sich die Instanz über die Methode `recordMyExistence` beim `HistorySender` an und bekommt damit die programmweit eindeutige Kennung zugewiesen. Mit dieser Kennung werden später die Zugriffsereignisse propagiert und zugeordnet. Uhren zur Vergabe der Zeitstempel von Zugriffsereignissen werden nicht von den Datenstrukturinstanzen gehalten. Der `HistorySender` stellt programmweit eine Uhr zur Verfügung. Damit können im Nachhinein die Zugriffe von unterschiedlichen Instanzen in eine zeitliche Beziehung gebracht werden.

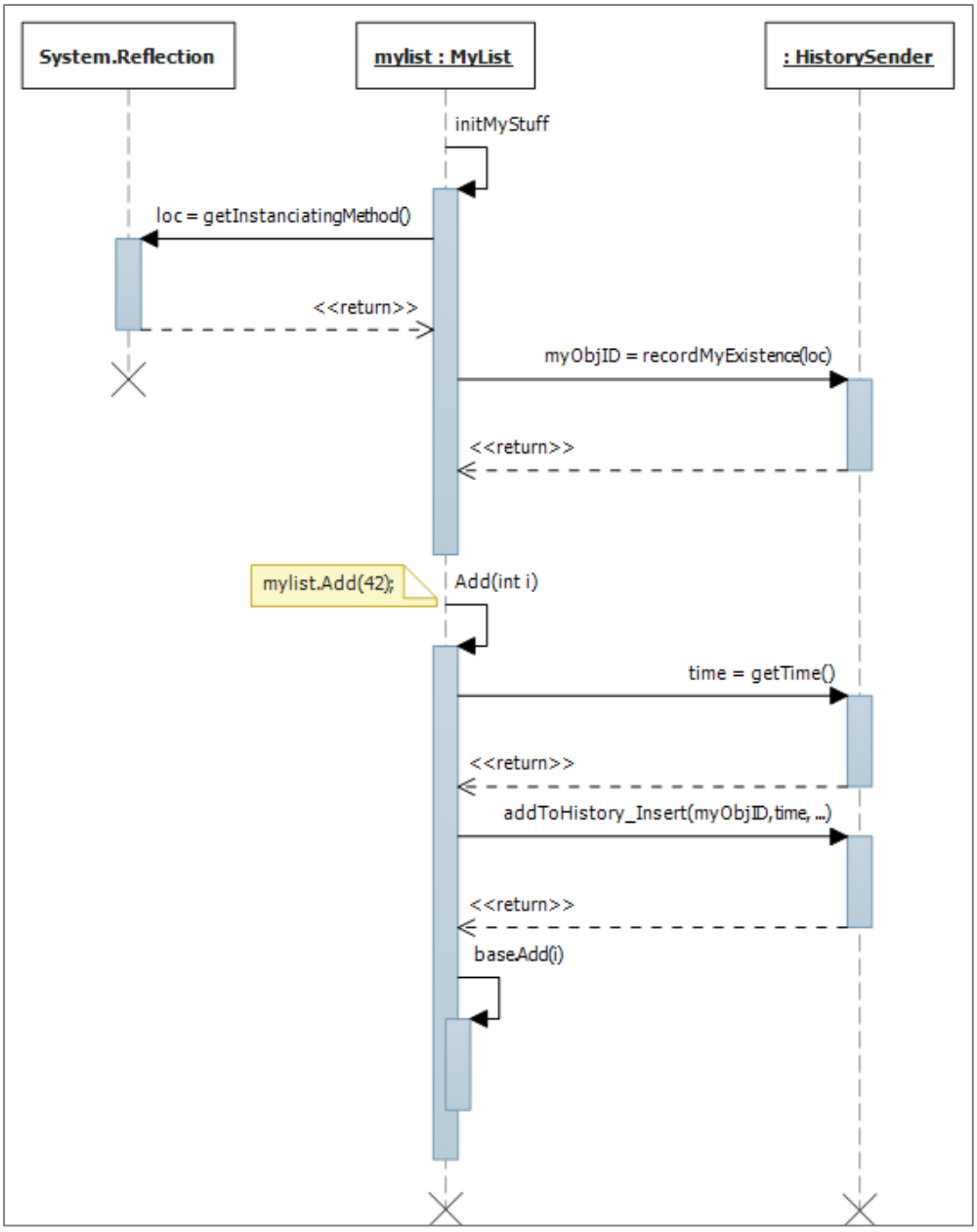

Abbildung 11: Instanziierung und Protokollierung der Zugriffe am Beispiel von MyList

MyList

`MyList` implementiert folgende Schnittstellen: `List<T>`, `IList<T>`, `ICollection<T>`, `IEnumerable<T>`, `IList`, `ICollection` und `IEnumerable`. Bei einem Methodenaufrufe wird dies dem `HistorySender` mitgeteilt und danach der entsprechenden Oberklassen von `List` weitergeleitet (siehe Abbildung 11). Je nach Zugriffsart (siehe Kapitel 5.2) werden dafür die Methoden, die mit dem Suffix „addToHistory_“ beginnen, verwendet. Neben allen Methoden stellt `MyList` auch einen Enumerator zur Verfügung, wie es `IEnumerable` fordert. Dieser imitiert den Enumerator der `List`, sodass Methodenaufrufe implizit auf die `MyList`-Instanz weitergeleitet werden. Somit wird die Verwendung eines Enumerators auf einer Variablen vom Typ `MyList` ebenfalls protokolliert.

MyArray

Im Gegensatz zu `MyList` hält `MyArray` ein öffentliches Array vor, welches letztlich gekapselt wird. Der Grund hierfür ist zum einen, dass C# das Ableiten von `Array` nicht erlaubt und daher die Aufrufe nicht an Oberklassen weitergereicht werden können. Zum anderen werden Arrays als Referenzparameter übergeben. In solchen Fällen muss eine Array-Instanz vorliegen. Die Verwendung einer Instanz von `MyArray` an diesen Stellen führt zu einem Typfehler. Für diesen Zweck bietet `MyArray` die Eigenschaft `T[] origArray` an, welche den direkten Zugriff auf das interne Array protokolliert. Es gibt aber zudem auch Fälle, bei denen die Übergabe dieser Eigenschaft nicht ausreicht. Die öffentliche Variable `myArray` kann dafür verwendet werden. Dies sollte allerdings so weit wie möglich vermieden werden, da sich dann die Operationen auf dem Array der Beobachtung entziehen und keine Protokollierung stattfindet.

Ein weiterer Unterschied zu `MyList` ist, dass `MyArray` eine implizite Typkonvertierung, von `Array` zu `MyArray` und umgekehrt, implementiert. Wie in 6.2.1 beschrieben, führt dies zu einer Vereinfachung der Instrumentierung von Arrays.

Die Implementierung eines Enumerators auf MyArray schließt auch diesen Stellvertreter ab, wodurch ein umfassend kompatibler Ersatz von Arrays durch `MyArray` gewährleistet wird.

6.2.3 Senden und Empfangen der Zugriffsereignisse

Für die Übertragung der Zugriffsereignisse mittels Interprozesskommunikation sind die statischen Klassen `HistorySender` im Paket `MyCollection` und `HistoryCollector` im Paket `HistoryCollector` verantwortlich. Der `HistorySender` wird automatisch von den Stellvertretern initialisiert. Der `HistoryCollector` muss zusätzlich mithilfe der Methode `HistoryCollector.init()` instanziiert werden. Beide Seiten des Kommunikationsmechanismus werden im Folgenden näher erläutert.

HistorySender

Die Zugriffsereignisse der Stellvertreter werden vom `HistorySender` gesammelt und stapelweise versendet. Der im folgenden beschriebene Mechanismus trägt dazu durch zwei Implementierungsaspekte der Nebenbedingung A1 (die Laufzeit des Programms soll wenig beeinflusst werden) Rechenschaft.

Der erste Aspekt bezieht sich auf den Laufzeitaufwand durch die Protokollierung. Damit der Mehraufwand dort gering gehalten wird, werden die zu sendenden Daten lediglich in einem Puffer zwischengespeichert.

Der zweite Aspekt beläuft sich auf die Sequenzialisierung paralleler Programme durch das Senden. Bei parallelen Programmen werden instrumentierte Datenstrukturen nebenläufig benutzt. Dies führt dazu, dass mehrere Fäden gleichzeitig Protokollierungsdaten senden wollen. Aus diesem Grund muss hier durch Sperrmechanismen sichergestellt werden, dass stets nur ein Faden auf den Sendekanal zugreift. Dies bedeutet aber, dass die protokollierende Instrumentie-

rung das zu untersuchende Programm künstlich sequenzialisiert. Um das Sperren so gering wie möglich zu halten, wird ein Mechanismus benutzt, der je Faden eine Sperre zur Synchronisierung benutzt, sodass Wartezeiten an Sperren praktisch entfallen. Zum Einsatz kommen dazu mehrere Puffer. Ein Puffer ist dabei einer Datenstrukturinstanz und einer Fadenkennung zugeordnet. Greift ein Faden zum ersten Mal auf eine Datenstruktur zu, so wird für diesen ein neuer Puffer (bezüglich der Datenstrukturinstanz) erzeugt. Dieser einmalige Schritt muss durch eine Sperre geschützt werden. Sobald der Puffer erzeugt wurde, kann der jeweilige Faden dort hineinschreiben, ohne dass ein anderer Faden (aus dem untersuchten Programm) intervenieren wird. Zwei Programmfäden werden infolgedessen niemals in einen gemeinsamen Puffer schreiben. Der Puffer muss auf der anderen Seite natürlich gelesen und sein Inhalt gesendet werden. Dazu gibt es einen eigenen Faden, der die Puffer abarbeitet und die dort zwischengespeicherten Daten verschickt. Dieser Faden (`SenderThread`) liest dazu bis zum aktuell vorletzten Element des Puffers. Dadurch kann keine Wettlaufsituation entstehen, weshalb das Lesen nicht synchronisiert zu werden braucht. Am Schluss löscht er die gelesenen Elemente.

Damit sich ein zu testendes Programm korrekt beenden kann, ohne auf einen unnötig laufenden `SenderThread` zu warten, beendet sich dieser immer dann von selbst, wenn keine Puffer mehr abzuarbeiten sind. Die Information, ob ein Puffer gelöscht werden darf, erhält er dabei von den Stellvertreterinstanzen. Deren Destruktoren registrieren in einer separaten Datenstruktur, dass ihre Lebenszeit zu Ende ist. Deren Puffer werden daraufhin entfernt. Meldet eine neue Stellvertreterinstanz ihre Erzeugung, prüft sie zusätzlich, ob ein `SenderThread` läuft. Ist dies nicht der Fall, startet sie eine neue Fadeninstanz. Auf diese Art ist gewährleistet, dass der sendende Faden nur solange läuft, wie von den noch lebenden Datenstrukturen benötigt und somit die Beendigung des zu untersuchenden Programms nicht blockiert wird.

HistoryCollector

Diese Klasse ist das Gegenstück zum `HistorySender` und dient dementsprechend zum Empfangen der Zugriffsereignisse. Der dedizierte Faden (`CollectorThreadWork`) ruft dazu zeilenweise den Nachrichtenkanal ab. Ein Identifikationszeichen gibt an, ob es sich um eine Stellvertreterinstanziierung handelt oder um einen Datenstrukturzugriff. Bei einer Instanziierung werden die neuen Einträge in den Buch führenden Datenstrukturen angelegt. Die Zugriffsereignisse werden dort entsprechend einsortiert. Wird der Nachrichtenkanal geschlossen, ist die Arbeit des Fadens `CollectorThreadWork` beendet. Die gesammelten Zugriffshistorien liegen statisch im `HistoryCollector` zur Verfügung und können über `getAllProtocols` abgerufen werden. Diese Methode wartet gegebenenfalls auf die Beendigung des empfangenden Fadens, da in einem solchen Fall die Übertragung noch nicht abgeschlossen ist.

6.2.4 Zusammenfassend: Interaktion der Komponenten

Nachdem die einzelnen Akteure für die Erstellung von Zugriffsereignissen vorgestellt wurden, soll nun abschließend deren Zusammenspiel betrachtet werden.

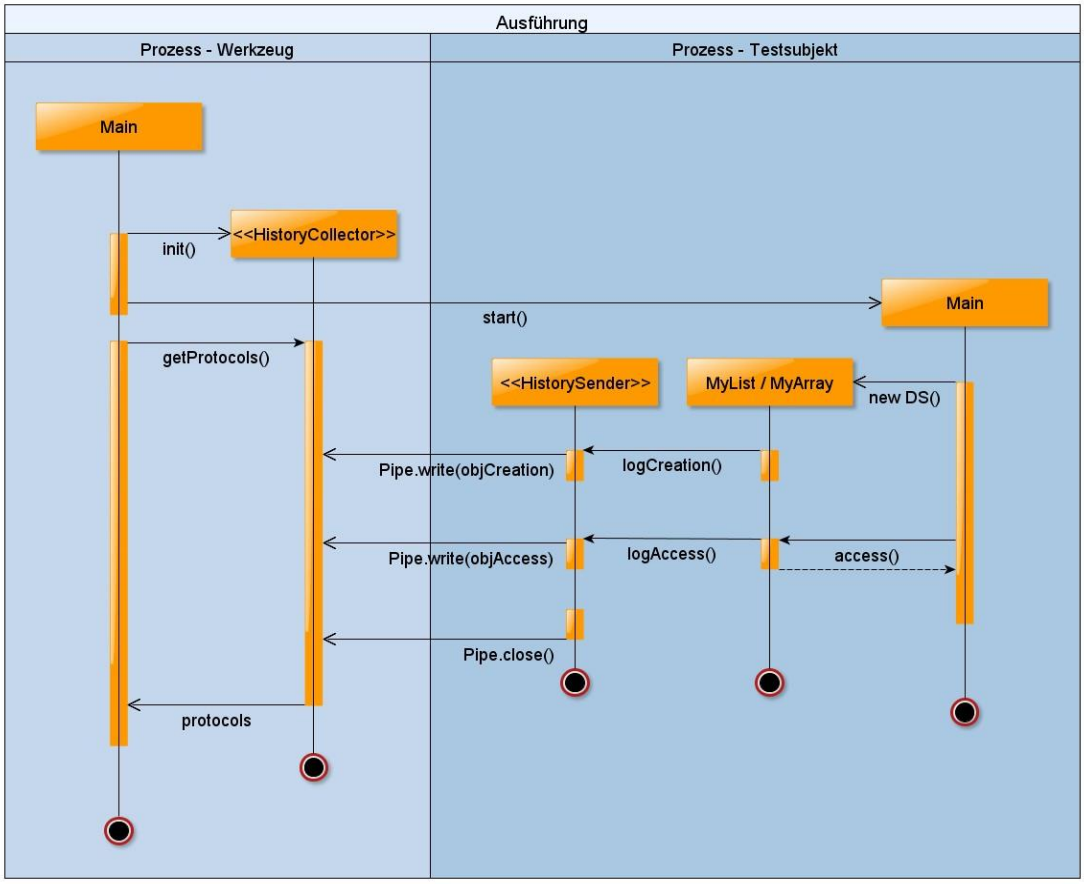

Abbildung 12: Sequenzdiagramm zur Erstellung der Zugriffshistorien

Abbildung 12 zeigt ein Sequenzdiagramm, das zur Veranschaulichung der Interaktion dient. Man sieht zunächst die beiden Prozessräume, welche als Schwimmbahnen in unterschiedlichen Blautönen markiert sind. Das Werkzeug startet als Erstes den `HistoryCollector` und danach das zu untersuchende Programm (Testsubjekt). Anschließend fordert es die Zugriffshistorien an, was dazu führt, dass das Werkzeug auf die Beendigung des Testsubjektes warten muss. Sobald das Testsubjekt einen Stellvertreter instanziiert, wird der `HistorySender` erzeugt. Die Information, dass ein neues Objekt instanziiert wurde, wird sofort dem `HistorySender` gesendet. Auf gleiche Weise wird mit den Datenstrukturzugriffen verfahren. Der `HistorySender` meldet sich nach Beendigung des Testsubjektes vom Kommunikationskanal ab. Dieses Ereignis registriert der `HistoryCollector` und beendet sich ebenfalls. Erst danach können die Protokolle der Hauptmethode des Werkzeugs übergeben werden. Die weiteren Schritte unterliegen den anderen Komponenten der folgenden Kapitel.

6.3 IS2 Ableiten von Fingerzeigen

Wie im Konzeptkapitel 5.4 geschildert, unterteilt sich das Ableiten von Fingerzeigen in zwei Schritte. Zum einen die Phasenerkennung, welche die Phasen innerhalb der Historien ermittelt und zum anderen die Fingerzeigerkennung an sich, welche die handlungsempfehlenden Schlüsse aus den Phasen zieht. Die Phasenerkennung wird im Namensraum `PhaseDetectionNS` des Pakets `PhaseDetection` definiert. Die eigentliche Fingerzeigerkennung wird im Namensraum `FingerzeiggenerierungNS` des Pakets `Fingerzeiggenerierung` implementiert. Die Umsetzung der beiden Funktionalitäten wird im Folgenden detailliert beleuchtet.

6.3.1 Phasenerkennung

Das Konzept der Erkennung einer Phase ist durch die Schnittstelle mit dem Namen `PhaseDetector` realisiert. Jedes Phasencharakteristikum wird durch eine eigene Klasse implementiert, welche dieses erkennt und von `PhaseDetector` erbt. Es hat sich herausgestellt, dass sich alle im Konzept angedachten Phasen mit einem Bildzeilenalgorithmus erkennen lassen. Aus diesem Grund gibt es die abstrakte Klasse `Detector_TristateTemplate`. Dort wird der Bildzeilenalgorithmus implementiert, wobei die phasenspezifischen Merkmalsabfragen in abstrakte Methoden ausgelagert sind. Alle Phasenerkenner erben von `Detector_TristateTemplate` und realisieren die spezifischen Methoden, sodass sie ihrem Phasencharakteristikum entsprechen. Ehe ein Beispiel gegeben werden kann, wird jedoch zunächst der Bildzeilenalgorithmus erklärt.

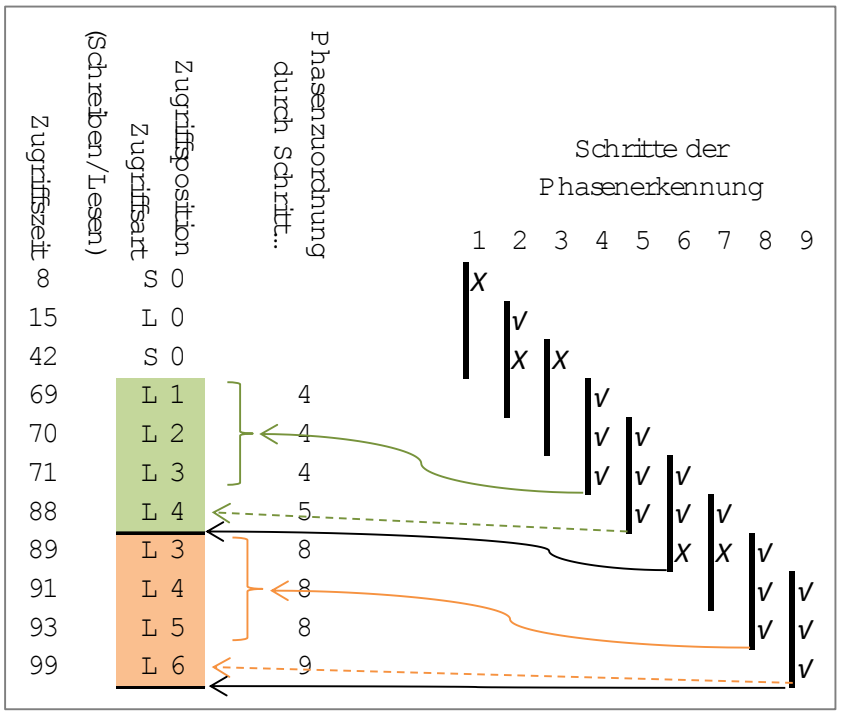

Abbildung 13: Ablauf der Erkennung von Lineares-Lesen-Vorwärts Phasen

Der Ablauf der Phasenerkennung ist zur Verdeutlichung in Abbildung 13 dargestellt und findet wie folgt statt: Ein Fenster von drei benachbarten Zugriffsereignissen der gleichen Zugriffsart wird über die zeitlich geordnete Historie geschoben. Das

Wort „benachbart“ ist wie in Kapitel 5.4 spezifiziert, da je nach Phasencharakteristikum ein bestimmter „Nachbarabstand“ erlaubt ist. Sobald drei gleichartige Zugriffe gefunden wurden (Schritt 4), wird geprüft, ob das Phasencharakteristikum stimmt. Beispielsweise müssen bei einer linear aufsteigenden Phase die Indizes aufsteigen. Ist das Phasencharakteristikum auf die drei Zugriffsereignisse anwendbar, wird eine neue Phase erstellt (farbiger durchgehender Pfeil), die diese drei Ereignisse umfasst. Solange das Charakteristikum bei nachfolgenden Zugriffsereignissen kontinuierlich gültig ist, wird die Phase erweitert (Schritt 5, farbig gestrichelter Pfeil). Sobald die Bedingungen nicht mehr dem Charakteristikum entsprechen, wird die aktuelle Phase komplettiert (Schritt 6, schwarzer kontinuierlicher Pfeil). Wie schon skizziert, beschränken sich die Bedingungen der Phasencharakteristika auf die Art und die Indizes dreier zeitlich geordneter Zugriffsereignisse. Aus diesem Grund beziehen sich die abstrakten Methoden des `Detector_TristateTemplate` auf genau diese beiden Merkmale.

Folgendes Beispiel soll die Beschreibungen der Phasen erkennenden Klassen verdeutlichen. Die erkennende Klasse von Phasen des Phasencharakteristikums `Einfügen-Vorn` heißt `Detector_EinfügenVorn`, erbt von `Detector_TristateTemplate` und implementiert die abstrakten Methoden. Die Arten der Zugriffe müssen dabei Einfügeoperationen und die Indizes alle gleich Null sein. Dem Konstruktor wird zusätzlich der Schwellenwert übergeben, der angibt, bis zu welchem Abstand zwei Zugriffsereignisse noch als benachbart gelten. Ein Schwellenwert von 1 bedeutet dabei direkte Nachbarschaft, wie es konzeptionell beim Charakteristikum `Einfügen-Vorn` erwartet wird.

Auf diese Weise lassen sich auch alle anderen Phasencharakteristika ausreichend im Quelltext kodieren. Eine Gesamtliste aller in dieser Arbeit definierten Charakteristika befindet sich in Kapitel 5.4. Wie eingangs beschrieben, gibt es eine koordinierende Instanz, welche in der Klasse `PhaseDetective` umgesetzt ist. Diese bekommt eine Protokollsammlung übergeben, auf welcher sie die Phasensuche ausgeführt. Die Enumeration der Phasencharakteristika wird dabei durchlaufen und eine Instanz des entsprechenden Phasenerkenners erzeugt und ausgeführt. Die Fabrikmethode `getCorrespDetector` liefert dabei zu einem gewünschten Phasentyp den entsprechenden Phasenerkenner. Jede definierte Phase muss einem Erkenner zugeordnet werden können. Da die Phasenerkenner nur lesend auf den Historien arbeiten und untereinander unabhängig sind, laufen diese parallel zueinander.

Schließlich kann der Schwellenwert `minPhaseSize` angegeben werden. Dieser besagt, ab wie viel enthaltenen Zugriffsereignissen eine Phase als solche bestehen bleiben darf. Das verhindert, dass bei unregelmäßigen Zugriffen sehr viele kleine und dadurch aussagenlose Phasen erkannt werden. Solche informationsarme Phasen werden am Ende wieder gelöscht.

6.3.2 Fingerzeigerkennung

Die Umsetzung der Fingerzeigerkennung gestaltet sich ähnlich zur Phasenerkennung. Zunächst gibt es eine abstrakte Klasse namens `AbstractGenerator`,

welche das Konzept des allgemeinen Fingerzeigerkenners repräsentiert, Schwellenwerte deklariert und die von Fingerzeigerkennern zu implementierende Methode `gibDeineFingerzeige` definiert. Alle Fingerzeigerkenner erben hiervon. Ihre Aufgabe besteht darin, zu einer übergebenen Historie die erkannten Fingerzeige als Liste gesammelt zurückzugeben. Dabei ignorieren sie die Information des Instanziierungsortes, da diese an späterer Stelle zugewiesen wird. Im Wesentlichen prüfen die Erkenner, ob eine Historie bestimmte Phasen enthält oder der Anteil von Phasen einen definierten Schwellenwert überschreitet.

Weiterhin gibt es die koordinierende Klasse `FingerzeigKoordinator`. Diese Klasse führt eine Liste über Instanzen von Fingerzeigerkennern und stößt deren Arbeit an. Dabei durchläuft sie die Historien der Protokollsammlung und übergibt sie den Erkennern. In den zurückgegebenen Fingerzeiglisten wird die Information des Instanziierungsortes ergänzt und die Ordnungskennzahl berechnet. Zudem werden abschließend doppelte Fingerzeige eines Instanziierungsortes entfernt. Doppelte Fingerzeige können entstehen, wenn es mehrere Instanzen einer Datenstruktur gibt, deren Historien gleichermaßen die Bedingungen eines Fingerzeigs erfüllen. Die bereinigte Liste der Fingerzeige wird schließlich zurückgegeben.

6.4 I3 Visualisierung

Die grafische Benutzeroberfläche im Paket `HistoryPresenter` implementiert die Darstellung von Historien, wie sie in Kapitel 4.2 dargelegt ist. Die Instanziierung eines neuen Objektes vom Typ `Presenter` führt dabei zur Erzeugung eines neuen Fensters. Als Parameter wird hierbei eine Protokollsammlung erwartet. Die Klasse `Form1` stellt schließlich den eigentlichen Quelltext für das Aussehen und die Funktionalität des Fensters bereit. Deren Konstruktor erzeugt aus der Protokollsammlung alle darzustellenden Elemente. Das *charting-framework* von Microsoft wird dazu verwendet. Damit der Benutzer ohne Wartezeiten für Berechnungen durch alle Historien navigieren kann, werden alle Historien zur Konstruktionszeit erstellt. Dies hat allerdings den Nachteil, dass die Erzeugung der Oberfläche lange dauert und sehr viel Arbeitsspeicher beanspruchen kann. Damit speicherintensive Visualisierungen den Arbeitsspeicher nicht überlasten, gibt es die Möglichkeit, die Anzahl von Historien je Instanziierungsort und die Anzahl von Datenpunkten in Historien zu beschränken. Die Variablen `maxParts` bzw. `maxLines` sind dafür verantwortlich. Zur Darstellung werden sogenannte `chartArea`-Objekte erzeugt. Diese stellen ein zweidimensionales kartesisches Koordinatensystem zur Verfügung. Dort können alle Datenpunkte hinzugefügt werden. Kurze Historien werden dabei als Balken, lange Historien als kleine quadratische Punkte visualisiert. Balken sind zwar zugänglicher, allerdings werden Punkte deutlich schneller dargestellt, was sich bei langen Historien dramatisch bemerkbar macht.

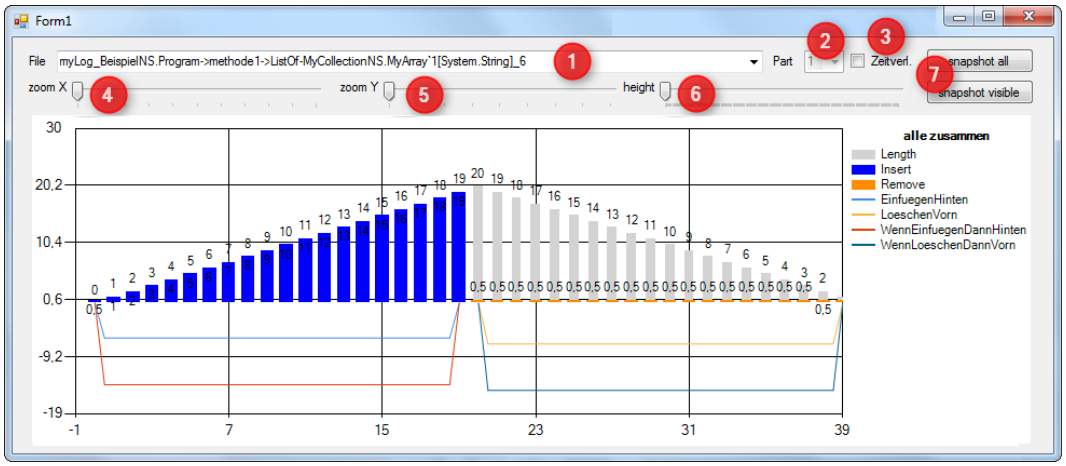

Abbildung 14: Bildschirmfoto der Visualisierung

Abbildung 14 zeigt die Visualisierung des Beispielprogramms (Quelltext 1) aus Kapitel 6.2.1. Zu sehen sind die zwanzig Einfüge- und Löschoperationen der Variable `liste`. Die Legenden zeigen die Bedeutung des Dargestellten im stets gleichen Farbschema an. Dabei stehen die Rechtecke für Zugriffsarten der Balken, die Linien repräsentieren Phasencharakteristika. Die Phasen werden unterhalb der Nulllinie als entsprechend lange, klammernde Striche gezeigt. Aus diesem Grund wird die Skalierung der y-Achse negativ.

Folgende Bedienelemente, welche durchnummeriert beschriftet sind, sind auf der Oberfläche zu sehen. Unter (1) kann man den Instanziierungsort auswählen, zu welchem die Historien anschließend unter (2) aufgelistet werden. Sobald mehrere Instanzen am jeweiligen Instanziierungsort erzeugt wurden, würden diese dort wählbar sein. Mit dem Schalter bei (3) kann eingestellt werden, ob die reale oder die virtuelle Zeit angezeigt wird. Mit den Reglern (4) und (5) lässt sich in X- bzw. y-Richtung in die Diagramme hinein zoomen. Zusätzlich kann man durch Anklicken und Markieren eines Bereichs im Diagramm ebenfalls einen beliebigen Bereich in x-Richtung vergrößern. Falls sehr viele Fäden auf die Datenstruktur zugegriffen haben, wird Regler (6) interessant. Dieser justiert die Höhe der weißen Zeichenfläche, auf welchem die Diagramme zu sehen sind. Bei vielen Fäden kann man die Höhe soweit vergrößern, dass die Historien der Fäden eine angemessene Größe haben. Schließlich gibt es noch zwei Knöpfe zum Abfotografieren des aktuell dargestellten Inhalts. Der obere Knopf fotografiert dabei den gesamten Fensterinhalt inklusive der Bedienelemente. Der untere Knopf macht lediglich ein Abbild der weißen Zeichenfläche mit den Koordinatensystemen und den Legenden. Die Bilder werden im Hauptspeicher abgelegt und können durch die Einfügefunktion eines Bildbearbeitungs-, Präsentation- oder Schreibprogramms eingefügt und weiterverarbeitet werden. Ein erweitertes Beispiel, bei dem eine parallel verwendete Liste zu sehen ist, findet man im Anhang auf Seite 81, Abbildung 18.

6.5 Erweiterbarkeit

In diesem abschließenden Kapitel zur Umsetzung der Konzeption wird auf die Erweiterbarkeit eingegangen. Es richtet sich an diejenigen, die die Implementie-

rung anpassen und ergänzen wollen – solange das Konzept an sich bestehen bleibt. Wenn weiteres Parallelisierungspotenzial identifiziert werden soll, wird es u. U. nötig sein

- neue Datenstrukturen zu unterstützen, wobei entsprechende Stellvertreter implementiert werden müssen,
- neue Zugriffsarten zu definieren, um mehr Informationen der Fingerzeigerkennung zur Verfügung zu stellen,
- neue Phasencharakteristika zu definieren, wenn weitere Verhaltensmuster in den Zugriffen erkannt werden sollen,
- und/oder neue Fingerzeige in den Katalog aufzunehmen.

Die folgenden Kapitel beschreiben zu unterschiedlichen Erweiterungswünschen, welche Änderungen wo im Quelltext vorgenommen werden müssen.

Unterstützung weiterer Datenstrukturen

Die Stellvertreter weiterer Datenstrukturen sollten in `MyCollectionNS` hinzugefügt werden. Deren Konstruktoren müssen sich, ähnlich zu den bestehenden, beim `HistorySender` anmelden, wodurch sie eine Kennung erhalten. Mit dieser Kennung können für die Methoden entsprechende Zugriffsereignisse erzeugt werden. Je nach Zugriffsart wird dazu die Methode `HistorySender.addToHistory_...` verwendet.

Zusätzlich zum Stellvertreter muss der Instrumentierer erweitert werden, sodass der neue, zu beobachtende Datentyp injiziert werden kann. Die dort bestehenden Besucher-Klassen können als Vorlage dienen. Abschließend ist es nötig, die Verwendung der neuen Besucher in der Hauptfunktion von `Instrumentierer` anzustoßen.

Neue Zugriffsarten

Die Enumeration `AccessType` im Paket `BasicStuff` enthält alle definierten Zugriffsarten. Dort können neue Arten registriert werden. Üblicherweise will man Zugriffsereignisse mit den neuen Zugriffsarten erzeugen lassen. Dazu wird bei `HistorySender` lediglich eine neue Methode erstellt, die dem Namensmuster `addToHistory_xyz` gerecht wird. Darin kann der neue Zugriffstyp als Parameter verwendet werden. Zusätzlich muss in `BasicStuff.UTILS` das Array `indexedAccesses` um die neue Zugriffsart erweitert werden, falls es sich um eine indexbehafte Art handelt. Ansonsten gilt sie automatisch als indexlos. Für den Fall, dass eine neue Zugriffsart mehrere „primitive“ Arten beinhaltet (denkbar wäre zum Beispiel eine Operation zum Lesen mit gleichzeitigem Löschen), sollten die Methoden `isRead`, `isWrite` usw. der Klasse `BasicStuff.UTILS` ebenfalls angepasst werden. Schließlich muss einer neuen Zugriffsart die Farbe der Visualisierung zugewiesen werden. Dies geschieht in der Methode `generateCharts` von der Klasse `HistoryPresenter.Form1`.

Neue Phasencharakteristika

Neue Phasencharakteristika können in der Enumeration `Phasentyp` im Namensraum `BasicStuff` definiert werden. Die Fabrikmethode `getCorrespDetector` der Klasse `PhaseDetective` muss schließlich zu dieser neuen Phase den entsprechenden Phasenerkenner zurückgeben. Es ist unumgänglich, dass dieser von `Detector` erbt und sollte zudem in `Detectoren.cs` implementiert sein.

Neue Fingerzeige

Das Hinzufügen neuer Fingerzeige gestaltet sich gleichermaßen einfach wie für neue Phasencharakteristika. Im Paket `Fingerzeiggenerierung` muss zunächst eine neue Klasse implementiert werden, die von `AbstractGenerator` erbt. Im zweiten Schritt ist es erforderlich die Liste der Fingerzeiggeneratoren in der Methode `gibFingerzeige` der Klasse `FingerzeigKoordinator` um eine Instanz des neuen Generators zu erweitern. Danach werden die Historien der Protokolle auch bezüglich des neuen Fingerzeigs durchsucht.

6.6 Programmparameter

Im Folgenden werden die Konstanten aus unterschiedlichen Klassen und Methoden aufgelistet und deren Wirkung erklärt. Diese Stellschrauben sind insbesondere für grundlegende Modifikationen des Werkzeugs von Bedeutung. Variablen der Ein- und Ausgabeorte sind ebenfalls hier genannt.

Klasse FingerzeiggenerierungNS.AbstractGenerator

Kapitel 5.4.3 verwendet unscharfe Zahlenangaben, deren Schwellenwerte wie folgt im Quelltext realisiert sind. Die gesetzten Werte der Variablen sind Ergebnis einer manuellen Exploration.

int CONST_DSLengthThreashold = 50;
gibt an, wie lang eine Datenstruktur sein muss, um als „lang“ zu gelten.

int CONST_DSAccessMinimumCount = 1000;
gibt an, wie lang eine Historie sein muss, um als „lang“ zu gelten. Die entsprechende Datenstruktur gilt dann als „oft“ zugegriffen.

int CONST_frontDeletePercentage = 30;
gibt an, wie groß der Anteil der Lesen-Vorn Phasen in Bezug auf die Historienlänge sein muss, um als „häufiges Lesen vorn“ zu gelten.

int CONST_longInsertMinimumOpCnt = 100;
gibt die Mindestlänge einer Einfügephase an, um als „lang“ zu gelten.

int CONST_longInsertMinimumCnt = 1;
gibt die Mindestanzahl langer Einfügephasen an, um als „oft” zu gelten.

int CONST_queueOperationsPercentage = 60;
gibt an, wie groß der Anteil von Warteschlangenoperationen einer Historie in Bezug auf deren Länge sein muss, um als Warteschlange zu gelten.

int CONST_findPercentage = 2;
gibt an, wie groß der Anteil der Suchoperationen in Bezug auf die Historienlänge sein muss, um als „oft suchen“ zu gelten.

int CONST_longLinearReadPercentage = 50;
gibt an, wie groß der Anteil der Längensumme langer Lesephasen sein muss, damit eine Historie als „oft lang” gelesen gilt.

int CONST_longLinearReadMinimumCnt = 10;
gibt die Mindestlänge einer Lesephase an, damit diese nicht als „zu kurz“ gilt. Dieser Parameter ist in Kombination mit dem folgenden zu betrachten.

int CONST_singleLongReadPercentage = 50;
gibt an, wie groß der Anteil der Länge einer linearen Lesephase in Bezug auf die Länge der Datenstruktur sein muss, um als „lang“ zu gelten.

Klasse HistoryPresenterNS.Form1

int maxLines = 8000;
gibt die maximale Anzahl von Zugriffsereignissen an, die in einer Historie dargestellt warden sollen.

int maxParts = 500;
gibt die maximale Anzahl von Instanzen einer Datenstruktur an, die in der Oberfläche auswählbar sein sollen.

Klasse InstrumenterNS.Instrumentierer

Boolean INSTRUMENT_ARRAYS = true;
gibt an, ob Arrays instrumentiert werden sollen.

Methode InstrumenterNS.Instrumentierer.Main

String solutionFile = @"c:\Path\to\Solution.sln";
ist der Dateiname inklusive dem Pfad der Beschreibungsdatei eines zu instrumentierenden C#-Projektes.

String targetPath = @"c:\myInstrumentedProjects\";
ist der Ordnerpfad, in welchem das instrumentierte Projekt ausgegeben wird.

Klasse PhaseDetectionNS.PhaseDetective

int minPhaseSize = 5;
definiert die Anzahl von Zugriffsereignissen, die eine Phase mindestens enthalten muss, um bestehen zu bleiben.

Methode PhaseDetectionNS.PhaseDetective.dumpCSV

int maxLines = 10000;
definiert die maximale Anzahl von Zeilen in der Ausgabe für einen Instanziierungsort (unabhängig von der Anzahl der Instanzen).

Methode WholeProcessWalker.Main

List<String> paths;
enthält die Ausführungsdatei (inklusive Pfad), des instrumentierten Programms, welches gestartet werden soll.

7 Evaluation

Mit einer Sammlung von Testprogrammen aus verschiedenen Domänen werden im folgenden Fallbeispiele beleuchtet, für die gezeigt wird,

- wie groß der Faktor der Verlangsamung durch die Instrumentierung ist,
- wie viele richtig-positiv Treffer gefunden werden und
- welche Laufzeitvorteile durch die manuelle Parallelisierung, angeleitet durch die richtig-positiven Treffer, erreicht werden.

Dazu werden die Quelltexte der Testprogramme in das Werkzeug gegeben und die Analysenergebnisse untersucht. Die Laufzeitvorteile werden durch manuelle Implementierung der angebotenen Fingerzeige überprüft. Jedes Evaluierungsprogramm wird dazu in den folgenden Unterkapiteln separat betrachtet. Zunächst wird ein Überblick über die Programme gegeben und die Messungen zum Verlangsamungsfaktor gezeigt.

Programm	LOC	Domäne	Textuelle Ausgabe	Grafische Visualisierung	Art der Ausführung	#Instanziierungsorte (List/Array)
Algorithmia	2800	Bibliothek	-	-	UnitTest	16 (15/1)
Astrogrep	4800	Dateisuche	+	-	Manuell	21 (6/15)
Contentfinder	290	Dateisuche	+	-	Manuell	11 (5/6)
CPU-Benchmark	400	Benchmark	+	-	Manuell	7 (0/7)
Gpdotnet	7000	Simulation	+	+	Manuell	33 (14/19)
Mandelbrot	150	Problemlöser	+	+	Manuell	7 (0/7)
WordWheel-Solver	110	Problemlöser	+	-	Manuell	5 (5/0)

Tabelle 4: Übersicht der Testprogramme

Tabelle 4 zeigt eine Übersicht der von CodePlex und SourceForge heruntergeladenen Programme, die zur Evaluierung herangezogen werden, aufgeschlüsselt nach der Anzahl Zeilen Code (LOC[2]), der Domäne, der Verwendung einer textuellen und/oder grafischen Ausgabe bzw. Visualisierung, der Art der Programmausführung und der Anzahl instrumentierter Instanziierungsorte.

Die Hardwarekonfiguration des Testrechners zeichnet sich durch folgende Merkmale aus:

- Prozessor: AMD FX 8120h, 8 Prozessorkerne, je 3,1 GHz Taktfrequenz
- Hauptplatine: Acer Predator G3120
- Hauptspeicher: 8 GB (4x2GB DDR3, PC3-10700, Dual-Channel)
- Festplatte: Samsung SSD 830
- Betriebssystem: Windows 7 x64

Parallelisierungen der Evaluierungsprogramme nutzen alle acht Kerne, wodurch eine maximale Beschleunigung von 8 erreicht werden kann[3].

Wie groß die Verlangsamung durch die Instrumentierung bei Schreiboperationen ist, wird mithilfe eines selbst geschriebenen Testprogramms ermittelt. Dieses misst die Zeit von Schleifen bei mehrmaliger Traversierung und errechnet daraus die Durchschnittszeit pro Zugriffsoperation. Zugriffe auf das Array sind auf dem Testrechner 84-mal schneller als auf `MyArray`. Zugriffe auf `List` sind 63-mal schneller als auf `MyList`.

Programm	Einheit	Laufzeit (orig.)	Laufzeit (instr.)	Verlangsamungs-faktor
Algorithmia	sek	0,50	2,40	4,80
Astrogrep	sek	4,80	5,80	1,21
Contentfinder(v1)	sek	3,20	17,00	5,31
Contentfinder(v2)	sek	1,80	5,20	2,89
CPU-Benchmark	ms	10,00	550,00	55,00
Gpdotnet(SC)	sek	0,36	78,00	216,67
Gpdotnet(MC)	sek	0,30	64,38	214,60
Mandelbrot(SC)	ms	110,00	1200,00	10,91
Mandelbrot(MC)	ms	50,00	800,00	16,00
WordWheelSolver	ms	39,00	1500,00	38,46

Tabelle 5: Messungen des Mehraufwands durch die Instrumentierung

Tabelle 5 zeigt die Ergebnisse der Messungen zur Verlangsamung der Testprogramme durch die Instrumentierung[4]. Im Gesamtschnitt verlangsamt die

[2] ohne Leer- und Kommentarzeilen, Ermittelt über die Codemetrik von Visual Studio

[3] es sei denn Cache-Effekte führen zu superlinearen Beschleunigungen

[4] (MC) bedeutet Ausführung mit aktivierter Mehrkernoption, (SC) bedeutet einfädige Programmausführung

Instrumentierung die Laufzeit um den Faktor 57. Das Programm gpdotnet stellt einen Ausreißer dar, da hier die Instrumentierung mit einem Faktor von ca. 215 besonders stark interveniert. Einige Zugriffsarten, wie zum Beispiel `Remove-All(Predicate<T> match)`, werden mit deutlich mehr Aufwand protokolliert, als Schreibzugriffe. gpdotnet macht sehr viel Gebrauch von diesen Operationen, wodurch die starke Verlangsamung entsteht. Die restlichen Programme erfahren einen durchschnittlichen Verlangsamungsfaktor von ca. 17.

Die Programme gpdotnet und Mandelbrot können mehrfädig und einfädig gestartet werden. Beide Modi sind in Tabelle 5 als (MC) bzw. (SC) gekennzeichnet. Bei der manuellen Inspektion des Programms Contentfinder ergab sich ein algorithmisches Optimierungspotenzial, welches in Kapitel 7.3 nachzulesen ist. Die Verlangsamung der verbesserten Version wurde separat gemessen und ist als (v2) in der Übersicht gekennzeichnet.

Im Folgenden werden die Testprogramme einzeln beleuchtet. Es wird gezeigt, welche Fingerzeige ausgegeben wurden und welche Beschleunigung die manuelle Umsetzung der Hinweise erreicht. Die Position gibt dabei an, wo in der Methode die Instanziierung stattfindet. Die Ordnungskennzahl realisiert auf Grundlage der Länge einer Historie eine Ordnung der Fingerzeige.

7.1 Algorithmia

Diese Klassenbibliothek stellt diverse Datenstrukturen und Algorithmen zur Verfügung. Die Ausführung des Unit-Tests ergab folgende vier Fingerzeige.

Fingerzeig 1

- **Klasse**: SD.Tools.Algorithmia.Tests.SortingTests
- **Methode, Position**: CreateListOfRandomNumbers, 6
- **Datenstruktur**: List<System.Int32>
- **Grund, Ordnungskennzahl**: Langes Einfügen, 34034

Fingerzeig 2

- **Klasse**: SD.Tools.Algorithmia.PriorityQueues.SimplePriorityQueue`1
- **Methode, Position**: InitDataStructures, 7
- **Datenstruktur**: List<SD.Tools.Algorithmia.Tests.PriorityQueueTests+QueueElement>
- **Grund, Ordnungskennzahl**: Häufig langes Lesen, 5451

Fingerzeig 3

- **Klasse**: SD.Tools.Algorithmia.PriorityQueues.SimplePriorityQueue`1
- **Methode, Position**: InitDataStructures, 7
- **Datenstruktur**: List<SD.Tools.Algorithmia.Tests.PriorityQueueTests+QueueElement>
- **Grund, Ordnungskennzahl**: Langes Einfügen, 5451

Fingerzeig 4

- **Klasse**: SD.Tools.Algorithmia.Tests.PriorityQueueTests
- **Methode, Position**: PriorityQueueFunctionalityTest, 6
- **Datenstruktur**: List<System.Int32>
- **Grund, Ordnungskennzahl**: Langes Einfügen, 2562

Fingerzeig 1

Die Testmethoden dreier Sortieralgorithmen erhalten bei diesem Instanziierungsort ihre zu sortierenden Listen. Die Erzeugung dieser Listen wurde so parallelisiert, dass mehrere Fäden gleichzeitig entsprechend lange Teillisten mit randomisierten Zahlen füllen, sodass deren geschützte Konkatenation die gewünschte Länge besitzt. Die Ausführungszeit dieser Initialisierung reduziert sich dadurch bei 10 Mio. Elementen von 270 ms auf 200 ms. Dies ergibt einen Beschleunigungsfaktor von 1,35. Da die Initialisierung lediglich ca. 1 % der 88-sekündigen Laufzeit des Sortier-Tests ausmacht, fällt diese Parallelisierung kaum ins Gewicht.

Fingerzeig 2

Dieser Fingerzeig weist auf häufig langes Lesen in einer `List` hin, was von einer selbst implementierte Suche stammen könnte. Tatsächlich handelt es sich bei der `SimplePriorityQueue` um eine Suchdatenstruktur. Bei jeder Anfrage, das Element mit der höchsten Priorität zurückzugeben, wird die gesamte Datenstruktur durchsucht. Abgesehen vom linearen Aufwand, lässt sich diese Suche auf Teilabschnitten der Liste parallelisieren. Beim NUnit-Test mit 100.000 Elementen ergibt sich dabei eine Beschleunigung von 2,3. Die Laufzeit verbessert sich damit von 266 sek auf 117 sek.

Der Einsatz einer geeigneteren Suchdatenstruktur verbessert in diesem Fall auch ohne Parallelisierung die Laufzeit dramatisch. Bei 10.000 Elementen benötigt die parallelisierte Version 1,6 sek, die sortierte Vorrangwarteschlange 0,14 sek und der binäre Heap lediglich 0,05 sek. Zusätzlich kann die sortierte Vorrangwarteschlange bei 100.000 Elementen von 12 sek auf 4,4 sek verbessert werden, indem die interne Sortierung umgedreht wird. Da die Elemente am Ende gelöscht werden, müssen die verbleibenden Elemente weniger oft im Speicher verschoben werden, wodurch die 2,7-fache Beschleunigung entsteht.

Fingerzeig 3 und Fingerzeig 4

Diese beiden Fingerzeige beziehen sich auf Initialisierungsmethoden und nehmen daher keinen oder nur sehr geringen Einfluss auf die Gesamtlaufzeit. Aus diesem Grund wird an diesen Stellen nicht parallelisiert.

7.2 Astrogrep

Dieses Programm dient zur Suche nach Dateien mit bestimmten Inhalten. Dabei können eine Vielzahl von Suchoptionen, wie zum Beispiel die maximale Dateigröße oder die Aktivierung von regulären Ausdrücken, gesetzt werden. Die gefundenen Zeilen werden nach Dateien gebündelt in der Benutzeroberfläche listenartig angezeigt. Bei der Ausführung wird nach dem Wort „Main“ in cs-Dateien gesucht.

Die Suche in den 3.000 Ordnern mit insgesamt 14.500 Dateien (davon 5.930 cs-Dateien) ergibt 1.435 Treffer.

Fingerzeig 1

- **Klasse:** libAstroGrep.Grep
- **Methode, Position:** .ctor, 6
- **Datenstruktur:** List<System.String>
- **Grund, Ordnungskennzahl:** Häufiges Suchen, 6080

Fingerzeig 2

- **Klasse:** libAstroGrep.Grep
- **Methode, Position:** .ctor, 41
- **Datenstruktur:** List<libAstroGrep.HitObject>
- **Grund, Ordnungskennzahl**: Langes Einfügen, 1439

Fingerzeig 1

Die hier betroffene Datenstruktur wird im Konstruktor instanziiert und enthält die Endungen der zu ignorierenden Dateien. Für jede Datei wird geprüft, ob deren Endung in dieser Liste enthalten ist und damit übersprungen werden kann. Diese Suche kann parallelisiert werden. Allerdings konnte dabei keine Beschleunigung erreicht werden. Parallelisiert wurde mit Hilfe von Partitionen, auf denen unabhängig voneinander gesucht werden kann. Bei 1200 Dateiendungen war die parallele Version mit 450.000 bis 600.000 gemessenen Zeitscheiben (engl. *ticks*) immer noch langsamer als die Originalversion mit 350.000 *ticks*. Da noch mehr Dateiendungen in dieser Datenstruktur zu einem Verlust jeglichen Realitätsbezugs führt, wird konstatiert, dass an dieser Stelle eine Parallelisierung nicht sinnvoll ist. Grund ist, dass die Arbeitslast der Schleife zu klein im Verhältnis zum Aufwand der Parallelisierung ist. Im Gegensatz dazu bewirkt die binäre Suche in einer sortierten Liste einen Beschleunigungsfaktor von 6 bei 1.200 Dateiendungen.

Fingerzeig 2

Diese Datenstruktur enthält die Ergebnisse der Dateisuche und wird durch eine traversierende Bedingungsprüfung aller Dateien mit gefundenen Textstellen gefüllt. Dabei kommen zwei Schleifen zum Einsatz, deren Parallelisierung sich lohnt. Eine Schleife dient zum Durchlaufen der Dateien und die andere Schleife zum Durchlaufen aller Unterordner des aktuellen Ordners. Die einfache Parallelisierung mittels `Parallel.ForEach()` bewirkt eine Laufzeitverbesserung um den Faktor 2,9 von 26 sek auf 9 sek.

7.3 Contentfinder

Wie das Programm Astrogrep, sucht Contentfinder nach Dateien mit bestimmten Inhalten. Die Treffer werden ebenfalls in einer Listenansicht angezeigt. Durchsucht werden dieselben Dateien nach dem Wort „Main“ wie in Kapitel 7.2. Das Programm ist bereits vom Entwickler so parallelisiert worden, dass die eingegebenen Ordner parallel durchsucht werden. Da allerdings beim Evaluierungslauf nur ein Ordner

mit den beschriebenen Dateien durchsucht wird, bewirkt diese Parallelisierung keine Laufzeitvorteile.

Das Werkzeug gibt nur einen einzelnen Fingerzeig bei der Ausführung aus. Dieser wird im Folgenden beleuchtet. Bei der Durchsicht ergab sich zudem noch eine algorithmische Optimierung. Im Testprogramm wurde eine Ergebnisliste sukzessive mit dem Operator `Union()` um eine neue Teilliste erweitert:

```
allFileList = allFileList.Union(returnValue).ToList();
```

Diese Codezeile bewirkt, dass bei jeder Aktualisierung der Ergebnisliste eine neue Instanz angelegt wird. Folgendes einfaches Hinzufügen zur Ergebnisliste verkürzt die Gesamtlaufzeit des Programms bereits um einen Faktor von 2,8:

```
allFileList.AddRange(returnValue);
```

Die Programmversion mit dieser einzeiligen Änderung wird separat untersucht und ergibt zusätzlich zum ersten Fingerzeig den zweiten der nun folgenden Hinweise.

Fingerzeig 1

- **Klasse:** ContentFinder.MainWindow
- **Methode, Position:** ProcessEveryFolder, 59
- **Datenstruktur:** List<ContentFinder.ContentUsage>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 2405

Fingerzeig 2

- **Klasse:** ContentFinder.MainWindow
- **Methode, Position:** GetContainsMyContent, 6
- **Datenstruktur:** List<ContentFinder.ContentUsage>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 4815

Fingerzeig 1

Die hier verantwortliche Schleife befüllt die Ergebnisliste, welche schließlich in der Benutzeroberfläche angezeigt wird.

Die Parallelisierung mittels `Parallel.ForEach()` ergibt einen Geschwindigkeitszuwachs von 5 sek auf 3,2 sek. Dies entspricht einem Faktor von 1,56.

Fingerzeig 2

Die sequenzielle Optimierung bedingt eine bereits sequenziell schnellere Ausführung. Dieser Fingerzeig wurde aufgrund dieser Optimierung erst erzeugt. Die verantwortliche Datenstruktur ist die bereits oben genannte Variable `allFileList`. Der Fingerzeig führt zur selben Parallelisierung wie der vorige, da der Inhalt dieser Variablen in die Datenstruktur des obigen Fingerzeigs hinzugefügt wird. Es ergibt sich trotz der sequenziell schnelleren Ausführung von 1,8 sek eine Beschleunigung um 1,2 auf nur noch 1,5 sek.

7.4 CPU-Benchmark

Dieses Programm implementiert den Linpack- und den Whetstone-Benchmark in C#. Eine grafische Benutzeroberfläche stellt zwei Knöpfe für diese beiden Verfahren

zur Verfügung und zeigt die Ergebnisse jeweils in einem dafür vorgesehenen Feld an. Berechnende Methoden (hier „Benchmarkmethoden“ genannt), deren Zeit gemessen wird, um die Benchmark-Punktzahl zu ermitteln, werden nicht parallelisiert. Würden die Benchmarkmethoden parallelisiert werden, würde ein neuer Benchmark entstehen, der ganz andere Aspekte des Rechners vermisst als ursprünglich vorgesehen. Aus diesem Grund werden lediglich die Methoden parallelisiert, die keinen Einfluss auf das Benchmarkergebnis nehmen, wie zum Beispiel Initialisierungs- und Auswertungsmethoden. Die im Folgenden gemessenen Zeiten beziehen sich dementsprechend auf die Gesamtlaufzeit abzüglich der Zeit für die Benchmarkmethoden. Die Konstante im Quelltext, welche die Arraygröße definiert, wird zu Evaluierungszwecken auf 1.200 festgelegt.

Folgende Fingerzeige werden gefunden:

Fingerzeig 1
- **Klasse:** CPU_Benchmark.Linpack
- **Methode, Position:** RunBenchmark, 23
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 20548

Fingerzeig 2
- **Klasse:** CPU_Benchmark.Linpack
- **Methode, Position:** RunBenchmark, 16
- **Datenstruktur:** Array<MyCollectionNS.MyArray`1[System.Double]>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 18984

Fingerzeig 3
- **Klasse:** CPU_Benchmark.Linpack
- **Methode, Position:** RunBenchmark, 16
- **Datenstruktur:** Array<MyCollectionNS.MyArray`1[System.Double]>
- **Grund, Ordnungskennzahl:** Häufig langes Lesen, 18984

Fingerzeig 4
- **Klasse:** CPU_Benchmark.Linpack
- **Methode, Position:** RunBenchmark, 55
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Häufig langes Lesen, 3008

Fingerzeig 5
- **Klasse:** CPU_Benchmark.Linpack
- **Methode, Position:** RunBenchmark, 31
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 2605

Alle fünf Fingerzeige gehören zu den drei Datenstrukturen, die im Benchmark gemeinsam verwendet werden. Deren vorbereitende Berechnungen werden parallelisiert. Die Hinweise zum langen Einfügen und dem häufigen Lesen

resultieren teilweise aus diesen Vorbereitungen, welche die Arrays mehrmals traversieren. Es liegen somit keine Suchdatenstrukturen vor. Zehn der 42 im Programm enthaltenen Schleifen nehmen dabei Bezug auf diese Datenstrukturen. Zeitmessungen ergeben, dass drei der Schleifen Parallelisierungspotenzial enthalten. Die Parallelisierung dieser drei Schleifen ergab einen Geschwindigkeitszuwachs von 460 ms um einen Faktor von 1,2 auf 380 ms bei acht sekündiger Programmlaufzeit.

Der erste und der letzte Fingerzeig werden nicht als richtig-positive Treffer gewertet, auch wenn sie den Programmierer auf die relevanten Datenstrukturen lenken. Das Problem ist die Art des Fingerzeigs – das lange Einfügen, welches parallelisiert werden soll. Die initial schreibenden Schleifen, die die beiden Arrays befüllen, sollten nicht parallelisiert werden, da der Mehraufwand der Parallelisierung unverhältnismäßig groß ist.

7.5 gpdotnet

Dieses Programm verwendet genetische Optimierungsalgorithmen, um Modelle zur Beschreibung von Zeitreihen und diskreten Datenreihen zu berechnen. Es wurde mit den mitgelieferten Beispieldaten im Modus „Single core“ (restliche Einstellungen mit den Standardwerten) ausgeführt. Die dabei erzeugten fünf Fingerzeige werden im Folgenden diskutiert.

Fingerzeig 1

- **Klasse:** GPdotNET.Engine.GPModelGlobals
- **Methode, Position:** GenerateTerminalSet, 120
- **Datenstruktur:** Array<MyCollectionNS.MyArray`1[System.Double]>
- **Grund, Ordnungskennzahl:** Häufig langes Lesen, 119763

Fingerzeig 2

- **Klasse:** GPdotNET.Engine.CHPopulation
- **Methode, Position:** .ctor, 14
- **Datenstruktur:** List<GPdotNET.Core.IChromosome>
- **Grund, Ordnungskennzahl:** Häufig langes Lesen, 67187

Fingerzeig 3

- **Klasse:** GPdotNET.Engine.CHPopulation
- **Methode, Position:** .ctor, 14
- **Datenstruktur:** List<GPdotNET.Core.IChromosome>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 67187

Fingerzeig 4

- **Klasse:** GPdotNET.Engine.CHPopulation
- **Methode, Position:** FitnessProportionateSelection, 68
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Häufig langes Lesen, 1391

Fingerzeig 5

- **Klasse:** GPdotNET.Engine.CHPopulation
- **Methode, Position:** FitnessProportionateSelection, 68
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 1081

Fingerzeig 1

Einige Schleifen iterieren hier über die Datenstruktur, um Aggregate wie z. B. Maximum oder Durchschnitt zu berechnen. Leider ist die Länge der Datenstruktur zu gering, sodass sich eine Parallelisierung nicht lohnt, sondern sogar eine Verlangsamung mit sich bringt. Eine Optimierung als Suchdatenstruktur ist ebenfalls nicht möglich.

Fingerzeig 2 und Fingerzeig 3

Auch hier liegt keine Suchdatenstruktur vor. In der Liste werden die Elemente immer wieder aktualisiert und zu löschende Elemente gesucht. Dadurch wird sie oft vollständig gelesen. Die Schleife, welche genug Arbeitslast verbraucht, um eine Parallelisierung zu rechtfertigen, wurde bereits vom Entwickler parallelisiert. Es ergibt sich daraus eine Beschleunigung der Gesamtlautzeit von 2,99 min auf 1,02 min, folglich ein Beschleunigungsfaktor von 2,93. Fingerzeig 2 weist damit auf Parallelisierungspotenzial hin, welches auch vom Entwickler als solches eingeschätzt und umgesetzt wurde.

Das Einfügen in die Datenstruktur erfolgt initial zu Beginn. Hierbei wird eine Population für den genetischen Algorithmus erstellt, deren Elemente zufällige Werte enthalten. Da sie nur einen geringen Einfluss auf die Gesamtlaufzeit nehmen, wird die initialisierende Schleife nicht parallelisiert.

Fingerzeig 4 und Fingerzeig 5

Beide Fingerzeige wurden bezüglich derselben Datenstruktur gefunden. Relevant ist diese Datenstruktur bei der Erzeugung einer neuen Population für den genetischen Algorithmus. Dabei wird die alte Population traversiert und für jedes Element ein neues Element angelegt. Die Parallelisierung erfolgt mithilfe einer Partitionierung des Arrays, sodass die Fäden unabhängig auf den Teilbereichen operieren können. Die von ihnen erstellten Teilpopulationen werden am Ende geschützt konkateniert. Für die Schleife ergibt sich bei einer Populationsgröße von 5.000 Elementen eine Laufzeitverbesserung von 164 ms auf 50 ms. Dieser Faktor von 3,28 der Beschleunigung nimmt jedoch wenig Einfluss auf die Gesamtlaufzeit, da der relevante Codeabschnitt selten aufgerufen wird. Bei der Vermessung von 100 Iterationen des parallelen genetischen Algorithmus ergibt sich eine Gesamtlaufzeitverbesserung von 2,57 min auf 2,4 min, was einem Faktor von 1,07 entspricht.

Der Fingerzeig des häufig langen Lesens wurde ausgegeben, weil der Anteil des langen Lesens sehr hoch ist. Da aber die Instanzen nur einmal befüllt und danach zur Konkatenation einmal traversiert werden, wird dieser Fingerzeig als falschpositiv gewertet.

7.6 Mandelbrot

Dieses Programm berechnet das Mandelbrot-Fraktal und stellt es in der Benutzeroberfläche grafisch dar. Die Berechnung kann dabei wahlweise sequenziell oder parallel erfolgen. Die Zeitmessungen zu den folgenden Fingerzeigen wurden mit einer Bildgröße von 1858x1028 Bildpunkten vorgenommen.

Fingerzeig 1

- **Klasse:** MandelbrotDemo.MandelbrotSolver
- **Methode, Position:** Solve, 8
- **Datenstruktur:** Array<System.Int32>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 9738

Fingerzeig 2

- **Klasse:** MandelbrotDemo.SolverFrame
- **Methode, Position:** Initialize, 43
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 7422

Fingerzeig 3

- **Klasse:** MandelbrotDemo.SolverFrame
- **Methode, Position:** Initialize, 14
- **Datenstruktur:** Array<System.Double>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 7422

Fingerzeig 4

- **Klasse:** MandelbrotDemo.BitmapGenerator
- **Methode, Position:** Create, 1
- **Datenstruktur:** Array<System. Byte>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 3713

Fingerzeig 1

Dieser Fingerzeig führt den Programmierer zu der Stelle im Code, die bereits vom Entwickler parallelisiert wurde. Diese Parallelisierung ergibt eine Beschleunigung von 2,9. Die Gesamtlaufzeit verringert sich von 490 ms auf 170 ms.

Fingerzeig 2 und Fingerzeig 3

Auch diese Fingerzeige führen den Benutzer zu einer Schleife im Quelltext, deren Parallelisierung bereits über eine Option des Kompilierers vom Entwickler aktivierbar gemacht wurde. Zur Testlaufzeit war der sequenzielle Code verwendet worden. Das bedeutet, dass diese beiden Fingerzeige ebenfalls mit der Parallelisierungspotenzialabschätzung des Entwicklers übereinstimmen. Die Parallelisierung verringert die Ausführungszeit der Schleife von 60 ms auf 34 ms, folglich um einen Faktor von 1,77.

Fingerzeig 4

Die hier verantwortliche Datenstruktur wird durch eine Schleife initialisiert. Die Parallelisierung dieser Schleife verbessert ihre Laufzeit von 46 ms auf 33 ms, was einem Faktor von 1,4 entspricht.

Die beiden letzten Fingerzeige ergeben eine Verbesserung des kleinen Laufzeitanteils. Dennoch erreicht die Beschleunigung der Gesamtlaufzeit durch alle Fingerzeige einen Faktor von 3.

7.7 WordWheelSolver

Dieses Programm sucht in einem Wörterbuch nach Worten, die aus bestimmten Buchstaben bestehen und eine angegebene Mindestlänge haben.

Fingerzeig 1

- **Klasse:** WordWheelSolver.Program
- **Methode, Position:** readDictionary, 19
- **Datenstruktur:** List<System.String>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 147610

Fingerzeig 2

- **Klasse:** WordWheelSolver.Program
- **Methode, Position:** trimDictionary, 21
- **Datenstruktur:** List<System.String>
- **Grund, Ordnungskennzahl:** Langes Einfügen, 13293

Fingerzeig 1

Die verantwortliche Datenstruktur wird mit Buchstabenketten aus einer Datei befüllt. Da dieser Prozess nicht parallelisiert werden kann, wird dieser Hinweis als falsch-positiv gewertet.

Fingerzeig 2

Die `List` vom vorangegangenen Fingerzeig wird hier traversiert und gültige Elemente in eine neue Datenstruktur kopiert. Dieser Einfügeprozess beschränkt sich auf die Überprüfung der Länge der Wörter. Eine Parallelisierung lohnt sich deswegen hier nur dann, wenn viele Worte in die neue Datenstruktur übernommen werden. Beim akademischen Fall mit einem Wörterbuch aus 1 Mio. Elementen und der Verwendung fast aller Buchstaben des Alphabets ist eine Beschleunigung knapp über 1 messbar. Bei normalen Wörterbüchern mit ca. 200.000 Einträgen wirkt sich die Parallelisierung der Schleife sogar verlangsamend auf die Programmlaufzeit aus.

Dennoch enthält der Fingerzeig eine wichtige Information. Die Datenstruktur, die hier befüllt wird, wird im folgenden Schritt des Algorithmus gelesen. Für jedes Wort wird dabei geprüft, ob die enthaltenen Zeichen gültig sind. Dies erfordert viel Rechenaufwand, weshalb sich die Parallelisierung dieser Schleife sehr schnell amortisiert. Schon bei einem kleinen Wörterbuch mit 230.000 Einträgen und der Suche nach Worten bestehend aus den ersten neun Buchstaben ergibt sich eine

Beschleunigung von 140 ms auf 90 ms. Dieser Beschleunigungsfaktor von 1,5 lässt sich durch Vergrößern des Wörterbuchs oder des erlaubten Zeichensatzes stark steigern. Bei der oben genannten akademischen Situation ist ein Beschleunigungsfaktor von 4 messbar.

7.8 Auswertung

Die Evaluierung zeigt, dass die Umsetzung von Parallelisierungspotenzialen, welche dem Nutzer als Fingerzeige propagiert werden, Beschleunigungen der Programmlaufzeit erzielt. Dabei weisen sogar zwei Fingerzeige aus gpdotnet und drei Fingerzeige aus Mandelbrot auf eine bzw. zwei Stellen im Code hin, wo die Entwickler bereits Parallelisierungsmöglichkeiten aktivierbar gemacht hatten. Dies bestätigt, dass die Fingerzeige reales Parallelisierungspotenzial aufdecken können.

Programm	#Fingerzeige (dabei untersch. Orte)	#richtig-positive Orte (davon mit falschem Grund)	#Parallelitäts-umsetzungen	#falsch-positive Orte
Algorithmia	4 (3)	2 (0)	2	1
Astrogrep	2 (2)	1 (0)	1	1
Contentfinder	2 (2)	2 (0)	1	0
CPU-Benchmark	5 (4)	4 (2)	1	0
Gpdotnet	5 (3)	2 (0)	2	1
Mandelbrot	4 (4)	4 (0)	3	0
WordWheelSolver	2 (2)	1 (1)	1	1

Tabelle 6: Nützlichkeit der Fingerzeige nach manueller Inspektion

Tabelle 6 zeigt die in den Evaluierungsprogrammen gefundenen Fingerzeige und deren „Richtigkeit“. Da mehrere Fingerzeige bezüglich der gleichen Datenstrukturinstanz ausgegeben werden, ist in Klammern angegeben, wie viele unterschiedliche Instanziierungsorte enthalten sind. Die manuelle Inspektion der Fingerzeige führt zu einer Zuweisung dieser in eine von drei Kategorien. Richtig-positive Treffer sind dabei Fingerzeige, deren Umsetzung eine Beschleunigung bewirkt. Richtig-positive Treffer mit falscher Begründung sind Fingerzeige, die auf eine Datenstruktur hingewiesen haben, deren Verwendung zwar ein Parallelisierungspotenzial bietet, aber der Grund bzw. die Art des Fingerzeigs zunächst falsch ist. Falsch-positiv Treffer sind die Fingerzeige, deren Umsetzung nicht möglich ist oder bei denen sich eine Verlangsamung einstellt. Die Anzahl der Parallelitätsumsetzungen gibt an, aus wie vielen Fingerzeigen parallelisierende Codeanpassungen resultierten.

Von den insgesamt 24 ausgegebenen Fingerzeigen sind zwei Drittel richtig-positive Treffer. Nur drei Fingerzeige sind korrekte Treffer mit falscher Begründung und nur vier Fingerzeige sind falsch-positiv. Es kann somit festgehalten werden, dass ca. 80 % der Fingerzeige aus den Testprogrammen nützlich zur Parallelisierung sind.

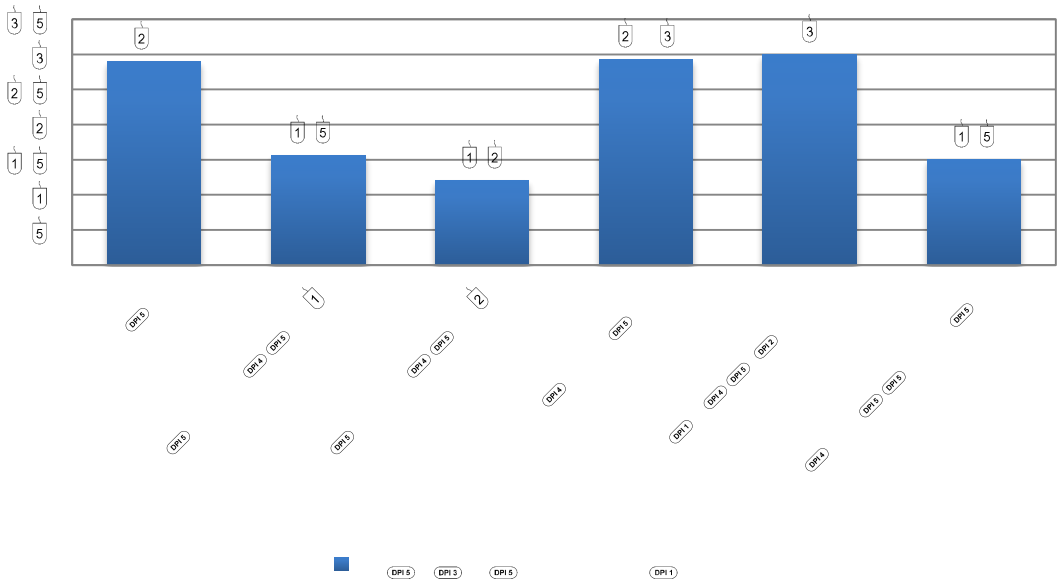

Abbildung 15: Erreichte Beschleunigungen in den Gesamtlaufzeiten der Testprogramme

Die Umsetzung der Fingerzeige ergibt in vielen Fällen eine Beschleunigung der Laufzeit. Abbildung 15 zeigt zu den fünf Evaluierungsprogrammen den erreichten Beschleunigungsfaktor nach Umsetzung aller Hinweise. Dieser liegt zwischen 1,2 beim Contentfinder und 3 bei gpdotnet. Algorithmia und CPU-Benchmark sind nicht in dem Diagramm aufgenommen, da dort keine Messungen einer sinnvollen Gesamtlaufzeit durchgeführt werden können. Für diese beiden Programme kann aber ebenfalls festgehalten werden, dass eine Beschleunigung erreichbar ist. Die Initialisierung des NUnit-Tests für Prioritätswarteschlangen in Algorithmia konnte um den Faktor 1,35 und die einfache Prioritätswarteschlange um den Faktor 2,3 beschleunigt werden. Die Initialisierungsarbeiten bei CPU-Benchmark konnten ebenfalls mit einem Faktor von 1,2 leicht verbessert werden.

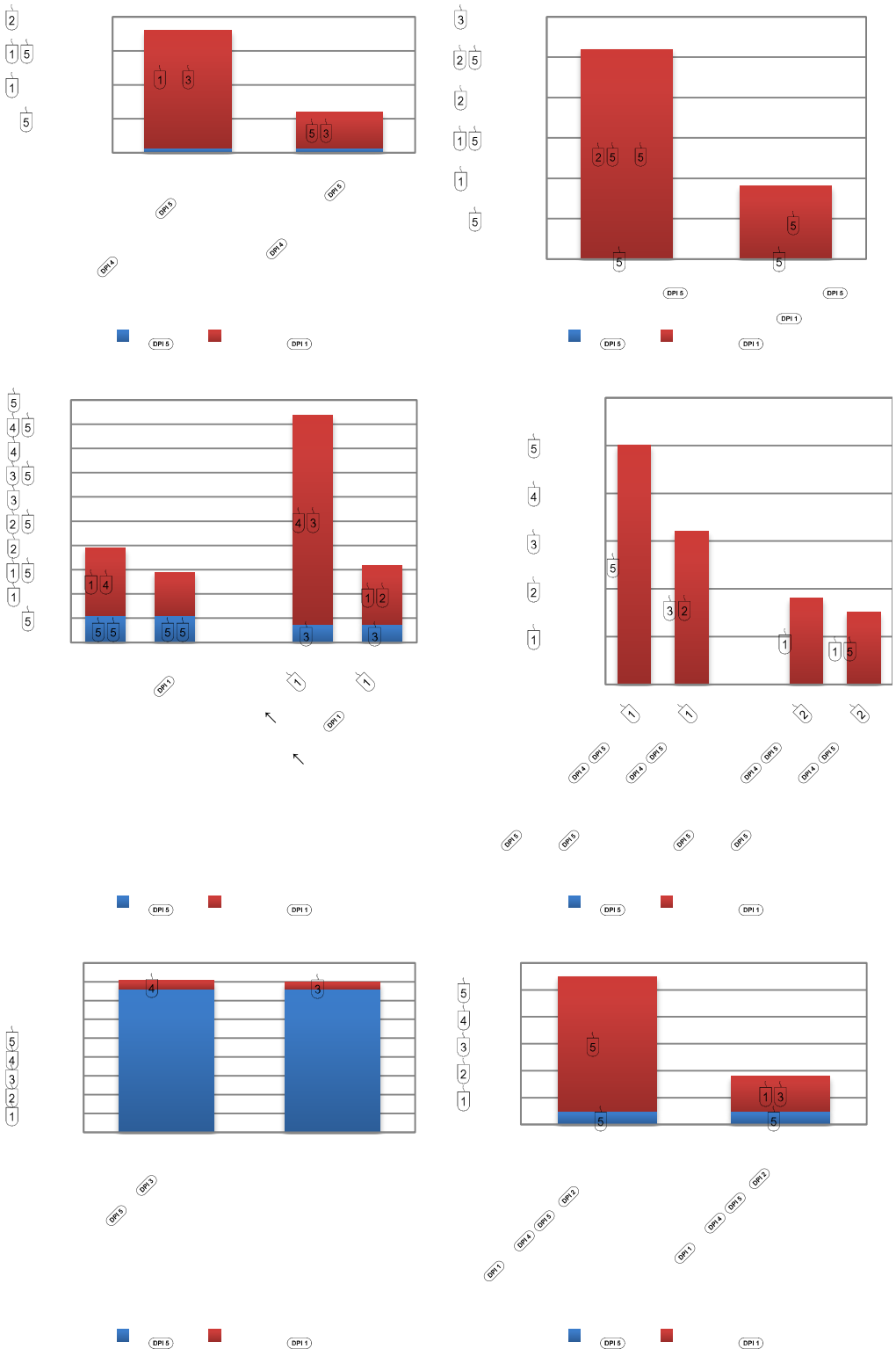

Abbildung 16: Beschleunigungen der parallel ausführbaren Zeitanteile in ms[5]

[5] „WWS“ = WorldWheelSolver; „WWS (input↑)“ = WordWheelSolver mit großer Eingabedatenmenge; „x10“ bedeutet, dass gemessene Zeiten das zehnfache der abgelesenen Zeiten sind; „CPU-Bench.“ = CPU-Benchmark

Abbildung 16 zeigt, wie viel Ausführungszeit bei den einzelnen Testprogrammen beschleunigt werden kann. Die blauen Balkenanteile (seq) repräsentieren dabei die Laufzeit, welche nicht parallelisierbar ist oder Parallelität sich nicht lohnt. Der rote Anteil (mglPar) ist die Zeit, welche für die Ausführung von parallelisierbaren Codeabschnitten benötigt wird. Zu jedem Testprogramm sind die Laufzeitanteile der sequenziellen Version (gekennzeichnet durch „seq“) und die Laufzeitanteile der parallelisierten Version (gekennzeichnet durch „par“) als zwei benachbarte Balken dargestellt. Man sieht beispielsweise, dass im CPU-Benchmark der sequenziell auszuführende Laufzeitanteil so groß ist, dass eine Parallelisierung kaum merkbar Einfluss auf die Gesamtlaufzeit hat. Im Gegensatz dazu ist der sequenzielle Laufzeitanteil beim Testprogramm Contentfinder sehr gering. Es enthält somit sehr viel Parallelisierungspotenzial. Welche Programmabschnitte parallelisierbar sind, wurde nachträglich manuell ermittelt und stellt dementsprechend subjektiv das gesamte Parallelisierungspotenzial dar. Die gemessenen Zeiten und erreichten Beschleunigungen sind programmübergreifend nicht vergleichbar, da die Laufzeitanteile individuell vermessen werden. Versuche, einen Automatismus zur Ermittlung der sequenziellen und parallelen Laufzeitanteile (für die Evaluierung) zu entwickeln, scheiterten. Zum einen, weil aus Zugriffshistorien nicht ablesbar ist, welche Codeabschnitte für die Fingerzeig-relevanten Phasen verantwortlich sind. Zum anderen ist es schwierig einen Algorithmus anzugeben, der Stoppuhren im Quelltext platziert, sodass die Grenze zwischen sequenzieller und parallelisierbarer Programmlaufzeit stets angemessen ist und somit das tatsächliche Parallelisierungspotenzial abgrenzt. Die Messung der Ausführungszeiten aller parallel zueinander ausführbaren Instruktionen kommt der theoretischen Obergrenze nahe, ist aber aufgrund des Mehraufwandes einer Parallelisierung realitätsfern. Dementsprechend ist das reale Parallelisierungspotenzial wegen des Einflusses vieler Faktoren schwer zu ermitteln. Ein geeignetes Mittelmaß wurde daher manuell vermessen.

Die Messungen der Verlangsamungsfaktoren der Evaluierungsprogramme zeigen, mit Ausnahme des Programms gpdotnet, dass die Instrumentierung lediglich eine reale Verlangsamung um den Faktor von 1,2 bis 55 (im Durchschnitt 17) bewirkt.

Nicht zu Letzt kann festgehalten werden, dass durch Fingerzeige in Algorithmia und Astrogrep Optimierungspotenzial gefunden wurde. Beim Umsetzen des Parallelisierungspotenzials wurde ebenfalls versucht durch sequenzielle Optimierungen Laufzeitverbesserungen zu erreichen. In diesen beiden Fällen ist der Erfolg deutlich erkennbar. Das bedeutet, dass Fingerzeige offenbar nicht nur Parallelisierungspotenzial, sondern auch sequenzielles Optimierungspotenzial orten können.

8 Zusammenfassung und Ausblick

Ziel dieser Diplomarbeit war die automatische Lokalisierung von Parallelisierungspotenzial anhand des zur Laufzeit beobachteten Datenstrukturzugriffsverhaltens in sequenziellen, objektorientierten Programmen.

Die Studie „Welche Datenstrukturen werden verwendet?“ hat gezeigt, dass Programmierer einige wenige Datenstrukturen benutzen. Lineare Datenstrukturen wie `List` und Array sind die meist benutzten Datencontainer, weshalb sich diese Arbeit auf diese beiden Datenstrukturen konzentriert. Die Studie „Treten Regelmäßigkeiten in Zugriffshistorien auf?“ hat gezeigt, dass regelmäßige Strukturen in Zugriffshistorien zu finden sind. Dies motivierte eine Beschreibungsmethodik in Form von Phasencharakteristika zur Formulierung solcher Regelmäßigkeiten. Dadurch wurde es möglich, nach Verhaltensmustern zu suchen und gefundene Parallelisierungspotenziale in Form von Fingerzeigen auszugeben.

Auf den Ergebnissen der Vorstudien aufbauend, gliedert sich das Konzept in zwei Schritte. Als Erstes werden Zugriffshistorien erzeugt. Das bestehende Programm wird dazu instrumentiert und ausgeführt. Im zweiten Schritt werden in den gespeicherten Historien Muster gesucht und daraus Fingerzeige abgeleitet. Dafür werden Zugriffsereignisse zunächst zu Phasen zusammengefasst. Ein Katalog beschreibt schließlich, wann Fingerzeige ausgegeben werden. Es wurde ein Werkzeug implementiert, welches dieses Konzept zur Evaluierung umsetzt.

Die Evaluierung anhand sieben unterschiedlicher Programme zeigt, dass durch die Umsetzung von Parallelisierungspotenzialen angeleitet durch Fingerzeige, Programmbeschleunigungen erzielt werden. Von den insgesamt 24 aufgezeigten Orten sind zwei Drittel richtig-positive Treffer und nur drei Fingerzeige sind korrekte Treffer mit falscher Begründung. Damit sind ca. 80 % der Fingerzeige aus den Testprogrammen zur Parallelisierung nützlich. Die erreichten Beschleunigungsfaktoren nach Umsetzung aller Hinweise liegen zwischen 1,2 und 3. Fünf Fingerzeige wiesen auf Datenstrukturen hin, welche bereits von den Entwicklern als parallelisierungstauglich gekennzeichnet wurden. Dies unterstreicht, dass mit Fingerzeigen reales Parallelisierungspotenzial geortet wird. Zudem wurden bei der Umsetzung der Fingerzeige auch sequenzielle Optimierungspotenziale gefunden.

Das bedeutet, dass Fingerzeige offenbar nicht nur Parallelisierungspotenziale, sondern auch Optimierungspotenziale orten können.

Diese Feststellung motiviert nachfolgende Forschungen an dieser Arbeit anzuknüpfen und weitere Fingerzeige zu untersuchen, die nicht nur paralleler Natur sein müssen. Zusätzlich zu weiteren Fingerzeigen können Zugriffsarten, Phasencharakteristika und zu beobachtende Datenstrukturen ergänzt werden. Der konzeptionelle Prozess ist davon unabhängig. Wie in Kapitel 6.5 beschrieben, ist die Implementierung darauf ausgelegt derartige Erweiterungen problemlos umzusetzen. Vor allem neuartige Phasencharakteristika ermöglichen neue gewinnbringende Fingerzeige.

Weiterhin können anknüpfende Arbeiten die Schwellenwerte der Fingerzeiggenerierung (siehe Kapitel 5.4.3) thematisieren. Mikrobenchmarks könnten Messungen vornehmen, mit denen die Schwellenwerte fundamentiert angepasst werden, sodass qualitativ hochwertigere Fingerzeige ausgegeben werden. Zudem ist ein System denkbar, welches automatischen die Schwellenwerte, beispielsweise im Hinblick auf die Zielhardware, einstellt.

Die Untersuchung der Qualität der Fingerzeige ist ein weiterer Aspekt zukünftiger Arbeiten. Messmethoden der Fingerzeigqualität müssen untersucht werden. In dieser Arbeit wurden die Testprogramme individuell und vor allem manuell inspiziert. Ein Automatismus ist wünschenswert. Dazu sind Verfahren nötig, welche vorhandenes Parallelisierungspotenzial objektiv messen und Fingerzeige bezüglich eines anzugebenen Optimierungsziels bewerten. Die Parameterwerte des Werkzeugs könnten mit einem solchen Automatismus sogar optimiert werden.

Zudem sind weitere Evaluationen denkbar. Es wäre interessant zu ermitteln, um wie viel kürzer die benötigte Arbeitszeit zur Parallelisierung mit Hilfe der Fingerzeige wird, gegenüber Parallelisierung ohne Fingerzeige oder mithilfe anderer Werkzeuge. Dazu könnte man drei Gruppen von Entwicklern damit beauftragen, ein ihnen unbekanntes Programm zu parallelisieren. Die erste Gruppe darf nur die Entwicklungsumgebung verwenden, die zweite Gruppe darf Fingerzeige verwenden und die dritte ein anderes Werkzeug. Benötigte Arbeitszeit, erreichte Beschleunigung und Angaben der Probanden zur gefühlten Nützlichkeit könnten untersucht und verglichen werden.

Nicht nur Erweiterungen dieser Arbeit sind denkbar, sondern auch eine Weiterentwicklung des Konzepts sowie der zukünftige Einsatz dieser Technologie in größeren Parallelisierungswerkzeugen. In einem nächsten Schritt könnte versucht werden das Wechselspiel von Datenstrukturen zu erfassen. Dies ermöglicht Aussagen über verschiedene Datenstrukturinstanzen als Einheit zu treffen. Ziel ist dabei die Findung von Parallelisierungspotenzial in solchen Gruppen, wenn die einzelnen Datenstrukturen für sich betrachtet kein Parallelisierungspotenzial bieten. Diese Konzeptidee ist schließlich auch für einen darüberliegenden Parallelisierungsprozess interessant, der beweisbare Aussagen über die Parallelisierbarkeit von Quelltext treffen kann. Die Ergebnisse der dynamischen Analysen

dieses Konzeptes ermöglichen dem Parallelisierer zielgerichtet laufzeitrelevante Stellen des Codes weiterzuverfolgen.

Abschließend wird konstatiert, dass aus dem Zugriffsverhalten auf Datenstrukturen werkzeugunterstützt Parallelisierungspotenziale geortet werden können, um manuelle Parallelisierung zu unterstützen. Die Implementierung des vorgestellten Ansatzes erreicht dies und bietet zudem Raum für zukünftige Erweiterungsmöglichkeiten, um dem Programmierer bei der Verbesserung ihm unvertrauter Programme zu helfen.

ANHÄNGE

A. Ergänzungen

	#classes*	List #var	List #T	LinkedList #var3	LinkedList #T4	ArrayList #var5	ArrayList #T6	Dictionary #var7	Dictionary #T8	Stack #var9	Stack #T10	Queue #var11	Queue #T12	HashSet #var13	HashSet #T14	SortedDictionary #var15	SortedDictionary #T16	SortedList #var17	SortedList #T18	SortedSet #var19	SortedSet #T20	KeyedByTypeCollection	Hashtable
dotspatial	1500	400	90	2	1	150		80	42	7	3	1	1	2	2	3	2	14	2	4	2		
OsmExplorer	227	100	45					30	13	9	3	3	1	13	5	2	2	6	2	6	1		
ManicDigger2011	220	100	50			1		45	30	2	2	5	3										
theAirline	200	100	35					30	25														
graphsharp	150	60	15					60	20	4	3	16	7	20	7								
greatmaps	250	50	25			2		20	15	1	1	3	3			1	1						
cognitionmaster	200	50	20					8	5			2	1										
SequenceViz	100	50	25			1		5	5			1	1										
dddpds (SmartCA)	250	30	25					4	3														
netinfotrace	100	30	25																				
gpdotnet	45	30	6							5	2	2	1										
evo	40	30	5					1	1														
csparser	200	25	32	1	1			11	7	14	9												
waveletstudio	60	25	9					2	2							1							
ittycoon.net	54	25	17					2	2														
clipper	25	20	8																				
borys-MeshRouting	42	19	9																				
orazio1	65	15	9			2		13	10					1	1	1							
metaclip	25	14	5																				
TreeLayoutHelper	18	14	5							2	1	6	3										
TerraBIB	60	13	9																				
Contentfinder	27	11	2																				
sharpener	25	10	4					6	5														
compgeo	12	9	2							4	2												
dotqcf	350	8	7			27																	
‚dsa'	20	8	2									1	1	1	1								
twodsphsim	15	8	2																				
rushHour	25	5	3					1	1			1	1	1	1								
rrsroguelike	12	5	4																				
fire	24	4	4					4	2														
ProcessHacker	21	4	4																				
BeHappy	55	1	1			5		1	1														
Arcanum	14	1	1					1	1														
Net_With_UI	9	1	1																				
zedgraph	100	0	0			2																	
starsystemsimulator	18	0	0							1	1												
7zip	11	0	0			2																	

Tabelle 7: Softwareprojekte mit der jeweiligen Anzahl enthaltener Variablen (#var) und der Anzahl unterschiedlicher generischer Parametrisierungen (#T) je Datenstruktur

	LE	IS	SnE	HS	HlL	Σ
QIT	6	1			1	**8**
ManicDigger2011	3	1	1		1	**6**
csparser	5					**5**
clipper	4			1		**5**
gpdotnet	4				1	**5**
CPU-Benchmark (netlinwhetcpu)	3				2	**5**
Mandelbrot	3					**3**
quickgraph	3					**3**
astrogrep				2	1	**3**
mesh routing		1			2	**3**
Contentfinder	2					**2**
DambachMulti	2					**2**
LinearAlgebra	2					**2**
MathNetIridium	2					**2**
Net_UI	2					**2**
fire	1				1	**2**
DesktopSuche	1					**1**
FIPL	1					**1**
FreeFlowSPH	1					**1**
networkminer	1					**1**
rrrsroguelike	1					**1**
WordWheelSolver	1					**1**
wordSorter	1					**1**
Algorithmia					1	**1**
adamrmoss-DataStructures						**0**
alglib						**0**
AlphaBetaSearch						**0**
Antisocial Robots						**0**
Arcanum						**0**
ClostestPair						**0**
compgeo						**0**
dsa						**0**
evo						**0**
Gauss						**0**
graphsharp						**0**
Matrix Parallel						**0**
NDamen						**0**
ogazitt-DataStructures						**0**
orazio (mAdcOW)						**0**
purelyFunctional-ds						**0**
royosherove (Regulator)						**0**
rushHour						**0**
sharpener						**0**
Sift						**0**
Starsystemsimulator						**0**
tic tac toe						**0**
TreeLayoutHelper						**0**
winrichcopy						**0**
Σ	**49**	**3**	**1**	**3**	**10**	**66**

Tabelle 8: Übersicht über die Anzahl der ausgegebenen Fingerzeige bei verschiedenen Programmen

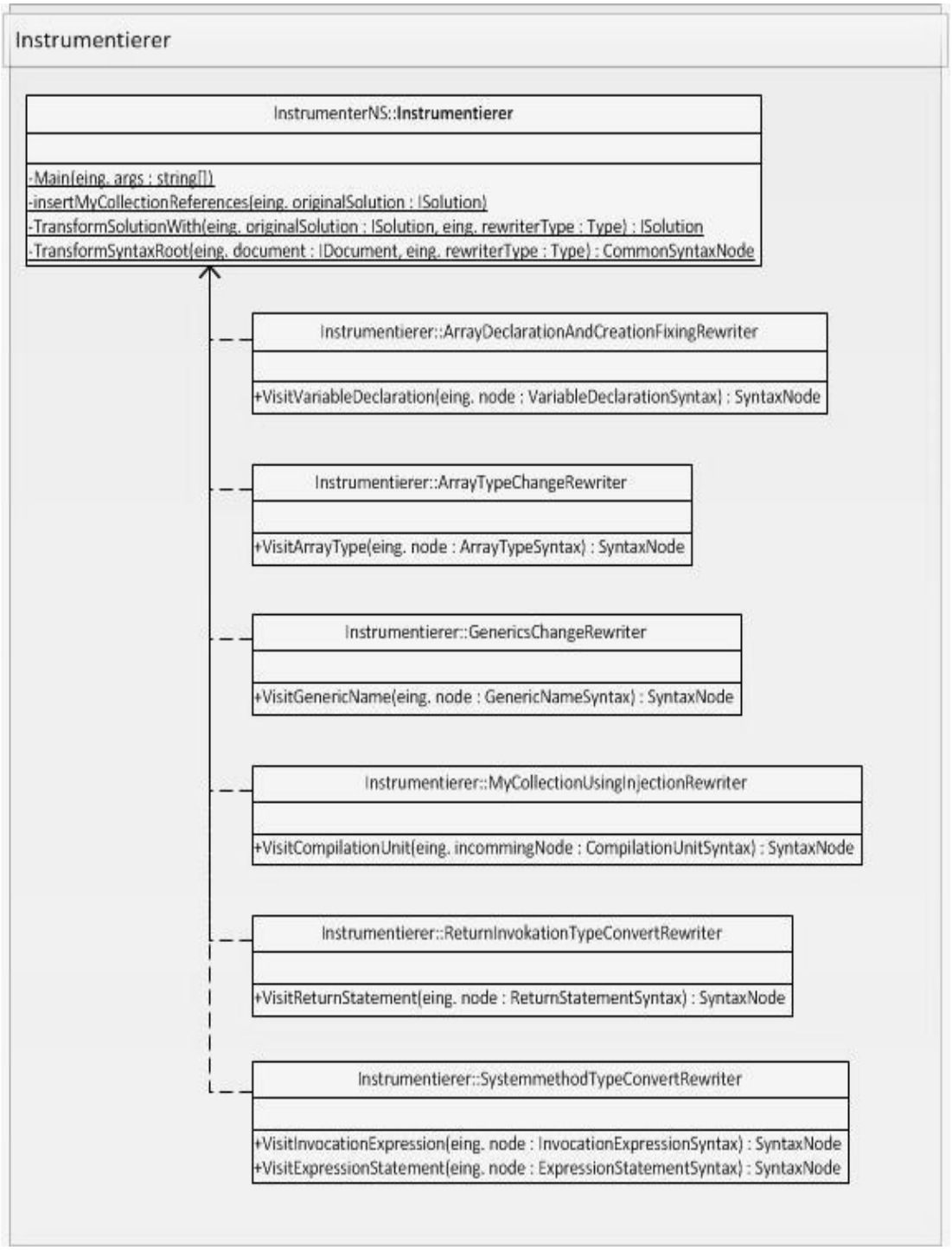

Abbildung 17: Klassendiagramm des Pakets Instrumentierer

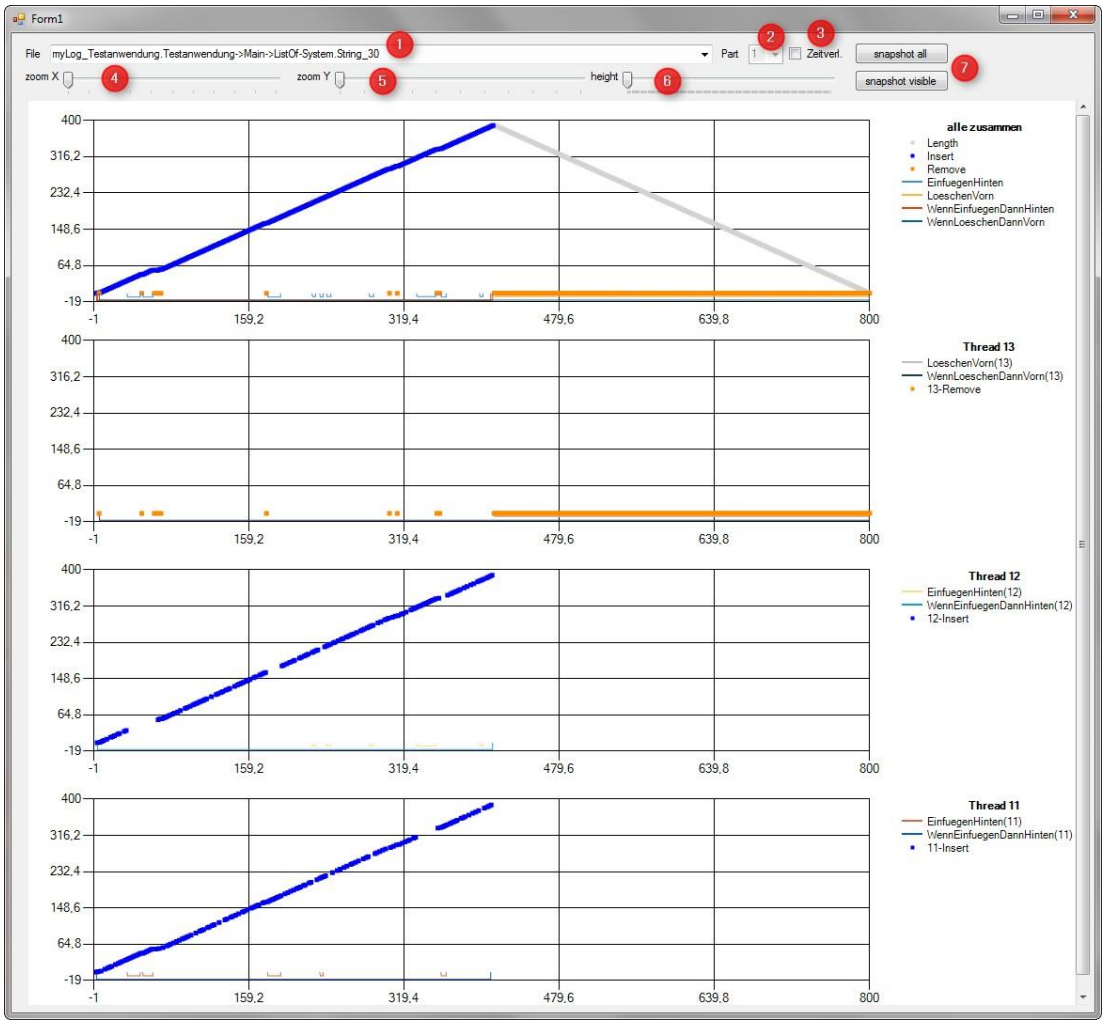

Abbildung 18: Bildschirmfoto der Visualisierung einer Schlange

Zu Abbildung 18: Dargestellt ist die Historie einer Liste, die als gemeinsame Schlange in einem parallelen Erzeuger-Verbraucher-Programm eingesetzt wird. Das oberste der vier Diagramme zeigt alle Zugriffe der Fäden in einer Historie an. Die unteren drei Koordinatensysteme beinhalten die getrennten Historien des Verbrauchers und der zwei Erzeuger. Die Erzeuger fügen dabei je 200 Elemente in die Schlange ein. Die x-Koordinaten der Datenpunkte zwischen den Koordinatensystemen sind synchron. Dadurch kann man sehr gut sehen, wie die beiden Erzeuger parallel in die Schlange einfügen, während der Verbraucher bereits einzelne Elemente herausnimmt.

B. Abkürzungsverzeichnis

Abkürzung	Langbezeichnung bzw. Begriffserklärung
A0, …, A6	Anforderungen, siehe Kapitel 5.1.1 und 5.1.2
HlL	Fingerzeig: Häufig langes Lesen, siehe Kapitel 5.4.3
HS	Fingerzeig: Häufiges Suchen, siehe Kapitel 5.4.3
IS	Fingerzeig: Implementierung einer Schlange, siehe Kapitel 5.4.3
IS1, IS2	Implementierung der Teilkonzepte KS1 und KS2, siehe Kapitel 6.2 und 6.3
KS1, KS2	Teile des Konzepts bezüglich der Schritte S1 und S2, siehe Kapitel 5.3 und 5.4
LE	Fingerzeig: Langes Einfügen, siehe Kapitel 5.4.3
LOC	Anzahl Quelltextzeilen (engl.: lines of code)
S1, S2	Teilschritte des Konzepts, siehe Kapitel 5.1
SnE	Fingerzeig: Sortierung nach dem Einfügen, siehe Kapitel 5.4.3

C. Abbildungsverzeichnis

D. Quelltextverzeichnis

E. Tabellenverzeichnis

F. Literaturverzeichnis

[AC+09]	J. Ansel, C. Chan, Y. L. Wong “PetaBricks: A Language and Compiler for Algorithmic Choice” Programming language design and implementation PLDI (2009) 38-49
[CC+11]	J. Ceng, J. Castrillon, W. Sheng “MAPS: an integrated framework for MPSoC application parallelization” Design Automation Conference DAC (2011) 754-759
[DV94]	R. Dekker, F. Ververs “Abstract data structure recognition” Knowledge-Based Software Engineering Conference KBSE (1994) 133-140
[H11]	J. Huck “Automatisierte Parallelisierung mit Auto-Futures“ Karlsruher Institut für Technologie (KIT), Institut für Programmstrukturen und Datenorganisation (IPD), Lehrstuhl für Programmiersysteme (2011)
[IS07]	A. Itai, M. Slavkin “Detecting Data Structures from Traces” Workshop on Approaches and Applications of Inductive Programming AAIP (2007) 39-50
[JC09]	C. Jung, N. Clark “DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage” International Symposium on Microarchitecture MICRO (2009) 56-66
[JR+11]	C. Jung, S. Rus, B. P. Railing, N. Clark, S. Pande “Brainy: Effective Selection of Data Structures” Conference on Programming Language Design and Implementation PLDI (2011) 86-97

[KM04]	A. J. Ko, B. A. Myers “Designing the whyline: a debugging interface for asking questions about program behavior” SIGCHI Conference on Human Factors in Computing Systems CHI (2004) 151-158
[LH+11]	Y. Liu, Z. Hu, K. Matsuzaki “Towards Systematic Parallel Programming over MapReduce“ Parallel processing - Volume Part II Euro-Par (2011) 39-50
[MS93]	S. Mukherjea, J. T. Stasko “Applying algorithm animation techniques for program tracing, debugging, and understanding” International conference on Software Engineering ICSE (1993) 456-465
[RA05]	E. Raman, D. I. August “Recursive data structure profiling” ACM SIGPLAN Workshop on Memory Systems Performance MSP (2005) 5-14
[RB11]	A. Rane, J. Browne “Performance optimization of data structures using memory access characterization” Cluster Computing CLUSTER (2011) 570-574
[SG96]	B. Sinharoy, R. Govindaraju “Improving Software MP Efficiency for Shared Memory Systems” Hawaii International Conference on System Sciences HICSS (1996) 111-120
[TB+11]	G. Tanas, A. Buss, A. Fidel “The STAPL parallel container framework” Principles and practice of parallel programming PPoPP (2011) 235-246
[TF10]	G. Tournavitis, B. Franke “Semi-Automatic Extraction and Exploitation of Hierarchical Pipeline Parallelism Using Profiling Information” Parallel architectures and compilation techniques PACT (2010) 377-388
[WM98]	R. J. Walker, G. C. Murphy “Visualizing dynamic software system information through high-level models” ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications OOPSLA (1998) 271-283
[ZD+11]	Y. Zhang, W. Ding, J. Liu, M. Kandemir “Optimizing Data Layouts for Parallel Computation on Multicores” Parallel Architectures and Compilation Techniques PACT (2011) 143-154