Karlsruhe Institute of Technology

Automatische Performanzoptimierung Paralleler Architekturen

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften/
Doktors der Naturwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

vorgelegte

Dissertation

von

Christoph A. Schaefer

Tag der mündlichen Prüfung: 30. Juni 2010

Erstgutachter: Prof. Dr. Walter F. Tichy
Zweitgutachter: Prof. Dr. Ralf H. Reussner

Die Parallelisierung von sequentiellen Anwendungen sowie die anschließend erforderliche Optimierung stellt Software-Entwickler vor große Herausforderungen.

Diese Arbeit befasst sich daher mit Problemstellungen im Bereich der automatischen Performanzoptimierung (Auto-Tuning) paralleler Architekturen. Die Grundidee des Auto-Tuning wird derart erweitert, dass nicht mehr nur algorithmische Programme, sondern auch komplexe parallele Architekturen mittels eines automatisierten Verfahrens entworfen, implementiert und optimiert werden können – unabhängig von Größe, Anwendungsgebiet oder Zielplattform der Applikation.

Hierzu werden neben einer Instrumentierungssprache zur Spezifikation von Tuning-Instruktionen im Programmquelltext ein Verfahren für den Entwurf paralleler optimierbarer Architekturen sowie deren automatisierte Implementierung vorgestellt; das Konzept eines suchbasierten Auto-Tuners für parallele Architekturen rundet die Arbeit ab.

Die Funktionalität der Konzepte wird an Hand einer prototypischen Implementierung unter Beweis gestellt, während die Verfahren mittels Fallstudien evaluiert werden. Die experimentellen Ergebnisse erweisen sich als vielversprechend und belegen die Unentbehrlichkeit der Kombination aus Parallelisierung und Optimierung.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Die vorliegende Arbeit zeigt Problemstellungen im Bereich der automatisierten Performanzoptimierung paralleler Applikationen auf, erörtert geeignete Lösungsansätze und Konzepte und stellt eine prototypische Implementierung vor. Des Weiteren wird die Zielsetzung und der Beitrag dieser Arbeit in den Kontext existierender Ansätze eingeordnet. Mittels Fallstudien werden experimentelle Ergebnisse ermittelt, welche die Effizienz des Prototyps und der zu Grunde liegenden Konzepte belegen, sowie deren Nutzen für die Softwaretechnik unterstreichen.

1.1 Motivation

Bereits 1965 formulierte Gordon Moore in einem Artikel der Zeitschrift *Electronics* die Annahme, dass sich die Anzahl der Transistoren auf einem Prozessor-Chip jährlich verdoppeln werde [Moor65]. Zehn Jahre später korrigierte er seine Angabe auf eine Verdopplung innerhalb einer Zeitspanne von zwei Jahren [Moor75]. Diese Prognose über die Transistordichte wurde als das *Mooresche Gesetz* bekannt. Im Jahre 2007 versicherte Moore, dass sein Gesetz für weitere 10 bis 15 Jahre gültig sein werde. Erst dann seien physikalische Grenzen bei der Herstellung von Transistoren erreicht, die es unmöglich machen, noch weitere Transistoren auf einer gleichgroßen Chip-Fläche unterzubringen.

Allerdings stoßen die Hersteller von Mikroprozessoren bereits heute an physikalische Grenzen. Jedoch stellen nicht etwa die immer kleineren Transistoren das größte Problem dar, sondern die Abwärme, die ein Prozessor produziert. Abbildung 1.1 veranschaulicht den Anstieg der Energiedichte bei Mikroprozessoren. Der Grund für den steilen Anstieg der Wärmeentwicklung liegt in der immer weiter gestiegenen Taktrate der Prozessoren, die 2002 bei etwa 4 GHz stagnierte.

Da eine Leistungssteigerung der Prozessoren über die Erhöhung der Taktrate aus besagten Gründen nicht mehr möglich ist, haben die Hersteller einen anderen Weg eingeschlagen. Der Zugewinn an Leistung soll nun über Hardwareparallelisierung erreicht werden, indem mehrere unabhängige Prozessorkerne in einem Chip integriert werden. Man spricht hierbei von Mehrkernprozessoren. Die Prozessorkerne können parallel arbeiten. Im Jahr 2005 startete der Verkauf von von Mehrkernprozessoren in PCs; Einkernprozessoren sind in heutigen PCs gar nicht mehr anzutreffen.

Obwohl die Taktrate der einzelnen Kerne deutlich herabgesenkt wurde, kann – zumindest theoretisch – mit zwei Prozessorkernen die doppelte Rechenleistung (und damit die

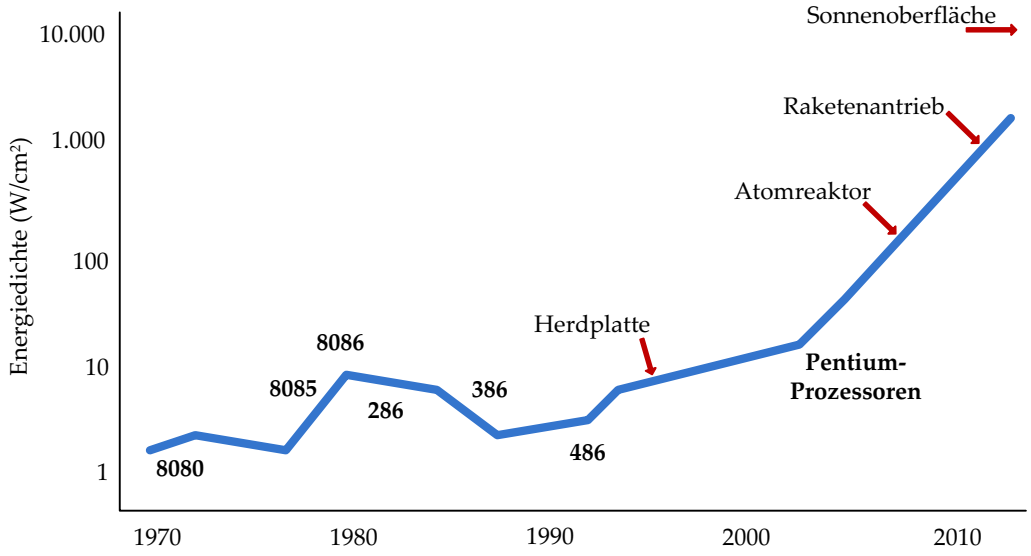


Abbildung 1.1: Anstieg der Energiedichte bei Mikroprozessoren (nach [Gels04]).

doppelte Geschwindigkeit) erzielt werden. Entsprechend steigt die Leistung bei Prozessoren mit vier oder gar acht Kernen. Diesen Leistungsschub bekommen Entwickler sowie Anwender von Programmen allerdings nicht mehr geschenkt. Herb Sutter beschrieb diese Tatsache mit dem folgenden Zitat:

> *„The free lunch is over."* [Sutt05]

Damit Programme die Mehrkernarchitektur nutzen können, müssen diese derart strukturiert sein, dass Berechnungen oder ganze Programmteile auf den zur Verfügung stehenden Kernen parallel ablaufen können; die Programme müssen *parallelisiert* werden.

Die Parallelisierung von sequentiellen Anwendungen (also Programme, die noch nicht parallel arbeiten können) oder auch die Neuentwicklung paralleler Anwendungen stellt jedoch die Software-Entwickler vor eine Herausforderung, da aktuelle Techniken zur Software-Parallelisierung eher rudimentär ausfallen. Dies führt dazu, dass parallelisierte Programme selten unmittelbar den gewünschten Geschwindigkeitsgewinn erzielen. Vielmehr muss das Programm hinsichtlich der Hardware-Plattform, auf der es eingesetzt werden soll, auf Performanz optimiert werden. Eine Software-Parallelisierung ohne gezielte Optimierung führt meist nur zu einer schlechten bis mittelmäßigen Leistungsausbeute der Prozessorkerne.

Im nächsten Abschnitt wird das Problem der Optimierung paralleler Applikationen vorgestellt. Außerdem wird dargelegt, weshalb Ansätze zur Lösung dringend benötigt werden.

1.2 Problemstellung

Die starke Verbreitung der Mehrkernrechner zwingt zu einem Umdenken in der Softwaretechnik. Die Entwicklung paralleler Anwendungen rückt in den Mittelpunkt des Interesses, da nur durch effiziente und geschickte Parallelisierung die steigende Zahl an Prozessorkernen ausgenutzt und ein entsprechender Geschwindigkeitsgewinn erreicht werden kann.

Gängige Techniken zur Parallelisierung von Programmen, wie zum Beispiel die Verwendung mehrerer Ausführungsfäden und entsprechende Synchronisations-Mechanismen dienen jedoch nur dem Zweck, potentielle Nebenläufigkeiten des Programms auszunutzen, so dass unabhängige Aufgaben oder zustandslose Berechnungen auf großen Datenstrukturen parallel ausgeführt werden können. Innerhalb des Programms entstehen auf diese Weise so genannte parallele Sektionen.

Problematisch ist nun, dass parallele Sektionen je nach Art des Programms und Eigenschaften der zu Grunde liegenden Hardware-Architektur (beispielsweise die Anzahl der Prozessorkerne oder die Größe der Caches) unterschiedlich konfiguriert werden müssen, um dem Programm zu bestmöglicher Leistung zu verhelfen. Die Konfiguration einer parallelen Sektion erfolgt über Parameter, welche die Leistung der Sektion und damit des gesamtem Programms beeinflussen. Solche Parameter werden *Tuning-Parameter* genannt. Typische Tuning-Parameter sind zum Beispiel die Anzahl der Ausführungsfäden, die für eine Berechnung verwendet werden, die Größe der Datenblöcke, die parallel verarbeitet werden können, oder die Wahl der Lastenausgleichsstrategie, um alle Ausführungsfäden möglichst gleichmäßig mit Arbeit zu versorgen.

Da eine parallele Sektion mehrere Tuning-Parameter und ein Programm wiederum mehrere parallele Sektionen enthalten kann, steigt die Anzahl an Tuning-Parametern und damit die Anzahl an Möglichkeiten, auf die Leistung des Programms Einfluss zu nehmen, sehr schnell. In diesem Zusammenhang ergeben sich nun die folgenden Probleme:

- Welches sind leistungsrelevante Tuning-Parameter eines Programms, wie identifiziert und spezifiziert man diese? In einem existierenden Programm ohne bekannte Struktur kann es schwer sein, die wichtigen „Stellschrauben“ zu finden.
- Ein Tuning-Parameter entspricht einer Variablen im Programm. Wird der Wert der Variable geändert, muss das Programm entsprechend reagieren und unter Umständen sein Verhalten ändern. Hinsichtlich der Tuning-Parameter muss das Programm adaptierbar sein. Es stellt sich daher die Frage, welche Konzepte einem Software-Entwickler an die Hand gegeben werden können, um parallele optimierbare Programme effizient zu entwerfen und zu implementieren.
- Einen entscheidenden Punkt stellt die Optimierung selbst dar. Auf Grund der großen Zahl an Tuning-Parametern, die in einem Programm existieren können, sowie deren potentieller Abhängigkeit, wird die Menge an möglichen Konfigurationen sehr groß. Dies macht eine manuelle Optimierung durch sukzessives Ausprobieren nahezu unmöglich. Es müssen also Wege gefunden werden, die Optimierung ganzer Applikationen automatisiert durchzuführen. Hierbei spielen die Dauer des Optimierungsprozesses sowie die Qualität des Ergebnisses eine wichtige Rolle. Der Prozess der automatisierten Performanzoptimierung wird *Auto-Tuning* genannt.
- Die große Menge an Tuning-Parametern führt außerdem dazu, dass selbst automatisierte Suchstrategien nur mit sehr hohem zeitlichen Aufwand eine hinreichend gute Konfiguration für ein Programm finden. Es ist daher notwendig, den Suchraum durch Ausnutzung von Kontext- und Strukturinformationen über das Programm bereits im Vorfeld der eigentlichen Optimierung derart zu reduzieren, dass die verwendete Suchstrategie nach Möglichkeit nur noch den minimal notwendigen Suchraum zu bearbeiten hat.

In früheren Arbeiten [PSJT08, ScPT09] konnten wir bereits zeigen, dass parallelisierte Programme allein durch Auto-Tuning bis zu 450% Leistungssteigerung erfahren haben.

Diese Tatsache macht deutlich, dass Auto-Tuning gerade im Bereich der parallelen Programmierung nicht fehlen darf.

In Kapitel 4 werden die wichtigsten Vertreter verwandter Arbeiten vorgestellt und diskutiert. Grundsätzlich kann jedoch bereits an dieser Stelle festgehalten werden, dass die meisten existierenden Ansätze sich auf das Optimieren spezieller Algorithmen oder kleiner Programme aus bestimmten Anwendungsbereichen konzentrieren. Dies gilt sowohl für den Entwurf paralleler und optimierbarer Programme als auch für das eigentliche Auto-Tuning.

Die Grundidee des Auto-Tuning muss also derart erweitert werden, dass Konzepte entstehen, die von softwaretechnischer Relevanz sind und auf Applikationsebene sowie beim architektonischen Entwurf des Programms eingesetzt werden können.

1.3 Gliederung der Arbeit

Die Arbeit ist wie folgt gegliedert. In Kapitel 2 werden Zielsetzung und Beitrag erläutert und die zentralen Thesen aufgestellt. Kapitel 3 gibt eine Einführung in das Thema der automatischen Performanzoptimierung; die verwandten Arbeiten werden in Kapitel 4 diskutiert. Kapitel 5 erläutert die Konzepte und Lösungsansätze der Arbeit und Kapitel 6 die zugehörigen Implementierungstechniken. Anschließend werden in Kapitel 7 die vorgestellten Konzepte evaluiert und entsprechende Ergebnisse präsentiert. Die Arbeit schließt mit einer Zusammenfassung sowie einem Ausblick auf mögliche aufbauende Forschungsarbeiten (Kapitel 8).

2. Zielsetzung und Beitrag

In diesem Kapitel wird die Zielsetzung der Arbeit beschrieben, die sich an der im vorherigen Kapitel besprochenen Problemstellung orientiert. Des Weiteren wird der Beitrag der Arbeit für den Forschungsbereich der automatisierten Performanzoptimierung paralleler Programme dargelegt.

2.1 Zielsetzung

Die meisten existierenden Arbeiten und Forschungsprojekte im Bereich des Auto-Tuning setzen ihren Schwerpunkt auf numerische Programme, auf Algorithmen im Umfeld des *High-Performance-Computing (HPC)*, sowie auf domänenspezifische Applikationen und Erweiterungen von Übersetzern (engl. *compiler*). Diese Arbeiten sind in Kapitel 4 Gegenstand unserer Diskussion.

Durch die Verbreitung von Mehrkernrechnern sind nicht mehr nur Anwendungen im HPC-Bereich und in der Algorithmentechnik im Fokus von Parallelisierung und Optimierung, sondern insbesondere auch komplexe, große Applikationen, deren Entwurf und Entwicklung in den Bereich der Softwaretechnik fallen. Eine wichtige Zielsetzung der Arbeit umfasst daher die Erweiterung der Grundidee des Auto-Tunings: Die Konzepte zur automatischen Performanzoptimierung müssen von konkreten Anwendungsbereichen und Hardware-Plattformen abstrahieren und für umfangreiche und komplexe Applikationen ausgelegt sein.

In Abbildung 2.1 ist die Zielsetzung der Arbeit schematisch dargestellt. Mit der Erweiterung des Auto-Tuning-Ansatzes sind drei Teilkonzepte verbunden, deren Entwurf und Umsetzung den Rahmen dieser Arbeit bilden sollen.

2.1.1 Teilkonzept 1

Der erste Schwerpunkt der Arbeit ist der Entwurf einer Sprache zur Spezifikation von Tuning-Instruktionen im Quelltext des parallelen Programms. Diese so genannte *Tuning-Instrumentierungssprache* soll es ermöglichen, beliebige Tuning-Parameter im Programm, aber auch Messpunkte zur Ermittlung der Performanz in möglichst einfacher Weise zu deklarieren. Folgende Aspekte und Anforderungen sind zu berücksichtigen:

- Einfache Integration in verschiedene Wirtssprachen: Dies erfordert eine starke Trennung zwischen Tuning- und Wirtssprache.

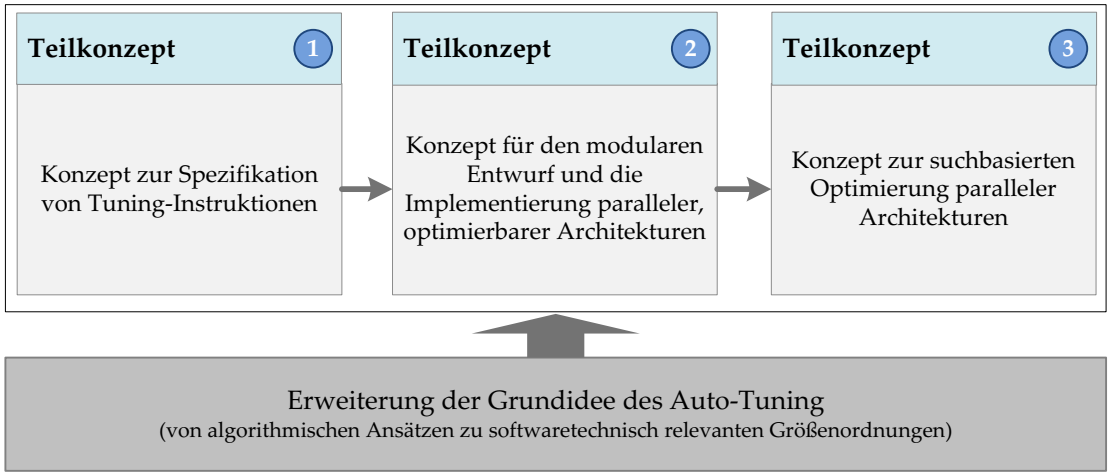


Abbildung 2.1: Darstellung der Teilkonzepte und deren Zusammenhang als Zielsetzung dieser Arbeit.

- Im Gegensatz zu den meisten existierenden Ansätzen muss das Sprachkonzept die Deklaration von Tuning-Parametern *und* Messpunkten unterstützen.
- Bereitstellen von Sprachkonstrukten zur Abbildung der Programmstruktur: hierzu gehören insbesondere die Deklaration von Parameterabhängigkeiten sowie das Markieren von parallelen Sektionen und den darin verwendeten Parallelisierungsstrategien.

Die Beschreibung der Konzepte und Lösungsansätze zum Teilkonzept der Tuning-Instrumentierungssprache werden in Kapitel 5.2 erörtert.

2.1.2 Teilkonzept 2

Aktuelle Untersuchungen [PSJT08, PaJT09, Scha09] haben gezeigt, dass bei größeren parallelen Programmen die Anpassung von Tuning-Parametern alleine nicht ausreicht, um eine optimale Variante der Anwendung zu erhalten.

Vielmehr sind zusätzlich Adaptionen auf architektonischer Ebene erforderlich, welche die Auswahl der Parallelisierungsstrategien sowie deren Kombinationen beeinflussen. Eine aus vordefinierten grobgranularen Parallelisierungsbausteinen zusammengesetzte Architektur kann die Effizienz des Optimierungsprozesses erheblich steigern. Als Grundlage und Modell für derartige Programmbausteine bieten sich parallele Entwurfsmuster [MaSM04] an, die um Tuning-Parameter und entsprechende Adaptierbarkeit erweitert werden.

Die nötigen Entwurfsentscheidungen eines Entwicklers beschränken sich somit auf (i) die Identifikation von Parallelisierungspotential im Programm und (ii) darauf, welche Parallelisierungsstrategie verwendet werden soll, um das Potential auszuschöpfen. Von konkreten Parallelisierungstechniken sowie vom Optimierungsprozess wird abstrahiert.

Ein Schwerpunkt der Arbeit ist daher die Entwicklung und Umsetzung eines Konzeptes zum Entwurf paralleler optimierbarer Programme und deren Implementierung. Folgende Ziele sind hierbei zu erreichen:

- Identifikation geeigneter paralleler Entwurfsmuster.
- Performanzanalyse der gewählten parallelen Entwurfsmuster und Herleitung leistungsrelevanter Tuning-Parameter mit dem Ziel, Spezifikationen parametrisierter Parallelisierungsstrategien zu entwerfen.
- Entwicklung eines Architektur-Entwurfskonzeptes in Form einer Beschreibungssprache. Ausgehend von sequentiellen, atomaren Programmteilen sollen Konzepte bereitgestellt werden, mit denen die Interaktion sowie die Art der parallelen Verarbeitung dieser Programmteile beschrieben werden können.
- Entwicklung eines Konzeptes zur Umsetzung der Architekturbeschreibung in ein lauffähiges, paralleles, optimierbares Programm. Ein wichtiger Punkt ist hierbei die Integration der Tuning-Instrumentierungssprache (vgl. Abschnitt 2.1.1), so dass bei der Umsetzung automatisch die benötigten Instrumentierungen erzeugt werden.

Eine parallele und gleichzeitig konfigurierbare Architektur bezeichnen wir im Folgenden als *optimierbare Architektur*. Die Beschreibung der Konzepte und Lösungsansätze für den Entwurf und die Implementierung optimierbarer Architekturen werden in Kapitel 5.3 erläutert.

2.1.3 Teilkonzept 3

Der dritte Schwerpunkt der Arbeit ist die automatische Performanzoptimierung selbst. Hierbei gilt es insbesondere, ein Optimierungskonzept zu entwerfen, welches für parallele Anwendungen im Allgemeinen anwendbar ist. Der hieraus entstehende Optimierer (*Auto-Tuner*) muss in der Lage sein, die Informationen der Tuning-Instrumentierungssprache zu verarbeiten (siehe Abschnitt 2.1.1). Folgende Anforderungen sind hierbei zu beachten:

- Erweiterung der Auto-Tuning-Idee zu einem suchbasierten Optimierungsprozess für allgemeine parallele Programme.
- Analyse und Verarbeitung der Tuning-Instruktionen, die aus den Instrumentierungen des parallelen Programms gewonnen wurden.
- Analyse von Suchstrategien hinsichtlich deren Anwendbarkeit und Effizienz im Bereich Auto-Tuning und anschließende Einbindung in den Optimierungsprozess.
- Ausnutzen der optimierbaren Architekturen (vgl. Abschnitt 2.1.2): Verwendung von Heuristiken, um das Kontextwissen über verwendete Parallelisierungsstrategien und deren Tuning-Parameter sowie der gesamten Programmstruktur auszunutzen. Es sollen ausschließlich notwendige Kombinationen von Tuning-Parametern in den Suchraum aufgenommen werden, um das so genannte *Black Box Tuning* zu vermeiden.

Die Beschreibung der Konzepte und Lösungsansätze zum Teilkonzept des suchbasierten Auto-Tunings werden in Kapitel 5.4 behandelt.

Wie aus Abbildung 2.1 bereits ersichtlich, bauen die drei vorgestellten Teilkonzepte jeweils aufeinander auf und ergeben ein umfassendes Gesamtkonzept. Fasst man die Teilkonzepte zusammen, so besteht die Zielsetzung der Dissertation darin, den gesamten Prozess der Parallelisierung und Optimierung – angefangen beim Entwurf einer Optimierbaren Architektur bis hin zur optimierten parallelen Applikation – durch erweiterte und neuartige Methoden zu unterstützen bzw. zu ermöglichen.

Die Teilkonzepte dieser Arbeit lassen sich im Software-Entwicklungsprozess den Phasen *Entwurf* (Teilkonzept 2), *Implementierung* (Teilkonzept 1 und 2) und *Testen* (Teilkonzept 3) zuordnen. Die Umsetzung der Konzepte dient daher der Unterstützung des Software-Entwicklers bei Entwurf, Parallelisierung und Optimierung von Programmen.

2.2 Beitrag

Die Arbeit soll einen grundlegenden Beitrag im Bereich der automatisierten Performanzoptimierung auf Applikationsebene leisten, indem zunächst Auto-Tuning-Konzepte klassifiziert und hinsichtlich ihrer Anwendbarkeit auf allgemeine parallele Programme bewertet werden. Die Arbeit soll außerdem darlegen, welche Herausforderungen und Probleme bei der Erweiterung herkömmlicher Auto-Tuning-Ansätze entstehen.

Durch die Erarbeitung und Umsetzung der in Abschnitt 2.1 eingeführten Konzepte werden Methoden entwickelt, mit deren Hilfe der Entwurf und die Optimierung paralleler Architekturen auf softwaretechnischer Ebene erreicht werden können. Generell wird mit dieser Arbeit gezeigt, dass durch Abstraktion auch komplexe und große Applikationen parallelisiert und mit großer Genauigkeit optimiert werden können. Die Beschränkung auf numerische Anwendungen oder Algorithmen ist nicht mehr nötig.

Nicht zuletzt soll diese Arbeit die Grundlage für weitere Forschung bilden. Das vorgestellte Thema wurde im Rahmen dieser Arbeit tiefgehend untersucht und analysiert, um weiterführenden Arbeiten einen fachlich umfassenden Zugang zu bieten.

2.3 Thesen

Im Folgenden werden die zu Grunde liegenden Thesen dieser Arbeit aufgestellt. Sie ergeben sich aus den Zielsetzungen, die in den vorherigen Abschnitten besprochen wurden.

- **These 1.** Es ist möglich, mittels Auto-Tuning diverse Klassen paralleler Programme automatisch für Mehrkernarchitekturen zu optimieren. Eine Beschränkung auf konkrete Anwendungsbereiche wissenschaftlicher oder numerischer Applikationen ist nicht notwendig.
- **These 2.** Mit dem Einsatz einer Tuning-Instrumentierungs-Sprache ist es möglich, (i) neue und wichtige tuning-relevante Informationen zu spezifizieren, die für die effiziente automatische Optimierung paralleler Programme benötigt werden, und (ii) für den Tuning-Prozess relevante Komponenten vom zu optimierenden Programm zu trennen.
- **These 3.** Ausgehend von einem geeigneten Konzept zur Beschreibung von Programmarchitekturen können parallele, optimierbare Anwendungen entworfen und automatisiert umgesetzt werden, wobei von konkreten Parallelisierungsprimitiven und Optimierungsprozessen abstrahiert werden kann.
- **These 4.** Durch Ausnutzung von Kontextinformationen über die Struktur der Programms sowie über die Parallelisierungsstrategie ist es möglich, den Suchraum für einen Auto-Tuner derart zu reduzieren, dass auch große parallele Anwendungen mit suchbasierten Methoden optimiert werden können.

Die Thesen werden durch experimentelle Ergebnisse belegt. Die hierfür durchgeführten Fallstudien werden in Kapitel 7 ausführlich besprochen.

3. Grundlagen der automatischen Performanzoptimierung

Das Gebiet der automatischen Performanzoptimierung (*Auto-Tuning*) umfasst ein weites Spektrum an unterschiedlichen Methoden und Konzepten. Allen gemein ist jedoch das Ziel, einen Algorithmus oder ein Programm hinsichtlich bestimmter Leistungskriterien durch Modifikation relevanter Tuning-Parameter automatisiert zu optimieren.

In diesem Kapitel werden zunächst die wichtigsten Grundbegriffe des Auto-Tuning betrachtet sowie die gängigen Strategien zur Ermittlung des Leistungsoptimums dargestellt. Des Weiteren werden verschiedene Auto-Tuning-Verfahren analysiert sowie klassifiziert und die Einführung einer entsprechenden Taxonomie gegeben. Danach werden die Grundlagen optimierbarer Programme und deren Architektur besprochen und der Nutzen paralleler Entwurfsmuster als Architekturbausteine erläutert. Schließlich soll die Relevanz der Eingabedaten für Tuning-Verfahren diskutiert werden.

3.1 Grundbegriffe

Im Folgenden werden alle wichtigen Grundlagen der automatischen Performanzoptimierung aufeinander aufbauend definiert, so dass ein in sich stimmiges Grundgerüst entsteht, das die formale Basis für die Konzepte dieser Arbeit bildet.

3.1.1 Tuning-Parameter

Ein Tuning-Parameter repräsentiert das grundlegende Konstrukt zur Optimierung von Programmen.

Definition 3.1 *Ein Tuning-Parameter p ist eine Programmvariable mit einer diskreten Wertemenge V_p. Ein $v \in V_p$ wird als ein Parameterwert von p bezeichnet. Durch die Zuweisung eines Parameterwertes $v \in V_p$ zu p ändert sich das Verhalten des Programms, wobei die Semantik des Programms erhalten bleibt.*

Definition 3.2 *Ein Tuning-Parameter p besitzt einen Standardwert, der durch $def(p) \in V_p$ definiert ist.*

Ein Programm kann beliebig viele Tuning-Parameter besitzen, wobei jeder das Verhalten des Programms auf andere Weise beeinflussen kann. Ein oft verwendeter Tuning-Parameter in parallelen Programmen ist beispielsweise die Anzahl an Fäden, die das Programm zur Durchführung einer Berechnung nutzt.

Tuning-Parameter besitzen wie alle Variablen ein Skalenniveau. Im Hinblick auf die spätere Optimierung ist die Unterscheidung zwischen der Nominal- und der Ordinalskala relevant.

Ein Tuning-Parameter ist nominalskaliert, wenn seine möglichen Werte zwar unterschieden, nicht aber in eine Rangfolge gebracht werden können. Beispielsweise ist ein Parameter, der eine Lastausgleichsstrategie mit dem Wertebereich {`'static'`,`'dynamic'`, `'workstealing'`} bestimmt, nominalskaliert, da wir seine Werte zwar unterscheiden, also auf Gleichheit oder Ungleichheit überprüfen, diese aber nicht der Größe nach sortieren können.

Ein ordinalskalierter Tuning-Parameter besitzt einen Wertebereich, auf dessen Werten eine natürliche Rangordnung besteht. Ein Parameter, der die Anzahl an Ausführungsfäden für eine Berechnung bestimmt und zum Beispiel einen Wertebereich $\{2, 3, ..., 32\}$ besitzt, ist ordinalskaliert, da eine Ordnung $2 \leq 3 \leq ... \leq 32$ (Fäden) definiert ist.

3.1.2 Parameterkonfiguration

Eine Parameterkonfiguration K bezeichnet ein Tupel aus je einem Parameterwert aller Tuning-Parameter des Programms.

Definition 3.3 *Sei $P = \{p_1, p_2, ..., p_n\}$ die Menge aller Tuning-Parameter in einem Programm, so ist eine Parameterkonfiguration K definiert als ein n-Tupel $(v_i)_{i \in \{1,2,...,n\}}$, wobei $v_i \in V_{p_i}$ und $i \in \{1, ..., n\}$.*

Als Spezialfall führen wir die *maskierte Parameterkonfiguration* ein. In einer maskierten Parameterkonfiguration $\bar{K}_{p_i}$ kann p_i einen beliebigen Wert aus seiner Wertemenge V_{p_i} annehmen, während alle anderen Parameter auf ihren Standardwert def (siehe Definition 3.2) gesetzt sind.

Definition 3.4 *Die Menge $\bar{\mathcal{K}}_{p_i}$ aller maskierten Parameterkonfigurationen über p_i ist durch*

$$\bar{\mathcal{K}}_{p_i} = \{(def(p_1), ..., def(p_{i-1}), x, def(p_{i+1}), ..., def(p_n)) | x \in V_{p_i}\}$$

definiert.

Des Weiteren können wir auch die Menge der maskierten Parameterkonfigurationen über einer Untermenge der Tuning-Parameter in einem Programm definieren.

Definition 3.5 *Es sei $T \subset P$ eine Untermenge der Menge an Tuning-Parametern. Wir definieren zunächst die beiden Hilfsmengen*

$$L = \{p_i | p_i \in T, p_i = v \in V_{p_i}\}$$
$$M = \{p_j | p_j \in P \backslash T, p_j = def(p_j)\}$$

wobei $i \in \{1, ..., k\}$ und $j \in \{k + 1, ..., n\}$.

Alle Tuning-Parameter in T können beliebige Werte aus ihrer Wertemenge annehmen, alle übrigen Tuning-Parameter sind auf ihren Standardwerten fixiert.

Die Menge der maskierten Parameterkonfigurationen über T ergibt sich dann zu $\bar{\mathcal{K}}_T = L \cup M$.

Maskierte Parameterkonfigurationen werden verwendet, um die Auswirkungen eines einzigen Parameters oder einer Untermenge der Tuning-Parameter eines Programms zu untersuchen und alle übrigen Parameter auf ihren jeweiligen Standardwert zu fixieren.

3.1.3 Suchraum

Als Suchraum $\mathcal{S}$ wird das kartesische Produkt der Wertemengen aller Parameter in einem Programm bezeichnet.

Definition 3.6 *Sei $P = \{p_1, p_2, ..., p_n\}$ die Menge aller Tuning-Parameter in einem Programm und V_{p_i} der Wertebereich von p_i ($i \in \{1, 2, ..., n\}$), so ist der Suchraum durch*

$$\mathcal{S} = \prod_i V_{p_i} = V_{p_1} \times V_{p_2} \times ... \times V_{p_n}$$

definiert.

Definition 3.7 *Gemäß der Definition des kartesischen Produkts ist die Größe $|\mathcal{S}|$ des Suchraums durch*

$$|\mathcal{S}| = |V_{p_1} \times V_{p_2} \times ... \times V_{p_n}| = |V_{p_1}| \cdot |V_{p_2}| \cdot ... \cdot |V_{p_n}|$$

gegeben. Mit Definition 3.3 kann der Suchraum auch als Menge aller Parameterkonfigurationen

$$\mathcal{S} = \{K_1, ..., K_m\}, m = |\mathcal{S}|$$

beschrieben werden.

Da die Wertemengen der Tuning-Parameter diskret sind, so ist auch die Menge aller Parameterkonfigurationen diskret.

3.1.4 Leistungskriterium

Das Leistungskriterium ist eine Metrik, anhand derer beurteilt wird, wie gut sich eine Parameterkonfiguration K auf die entsprechende Leistung des Programms auswirkt.

In erster Linie dient die automatische Performanzoptimierung paralleler Programme dazu, deren Ausführungszeit zu minimieren. Ein paralleles Programm soll so schnell wie möglich arbeiten, um eine möglichst hohe Beschleunigung (engl. *speedup*) gegenüber der sequentiellen Version des Programms zu erreichen. Ist t_s die Ausführungszeit der sequentiellen Programmversion und t_p die der parallelen Programmversion, so ist Beschleunigung B wie folgt definiert:

$$B = \frac{t_s}{t_p}.$$

Neben Geschwindigkeit und Beschleunigung sind noch weitere Leistungsmerkmale von Interesse, wie zum Beispiel der Speicherverbrauch eines Programms. Grundsätzlich können die nahezu alle Leistungszähler (engl. *performance counter*) eines System als Leistungsmerkmal definiert werden.

So ist es beispielsweise möglich, durch das Auslesen spezieller Leistungszähler auf ein übergeordnetes Leistungskriterium zu schließen. Kann beispielsweise mittels eines Leistungszählers die Anzahl der *Cache-Misses* festgestellt werden, die ein Programm verursacht, und besitzt selbiges Programm einen Tuning-Parameter, der auf dieses Verhalten Einfluss nehmen kann, so wäre es möglich, über die Reduzierung der Cache-Misses die Laufzeit des Programms zu verringern [TJTK+05]. Auf diese Weise kann zum Beispiel der Effekt des *False Sharing* entdeckt werden, der die Laufzeit des Programms unter Umständen merklich verlängert.

An dieser Stelle sind nahezu beliebige Szenarien denkbar, bei denen Leistungszähler dazu verwendet werden, Aussagen über die Gesamtleistung des Programms zu machen.

3.1.5 Messpunkt

Ein Messpunkt bezeichnet eine bestimmte Stelle im Programm, an der gemäß des Leistungskriteriums Daten erfasst werden, um die Leistung des Programms zu messen. Ist als Leistungskriterium beispielsweise die Laufzeit des Programms definiert, so sind innerhalb des Programms mindestens zwei Messpunkte erforderlich, die jeweils die aktuelle Zeit messen. Die Laufzeit zwischen den zwei Messpunkten ergibt sich dann aus der Differenz der beiden Messwerte. Die Position der Messpunkte im Programm hängt davon ab, welche Teile mit welcher Granularität gemessen werden sollen. Ein Programm kann beliebig viele Messpunkte enthalten. Dies setzt allerdings einen Definition voraus, wie mit den einzelnen Werten verfahren wird (Aufsummierung, Durchschnittsbildung, usw.).

3.1.6 Leistungswert

Der Leistungswert ist eine Größe, um die Leistung eines Programms hinsichtlich einer Parameterkonfiguration zu beschreiben.

Definition 3.8 *Der Leistungswert $\ell \in \mathbb{R}$ eines Programms $\wp$ ist definiert als*

$$\ell_{\wp,K} = \omega_\wp(K) : \mathcal{K} \to \mathbb{R}, \mathcal{K} = \{K | K \in \mathcal{S}\}.$$

Für ein gegebenes Programm $\wp$ bildet die Funktion $\omega_\wp$ eine Parameterkonfiguration K auf eine reelle Zahl, den Leistungswert $\ell_{\wp,K}$, ab. Auf diese Weise wird jeder Parameterkonfiguration $K \in \mathcal{S}$ ein Leistungswert zugeordnet. Die Semantik des Leistungswertes wird durch das gewählte Leistungskriterium (z.B. Laufzeit) definiert.

Für maskierte Parameterkonfigurationen kann der Leistungswert in Abhängigkeit des variablen Tuning-Parameters und dessen Wert in der Parameterkonfiguration definiert werden. Für die maskierte Parameterkonfiguration $\bar{K}_{p_i,j}$ des Parameters p_i mit der Wertemenge $j = V_{p_i} = \{v_1, v_2, ..., v_k\}$ ist der Leistungswert definiert als

$$\ell_{\wp,\bar{K}_{p_i,j}} = \omega_\wp(p_i) : \bar{\mathcal{K}}_{p_i} \to \mathbb{R}, \bar{\mathcal{K}}_{p_i} \in \mathcal{S}, j \in \{1, ..., k\}$$

3.1.7 Parametersensitivität

Die Parametersensitivität $\Delta\ell_{\wp,p}$ eines Tuning-Parameters p beschreibt, wie stark p den Leistungswert $\ell_\wp$ des Programms $\wp$ beeinflusst.

Definition 3.9 *Es sei $P = \{p_1, p_2, ..., p_n\}$ die Menge aller Tuning-Parameter in einem Programm $\wp$ und $V_{p_i} = \{v_1, ..., v_k\}$ der Wertebereich von $p_i, i \in \{1, 2, ..., n\}$. Außerdem sei $\mathcal{L}_{\wp,\bar{\mathcal{K}}_{p_i}} = \{\ell_{\wp,\bar{K}_{p_i,1}}, ..., \ell_{\wp,\bar{K}_{p_i,k}}\}$ die Menge aller Leistungswerte, die jeweils den maskierten Parameterkonfigurationen in $\bar{\mathcal{K}}_{p_i}$ zugeordnet werden. Die Parametersensitivität von p_i ist dann definiert durch*

$$\Delta\ell_{\wp,p_i} = \max \mathcal{L}_{\wp,\bar{\mathcal{K}}_{p_i}} - \min \mathcal{L}_{\wp,\bar{\mathcal{K}}_{p_i}}.$$

3.2 Definition des Suchproblems

Mit Hilfe der im vorigen Abschnitt eingeführten Grundbegriffe lässt sich das Problem der Suche nach der besten Parameterkonfiguration wie folgt definieren.

Definition 3.10 *Für ein Programm $\wp$ auf einer definierten Hardware-Plattform gilt es diejenige Parameterkonfiguration $K^* \in \mathcal{S}$ zu finden, mit der gemäß dem Leistungskriterium der beste Leistungswert $\ell_{\wp,K^*}$ erreicht wird. Die optimale Parameterkonfiguration K^* für $\wp$ ist gefunden, wenn für alle $K \in \mathcal{S}$*

$$\omega_\wp(K^*) \leq \omega_\wp(K)$$

gilt.

Soll beispielsweise die Ausführungszeit eines Programms optimiert werden, so wird nach der Parameterkonfiguration gesucht, mit der das Programm am schnellsten läuft. Auto-Tuning kann demnach als ein kombinatorisches Optimierungsproblem bezeichnet werden. Jede Parameterkonfiguration stellt eine potentielle Lösung des Problems dar, es wird jedoch die beste Lösung gesucht.

Die Tatsache, dass die Funktion $\omega_\wp$ und daher auch ihr Gradient $\nabla\omega_\wp$ üblicherweise nicht bekannt ist, macht das Tuning von Programmen überhaupt erst erforderlich. Wäre $\omega_\wp$ differenzierbar und $\nabla\omega_\wp$ demnach bekannt, so gäbe es eine Reihe von Algorithmen, mit denen das Problem gelöst werden könnte. In diesem Fall würde man den Einfluss der Parameter sowie deren Abhängigkeiten untereinander bereits kennen, so dass $\omega_\wp$ ein vollständiges analytisches Modell des Programms darstellen würde, welches die Leistung in Abhängigkeit der Parameter beschreibt.

Grundsätzlich ist anzumerken, dass die Performanzoptimierung eines Programms stets für eine konkrete Hardware-Plattform durchgeführt wird. Die optimale Parameterkonfiguration $K^* \in \mathcal{S}$ für ein Programm $\wp$ muss daher für jede Hardware-Plattform, auf der $\wp$ ausgeführt werden soll, erneut ermittelt werden. Dies ist insbesondere dann unumgänglich, wenn sich die Hardware-Architekturen in signifikanten Punkten unterscheiden, wie z.B. in der Anzahl der Kerne oder der Struktur und Größe der Caches. Der beste Leistungswert $\ell^*_\wp$ hinsichtlich eines Leistungskriteriums ist also immer im Kontext einer konkreten Hardware-Plattform zu sehen.

Die Vielzahl an unterschiedlichen Hardware-Plattformen setzen flexibel adaptierbare Programme sowie ein portierbares und plattform-unabhängiges Auto-Tuning-Konzept voraus. Beides wird im Rahmen dieser Arbeit vorgestellt.

3.3 Verarbeiten des Suchraums

Für die Ermittlung der Parameterkonfiguration, die die beste Leistung eines Programms bewirkt, stehen verschiedene Methoden zu Verfügung. An dieser Stelle werden die für das Optimierungsproblem des Auto-Tunings wichtigsten Verfahren aufgeführt und kurz erklärt. Es besteht kein Anspruch auf Vollständigkeit, da eine große Zahl unterschiedlicher Verfahren existiert, von denen es wiederum zahlreiche Varianten gibt. Als zentrale Literaturstelle für das Thema Optimierung und Problemlösung sei auf [MiFo00] verwiesen.

Grundsätzlich gilt, dass Verfahren zur Lösung kombinatorischer Optimierungsprobleme an das konkrete Problem angepasst werden müssen. Aus diesem Grund werden alle Verfahren an Hand des Auto-Tuning-Kontextes unter Verwendung der im vorherigen Abschnitt eingeführten Grundbegriffe erläutert.

3.3.1 Vollständige Enumeration (Exhaustionsmethode)

Die einfachste Methode besteht in der Enumeration des gesamten Suchraums und dem anschließenden Testen aller enthaltenen Parameterkonfigurationen. Für alle $K \in \mathcal{S}$ wird der zugehörige Leistungswert $\ell_{\wp,K}$ des Programms $\wp$ ermittelt.

Das Verfahren stellt sicher, dass auf jeden Fall die optimale Parameterkonfiguration gefunden wird. Allerdings kann diese Methode nur für sehr kleine Suchräume verwendet werden, da keine Reduktion des Suchraums durchgeführt wird. Dies bedeutet jedoch nicht, dass die vollständige Enumeration nie zum Einsatz kommt. Kann der Suchraum im Vorfeld der eigentlichen Suche auf wenige, aber relevante Parameterkonfigurationen reduziert werden (entsprechende Konzepte werden in einem späteren Kapitel erörtert), ist es meist sinnvoll, all diese Konfigurationen zu testen, um ein möglichst gutes Ergebnis zu erzielen.

Im Kontext des Auto-Tuning ist eine vollständige Enumeration des Suchraums möglich, da die Parameterkonfigurationen im Suchraum eine diskrete Menge bilden. Einzelne Parameterkonfigurationen können also exakt von einander unterschieden werden. Hinzu kommt, dass der Suchraum in der Regel endlich ist, da gängige Tuning-Parameter einen endlichen Wertebereich besitzen. Bei der Enumeration des Suchraums müssen nun gemäß der Definition in 3.1.2 aus den Wertemengen der Tuning-Parameter alle zulässigen Permutationen der Parameterwerte erzeugt werden, die dann die Menge der Parameterkonfigurationen ergeben.

In späteren Kapiteln werden wir feststellen, dass es konditionale Abhängigkeiten zwischen Tuning-Parametern geben kann. Diese müssen bei einer vollständigen Enumeration des Suchraums berücksichtigt werden.

3.3.2 Metaheuristische Verfahren

Eine effizientere Methode zur Erforschung des Suchraums stellen Metaheuristiken dar. Eine Metaheuristik beschreibt einen Algorithmus zur näherungsweisen Lösung eines kombinatorischen Optimierungsproblems. Hierbei werden alle Elemente des Suchraums als potentielle Lösungen angesehen. Es werden so lange Lösungen getestet, bis der Algorithmus konvergiert, d.h. bis keine bessere Lösung mehr gefunden werden kann. Es wird also versucht, durch „systematisches Ausprobieren“ die optimale Lösung zu finden. Eine Metaheuristik garantiert allerdings nicht, dass die optimale Lösung tatsächlich gefunden wird. In der Informatik werden Metaheuristiken üblicherweise als Suchalgorithmen bezeichnet.

Metaheuristiken haben sich für das Suchproblem des Auto-Tuning bereits als äußerst geeignet erwiesen [ScPT09, ScPT10]. Jede Parameterkonfiguration K aus dem Suchraum $\mathcal{S}$ stellt eine potentielle Lösung des Suchproblems dar, um das Programm $\wp$ hinsichtlich eines Leistungskriteriums optimal zu konfigurieren. Die Qualität einer Parameterkonfiguration K wird anhand des Leistungswertes $\ell_{\wp,K}$ ermittelt. Die beste Parameterkonfiguration wird gemäß Definition 3.10 bestimmt.

Zur Bestimmung der Qualität einer Metaheuristik dienen zwei Metriken. Zum einen ist entscheidend, wie nahe sich der Suchalgorithmus an das tatsächliche Optimum annähert. Hier kann beispielsweise eine mittlere Fehlerrate angegeben werden. Zum anderen ist maßgeblich, wie schnell das Verfahren konvergiert, d.h. wie viele Parameterkonfigurationen getestet werden müssen, bis der Suchalgorithmus terminiert und das bisher beste Ergebnis hinsichtlich des Leistungskriteriums zurückgibt.

Im Folgenden betrachten wir unterschiedliche Klassen von Suchalgorithmen und nennen die wichtigsten Vertreter.

3.3.2.1 Zufallsmethode

Mit der Zufallsmethode wird eine bestimmte Anzahl an Parameterkonfigurationen aus dem Suchraum zufällig ausgewählt und getestet. Die Parameterkonfiguration, die den

besten Leistungswert erzielt hat, wird als Ergebnis ausgegeben. Auch wenn das Verfahren auf den ersten Blick nicht zielführend erscheint, so können insbesondere bei kleinen Suchräumen, deren Parameterkonfigurationen zu ähnlichen Leistungswerten führen, brauchbare Ergebnisse erreicht werden.

3.3.2.2 Prinzip der lokalen Suche

Die lokale Suche konzentriert sich auf einen Bereich um eine gegebene Startkonfiguration. Es werden nur Parameterkonfigurationen betrachtet, die sich in einer geeignet definierten Nachbarschaft der Startkonfiguration befinden. Diese Prozedur kann allgemein durch vier Schritte beschrieben werden:

1. Wähle eine initiale Parameterkonfiguration $K \in \mathcal{S}$ und definiere diese Parameterkonfiguration als die *aktuelle Konfiguration* K_a. Bestimme den zugehörigen Leistungswert $\ell_{\wp,K_a} = \omega_\wp(K_a)$.

2. Transformiere die aktuelle Konfiguration, um eine neue Parameterkonfiguration $K_n \in \mathcal{S}$ zu erhalten und bestimme den zugehörigen Leistungswert $\ell_{\wp,K_n} = \omega_\wp(K_n)$.

3. Ist $\ell_{\wp,K_n} \leq \ell_{\wp,K_a}$, ersetze die aktuelle durch die neue Parameterkonfiguration; andernfalls verwerfe die neue Parameterkonfiguration.

4. Wiederhole Schritt 2 und 3, bis keine weitere Transformation den Leistungswert der aktuellen Konfiguration verbessert.

Ein Verfahren, das auf dem Prinzip der lokalen Suche basiert, wird durch folgende Eigenschaften charakterisiert:

- **Schrittweite.** Mit der Schrittweite wird definiert, wie groß der Schritt von der aktuellen Konfiguration K_a zu einer neuen Konfiguration K_n sein soll. Die Angabe einer Schrittweite ist nur dann möglich, wenn auf den Wertemengen der einzelnen Tuning-Parameter eine Metrik definiert ist. Mögliche Alternativen sind beispielsweise, die Schrittweite immer gleich groß, zufällig groß, kleiner werdend (in der Nähe des Optimums), oder wachsend (bei vielversprechender Suchrichtung) zu wählen.

- **Selektionsstrategie.** Die Selektionsstrategie bestimmt, wie die initiale Parameterkonfiguration generiert wird und wann ggf. an neuer Position im Suchraum weitergesucht wird (also eine neue initiale Konfiguration generiert wird).

- **Individuenanzahl.** Die oben beschriebene Prozedur der lokalen Suche kann basierend auf mehreren initialen Parameterkonfigurationen (Individuen) im Suchraum gleichzeitig oder nacheinander gestartet werden. Der Einsatz mehrerer Individuen erhöht die Laufzeit, kann aber unter Umständen zu besseren Ergebnissen führen.

- **Abbruchkriterium.** Das Abbruchkriterium bestimmt, wann die Suche beendet ist. Dies kann beispielsweise nach einer initial festgelegten Anzahl an Algorithmen-Iterationen oder bei einer Stagnation der Verbesserungsrate sein.

Der Vorteil von Verfahren mit lokaler Suche ist die relativ einfache Implementierung sowie die leichte Adaptierbarkeit an das konkrete Problem – in unserem Fall das Auto-Tuning. Allerdings sind auch einige Nachteile zu nennen, die bei der Verwendung berücksichtigt werden sollten:

- Verfahren mit lokaler Suche terminieren in der Regel bei Lösungen, die nur ein lokales Optimum darstellen. Die Algorithmen können sich gewissermaßen in lokalen Optima „verfangen“.
- Die Verfahren bieten keine Information darüber, wie weit das gefundene (lokale) Optimum vom globalen Optimum entfernt ist (also um wie viel die beste gefundene Parameterkonfiguration schlechter ist als die global beste Konfiguration).
- Der Erfolg des Verfahrens hängt von der Wahl der initialen Parameterkonfiguration(en) oder Individuen ab.
- Es ist in der Regel nicht möglich, eine obere Schranke für die Dauer der Berechnung (Zeit, bis das Verfahren konvergiert) anzugeben.

Einige bekannte Verfahren, die auf dem Prinzip der lokalen Suche basieren und für das Optimierungsproblem des Auto-Tuning geeignet sind, werden im Folgenden aufgeführt. Alle Verfahren sind grundlegend ähnlich, besitzen aber zum Teil Eigenschaften, die den oben genannten Nachteilen entgegenwirken sollen.

Bergsteigeralgorithmus (engl. *Hillclimbing*)

Der Bergsteigeralgorithmus gehört zu den klassischen Verfahren mit lokaler Suche und orientiert sich im Wesentlichen an den vier Schritten, die oben vorgestellt wurden. Die initiale Parameterkonfiguration wird meist per Zufall gewählt. Anschließend werden alle möglichen Nachbarn der initialen Konfiguration K_a betrachtet, wobei die beste der Nachbarkonfigurationen mit der initialen verglichen wird. Ist die neue Konfiguration aus der Nachbarschaft besser als die initiale Konfiguration, wird mit der neuen Konfiguration fortgefahren. Andernfalls kann keine Verbesserung mehr erzielt werden und der Algorithmus terminiert: Ein lokales oder globales Optimum ist erreicht.

Es wird ersichtlich, dass der Bergsteigeralgorithmus keine Methode besitzt, die das Terminieren in lokalen Optima verhindert. Durch erneutes Starten des Algorithmus mit einer neuen zufällig gewählten initialen Parameterkonfiguration können zumindest noch andere Bereiche des Suchraums abgearbeitet werden, so dass das globale Optimum mit höherer Wahrscheinlichkeit gefunden wird.

Simulierte Abkühlung (engl. *Simulated Annealing*)

Die Grundidee der simulierten Abkühlung ist die Nachbildung eines Abkühlungsprozesses. Beim Erzeugen von Kristallen wird nach dem Erhitzen des Materials durch langsame Abkühlung erreicht, dass die Moleküle eine stabile Kristallstruktur bilden können. Würde der Abkühlungsprozess zu schnell stattfinden, könnten sich Unregelmäßigkeiten in der Kristallstruktur bilden. Nur bei langsamer Abkühlung wird ein energiearmer Zustand erreicht, der nahe am Optimum liegt.

Überträgt man den Abkühlungsprozess auf das Optimierungsverfahren, so entspricht die Temperatur einer Wahrscheinlichkeit, mit der sich die Qualität einer Lösung im Gegensatz zum Bergsteigeralgorithmus auch verschlechtern darf. Dies hat zur Folge, dass bei der simulierten Abkühlung ein lokales Optimum auch wieder verlassen werden kann.

Der Algorithmus der simulierten Abkühlung weist im Vergleich zum Bergsteigeralgorithmus eine wesentliche Änderungen auf. Es werden nicht *alle* Parameterkonfiguration in der Nachbarschaft überprüft, um dann die beste mit der aktuellen Konfiguration K_a zu vergleichen, sondern nur *eine* Parameterkonfiguration K_n. Unabhängig von ihrer Qualität wird diese neue Parameterkonfiguration K_n nur mit einer Wahrscheinlichkeit $prob$ für

den Vergleich mit K_a akzeptiert. $prob$ ist abhängig von den ermittelten Leistungswerten der beiden Konfigurationen K_a und K_n. Ist K_n einmal akzeptiert, findet kein zusätzlicher Vergleich mehr mit K_a statt. K_n kann also auch einen schlechteren Leistungswert aufweisen als K_a, da ausschließlich $prob$ darüber entscheidet, ob K_n als neue Parameterkonfiguration verwendet wird. Somit kann sich die Qualität der überprüften Parameterkonfigurationen temporär verschlechtern. Wird K_n nicht akzeptiert, wird eine neue Konfiguration aus der Nachbarschaft von K_a ausgewählt und K_n verworfen.

Tabu-Suche (engl. *Tabu Search*)

Der Prozess der Tabu-Suche [Glov86, GlKL95] folgt grundlegend dem der lokalen Suche. Auch die Tabu-Suche ist ein iteratives Verfahren, das in jeder Iteration eine neue Parameterkonfiguration in der Nachbarschaft der aktuellen Konfiguration überprüft. Um jedoch Zyklen beim Durchlaufen des Suchraums zu vermeiden, wird eine so genannte *Tabu-Liste* geführt. In dieser Liste werden für eine bestimmte Dauer (Anzahl an Iterationen) Parameterkonfigurationen tabuisiert (also von einer Überprüfung durch den Algorithmus ausgeschlossen). Im einfachsten Fall werden bereits untersuchte Parameterkonfigurationen in die Tabu-Liste eingetragen, damit diese im Folgenden nicht noch einmal untersucht werden. Die Dauer der Tabuisierung hängt davon ab, wie weit sich das Verfahren von einer bereits untersuchten Parameterkonfiguration mit einer bestimmten Anzahl an Iterationen entfernen kann und somit die Wahrscheinlichkeit sinkt, dass eben diese Parameterkonfiguration ein zweites Mal getestet wird.

Auf Grund dieser Erweiterung des Prinzips der allgemeinen lokalen Suche ändert sich das Auswahlkriterium in Schritt (3) (siehe oben): Die neue Parameterkonfiguration K_n aus der Nachbarschaft um die aktuelle Konfiguration K_a wird nur dann ausgewählt, wenn K_n die beste Konfiguration in der Nachbarschaft darstellt, $\ell_{\wp,K_n} \leq \ell_{\wp,K_a}$ gilt und K_n nicht als Tabu definiert wurde.

Simplex-Algorithmus nach Nelder und Mead (engl. *Downhill Simplex*)

Der Simplex-Algorithmus nach Nelder und Mead [NeMe65] ist verwandt mit dem klassischen Bergsteiger-Algorithmus und daher nicht zu verwechseln mit dem namensgleichen Verfahren zur Lösung linearer Optimierungsprobleme. Das Verfahren basiert auf einem Simplex, dem einfachsten Volumen im N-dimensionalen Suchraum, das aus $N+1$ Parameterkonfigurationen (Punkten im Raum) aufgespannt wird. Die ersten $N+1$ Iterationen des Algorithmus werden dazu verwendet, den Simplex zu erzeugen.

Dieser initiale Simplex wird anschließend in weiteren Iteration modifiziert, indem man die Parameterkonfiguration mit dem schlechtesten Leistungswert durch eine neue Konfiguration mit einem besseren Leistungswert ersetzt. Hierbei behält man die Parameterkonfiguration mit dem besten Leistungswert stets als bisher beste Lösung des Optimierungsproblems bei.

Im Allgemeinen konvergiert das Verfahren nicht extrem schnell, ist dafür aber einfach und relativ robust. Insbesondere wird durch die Verwendung mehrerer Parameterkonfigurationen die Gefahr reduziert, dass der Algorithmus in einem lokalen Optimum konvergiert.

Lokale Suchverfahren bilden klassischerweise die Grundlage zur Lösung kombinatorischer Optimierungsprobleme. Trotz der genannten Nachteile sind derartige Verfahren für das Auto-Tuning geeignet.

3.3.2.3 Prinzip der globalen Suche

Das Prinzip der globalen Suche basiert auf dem der lokalen Suche, weshalb die im vorherigen Abschnitt formulierten Schritte im Wesentlichen auch für Verfahren mit globaler Suche gelten.

Im Gegensatz zu lokaler Suche wird jedoch von Beginn an der gesamte Suchraum berücksichtigt, indem initial nicht nur eine, sondern mehrere Parameterkonfigurationen ausgewählt werden, von denen aus die Suche gestartet wird. Verfahren mit globaler Suche laufen in der Regel nicht Gefahr, in einem lokalen Optimum zu konvergieren.

Globale Suchverfahren werden durch ähnliche Eigenschaften wie lokale Suchverfahren charakterisiert. Dazu gehören die Anzahl der Individuen sowie die Definition eines Abbruchkriteriums. Des Weiteren ist die Selektionsstrategie entscheidend, mit der neue Individuen ausgewählt werden.

Es existieren eine große Zahl an Verfahren mit globaler Suche, deren Auflistung den Rahmen dieses Kapitels weit überschreiten würde. Bekannte Vertreter globaler Suchverfahren sind Algorithmen, die Schwarmintelligenz ausnutzen (z.B. Partikel-Schwarm-Optimierung [KeEb95] oder Ameisenalgorithmen [CoDM91]) sowie evolutionäre Verfahren [FoOW66] oder genetische Algorithmen [Holl92].

Die meisten Schwarmintelligenzverfahren bieten den Vorteil, dass sie dynamisch auf eine sich ändernde Problemstellung reagieren können, weshalb der Einsatz derartiger Algorithmen neben Offline-Tuning auch für Online-Tuning denkbar ist.

Grundsätzlich erfordern globale Suchverfahren einen etwas höheren Implementierungsaufwand als entsprechende Ansätze mit lokaler Suche. Die meisten Nachteile der lokalen Suchverfahren werden jedoch durch die Verwendung mehrerer zusammenhängender Individuen kompensiert. Aus diesen Gründen sind Verfahren mit globaler Suche auch für große Suchräume geeignet, die viele hinreichend gute Parameterkonfigurationen beinhalten.

In Kapitel 4 werden existierende Ansätze vorgestellt, die auf lokalen Suchverfahren basieren. In Kapitel 7 zeigen wir an Hand von Experimenten, dass sowohl lokale als auch globale Suchverfahren auf das kombinatorische Optimierungsproblem des Auto-Tuning anwendbar sind.

3.3.3 Metaheuristiken mit geführter Suche

Metaheuristische Verfahren eignen sich für das plattform- und domänenunabhängige Auto-Tuning von parallelen Programmen insbesondere aus dem Grund, weil derartige Suchverfahren vom eigentlichen Problem abstrahieren und für jedes kombinatorischer Optimierungsproblem herangezogen werden können. Es ist daher kein problemspezifisches Wissen notwendig, um die vorgestellten Suchverfahren anzuwenden.

Genau diese Eigenschaft kann aber auch zum Nachteil werden. Da die Suchverfahren keine Information über die Struktur des Suchraums und potentielle Zusammenhänge von Parameterkonfigurationen und deren Leistungswerte haben, ist es wahrscheinlich, dass auch Teile des Suchraums durchlaufen werden, die hinsichtlich des aktuellen Problems – also des zu optimierenden Programms – irrelevant sind. Insbesondere bei Verfahren mit lokaler Suche kann dies problematisch sein, wenn die initiale Parameterkonfiguration ungünstig gewählt wird.

Abbildung 3.1 veranschaulicht diesen Sachverhalt. Innerhalb des *gesamten Suchraums* $\mathcal{S}$, der gemäß Definition 3.6 alle Parameterkonfigurationen enthält, befindet sich der *relevante Suchraum* $\mathcal{R}$, der nur diejenigen Parameterkonfigurationen enthält, die für die Optimierung relevant sind. Unnötige oder gar sinnlose Parameterkonfigurationen, die sich

aus den einzelnen Parameterwerten jedoch bilden lassen, sind in $\mathcal{R}$ nicht enthalten. Es gilt also $\mathcal{R} \subseteq \mathcal{S}$.

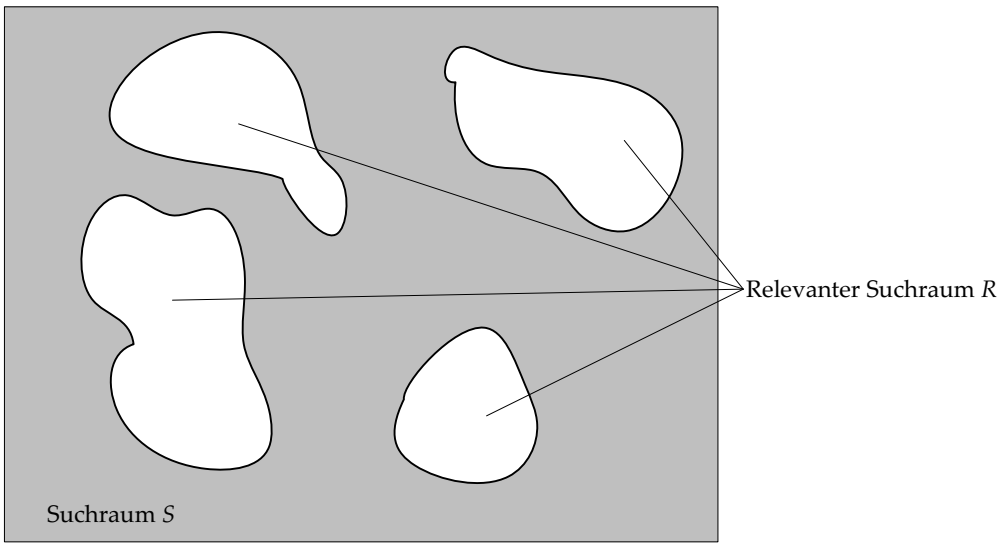


Abbildung 3.1: Darstellung des gesamtem Suchraums $\mathcal{S}$ und einer relevanten Untermenge $\mathcal{R}$.

Um zu vermeiden, dass Suchverfahren Parameterkonfigurationen aus dem irrelevanten Bereich des Suchraums ($\mathcal{S} \backslash \mathcal{R}$) verarbeiten, muss der Suchraum vor der eigentlichen Suche auf seine relevante Untermenge reduziert werden. Man spricht in diesem Fall von einer geführten Suche, da das Suchverfahren bereits zu den wichtigen Bereichen des Suchraums (in denen sich mit großer Wahrscheinlichkeit auch das Optimum befindet) geführt wird.

Für das Optimierungsproblem des Auto-Tuning ist eine geführte Suche – also die problemabhängige Reduktion des Suchraums auf wesentliche Parameterkonfigurationen – nicht trivial, da diese nur mittels problemspezifischem Kontextwissen möglich ist. Es werden daher heuristische Methoden benötigt, die auf der Grundlage von derartigem Kontextwissen Annahmen darüber treffen, welche Parameterkonfigurationen relevant sind. Die Frage ist schließlich, welche Parameterkonfigurationen aus dem Suchraum tatsächlich getestet werden müssen, um die optimale oder zumindest eine hinreichend gute Konfiguration des Programms zu erhalten, und welche Konfigurationen von vorne herein ausgeschlossen werden können.

In Kapitel 5 werden grundlegende Konzepte und Ansätze vorgestellt, die eine signifikante und präzise Reduktion des Suchraums erreichen. Suchverfahren, die auf dem reduzierten Suchraum angewendet werden, konvergieren wesentlich schneller und bieten eine größere Genauigkeit [Scha09].

3.3.4 Performanzmodelle

Wie im vorherigen Abschnitt beschrieben, kann bei einer geführten Suche mittels geeigneter Annahmen über das Programm und seine leistungsrelevanten Parameter der Suchraum beträchtlich reduziert werden. Im verbleibenden Suchraum muss schließlich mit herkömmlichen Suchverfahren (siehe Abschnitt 3.3.2) oder der Exhaustionsmethode (siehe Abschnitt 3.3.1) nach der optimalen Parameterkonfiguration gesucht werden.

Um auf eine Suche gänzlich verzichten zu können, muss die Leistung eines Programms in Abhängigkeit seiner Tuning-Parameter berechnet werden. Dies kann mittels analytischer Performanzmodelle geschehen. Derartige Modelle setzen die Tuning-Parameter sowie Systemparameter in eine mathematische Abhängigkeit und bieten somit die Möglichkeit, die Leistung des Programms im Voraus zu berechnen und eine optimale Belegung der Tuning-Parameter zu ermitteln.

Der Vorteil gegenüber suchbasierten Verfahren besteht darin, dass das Programm zur Ermittlung der optimalen Parameterkonfiguration nicht ausgeführt werden muss und verhältnismäßig wenig Zeit für das Auffinden des Optimums benötigt wird.

Diesem Vorteil steht jedoch der Nachteil gegenüber, dass Modelle im Allgemeinen statisch und üblicherweise auf bestimmte Anwendungsbereiche, Hardware-Plattformen oder Programmiermodelle beschränkt sind. Um brauchbare Vorhersagen hinsichtlich der Leistung eines Programms zu erhalten, müssen Modelle möglichst viele problemspezifische Parameter berücksichtigen. Die Vielzahl an unterschiedlichen Programmen und Parallelisierungsstrategien sowie die stets wachsende Zahl an unterschiedlichen Hardware-Plattformen macht es daher schwierig, Modelle zu entwickeln, die hinreichend flexibel sind.

Modellbasierte Optimierung wird daher meist in domänenspezifischen Ansätzen verwendet, die auf eine bestimmte Anwendungs- oder Problemklasse auf einer konkreten Hardware-Plattform spezialisiert sind. Ein Beispiel hierfür wäre eine verteilte Anwendung, die mittels *Message Parsing Interface (MPI)* [MPI 09] Nachrichten austauscht, auf einer Cluster-Architektur läuft und somit auf einem verteilten Master/Worker-Prinzip aufbaut. Für ein derartiges Szenario lassen sich Modelle entwickeln, die ein Reihe von Umgebungs- und Systemparametern berücksichtigen (z.B. Anzahl der verfügbaren Rechnerknoten oder Netzwerkparameter wie Latenz oder Bandbreite) und gute Werte für die vorhandenen Tuning-Parameter liefern.

3.4 Klassifikation von Auto-Tuning-Methoden

Nachdem die grundlegenden Begriffe des Auto-Tuning sowie verschiedene Methoden zur Verarbeitung des Suchraums behandelt wurden, wird in diesem Abschnitt eine Klassifikation des allgemeinen Auto-Tuning-Ansatzes an Hand verschiedener Eigenschaften vorgenommen. Die bisher in dieser Form noch nicht definierte Klassifikation hilft, die Konzepte dieser Arbeit sowie verwandter Ansätze in das thematische Gesamtbild einzuordnen.

3.4.1 Auto-Tuning-Taxonomie

Für die übersichtliche Klassifikation von Auto-Tuning-Verfahren führen wir die *Auto-Tuning-Taxonomie* ein, die in Abbildung 3.2 dargestellt ist. Jede Dimension des Taxonomie-Würfels entspricht einer Eigenschaft, nach der klassifiziert werden kann. Die Auto-Tuning-Ansätze werden nach dem *Tuning-Typ*, dem *Tuning-Zeitpunkt* sowie der *Tuning-Strategie* unterschieden.

Im Folgenden werden die Charakteristiken beschrieben und diskutiert.

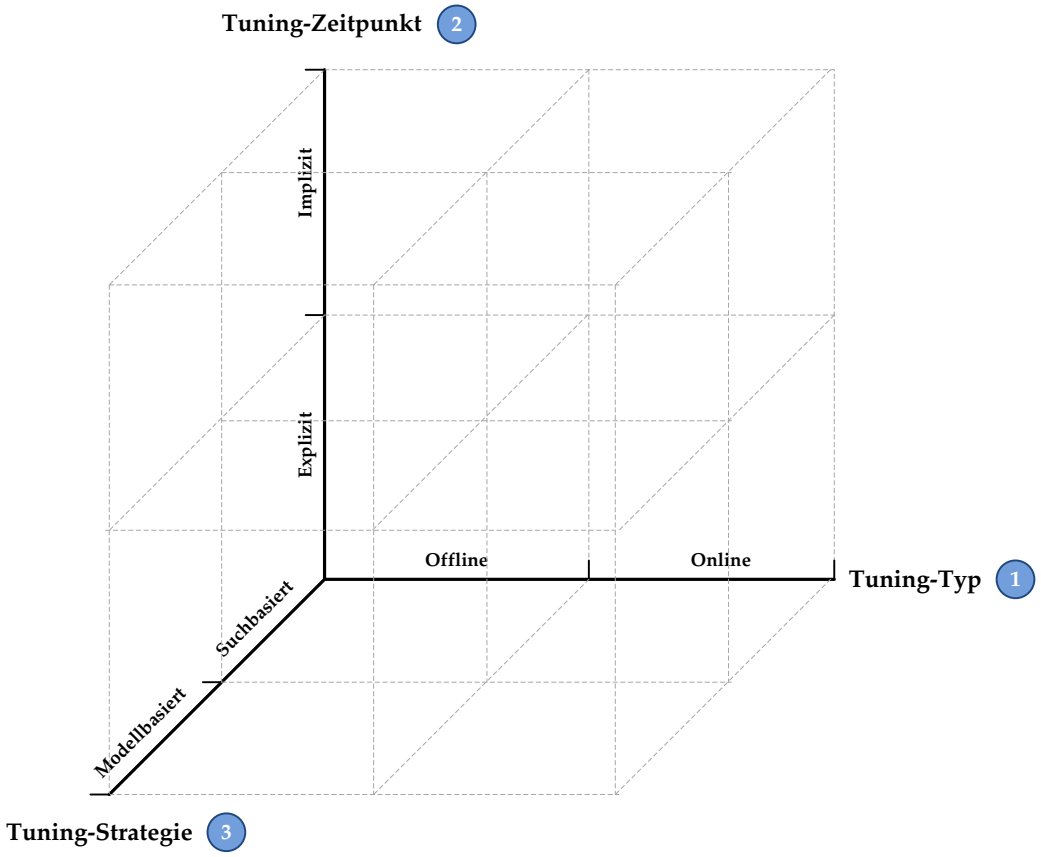
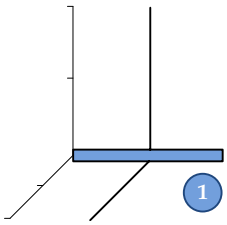


Abbildung 3.2: Dreidimensionale Auto-Tuning-Taxonomie.

3.4.1.1 Tuning-Typ (Dimension 1)



Mit dem *Tuning-Typ* wird das grundlegende Funktionsprinzip des Auto-Tuning-Verfahrens festgelegt. Der Typ entscheidet darüber, ob die Werte von Tuning-Parametern zur Laufzeit des Programms oder zwischen den Programmläufen modifiziert werden. Es wird daher zwischen *Offline-* und *Online-Tuning* unterschieden:

- **Offline-Tuning.** Die Änderung der Parameterwerte erfolgt stets *zwischen* den Programmläufen.

  Dieses Verfahren bietet eine hohe Flexibilität, da nach jedem Programmlauf festgestellt werden kann, wie sich die Parameterwerte auf die Gesamtleistung ausgewirkt haben, unabhängig davon, welche Parameter berücksichtigt wurden, an welcher Stelle im Programm sich diese befinden und auf Basis welcher Metrik optimiert wird. Des Weiteren setzt Offline-Tuning keine Mindestdauer des Programms oder eine spezielle Programmstruktur voraus. Da jedoch für jede zu testende Parameterkonfiguration ein vollständiger Programmdurchlauf benötigt wird, sind langlaufende Programme wie beispielsweise Echtzeit-Simulationen für Offline-Tuning weniger geeignet.

- **Online-Tuning.** Die Änderungen der Parameterwerte werden *zur Laufzeit* des Programms durchführt. Das Verfahren bietet den Vorteil, dass je nach Laufzeit und

Struktur des Programms während eines Programmlaufs mehrere Parameterkonfigurationen getestet oder sogar der gesamte Optimierungsprozess bis zum Auffinden der besten Parameterkonfiguration durchgeführt werden kann.

Bei sehr langlaufenden Programmen kann ein Online-Tuner auch als Regler im Sinne der Regelungstechnik eingesetzt werden. Stehen sich ändernde Umgebungsparameter (wie beispielsweise die Anzahl verfügbarer Prozessorkerne, die Problemgröße oder die Anzahl benötigter Speicherzugriffe) zur Verfügung, kann mit einem Online-Tuning-Verfahren auf geänderte äußere Bedingungen reagiert werden, indem entsprechende Tuning-Parameter des Programms angepasst werden. Aus diesem Grund besteht beim Online-Tuning nicht unbedingt eine Abhängigkeit von den Eingabedaten, da individuell reagiert werden kann.

Der Einsatzbereich des Online-Tunings ist allerdings deutlich eingeschränkter als der des Offline-Tunings. Zunächst setzt Online-Tuning eine Mindestlaufzeit des Programms voraus, um Parameterkonfigurationen wirksam zu testen. Darüber hinaus muss das Programm eine bestimmte Struktur aufweisen, damit Online-Tuning sinnvoll eingesetzt werden kann. Der zu optimierende und parametrisierte Programmteil sollte oft wiederholt ausgeführt werden (z.B. eine mit Tuning-Parametern versehene Schleife, die eine Berechnung durchführt). Andernfalls erhält das Online-Tuning-Verfahren nicht ausreichend Gelegenheit, unterschiedliche Parameterkonfigurationen zu testen.

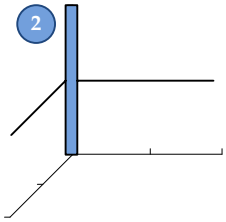
Problematisch ist bei der Methode des Online-Tuning die fehlende Rückkopplung hinsichtlich der Gesamtleistung des Programms. Da zur Laufzeit mehrere Parameterkonfigurationen getestet werden, ist eine Bewertung schwierig, da die Information über den Einfluss einer konkreten Konfiguration auf die Gesamtleistung des Programms nicht vorliegt. Diese Information ist aber wichtig, um die Qualität zu beurteilen und schließlich den nächsten Tuning-Schritt durchzuführen. Hier kann nur mit geschickt gewählten Heuristiken zur Abschätzung des Einflusses einer Parameterkonfiguration Abhilfe geschaffen werden.

Es zeigt sich, dass weder Offline- noch Online-Tuning das jeweils andere Verfahren ersetzen kann. Beide Methoden bieten signifikante Vorteile, aber auch Nachteile, die den Einsatz mehr oder weniger einschränken. Aus diesem Grund sind beide Verfahren bei Überlegungen im Bereich Auto-Tuning zu berücksichtigen.

Hierbei spielt Offline-Tuning insbesondere dann eine große Rolle, wenn Auto-Tuning als Teil des Entwicklungsprozesses fungiert und die zu optimierenden Programme von unterschiedlichster Struktur und Klasse sind.

Online-Tuning ist für langlaufende Programme und Simulationen unumgänglich, insbesondere dann, wenn die Leistung des Programms in beträchtlichem Maße von der Größe und Komplexität der Eingabedaten abhängt und diese starken Schwankungen unterworfen sind.

3.4.1.2 Tuning-Zeitpunkt (Dimension 2)



Die Eigenschaft des *Tuning-Zeitpunkts* entscheidet darüber, ob vor dem produktiven Einsatz des Programms explizite Tuning-Läufe stattfinden oder ob Programmläufe im Produktivbetrieb selbst genutzt werden, um Parameterkonfigurationen zu testen. Wir unterscheiden daher zwischen *Explizitem Tuning* und *Implizitem Tuning*:

- **Explizites Tuning.** Wird das Programm ausschließlich zum Zweck der Modifikation von Parameterwerten gestartet, spricht man von explizitem Tuning. Das Programm befindet sich hierbei nicht im Produktivbetrieb, sondern läuft auf einem Test-System, welches dem späteren Produktivsystem gleichen sollte. Das Programm wird unter kontrollierten und genau definierten Bedingungen gestartet. Sind wie beim Offline-Tuning mehrere Programmdurchläufe nötig, muss für gleichbleibende Bedingungen gesorgt werden, um eine Vergleichbarkeit der Ergebnisse zu garantieren. Explizite Tuning-Läufe eignen sich insbesondere, um in der Endphase der Entwicklung eine möglichst gute Standardkonfiguration des Programms hinsichtlich der Zielplattform zu finden und es somit auf den produktiven Einsatz vorzubereiten.

  Möglich ist auch eine Voroptimierung, bei der für mehrere unterschiedliche Produktivumgebungen die jeweils beste Parameterkonfiguration ermittelt wird. Diese werden anschließend mit dem Programm ausgeliefert. Während der Installation des Programms werden dann relevante Systemparameter analysiert und die passende Parameterkonfiguration ausgewählt und festgeschrieben.

  Der generelle Vorteil von explizitem Tuning besteht darin, dass auf dem Produktivsystem keine Auto-Tuning-Komponente mehr benötigt wird, da die Anwendung bereits auf dem Testsystem optimiert wurde. Weichen die Bedingungen in der Produktivumgebung allerdings von denen auf dem Testsystems ab, muss nachoptimiert werden.

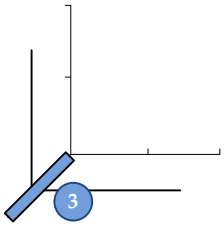
- **Implizites Tuning.** Beim impliziten Tuning werden Produktivläufe des Programms genutzt, um Parameterkonfigurationen zu testen. Dies spart Zeit, da nicht alle nötigen Tests in separaten Tuning-Läufen durchgeführt werden müssen. Allerdings setzt implizites Tuning voraus, dass auf dem Produktivsystem eine Auto-Tuning-Komponente in Betrieb ist. Des Weiteren müssen die Ergebnisse der einzelnen Tests von Parameterkonfigurationen an zentraler Stelle permanent gespeichert werden, da im Vergleich zu explizitem Tuning die Programmläufe nicht zwingend hintereinander ausgeführt werden. Auch ein eventueller Neustart des Systems muss berücksichtigt werden. Da implizites Tuning bei Produktivläufen des Programms durchgeführt wird, eignet es sich offensichtlich zur Nutzung in Kombination mit Online-Tuning.

  Zu beachten ist außerdem, dass der Tuning-Prozess, also das Modifizieren der Parameterwerte, den Produktivbetrieb nicht allzu sehr beeinflussen darf. Es sollte daher sichergestellt werden, dass Parameterkonfigurationen, die einen extrem schlechten Leistungswert bewirken, von vorne herein ausgeschlossen werden. Zudem sollte der Test einer Parameterkonfiguration abgebrochen werden, sobald die aktuelle Laufzeit die bisher kürzeste Laufzeit überschreitet.

Es ist anzumerken, dass sowohl implizites als auch explizites Tuning jeweils mit Offline- oder Online-Tuning-Verfahren realisiert werden kann.

Ob implizites oder explizites Tuning im Einzelfall das geeignetere ist, hängt davon ab, in welcher Phase des Software-Entwicklungsprozesses die Optimierung eingebunden wird und zu welchem Zeitpunkt ausreichend Informationen über das spätere Produktivsystem zur Verfügung stehen. Explizites Tuning kann demnach als Werkzeug in der Entwicklungsphase des Programms eingesetzt werden, um beispielsweise eine möglichst gute Standardkonfiguration zu ermitteln. Implizites Tuning eignet sich üblicherweise für den Einsatz in der Einführungs- und Nutzungsphase des Programms, also nach der Auslieferung beim Kunden. Hierbei können Wartungsaspekte oder die Migration auf eine andere Hardware-Plattform eine Rolle spielen.

3.4.1.3 Tuning-Strategie (Dimension 3)



Die *Tuning-Strategie* legt fest, mit welcher Methode der Suchraum bearbeitet wird. Bereits in Abschnitt 3.3 haben wir die für Auto-Tuning wichtigen Verfahren kennengelernt. Betrachtet man die Methoden zur Verarbeitung des Suchraums, so existieren auf oberster Ebene zwei Klassen von Verfahren: wir unterscheiden zwischen *suchbasierten* (vgl. Abschnitte 3.3.1, 3.3.2 und 3.3.3) und *modellbasierten* Ansätzen (vgl. Abschnitt 3.3.4).

- **Suchbasierte Optimierung.** Das Verfahren der suchbasierte Optimierung (manchmal auch als empirische Optimierung bezeichnet) führt mittels Suchalgorithmen eine systematische Suche im gesamten Suchraum durch, um eine optimale Parameterkonfiguration zu finden. Charakteristisch ist, dass das Programm tatsächlich ausgeführt wird, um die Leistungswerte zu erhalten. In Kombination mit der Methode des Offline-Tuning wird das Programm für jede zu testende Parameterkonfiguration ausgeführt. Das Testen einer Parameterkonfiguration wird *Tuning-Iteration* genannt.

  Der suchbasierte Ansatz ist allgemein verwendbar und ist im Wesentlichen unabhängig von Anwendung, Problemklasse und Hardware-Plattform. Die Qualität der Ergebnisse hängt jedoch stark von den verwendeten Suchalgorithmen ab. Um die Anzahl der Tuning-Iterationen möglichst gering zu halten, ist bei großen parallelen Programmen eine geführte Suche nahezu unumgänglich.

  Wie bereits in obigen Abschnitten besprochen, ist bei suchbasiertem Tuning nicht garantiert, dass die optimale Parameterkonfiguration des Programms bezüglich der zu Grunde liegenden Hardware-Plattform gefunden wird.

- **Modellbasierte Optimierung.** Beim modellbasierten Ansatz werden für bestimmte Partitionen des Suchraums (z.B. Tuning-Parameter, die zu einem Muster gehören oder die von konkreten Hardware- oder Umgebungsparametern abhängen) analytische Performanzmodelle entwickelt, mit denen für die entsprechenden Partitionen des Suchraums die Leistung abhängig von den Tuning-Parametern vorhergesagt und eine optimale Parameterkonfiguration berechnet werden kann.

  Um zuverlässige Vorhersagen zu gewährleisten, werden Modelle üblicherweise auf bestimmte Anwendungsbereiche, Hardware-Plattformen oder Programmier-Modelle beschränkt, um aus der vereinfachten Welt möglichst viele Parameter und Informationen zu berücksichtigen. Analytische Modelle sind daher wenig flexibel und es bedarf eines beträchtlichen Aufwands, ein Modell auf eine geänderte Situation anzupassen.

  Zudem ist für die Teile des Suchraums, die nicht durch ein Modell abgedeckt sind, wiederum suchbasiertes Tuning nötig.

  Modellbasiertes Tuning wird in der Regel mit Online-Tuning kombiniert, da Online-Tuning ohnehin Heuristiken benötigt, um für eine konkrete Parameterkonfiguration Rückschlüsse auf deren Einfluss hinsichtlich der Gesamtleistung des Programms zu ziehen. In diesem Fall macht man sich die Eigenschaft der Performanzmodelle zu Nutze, die Leistung des Programms in einem bestimmten Rahmen vorherzusagen.

Der Fokus dieser Arbeit liegt auf suchbasiertem, explizitem Offline-Tuning, da sich mit diesen Methoden eine flexible und skalierbare konzeptionelle Grundlage für das Auto-Tuning großer paralleler Applikationen entwickeln lässt.

3.5 Integration von Auto-Tunern

Im vorherigen Abschnitt haben wir die wichtigsten Eigenschaften von Auto-Tuning-Methoden kennengelernt und mittels einer Taxonomie klassifiziert. Unabhängig von der Art der Auto-Tuning-Methode stellt sich jedoch die Frage, in welcher Form eine Auto-Tuning-Komponente verwendet und wie diese in das System integriert werden kann. Die Überlegungen sind insbesondere deshalb wichtig, da mit dieser Arbeit Konzepte präsentiert werden, die Auto-Tuning in allgemeinerer Form für ein breites Spektrum an parallelen Programmen ermöglichen sollen. Es ist zu beachten, dass sich die Art der Integration von Auto-Tunern orthogonal zur vorgestellten Taxonomie verhält.

Wir konnten bereits einige Vorschläge zur Integration von Auto-Tunern erläutern, die an dieser Stelle aufgegriffen werden [KaSP09].

3.5.1 Auto-Tuner als Bibliothek

Eine Bibliothek stellt die grundlegende Methode dar, einer Anwendung Auto-Tuning-Funktionalität zur Verfügung zu stellen. In einer unserer ersten Experimente wurde dieser Ansatz verfolgt und evaluiert [PSJT08]. Eine Auto-Tuning-Bibliothek bietet den Vorteil, dass die Anwendung zu jedem Zeitpunkt auf deren Funktionalität zugreifen kann. Dies ist insbesondere dann nützlich, wenn auch Produktivläufe zur Durchführung von Tuning-Iterationen genutzt werden sollen (siehe Abschnitt 3.4.1.2, *Implizites Tuning*). Es muss nicht dafür gesorgt werden, dass die Auto-Tuning-Komponente auf allen genutzten Systemen vorhanden ist. Abbildung 3.3 zeigt den einfachen schematischen Aufbau.

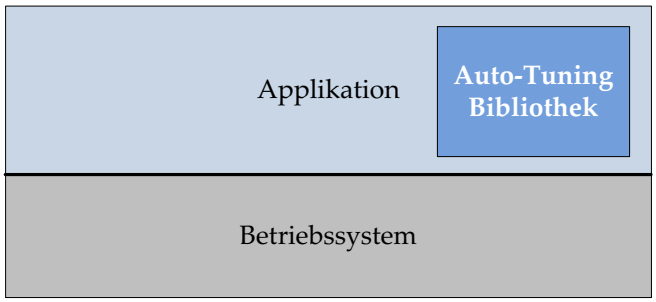


Abbildung 3.3: Darstellung einer Auto-Tuning-Bibliothek als Teil der Applikation.

Charakteristisch für den Bibliotheksansatz ist die Tatsache, dass die Anwendung die kontrollierende Instanz darstellt und die Auto-Tuning-Komponente bei Bedarf verwendet. Die Anwendung kann entweder in einem normalen oder einem Tuning-Modus gestartet werden. In Letzterem wird aus der Anwendung heraus die Auto-Tuning-Komponente gestartet, die daraufhin beginnt, Parameterkonfigurationen zu testen.

Ein Nachteil des Bibliotheksansatzes ist jedoch die starke Kopplung von Auto-Tuner und Anwendung, wodurch die eigentliche Programmlogik mit Auto-Tuning-Logik vermischt wird. Durch die feste Verdrahtung der Auto-Tuning-Komponente mit der Anwendung erfordert jegliche Änderung an den Deklarationen der Tuning-Parametern oder an der Schnittstelle der Auto-Tuning-Komponente die Modifikation des Programmquelltextes (z.B. Anpassung von Bibliotheksaufrufen oder Änderung von Parameter- bzw. Variablentypen).

Des Weiteren besteht eine Programmiersprachen-Abhängigkeit zwischen Bibliothek und Anwendung. Ist die Anwendung in einer Programmiersprache geschrieben, mit der die

Bibliothek nicht verwendet werden kann, steht die Auto-Tuning-Funktionalität für diese Anwendung nicht zur Verfügung.

Eine Auto-Tuning-Bibliothek bietet sich an, wenn eine spezielle Anwendung, deren Menge an Tuning-Parameter sich nicht ändert, auf vielen unterschiedlichen Plattformen zum Einsatz kommen soll und daher immer wieder neu optimiert werden muss.

3.5.2 Auto-Tuner als separate Applikation

Weitaus flexibler als ein Bibliotheksansatz ist das Auslagern der Auto-Tuning-Komponente in eine separate Applikation. Wie in Abbildung 3.4 dargestellt, trennt dieser Ansatz die Anwendung vollständig von der Auto-Tuning-Komponente – Quelltextmodifikationen zur Einbindung des Auto-Tuners sind nicht nötig. Zudem wird durch die Trennung ein flexibler Einsatz der Auto-Tuning-Komponente ermöglicht. Die Kommunikation zwischen Auto-Tuner und Applikation (Konfiguration der Parameter sowie Abfragen der Leistungswerte) findet über definierte Schnittstellen oder das Dateisystem statt.

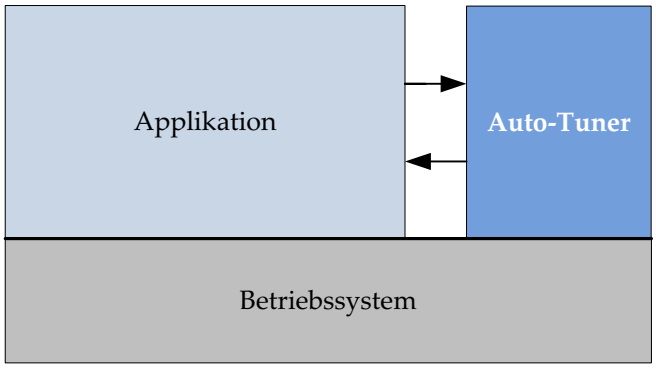


Abbildung 3.4: Darstellung des Auto-Tuners als eigenständiges Programm.

Im Gegensatz zu einer Auto-Tuning-Bibliothek repräsentiert in diesem Fall der Auto-Tuner die kontrollierende Instanz. Um die Anwendung zu optimieren, wird der Auto-Tuner gestartet, der wiederum die parametrisierte Anwendung ausführt, um eine Parameterkonfiguration zu testen. Im Falle von Offline-Tuning wird die Anwendung für jede zu testende Parameterkonfiguration neu gestartet.

Eine separate Auto-Tuning-Instanz bietet durch die Trennung von der Anwendung die nötige Abstraktion, um unterschiedlichste Anwendungen zu optimieren.

Das in dieser Arbeit vorgestellte Konzept basiert auf einer separaten Auto-Tuning-Komponente. In Kapitel 5 wird die Architektur sowie das Konzept zur Anbindung an eine Anwendung vorgestellt.

3.5.3 Auto-Tuner als Komponente eines Übersetzers

Im Forschungsbereich des Übersetzerbaus beschäftigt man sich seit geraumer Zeit mit Optimierungen auf Instruktionsebene, die ein Übersetzer automatisch durchführen kann [KeAl02]. Moderne Übersetzer sind in der Lage, eine ganze Reihe von Optimierungsstrategien anzuwenden, um die Performanz des Programmquelltextes zu steigern (z.B. das Ausrollen von Schleifen (engl. *loop unrolling*) [BaGS94]).

Für parallele Programme existieren jedoch noch wenige Ansätze zur automatischen Optimierung (beispielsweise [Ricc02, Fahr98]). Diese konzentrieren sich vornehmlich auf die

automatische Parallelisierung von Schleifen, die erfahrungsgemäß ein hohes Maß an Parallelisierungspotential bieten. Die automatische Parallelisierung von Schleifen und deren Optimierung stellt einen Übersetzer jedoch vor einige Schwierigkeiten. Die Analyse, mit der die Unabhängigkeit der einzelnen Schleifendurchläufe bestimmt werden muss, ist bei der Verwendung von Zeigern, Rekursion oder indirekten Funktionsaufrufen nicht trivial. Zudem ist die Anzahl der Schleifendurchläufe zur Übersetzungszeit oftmals nicht bekannt. Auf Grund der niedrigen Abstraktionsebene fehlen einem Übersetzer zudem Kontextinformationen über das Programm.

Es liegt daher nahe, eine Auto-Tuning-Komponente zur grobgranularen Optimierung paralleler Programme auf höherer Abstraktionsebene in einen Übersetzer zu integrieren. Wir haben bereits einen möglichen Ansatz auf Basis der *GNU Compiler Collection (GCC)* [GCC] vorgestellt [KaSP09] . Die Grundidee beinhaltet einen zweistufigen Prozess. Zunächst wird das Programm auf normalem Wege übersetzt, einschließlich der gängigen Optimierungen auf Instruktionsebene. Anschließend werden im übersetzten Binärcode des Programms zusätzliche Tuning-Informationen hinterlegt, mit denen der Auto-Tuner die Werte von Tuning-Parametern im Binärformat ändern kann. Das tatsächliche Optimieren des Programms findet dann direkt nach dem Übersetzen als Teil des gesamten Vorgangs statt. Abbildung 3.5 zeigt den Vorgang schematisch. Die Kommunikation zwischen Auto-Tuner und Programm muss auch hier die Konfiguration der Parameter in der übersetzten Applikation sowie die Abfrage der Leistungswerte ermöglichen.

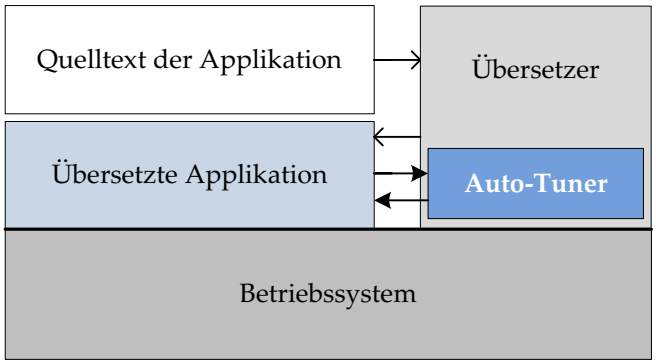


Abbildung 3.5: Darstellung des Auto-Tuners als Komponente des Übersetzers.

Der Vorteil dieses Ansatzes liegt im impliziten Tuning-Prozess. Durch die Integration in den Übersetzungsvorgang muss sich der Software-Entwickler nicht um die Optimierung kümmern, da diese bei einem Übersetzungsvorgang automatisch durchgeführt wird.

Der Nutzen dieses Ansatzes hängt zu einem großen Teil davon, wie schnell der Tuning-Vorgang durchgeführt werden kann. Wird Offline-Tuning eingesetzt, sollte bereits nach wenigen Tuning-Iterationen ein hinreichend gutes Ergebnis vorliegen. Sinnvoll wäre in diesem Zusammenhang ein Übersetzerbefehl, mit dem die Auto-Tuning-Komponente ein- und ausgeschaltet werden kann.

Die Integration eines Auto-Tuners für parallele Programme in einen Übersetzer ist ein vielversprechender Ansatz, der eine große Zahl an Möglichkeiten für zukünftige Forschung bietet.

3.5.4 Auto-Tuner als Komponente des Betriebssystems

Die Integration eines Auto-Tuners in ein Betriebssystem ist keine leichte Aufgabe, bietet aber durchaus einige Vorteile und eröffnet neue Möglichkeiten, die mit den bisherigen

Ansätzen nicht zu realisieren sind. Ein betriebsysteminterner Auto-Tuner kann als globale, anwendungsübergreifende Instanz betrachtet werden, bei der sich alle tuning-fähigen Applikationen registrieren können.

Als Teil des Betriebssystems hat die Auto-Tuning-Komponente auf systemweite Informationen und Ereignisse Zugriff. So kann zum Beispiel festgestellt werden, wann ein Programm gestartet wird. Eine transparente Tuning-Möglichkeit besteht deshalb darin, den Programmlader (unter Linux ist dies beispielsweise `/lib/ld-linux.so`) derart zu modifizieren, dass einer Applikation kurz vor der Ausführung neue Parameterwerte übergeben werden können. Dies kann, wie im vorherigen Abschnitt beschrieben, direkt im Binär-Code des Programms erfolgen. Dieser Ansatz eignet sich insbesondere in Verbindung mit Online-Tuning (siehe Abschnitt 3.4.1.1) und implizitem Tuning während produktiven Läufen des Programms (siehe Abschnitt 3.4.1.2).

Konsequenterweise kann man sich eine Auto-Tuning-Komponente auch als Teil des Ablaufplaners (engl. *scheduler*) vorstellen. Bei diesem Grad der Integration wäre der Auto-Tuner in der Lage, mehrere gerade ausgeführte parallele Programme aus einer globalen Sicht zu optimieren. Die optimale Parameterkonfiguration eines Programms würde auf diese Weise in Abhängigkeit von den beanspruchten Ressourcen anderer Programme ermittelt werden, was mit großer Wahrscheinlichkeit zu einer besseren Auslastung der gesamten Systemressourcen führt. In Verbindung mit Online-Tuning könnte auf sich ändernde Ressourcenverfügbarkeit und andere äußere Einflüsse dynamisch und unter Berücksichtigung aller aktiven Prozesse reagiert werden.

Abbildung 3.6 zeigt den Zusammenhang anhand eines schematischen Aufbaus. Es ist zu beachten, dass `Applikation 1` und `Applikation 2` auf eine gleichzeitige Ausführung hin optimiert werden. `Applikation 1` wird also bzgl. der benötigten Ressourcen in Abhängigkeit von `Applikation 2` optimiert und umgekehrt.

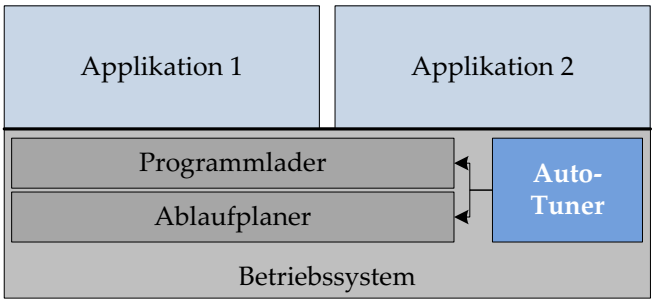


Abbildung 3.6: Darstellung des Auto-Tuners als Komponente des Betriebssystems.

Ähnlich wie die Integration eines Auto-Tuners in Übersetzer ist auch die Betriebssystemintegration bei Weitem noch nicht ausreichend erforscht. Die hier vorgeschlagenen Ansätze sollen als Anhaltspunkt und Grundlage für zukünftige Forschungsarbeiten dienen.

3.6 Programmanbindung von Auto-Tunern

In den vorangegangenen Abschnitten wurden verschiedene Methoden zur automatischen Performanzoptimierung sowie unterschiedliche Strategien zur Integration eines Auto-Tuners dargelegt.

Es bleibt jedoch die Frage, wie die Auto-Tuning-Komponente an das zu optimierende Programm angebunden wird. Um das Programm automatisch optimieren zu können, muss der Auto-Tuner folgende Schritte durchführen können:

- Ermitteln aller für den Optimierungsprozess relevanten Informationen über das Programm, kurz *Tuning-Instruktionen*. Zu den Tuning-Instruktionen gehören neben den Deklarationen der im Programm verfügbaren Tuning-Parameter (jeweils mit Bezeichner, Wertebereich sowie ggf. Standardwert und zusätzliche Kontextinformationen) auch Strukturinformationen über das Programm und dessen parallele Sektionen.

  Die Tuning-Instruktionen stellen die Eingabe des Optimierungsprozesses dar, aus denen der Auto-Tuner den zu erforschenden Suchraum generiert.

- Zuweisen und Anwenden der Parameterkonfiguration, die während des Optimierungsprozesses getestet werden sollen. Das Programm muss also gemäß der Tuning-Parameter durch den Auto-Tuner modifizierbar sein.

- Auslesen des Leistungskriteriums sowie der Leistungswerte aller Messpunkte.

Die beiden letzten Punkte spiegeln die bidirektionale Kommunikation zwischen Programm und Auto-Tuning wider, die bereits in den vorherigen Abschnitten im Kontext der Integrationsmöglichkeiten des Auto-Tuners deutlich wurde (siehe Abschnitt 3.5).

Die Anbindung des Auto-Tuners an das zu optimierende Programm darf nicht mit der eigentlichen Auto-Tuning-Funktionalität verwechselt werden. Erstere definiert, wie die Tuning-Instruktionen für den Auto-Tuner bereitgestellt werden und wie der Auto-Tuner mit dem Programm kommuniziert. Letztere stellt die Logik bereit, die entscheidet, mit welcher Strategie und unter Verwendung welcher Verfahren das Programm optimiert wird.

Der für die Anbindung zu wählende Ansatz hängt in großem Maße von der Art der Integration des Auto-Tuners in das Programm bzw. das zu Grunde liegende System ab (vgl. hierzu den vorherigen Abschnitt 3.5). Im Wesentlichen eignen sich zwei Ansätze, die im Folgenden kurz erläutert werden. In Kapitel 4 werden die wichtigsten existierenden Arbeiten in diesem Bereich aufgeführt, und Kapitel 5 erläutert den Ansatz, der in dieser Arbeit verfolgt wurde.

3.6.1 Direkte Anbindung im Programmquelltext

Die Anbindung des Auto-Tuners an das Programm kann durch Sprachkonstrukte der verwendeten Programmiersprache vorgenommen werden. Hierbei findet eine enge Verzahnung von Programm und Tuning-Informationen statt. Dieser Ansatz eignet sich demnach insbesondere dann, wenn der Auto-Tuner als Bibliothek eingebunden und somit ebenfalls Teil des Programms ist (vgl. 3.5.1).

Das C#-Listing 3.1 zeigt beispielhaft die Deklaration von Tuning-Parametern in Form eines C#-Attributs. Wir nehmen an, dass die (hier nicht näher spezifizierte) Implementierung des parallelen QuickSort (Klasse `ParallelQuickSort` mit Methode `Start()`) zwei Parameter besitzt, mit denen die Leistung beeinflusst werden kann: `numThreads` definiert die Anzahl der Ausführungsfäden (engl. *threads*), `batchSize` die Anzahl an Elementen, die von einem Faden auf einmal verarbeitet werden. Um die Deklaration der Tuning-Parameter vom funktionalen Quelltext des Programms möglichst klar zu trennen, wurde ein C#-Attribute `QuickSortTuningParameters` definiert, das einer Klasse zugewiesen werden kann und die beiden Parameter mit konkreten Werten als Argument akzeptiert. Die Parameterwerte werden über den Bibliotheksaufruf `AutoTuner.SetValue(string id,int min,int max)` gesetzt, der dem Auto-Tuner gleichzeitig den jeweiligen Wertebereich übermittelt. Die Methode `Start()` kann nun die Parameterwerte zur Laufzeit auslesen. Je nach Integration der Auto-Tuning-Komponente kann das Setzen der Parameterwerte auch über Kommandozeilenargumente geschehen.

Zu beachten ist, dass die Struktur sowie die Schnittstellen des Programms durch tuning-relevante Programmteile nicht beeinflusst werden. Beispielsweise bleibt die Signatur der Methode `Start()` unabhängig von den Tuning-Parametern, welche die Methode benötigt.

```
[QuickSortTuningParameters(
    numThreads=AutoTuner.SetValue("numThreads", 1, 8),
    batchSize=AutoTuner.SetValue("batchSize", 10, 100))
]
public class ParallelQuickSort
{
    public void Start(List<int> list)
    {
        Attribute[] attrs = Attribute.GetCustomAttributes(this);
        int numThreads = attrs["numThreads"];
        int batchSize = attrs["batchSize"];

        // Implementierung des parallelen Quicksorts
    }
}

public class Program
{
    public static void Main()
    {
        int[] array = GetLargeArray();
        ParallelQuickSort pqs = new ParallelQuickSort();
        pqs.Start(array);
    }
}
```

Listing 3.1: C#-Attribute zur Deklaration von Tuning-Parametern

Obwohl moderne Programmiersprachen Konzepte bereitstellen, mit denen tuning-spezifische Anweisungen intuitiv eingebunden werden können, so kann sich jedoch die direkte Anbindung durch Modifikation und Erweiterung des Programms als unflexibel erweisen, da die hierfür nötige Programmlogik (Deklaration der Tuning-Parameter, Setzen der Parameterwerte sowie Auslesen der Leistungswerte) manuell implementiert werden muss. Es entsteht somit ein hoher Implementierungsaufwand, der bei jedem zu optimierenden Programm erneut anfällt.

Des Weiteren können Informationen über parallele Sektionen und die allgemeine Programmstruktur nur in wenigen Fällen direkt in der verwendeten Programmiersprache beschrieben werden. Hierfür wird eine höhere Abstraktionsebene benötigt.

3.6.2 Instrumentierung des Quelltextes

Eine wesentlich höhere Flexibilität bietet die Instrumentierung des Quelltextes mittels einer domänenspezifischen Annotationssprache. Bei dieser Methode werden Annotation in den Quelltext eingefügt, die Anweisungen und Informationen für den Auto-Tuner enthalten. Durch die Instrumentierungen, die in der Regel an konkrete Quelltextbereiche oder einzelne Anweisungen gebunden sind, werden die tuning-relevanten Programmteile sowie Programmvariablen, die als Tuning-Parameter fungieren, markiert.

Im Gegensatz zur direkten Anbindung geht mit der Instrumentierung in den meisten Fällen eine Quelltexttransformation einher. Das bedeutet, dass die vom Auto-Tuner ge-

forderten Modifikationen von einem Quelltextgenerator erzeugt und an entsprechender Stelle im Programm eingefügt werden. Somit kann die Instrumentierung des Quelltextes als eine kompakte Darstellung mehrerer Programmvarianten betrachtet werden [DBRY+06].

Beispiele für gängige Transformationen sind das Ändern von Parameterwerten bzw. Wertzuweisungen oder das Anwenden von Schleifenoptimierungen (vgl. [HaPo09], [YSYV+07], [DBRY+06]). Mit entsprechenden Instrumentierungen werden Anweisungen deklariert und Informationen bereitgestellt, die ein herkömmlicher Übersetzer nicht besitzt (vgl. hierzu Abschnitt 3.5.3). Der Quelltextgenerator übernimmt dann die Aufgabe eines übergeordneten Übersetzers.

Auf grobgranularerer Ebene können mit Instruktionen auch Aufrufe zu Bibliotheken oder Komponenten angepasst werden, wie in [TaCH02] beschrieben.

Des Weiteren können mittels Instrumentierungen auch Messpunkte definiert werden. Bei der Transformation werden die in der Annotationssprache deklarierten Messpunkte beispielsweise durch Aufrufe zu einer Messbibliothek ersetzt [ScPT09].

Listing 3.2 greift das einfache Beispiel aus dem vorherigen Abschnitt nochmals auf und zeigt exemplarisch eine mögliche Instrumentierung der beiden Variablen `numThreads` und `batchSize`, die damit als Tuning-Parameter deklariert sind. Ein Auto-Tuner kann die Informationen einlesen und mittels Quelltexttransformation die Instrumentierungen durch konkrete Wertzuweisungen ersetzen.

```
public class ParallelQuickSort
{
    public void Start(int[] array)
    {
        int numThreads = 2; // Zuweisung eines Standardwertes
#pragma atune setvar numThreads values 2-8
        int batchSize = 10; // Zuweisung eines Standardwertes
#pragma atune setvar batchSize values 10-100

        // Implementierung des parallelen Quicksorts
    }
}

public class Program
{
    public static void Main()
    {
        int[] array = GetLargeArray();
        ParallelQuickSort pqs = new ParallelQuickSort();
        pqs.Start(array);
    }
}
```

Listing 3.2: Annotationen zur Deklaration von Tuning-Parametern

Der Vorteil einer separaten Annotationssprache liegt in der Trennung von Programmquelltext und Anweisungen für den Auto-Tuner. Der Quelltext des Programms bleibt beim Annotieren unverändert; die Lesbarkeit und Verständlichkeit des Quelltextes wird nicht durch auto-tuning-bezogenen Quelltext verschlechtert.

Außerdem verspricht eine Annotationssprache einen deutlich geringeren Implementierungsaufwand, da bei jedem zu optimierenden Programm nur die Instrumentierungen

eingefügt werden müssen. Anpassungen des Quelltextes sowie die Implementierung der Logik fallen nicht an.

Nicht zuletzt ist der Ansatz in jeweils angepasster Form für nahezu alle Arten der Auto-Tuner-Integration sowie Tuning-Strategien geeignet, da durch die separaten Annotationen vom verwendeten Auto-Tuning-Konzept sowie vom Programm selbst abstrahiert wird.

Allerdings ist das Verfahren komplexer als die direkte Anbindung des Auto-Tuners, da die Annotationen eingelesen und ausgewertet werden müssen. Neben syntaktischer muss eine semantische Überprüfung stattfinden, um ungültige Annotationen zu identifizieren. Schließlich müssen mittels passender Quelltexttransformationen korrekte Programmvarianten erzeugt werden, die dann getestet und deren Leistungswerte erfasst werden müssen.

Grundsätzlich ist zu bedenken, dass insbesondere feingranulare Transformationen auf Instruktionsebene (wie beispielsweise Schleifenoptimierungen) auch von einem Übersetzer durchgeführt werden können (vgl. hierzu Abschnitt 3.5.3). Es muss daher entschieden werden, welche Optimierungen der Übersetzer implizit durchführen kann und ab welcher Abstraktionsebene eine separate Auto-Tuning-Komponente sinnvoll ist. Im Rahmen dieser Arbeit beschäftigen wir uns ausschließlich mit Optimierungen auf höherer Abstraktionsebene, für deren effiziente Durchführung eine Reihe von Informationen benötigt werden, die einem herkömmlichen Übersetzer nicht zur Verfügung stehen.

Aus Gründen der Flexibilität und der Anforderung, den gesamten Auto-Tuning-Prozess von der Applikation zu trennen, basieren die Konzepte dieser Arbeit auf einer eigens entwickelten Instrumentierungssprache, die in Kapitel 5.2 erörtert wird.

3.7 Optimierbare Programme und deren Architektur

Ein paralleles Programm bietet in den meisten Fällen umfangreiches Optimierungspotential. Daraus folgt jedoch nicht, dass das Programm auch optimierbar ist, es also die Fähigkeit besitzt, sein Verhalten bezüglich bestimmter Parameter zu ändern.

Im einfachen Fall wird durch einen Tuning-Parameter lediglich der Wert einer Programmeigenschaft modifiziert, beispielsweise die Anzahl an Ausführungsfäden, die zur Verarbeitung eines parallelen Problems verwendet werden. Eine weitaus komplexere Situation ergibt sich, wenn durch einen Tuning-Parameter das Verhalten eines ganzen Programmteils angepasst oder die Auswahl einer geeigneten Parallelisierungsstrategie gesteuert werden soll.

In beiden Fällen muss das Programm in entsprechender Weise adaptierbar sein und entsprechende Funktionalität bereitstellen, mit der es auf einen geänderten Parameterwert reagieren und sein Verhalten anpassen kann. Die hierfür nötige Programmlogik bezeichnen wir als die *Implementierung des Tuning-Parameters*.

Besitzt ein Programm für all seine Tuning-Parameter die nötigen Implementierungen, so gilt das Programm als optimierbar.

Im Folgenden betrachten wir parallele Programme aus Sicht der Optimierbarkeit. Zunächst soll die Identifikation von Optimierungspotential diskutiert werden, danach in einer kurzen Abhandlung, welche softwaretechnischen Methoden zur effizienten Implementierung von Tuning-Parametern herangezogen werden können. Schließlich steht die Architektur paralleler Programme im Fokus unserer Untersuchung, deren Definition in den Kontext von Parallelisierung und Optimierung gesetzt wird.

3.7.1 Identifikation von Optimierungspotential

Bevor ein Programm parametrisiert und um entsprechende Konfigurationsmöglichkeiten erweitert werden kann, müssen zunächst alle Stellen im Programm identifiziert werden, die Optimierungspotential bieten. Dies sind leistungskritische Programmbereiche, deren optimale Implementierung hinsichtlich der Performanz des Programms für eine bestimmte Hardware-Plattform nicht eindeutig ist.

Wann immer man während der Implementierung eines Programms vor einer nicht eindeutigen Entscheidung steht, deren Ergebnis die Leistung des Programms beeinflussen kann (z.B. das Setzen eines Variablenwertes, die Wahl einer Algorithmenvariante, die Reihenfolge von Anweisungen oder die Wahl einer Parallelisierungsstrategie), ist Optimierungspotential gefunden, das durch einen Tuning-Parameter markiert und später ausgenutzt werden kann.

Es ist zu beachten, dass Tuning-Parameter auf nahezu allen softwaretechnischen Abstraktionsebenen einer parallelen Applikation auftreten können. Auf oberster Ebene können Tuning-Parameter die Wahl einer bestimmten Architekturvariante oder ganzer Parallelisierungsstrategien beeinflussen, während auf unterster Ebene der Ausrollfaktor einer Schleife oder die Anzahl an Fäden für einen Algorithmus angepasst werden können.

Da sich diese Arbeit mit der Parallelisierung und Optimierung großer paralleler Applikationen beschäftigt, werden in späteren Kapiteln Verfahren und Konzepte vorgestellt, die insbesondere eine Optimierung auf hoher architektonischer Programmebene ermöglichen.

Neben der eigentlichen Identifizierung von Tuning-Parametern und dem damit verbundenen Optimierungspotential ist für jeden Parameter die Einschätzung seiner Sensitivität von großer Wichtigkeit (siehe Definition 3.9). Sie beschreibt, wie stark ein Tuning-Parameter Einfluss auf die Gesamtleistung des Programms nimmt. Bei der späteren automatischen Performanzoptimierung kann der Sensitivitätswert eines Parameter zur Entscheidung beitragen, wie mit dem Parameter verfahren wird. Beispielsweise kann ein Tuning-Parameter mit geringer Sensitivität zunächst vom Optimierungsprozess ausgeschlossen werden, um schneller eine gute Konfiguration für die Parameter mit hoher Sensitivität zu finden.

Mit der Parametersensitivität kann also der Grad des Optimierungspotentials eines Tuning-Parameters spezifiziert werden. Die Angabe eines Sensitivitätswertes kann entweder manuell durch den Software-Entwickler oder automatisch durch den Optimierer erfolgen. Letzteres setzt zusätzliche Tuning-Iterationen voraus, in denen für jeden Parameter die jeweilige Sensitivität ermittelt wird. Hierbei kann beispielsweise wie folgt verfahren werden (vgl. [ChHo04]): Für jeden Parameter p_i wird eine bestimmte Anzahl an Werten getestet (z.B. auf Basis einer gegebenen Schrittweite), während alle anderen Parameter auf ihren Standardwert gesetzt sind (dies entspricht maskierten Parameterkonfigurationen über p_i). Nach den Testläufen für p_i wird die Differenz zwischen dem besten und dem schlechtesten der erzielten Leistungswerte errechnet, woraus sich schließlich durch Normalisierung ein Sensitivitätswert ableiten lässt.

3.7.2 Adaptierbarkeit von Programmen

Wie eingangs erwähnt, geht die Optimierbarkeit eines Programms mit seiner Konfigurierbarkeit bzw. Adaptierbarkeit einher. Grundsätzlich eignet sich das Prinzip der externen Steuerung zur Adaption des Programms an Hand von Messwerten. Abbildung 3.7 zeigt schematisch den entsprechenden Ablauf.

Konzepte für adaptierbare Programme verfolgen zwei wesentliche Zielsetzungen. Zum einen muss sichergestellt sein, dass die Adaptions-Funktionalität in eine Vielzahl von

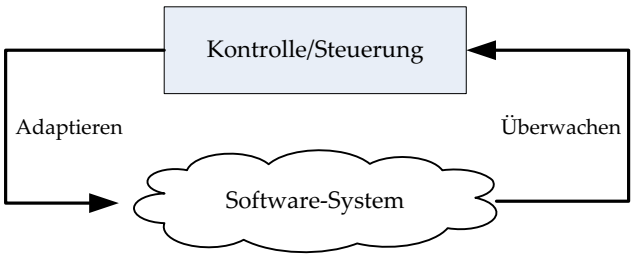


Abbildung 3.7: Schema der externen Steuerung und Überwachung eines Software-Systems zur dynamischen Adaption.

Programmen integriert werden kann, die sich in Struktur, Typ oder Anwendungsdomäne unterscheiden. Zum anderen darf die zusätzliche Funktionalität den eigentlichen Ablauf des Programms nicht beeinflussen und die Integration sollte möglichst mit moderatem Aufwand zu bewerkstelligen sein [GCHS+04].

Es ist notwendig, die Anforderungen an ein derartiges Konzept klar zu definieren. Beispielsweise genügt es für einen Offline-Tuning-Verfahren, wenn das Programm zwischen den einzelnen Tuning-Läufen adaptiert werden kann. Online-Tuning-Verfahren benötigen dynamisch rekonfigurierbare Programme, die zur Laufzeit ihr Verhalten ändern können, indem automatisch eine neue Konfiguration geladen und angewendet wird.

Selbstadaptierende Programme (engl. *runtime software adaptation*) weisen ähnliche Eigenschaften wie dynamisch rekonfigurierbare Programme auf, wobei erstere automatisch auf eine geänderte Umgebungssituation reagieren können und daher in erster Linie von Überwachungsdaten des Systems abhängen. Da diese Funktionalität jedoch zusätzliche Programmlogik erfordert, die wie in Abbildung 3.7 auch durch eine externe Komponente beigesteuert werden kann, ist der Übergang zwischen dynamisch rekonfigurierbaren und selbstadaptierenden Programmen als fließend zu betrachten.

Es existieren unterschiedliche Konzepte für Software-Adaption und Rekonfiguration. Der Artikel von Oreizy et al. [OrMT08], in dem neben einem eigenen Ansatz auch andere existierende Konzepte ausführlich beschrieben werden, bietet einen guten Überblick.

Ein verbreiteter Ansatz zur Spezifikation von Konfigurations- und Adaptionsfunktionalität basiert auf Modellen, die das Programm und seine Architektur beschreiben. Diese Modelle werden um entsprechende Adaptionsoperatoren, Randbedingungen und Konfigurationsstrategien erweitert [GCHS+04, GaSC09] und können im Gegensatz zur herkömmlichen Nutzung von Architekturmodellen während der Entwurfsphase auch zur Laufzeit eingesetzt werden.

Diese so genannten dynamischen Adaptionsmodelle (engl. *dynamic adaptation models*) ermöglichen eine starke Trennung der Adaptionsstrategien (die meist auch Gültigkeitsüberprüfungen beinhalten) von der tatsächlichen Implementierung des Programms. Die Rahmenbedingungen und Beschränkungen des Systems hinsichtlich Topologie und Verhalten werden durch die Verwendung eines Adaptionsmodells zur Laufzeit explizit verfügbar gemacht. Somit wird ein Handlungsspielraum definiert, innerhalb dessen gültige Änderungen am Programm durchgeführt werden können.

Eine weitere konzeptionelle Möglichkeit besteht in der direkten Spezifikation von Konfigurationsmöglichkeiten (ähnlich der Spezifikation von Tuning-Instruktionen, vgl. Abschnitt 3.6). Zu jedem Programm werden eine Reihe von Konfigurationsprofilen [RaPo03] erstellt, die von einer zentralen Instanz ausgelesen und unter Verwendung eines Rekon-

figurationsalgorithmus angewendet werden. Theoretische Überlegungen zu Rekonfigurationsalgorithmen sind beispielsweise in [Werm97] zu finden.

Zur Implementierung rekonfigurierbarer Programme werden verschiedene Verfahren eingesetzt. Zu den klassischen Ansätzen gehört die Verwendung eines mehr oder weniger komplexen Rahmenwerkes, das dem Entwickler zur Verfügung gestellt wird, um die konkrete Adaptions- bzw. Rekonfigurationsfunktionalität zu implementieren und anschließend in das Programm zu integrieren. Rahmenwerke gehen meist mit der Einbindung von Bibliotheken einher, die die übergeordnete Logik bereitstellen.

Einige neuere Ansätze beschäftigen sich mit der Verwendung von *Aspektorientierter Programmierung (AOP)* [KLMM+97], um eine dynamische Konfigurierbarkeit der Programme zu erreichen [MRSB+07, MuRC07, RaPo03].

Der in dieser Arbeit verfolgte Ansatz basiert konzeptionell auf einem Modell für konfigurierbare parallele Architekturen, mit dem die parallele Struktur eines Programms beschrieben wird. Alle Konfigurationsmöglichkeiten werden implizit integriert, da die optimale Konfiguration des implementierten Programms schließlich durch einen Auto-Tuner gefunden werden soll. Aus Sicht der Implementierung wird der Ansatz durch vordefinierte konfigurierbare Komponenten umgesetzt, die eine einheitliche Schnittstelle zur Verfügung stellen. In Kapitel 5.3 wird das Verfahren erörtert.

Zusammenfassend können wir festhalten, dass alle leistungskritischen Stellen eines parallelen Programms parametrisiert und adaptierbar sein müssen, um später mittels automatischer Performanzoptimierung die bestmögliche Programmkonfiguration zu finden.

3.7.3 Optimierung der Architektur paralleler Programme

Die Architektur eines parallelen Programms spielt bei der Optimierung eine entscheidende Rolle, da durch sie festgelegt wird, welche Programmteile mit welcher Strategie parallelisiert werden. Hinzu kommt, dass durch eine geeignete Struktur der Softwarearchitektur auf bewährte Elemente zurückgegriffen werden kann, um den Aufwand für Identifikation und Implementierung von Tuning-Parametern zu verringern.

Um diesen Sachverhalt genauer zu ergründen, wird zunächst der Begriff der Softwarearchitektur diskutiert und dieser dann in den Kontext optimierbarer Programme eingeordnet.

Die ersten relevanten Forschungsarbeiten auf dem Gebiet der Softwarearchitektur begannen nach Taylor et al. [TavdH07] in den frühen 1990er-Jahren, auch wenn der Begriff selbst deutlich älter ist. Allen voran sind hier insbesondere die Arbeiten von Shaw et al. [SDKR+95] sowie Perry und Wolf [PeWo92] zu nennen. Die Gruppen entwickelten jeweils ein Modell zur Definition von Architekturen. Diese Arbeiten bilden somit die Grundlage zur formalen Architekturenbeschreibung.

In der Literatur ist jedoch keine einheitliche Definition für Softwarearchitektur zu finden. Shaw und Garlan [ShGa96] formulierten eine Definition, die für den thematischen Rahmen dieser Arbeit geeignet erscheint und die daher den entsprechenden Konzepten der Arbeit zu Grunde gelegt werden soll:

> *„Software architecture [is a level of design that] involves the description of elements from which systems are built, interaction among those elements, patterns that guide their composition, and constraints on these patterns“.*

Eine Softwarearchitektur besteht demnach aus einer Beschreibung von Komponenten und deren Interaktion sowie von Mustern, die die Komposition unterschiedlicher Komponenten steuern. In der Regel repräsentiert ein entsprechendes Architekturmodell einen Graphen, dessen Knoten die *Komponenten* des Programms und dessen Kanten *Konnektoren* darstellen, die die Komponenten verbinden.

Diese Definition lässt sich auch auf Architekturen *paralleler* Programme erweitern. Als grundlegende Komponenten einer parallelen Architektur definieren wir die sequentiellen funktionalen Teile des Programms, welche elementare Aufgaben ausführen. Die Interaktion zwischen den Komponenten wird von vordefinierten Mustern gesteuert, wobei ein Muster eine konkrete Parallelisierungsstrategie implementiert. Die Architektur eines parallelen Programms bestimmt also, ob und in welcher Weise die elementaren Aufgaben des Programms auf grobgranularer Ebene parallel ausgeführt werden.

Betrachten wir die Definition aus der Sicht der Optimierung, so können wir Aspekte identifizieren, die bereits ohne konkrete Beschreibung der Architektur als allgemeine Indikatoren für Optimierungspotential auf architektonischer Ebene dienen können. Leistungsrelevant ist zum Beispiel die Wahl der parallelen Verarbeitungsstrategie. Sind mehrere Variationen möglich, muss die beste gefunden werden. Des Weiteren bieten sich die parallelen Verarbeitungsstrategien selbst an, um die Leistung des Programms zu optimieren. Es stellt sich umgehend die Frage nach potentiellen Tuning-Parametern der Verarbeitungsstrategie sowie deren optimaler Konfiguration.

Die Parallelisierung und Optimierung auf architektonischer Ebene kann äußerst vielversprechend sein. Unterstützt wird diese Beobachtung auch von aktuellen Studien (z.B. [PaJT09]), die zeigen, dass grobgranulare Parallelisierung und anschließende Optimierung, verglichen mit Verfahren auf algorithmischer Ebene, eine wesentlich effizientere Methode darstellt.

3.7.3.1 Parallele Entwurfsmuster

Eine wichtige Rolle bei der Optimierung paralleler Architekturen spielen die so genannten *parallelen Entwurfsmuster*, die gewissermaßen als Bindeglieder in einer parallelen Architektur verstanden werden können. Parallele Entwurfsmuster bilden seit einiger Zeit einen festen Bestandteil bei der Entwicklung paralleler Programme und eignen sich sowohl zur Architekturbeschreibung als auch zur strukturierten Programmoptimierung.

Das Konzept der Entwurfsmuster wurde ursprünglich von Christopher Alexander im Kontext der physischen Architektur eingeführt, dessen Definition wir an dieser Stelle zitieren [AlIS77]:

> *„Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice“.*

Gamma et al. [GHJV95] übertrugen dieses Konzept in den Bereich der Softwaretechnik und haben einen Entwurfsmusterkatalog zusammengestellt, der jedes enthaltene Muster einheitlich definiert und beschreibt. Entwurfsmuster stellen in diesem Kontext bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme in Softwarearchitektur und Softwareentwicklung dar. Einige der Entwurfsmuster unterstützen ausschließlich den Entwurfsprozess, andere bieten zudem konkrete Implementierungen an.

Mit der Spezifikation der parallelen Entwurfsmuster wurde das Prinzip auf die parallele Programmierung ausgeweitet. Ähnlich wie die Entwurfsmuster in der objektorientierten

Programmierung bieten auch die parallelen Muster jeweils eine vordefinierte Lösung für eine Klasse paralleler Probleme, indem sie eine bewährte Parallelisierungsstrategie definieren. Ein umfangreicher Katalog sowie Diskussionen und Erläuterungen zu parallelen Mustern ist in der Arbeit von Mattson et al. [MaSM04] zu finden.

Im Folgenden werden vier gebräuchliche parallele Entwurfsmuster vorgestellt und jeweils einem der drei Parallelitätstypen zugeordnet:

- *Fließbandparallelität* entsteht, wenn Datenelemente von mehreren Aufgaben nacheinander verarbeitet werden sollen, wobei unterschiedliche Datenelemente von unterschiedlichen Aufgaben zur gleichen Zeit verarbeitet werden können.
- *Aufgabenparallelität* ordnet unterschiedliche Aufgaben, die parallel durchgeführt werden können, mehreren Ausführungsfäden zu.
- *Datenparallelität* wird ausgenutzt, wenn die gleiche Aufgaben parallel auf alle Datenelemente einer Datenstruktur angewendet wird.

Fließband-Muster

Das *Fließband-Muster* repräsentiert das Konstrukt zur Ausnutzung von *Fließbandparallelität*. Das Konzept des Fließbandes ist dem einer Fertigungsstraße entliehen, auf der ein Produkt in mehreren Stufen zusammengesetzt wird.

Übertragen auf die Softwaretechnik stellen die Stufen hintereinander geschaltete Berechnungen und das Produkt ein Datenelement dar. Das Ergebnis einer Stufe ist somit die Eingabe der nächsten Stufe. Die Verarbeitung *eines* Datenelements erfolgt sequentiell; müssen jedoch mehrere Datenelemente verarbeitet werden, können die Berechnungen auf unterschiedlichen Datenelementen parallel ausgeführt werden. Jede Stufe muss daher in einem separaten Ausführungsfaden ablaufen.

Abbildung 3.8 verdeutlicht das Prinzip an Hand eines 3-stufigen Fließbandes mit 3 Datenelementen (D_1, D_2 und D_3). Zu keinem Zeitpunkt t_1 bis t_5 wird dasselbe Datenelement von zwei Stufen gleichzeitig verarbeitet. Parallelität entsteht durch die Verarbeitung unterschiedlicher Datenelemente zur gleichen Zeit in unterschiedlichen Stufen.

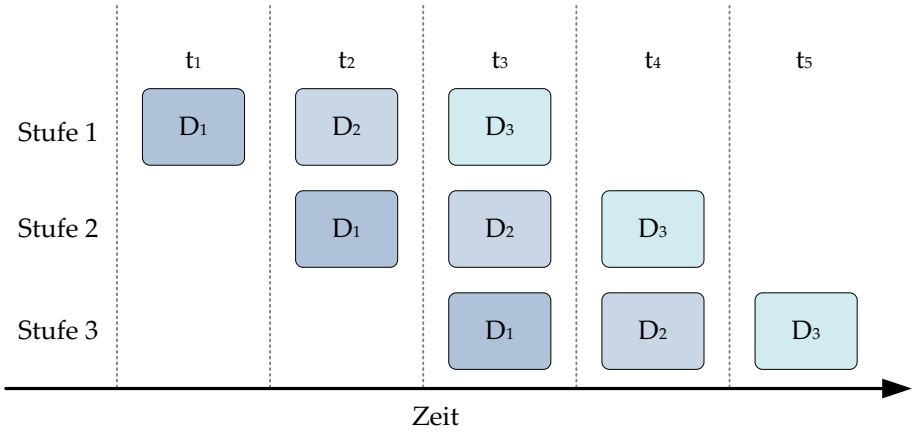


Abbildung 3.8: Darstellung der Fließbandparallelität an Hand eines 3-stufigen Fließbandes.

Abbildung 3.9 skizziert das Fließband-Muster: die einzelnen Stufen werden durch zwischengeschaltete Puffer zu einem Fließband verbunden. Die Puffer dienen zur Zwischenspeicherung der verarbeiteten Datenelemente einer Stufe, bis die nächste Stufe die Datenelemente entgegennehmen kann.

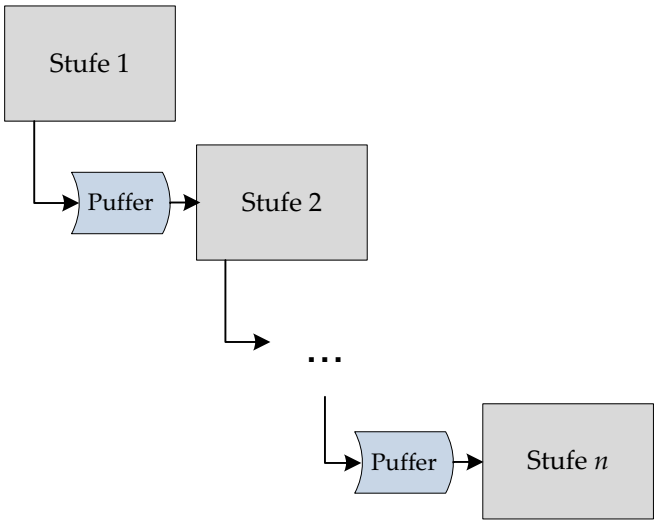


Abbildung 3.9: Schema des Fließband-Musters.

Man unterscheidet zwischen linearen und nicht-linearen Fließbändern. Erstere bestehen aus einer linearen Verkettung von Fließbandstufen, während bei letzteren der Datenfluss durch Verzweigungen und Vereinigungen aufgeteilt werden kann.

Erzeuger/Verbraucher-Muster

Das Erzeuger/Verbraucher-Muster (engl. *Producer/Consumer Pattern*) ist ein Spezialfall des Fließbandes und beschreibt ein klassisches Synchronisationskonstrukt zwischen zwei Aufgaben, die über einen gemeinsamen Puffer Daten austauschen. Hierbei legt der Erzeuger von ihm verarbeitete Datenelemente in den Puffer, aus dem der Verbraucher sie zur Weiterverarbeitung herausnimmt. Der Zugriff auf den Puffer muss synchronisiert erfolgen. Abbildung 3.10 zeigt das Schema des Musters.

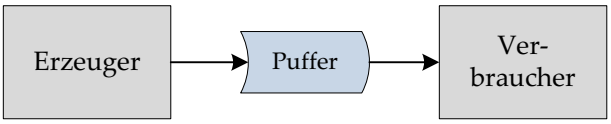


Abbildung 3.10: Schema des Erzeuger/Verbraucher-Musters.

Master/Worker-Muster

Das *Master/Worker-Muster* beschreibt das klassische Konstrukt zur Ausnutzung von *Aufgabenparallelität*, wie in Abbildung 3.11 dargestellt. Der Hauptfaden (*Master*) erzeugt mehrere Arbeiterfäden (*Worker*), denen er jeweils Teilaufgaben zuweist. Diese Aufgaben werden von den Arbeiterfäden parallel ausgeführt. Der Hauptfaden selbst wartet, bis die Teilaufgaben durchgeführt wurden, oder erledigt eine separate Aufgabe. Meist ist das Zusammenführen der Arbeiterfäden mit einer impliziten Barriere verbunden, so dass alle Fäden (einschließlich des Hauptfadens) warten, bis die Teilaufgaben abgearbeitet sind.

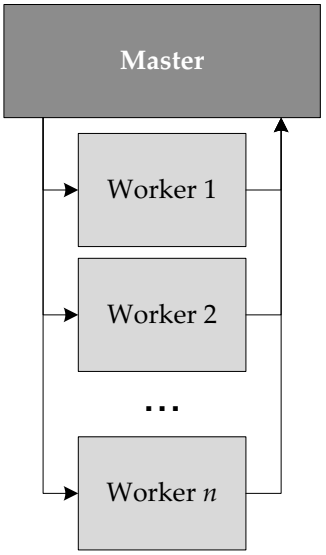


Abbildung 3.11: Schema des Master/Worker-Muster.

Eine Variante des Master/Worker-Musters ist das *Fork/Join-Muster*. Der Hauptfaden erzeugt hier ebenfalls mehrere Arbeiterfäden, beteiligt sich aber an der Abarbeitung der Teilaufgaben und wird auf diese Weise selbst zum Arbeiterfaden. Es findet also eine gabelartige Aufspaltung des Programmflusses in mehrere Fäden (*fork*) und nach paralleler Abarbeitung der Teilaufgaben eine Zusammenführung mit impliziter Barriere zu einem Hauptfaden (*join*) statt.

Gebietszerlegung

Das Muster der *Gebietszerlegung* beschreibt im Wesentlichen das Konstrukt zur Ausnutzung von *Datenparallelität*. Wie Abbildung 3.12 zeigt, findet eine Gebietszerlegung auf den Eingabedaten statt. Die resultierenden kleineren Datenpartitionen können anschließend von mehreren Instanzen derselben Aufgabe parallel verarbeitet werden. Jede Instanz wird hierbei einem separaten Ausführungsfaden zugewiesen. Eine Datenpartitionen kann auch nur aus einem Datenelement bestehen, so dass jedes Datenelement potentiell parallel verarbeitet werden kann.

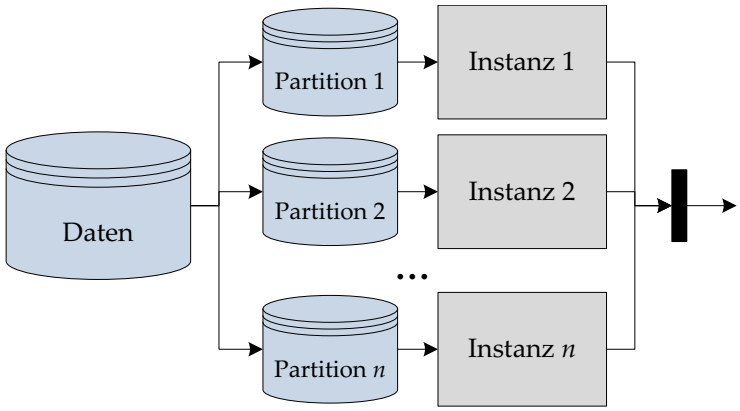


Abbildung 3.12: Schema der Gebietszerlegung.

Das Muster der Gebietszerlegung ähnelt dem Fork/Join-Muster: auch bei der Gebietszerlegung existiert eine implizite Barriere, so dass beim Zusammenführen der parallelen Ausführungsfäden gewartet wird, bis alle Datenpartitionen verarbeitet sind und die Ergebnisse vorliegen. Der Unterschied besteht in den Aufgaben. Während beim Fork/Join-Muster *unterschiedliche* Aufgaben parallel ausgeführt werden, wird bei der Gebietszerlegung *dieselbe* Aufgabe mehrfach repliziert, um Datenelemente parallel verarbeiten zu können.

Im Sinne der Taxonomie von Flynn [Flyn72] kann das Fork/Join-Muster als softwaretechnische Umsetzung von *Multiple-Instructions-Multiple-Data (MIMD)*, die Gebietszerlegung als *Single-Instruction-Multiple-Data (SIMD)* verstanden werden.

Neben den oben genannten parallelen Entwurfsmustern existieren noch einige weitere, beispielsweise das Wavefront-Muster [AMSS+02] oder das parallele Teile-und-Herrsche-Muster [MaSM04]. Da im Rahmen dieser Arbeit nicht alle bekannten parallelen Muster erwähnt werden können, wurden an dieser Stelle die wichtigsten Vertreter vorgestellt. Insbesondere wurde darauf geachtet, dass für jeden Parallelitätstyp mindestens ein passendes Muster aufgeführt wird.

Anhand der Beispiele wird ersichtlich, dass sich parallele Entwurfsmuster in bester Weise eignen, um hinsichtlich unserer Definition für die Architektur paralleler Programme die Verarbeitung, Interaktion sowie Kommunikation von Komponenten zu beschreiben und zu steuern.

Jedes parallele Entwurfsmuster beschreibt eine konkrete parallele Sektion in einem Programm. Da nahezu jede parallele Sektion Optimierungspotential bietet, liegt es nun nahe, parallele Muster hinsichtlich ihres leistungsrelevanten Verhaltens zu analysieren und mit entsprechenden Tuning-Parametern zu versehen. Ein paralleles Entwurfsmuster, das externalisierte Tuning-Parameter zur Verfügung stellt, wird als *optimierbares paralleles Muster* bezeichnet.

Für den Entwurf und die anschließende Implementierung einer parallelen Architektur ergeben sich durch optimierbare parallele Muster entscheidende Vorteile. Mittels dieser vordefinierten parametrisierten Bausteine können auf äußerst effiziente Weise parallele Architekturen entworfen und umgesetzt werden, die an den entscheidenden Stellen bereits implizit Optimierungsoptionen definieren. Einige Forschungsarbeiten beschäftigen sich mit der Idee, parametrisierte Muster zu modellieren, um deren Leistung unter bestimmten Bedingungen vorauszuberechnen [CMSL04, CéSL05, LiMa06, MCGS+08]. Auf diese Weise kann ein Programm, das basierend auf einem konkreten Muster implementiert wurde, an Hand des Leistungsmodells für das entsprechende Muster (vor-)optimiert werden. In Kapitel 4 wird näher auf die verwandten Arbeiten in diesem Gebiet eingegangen.

Auch Teile dieser Arbeit basieren auf parametrisierten parallelen Mustern, da sich gezeigt hat, dass die Optimierung großer paralleler Anwendungen besonders dann effizient umgesetzt werden kann, wenn sich die Architektur aus bereits bekannten Bausteinen zusammensetzt. In Kapitel 5.3 wird dieser Sachverhalt unter Hinzuziehung der entsprechenden Konzepte erklärt.

3.8 Relevanz der Eingabedaten

Der Leistungswert eines Programms hängt in großem Maße von den Eingabedaten ab, also von der konkreten Instanz des zu lösenden Problems. Dies führt dazu, dass bei der

Optimierung eines Programms die Eigenschaften der Eingabedaten berücksichtigt werden müssen, da sich bei unterschiedlichen Eingabedaten möglicherweise auch die optimale Konfiguration der Tuning-Parameter ändert. Hierzu sind folgende Überlegungen nötig.

3.8.1 Problemgröße und -komplexität

Bei der Beschreibung von Eingabedaten eines Algorithmus wird zwischen Problemgröße und Problemkomplexität unterschieden. Beide Begriffe stammen aus der Komplexitätstheorie und dienen als Maß, um die Laufzeit und die Effizienz von Algorithmen in Abhängigkeit der Eingabedaten zu bestimmen [CLRS01].

Die Problemgröße kann in der Regel mit einem oder mehreren skalaren Werten beschrieben werden, die Auskunft darüber geben, welchen Umfang die Eingabedaten besitzen. Die Komplexität des Problems beschreibt in Abhängigkeit von der Problemgröße den Aufwand, den der Algorithmus erbringen muss, um das konkrete Problem in Form der Eingabedaten zu lösen. Eine obere und untere Schranke der Komplexität eines Algorithmus wird mit der *$\mathcal{O}$-Notation* angegeben.

Zum besseren Verständnis soll der Sortieralgorithmus *Insertionsort* [CLRS01] betrachtet werden. Die Eingabedaten bestehen aus einer Sequenz von n Zahlen $\langle a_1, a_2, ..., a_n \rangle$. Die Ausgabe des Algorithmus stellt eine Permutation $\langle a'_1, a'_2, ..., a'_n \rangle$ der Eingabe dar, so dass $a'_1 \leq a'_2 \leq ... \leq a'_n$ gilt. Die Problemgröße einer konkreten Instanz des Sortierproblems entspricht der Anzahl an Elementen, die zu sortieren sind, also n. Die Problemkomplexität beschreibt in diesem Beispiel, welche Permutation der n Zahlen die Eingabesequenz repräsentiert. Die Eingabesequenz bestimmt daher den Sortieraufwand, den der Algorithmus zu bewältigen hat. Der beste Fall liegt vor, wenn die Eingabesequenz vollständig sortiert ist, so dass bereits $a'_1 \leq a'_2 \leq ... \leq a'_n$ gilt (Aufwand entspricht $\mathcal{O}(n)$). Im ungünstigsten Fall ist die Eingabesequenz absteigend sortiert (Aufwand entspricht $\mathcal{O}(n^2)$).

Das Beispiel zeigt, dass die Laufzeit eines Algorithmus durch die Eigenschaften der Eingabedaten stark beeinflusst werden kann. Dieser Tatsache gilt es bei der automatischen Performanzoptimierung Rechnung zu tragen.

Zu beachten ist allerdings, dass nicht bei allen Algorithmen ein abhängig von den Eingabedaten unterschiedlicher Aufwand eintritt. Beispielsweise besitzt der Sortieralgorithmus *Mergesort* stets einen Aufwand von $\mathcal{O}(n \log n)$ – unabhängig von der Komplexität des Problems [CLRS01]. Aus Sicht der Optimierung ist Mergesort demnach einfacher zu handhaben als Insertionsort.

3.8.2 Berücksichtigung von Eingabedaten

Die Optimierung eines Programms findet meist auf einer festen Menge an Eingabedaten statt. Daraus folgt, dass die ermittelte optimale Parameterkonfiguration streng genommen nur für die getesteten Eingabedaten die bestmögliche Leistung des Programms erzielt.

Beim Ansatz des Online-Tunings (vgl. Abschnitt 3.4.1.1) wiegt die Abhängigkeit von den Eingabedaten nicht in dem Maße, wie dies bei Offline-Tuning der Fall ist, da auf unterschiedliche Eingabedaten zur Laufzeit reagiert werden kann.

In jedem Fall müssen die Auswirkungen der Eingabedaten auf die Leistung des Programms berücksichtigt werden.

Die Problemgröße ist hierbei meist unkritisch, da eine vergrößerte Menge an Eingabedaten zwar die Laufzeit des Programms insgesamt verlängert, die Auswirkungen der einzelnen Tuning-Parameter auf ebendiese in der Regel jedoch unverändert bleiben. Zur

Veranschaulichung soll an dieser Stelle nochmals auf das Beispiel des Insertionsort-Algorithmus in vorherigem Abschnitt verwiesen werden. Eine größere Eingabesequenz führt insgesamt zu einer längeren Laufzeit, ändert aber nichts an der Beanspruchung des Algorithmus selbst. Ein Tuning-Parameter hätte mit hoher Wahrscheinlichkeit bei unterschiedlich großen Eingabesequenzen mit gleicher Komplexität denselben Einfluss auf die Gesamtleistung des Algorithmus.

Deutlich schwerwiegender kann sich die Änderung der Problemkomplexität auswirken, da ggf. Algorithmen des Programms stärker oder schwächer beansprucht werden als zuvor. Somit kann sich der Einfluss der an den entsprechenden Algorithmen platzierten Tuning-Parameter ändern. Ein Parameter, dessen Sensitivität zuvor gering war, kann unter den neuen Umständen äußerst relevant für die Gesamtleistung des Programms werden. Damit ändert sich eventuell auch sein optimaler Parameterwert.

3.8.2.1 Klassifizierung der Eingabedaten

Der Diversität der Eingabedaten kann durch eine Klassifizierung entgegengewirkt werden, wobei die Zuordnung von Eingabedaten zu einer Klasse an Hand von Problemgröße und Problemkomplexität durchgeführt wird.

Zunächst gilt es daher, die beiden Charakteristiken zu quantifizieren. Die Klassifizierung der Eingabedaten hängt offensichtlich stark von der Art des Programms und der Form der Eingabedaten selbst ab, weshalb die Zuordnung für jedes Programm separat erstellt werden muss.

Die Problemgröße lässt sich leicht in Partitionen einteilen. Bei einem Sortieralgorithmus kann beispielsweise ein Raster einer bestimmten Granularität (z.B. 10.000) erstellt werden, in das die jeweils konkrete Anzahl zu sortierender Elemente entsprechend eingeordnet wird. Für ein Programm, das auf einem Graphen arbeitet, kann dieselbe Methode mit der Anzahl an Knoten im konkreten Graphen angewendet werden. Bei Programmen, die große Mengen an Binärdaten analysieren, kann die Größe der Daten in Megabyte oder Gigabyte herangezogen werden. Für ein Programm, das Matrizen verarbeitet, kann analog die Größe der Matrizen in ein entsprechendes Raster eingeordnet werden.

Bei der Problemkomplexität fällt die Quantifizierung schwerer, da sie sich meist nicht als skalarer Wert darstellen lässt. Hier müssen anwendungsspezifische Klassen definiert werden, um ungefähre Abgrenzungen zu spezifizieren. Im Beispiel des Sortieralgorithmus lässt sich der Grad der Vorsortierung der Eingabedaten als Klassifizierungsmaß verwenden. Bei einem Programm, welches auf Graphen operiert, hängt die Klassifizierung von den verwendeten Algorithmen ab. Wird der kürzeste Weg zwischen zwei Knoten gesucht, kann die Gesamtzahl der Kanten im Graph oder andere Grapheigenschaften (Zyklen etc.) herangezogen werden. Bei Programmen, die mehrere unterschiedliche Algorithmen zur Lösung eines Problems einsetzen, kann eine Klassifizierung anhand der Beanspruchung einzelner Algorithmen vorgenommen werden. Bei der Verarbeitung von Matrizen können je nach verwendeten Algorithmen Matrizeneigenschaften (z.B. dünnbesetzte Matrix, Dreiecksmatrix usw.) als Klassifizierungsmaß dienen.

Im Allgemeinen kann für ein paralleles Programm auch ein möglichst realistischer Benchmark vorgegeben werden, für den das Programm schließlich optimiert wird. In diesem Fall existiert nur eine Klasse von Eingabedaten, nämlich die des Benchmarks.

Ist für das Programm eine passende Klassifizierung der Eingabedaten gefunden, können für alle oder einige der Klassen separate Tuning-Läufe durchgeführt werden. Hierzu sind entsprechende Testdaten vorzuhalten. Die Ergebnisse der jeweiligen Optimierung werden zusammen mit der Klasse der Eingabedaten sowie wichtiger Informationen über die

Tuning-Parameter in einer Datenbank gespeichert. Somit wird sukzessive eine Wissensbasis aufgebaut, in der für jede Eingabedaten-Klasse eines Programms Informationen über alle entsprechenden Tuning-Läufe abgelegt sind.

Bei späteren Tuning-Läufen – beispielsweise auf anderen Hardware-Plattformen – können die gespeicherten Informationen wieder verwendet werden, indem zunächst anhand der quantifizierten Charakteristiken die aktuelle Klasse der Eingabedaten bestimmt wird und anschließend die entsprechenden Informationen aus der Datenbank an den Auto-Tuner übergeben werden. Dies können neben der durchschnittlich besten gespeicherten Parameterkonfiguration für die konkrete Klasse zum Beispiel die Werte aller Parametersensitivitäten sein, so dass der Auto-Tuner in jedem Fall alle für die aktuellen Eingabedaten relevanten Tuning-Parameter berücksichtigt. Die beste gespeicherte Parameterkonfiguration kann als Ausgangspunkt für die Suche dienen, da es wahrscheinlich ist, dass das neue Optimum in der Nähe des alten liegt.

Einige existierende Arbeiten (wie z.B. [ChHo04]) beschäftigen sich intensiv mit der Analyse von Eingabedaten. Meist dient das Analysieren der Daten anhand gespeicherter Charakteristiken zur Beschleunigung des Optimierungsverfahrens, da – wie oben beschrieben – auf Grund der gespeicherten Informationen bereits vor der Suche Annahmen über hinreichend gute Parameterkonfigurationen getroffen werden können.

3.9 Zusammenfassung

In diesem Kapitel wurden die Grundlagen der automatischen Performanzoptimierung paralleler Programme dargelegt. Angefangen bei den aufeinander aufbauenden Definitionen der wichtigsten Grundbegriffe, über die Klassifizierung der Tuning-Verfahren, bis hin zu den Integrationsmöglichkeiten eines Auto-Tuners und den Überlegungen zur Berücksichtigung von Eingabedaten spannt das Kapitel einen umfassenden Bogen um das Fachgebiet des Auto-Tuning. Einige der vorgestellten Grundlagen basieren auch auf den Erkenntnissen, die während der Entstehung dieser Arbeit gewonnen wurden.

Im Folgenden werden die verwandten Arbeiten diskutiert, bevor die zentralen Konzepte dieser Arbeit in Kapitel 5 vorgestellt werden.

4. Diskussion verwandter Arbeiten

In diesem Kapitel sind verwandte Arbeiten Gegenstand der Diskussion. Unterteilt nach konzeptioneller Ausrichtung der Arbeiten, werden die Ansätze vorgestellt, erörtert und verglichen.

Das Thema dieser Arbeit berührt mehrere Forschungsfelder, weshalb Arbeiten aus verschiedenen Bereichen als relevant einzustufen sind. Ein Anspruch auf Vollständigkeit kann unter den gegebenen Bedingungen nur schwer erhoben werden, jedoch wurden aus allen relevanten Bereichen jeweils die wichtigsten Ansätze und Konzepte ausgewählt.

4.1 Ansätze zu Tuning-Sprachen

Die Zielsetzung dieser Arbeit (vgl. hierzu Kapitel 2) umfasst unter anderem die Entwicklung eines Sprachkonzeptes, um in Programmen Tuning-Instruktionen in allgemeiner Form zu spezifizieren. Wir stellen daher die wichtigsten Vertreter verwandter Arbeiten aus diesem Bereich vor.

Es sei jedoch angemerkt, dass eine klare Trennung zwischen Sprach-Ansätzen und Optimierungsverfahren nur schwer zu definieren ist, da manche Arbeiten im Bereich der Tuning-Sprachen auch ein Optimierungskonzept beinhalten. In diesem Abschnitt wurden daher diejenigen Arbeiten zusammengestellt, deren Schwerpunkt auf dem Konzept einer Tuning-Sprache liegt.

4.1.1 XLanguage

XLanguage [DBRY+06] ist eine Annotations-Sprache zur Formulierung von Quelltexttransformationen. Mittels Direktiven kann der Quelltext des Programms (ausschließlich C/C++) annotiert werden, um Transformationen wie beispielsweise das Ausrollen von Schleifen zu spezifizieren. XLanguage ermöglicht auf diese Weise eine kompakte Repräsentation mehrerer semantisch äquivalenter Programmversionen, aus denen schließlich mit einem suchbasierten Verfahren die performanteste Version ausgewählt werden kann. Basierend auf den deklarativen Anweisungen in den Annotationen werden die Programmversionen durch eine Quelltext-zu-Quelltext-Übersetzung erzeugt.

Das Konzept ermöglicht auch die Definition neuer Transformationen. Hierfür können in einer vorgegebenen Syntax Quelltextmuster angegeben werden, die durch andere ersetzt werden sollen. Dies verleiht der XLanguage Flexibilität, allerdings auf Kosten der Einfachheit, da die Syntax nicht intuitiv erscheint.

Der Schwerpunkt von XLanguage liegt auf feingranularer Schleifenoptimierung. Die Autoren selbst merken an, dass der transformierte Quelltext durchaus zu einer Komplexität neigt, die in Bezug auf die Fehlersuche schwer zu handhaben ist.

4.1.2 POET

POET (*Parameterized Optimizing for Empirical Tuning*, [YSYV+07]) stellt eine domänenspezifische Sprache zur Formulierung von Quelltexttransformation dar. Teile des ursprünglichen Quelltextes werden als Fragmente zusammen mit Transformationsanweisungen in Form eines POET-Skripts formuliert. Eher ungewöhnlich ist die Tatsache, dass ein POET-Skript die Fragmente der Wirtssprache enthält, anstatt der Quelltext des Programms die POET-Anweisungen. Der daraus resultierende Vorteil ist die weitgehende Unabhängigkeit von POET bzgl. der Wirtssprache.

POET unterstützt drei Schleifenoptimierungen: Ausrollen (engl. *loop unrolling*), Vertauschen von Schleifen bei geschachtelten Schleifen (engl. *loop interchange*) sowie Schleifenpartitionierung (engl. *loop blocking*)

Aus angegebener Literaturstelle ist zu entnehmen, dass ein POET-Skript als Ausgabe eines Übersetzers zu verstehen ist, der den Quelltext des Programms auf Optimierungsmöglichkeiten hin analysiert und die Ergebnisse in Form eines POET-Skripts aufzeichnet. Ein derartiger Übersetzer existiert allerdings noch nicht, weshalb eine Optimierung bisher nicht stattfindet und die POET-Skripte unhandlich erscheinen.

Auf Grund der Komplexität der Sprache ist POET nur für kleine Programme geeignet, die ein konkretes numerische Problem lösen. POET bietet keine expliziten Sprachkonstrukte, die auf Parallelität abgestimmt sind.

4.1.3 Orio

Die aktuellste der verwandten Arbeiten stellt das Tuning-System *Orio* [HaPo09] vor. Orio bietet eine Kombination aus einer Annotationssprache zur Generierung mehrerer semantisch äquivalenter Programmvarianten sowie einer Suchkomponente zur Ermittlung der performantesten Programmvariante. Orio ist daher konzeptionell mit dieser Arbeit verwandt (zumindest hinsichtlich der Grundideen von Tuning-Sprache und suchbasierter Optimierung), wenngleich Zielsetzung und Umsetzung deutlich divergieren.

Der Optimierungsprozess von Orio beinhaltet das Parsen der Annotationen im Quelltext (ausschließlich C-Quelltext), die Erzeugung von Quelltext zur Generierung der Programmvarianten sowie die Suche nach der besten Programmvariante.

Die Annotationssprache ist komplex. Für die Instrumentierung einer einzigen Schleife ist ein Vielfaches an Annotationen sowie ein Skript nötig.

Listing 4.1 zeigt ein mit Orio annotiertes Beispielprogramm für eine Matrix-Multiplikation sowie das separate Skript, das die Randbedingungen und die Spezifikation der Tuning-Parameter beschreibt. Die Quelltextannotationen selbst enthalten die entsprechenden Transformations-Instruktionen. Diese Trennung erlaubt die Wiederverwendung von Parameter-Spezifikationen und vermeidet die Überfrachtung des Quelltextes.

Für die Annotationen muss jedoch der gesamte zu transformierende Quelltext dupliziert werden, um die Transformations-Instruktionen zu beschreiben. Dies erzeugt erhebliche Redundanz, so dass der Ansatz für größere Anwendungen nicht geeignet ist.

```
/*@ begin Loop (
transform UnrollJam(ufactor=Ui)
for (i=0; i<=M-1; i++)
```

```
    transform UnrollJam(ufactor=Uj)
    for (j=0; j<=N-1; j++)
        transform UnrollJam(ufactor=Uk)
        for (k=0; k<=O-1; k++)
            A[i][j] += B[i][k]*C[k][j];
) @*/
for (i=0; i<=M-1; i++)
    for (j=0; j<=N-1; j++)
        for (k=0; k<=O-1; k++)
            A[i][j] += B[i][k]*C[k][j];
/*@ end @*/

/* Separate constraint specification */
def performance_params {
    param Ui[] = range(1,33);
    param Uj[] = range(1,33);
    param Uk[] = range(1,33);
    constraint reg_capacity = Ui*Uj+Ui*Uk+Uk*Uj<=32;
}
def input_params {
    param M[] = [10,50,100,500,1000];
    param N[] = [10,50,100,500,1000];
    param O[] = [10,50,100,500,1000];
    constraint square_matrices = (M==N) and (N==O);
}
```

Listing 4.1: Mit Orio annotierter Quelltext einer Matrix-Multiplikation (oben) sowie der dazugehörigen Definition der Tuning-Parameter (aus [HaPo09])

Wie POET unterstützt auch Orio automatische Quelltexttransformationen auf Schleifenebene. Hierfür nutzt Orio das externe Werkzeug *PLuTo* (siehe [HaPo09]), das in den Optimierungsprozess integriert wurde. Durch den modularen Aufbau können auch andere Transformationswerkzeuge verwendet werden.

Die Suchkomponente von Orio nutzt als Suchalgorithmen entweder das Verfahren der simulierten Abkühlung oder das Simplex-Verfahren nach Nelder und Mead (vgl. Kapitel 3, Abschnitt 3.3.2). Da in der angegebenen Literatur der Ansatz an Hand sehr kleiner Beispiele mit wenigen Tuning-Parametern evaluiert wurde, ist die Leistung der Suchalgorithmen nicht direkt auf große parallele Anwendungen zu übertragen.

4.1.4 Vergleich und Bewertung

Die meisten Ansätze im Bereich der Tuning-Sprachen fokussieren auf wissenschaftliche bzw. numerische Anwendungen. Dies bringt Vor- und Nachteile mit sich. Durch die Beschränkung auf bestimmte Anwendungsdomänen können offensichtlich gezieltere Tuning-Instruktionen definiert werden, was sich insbesondere auf die Quelltexttransformation auswirkt. Ein Quelltextgenerator kann auf Grund der detaillierten Informationen auf feingranularer Ebene auch komplexere und umfassendere Änderungen am Quelltext durchführen.

Ein Verfahren, welches hauptsächlich auf Schleifenoptimierungen spezialisiert ist (wie z.B. XLanguage), kann einen Quelltextgenerator zur Verfügung stellen, der die entsprechenden Transformationen (wie beispielsweise das Ausrollen von Schleifen oder das Einstellen der Blockgröße) vollständig und automatisiert durchführt. Bei allgemeineren Ansätzen wäre ein solcher Grad der Detaillierung kaum möglich, da es weit mehr Optimierungsmöglichkeiten zu berücksichtigen gäbe.

Diese Überlegung führt jedoch direkt zu den Nachteilen. Die Spezialisierung auf einzelne Optimierungsbereiche schmälert das Anwendungsspektrum. Daher fehlen Konzepte zur tuning-relevanten Strukturierung von großen parallelen Programmen sowie zur Suchraumreduktion.

In Tabelle 4.1 vergleichen wir die vorgestellten Ansätze an Hand der folgenden Charakteristiken:

- **Unterstützung für Parallele Programme.** Diese Unterstützung ist dann gegeben, wenn eine Sprache explizite Konstrukte zur Instrumentierung *paralleler* Sektionen zur Verfügung stellt.
- **Portabilität.** Eine Tuning-Sprache gilt als portabel, wenn sie zusammen mit unterschiedlichen Wirtssprachen sowie auf unterschiedlichen Hardware-Plattformen verwendet werden kann.
- **Domänenspezifisches Konzept.** Das Sprachkonzept wird als domänenspezifisch eingestuft, wenn es explizit auf bestimmte Algorithmen, Programmtypen oder Optimierungsbereiche spezialisiert ist.
- **Explizite Messpunkte.** Bietet eine Tuning-Sprache die Möglichkeit, Messpunkte innerhalb des Programms flexibel zu definieren, so wird das Setzen expliziter Messpunkte unterstützt.
- **Skript-Gebunden.** Ein Ansatz ist skript-gebunden, wenn zur Definition von Tuning-Instruktionen neben der Quelltextinstrumentierung zusätzliche Skripte angefertigt werden müssen.

	XLanguage	**POET**	**Orio**
Unterstützung für parallele Programme	-	-	-
Portabilität	-	✓	-
Domänenspezifisches Konzept	✓	✓	-
Explizite Messpunkte	-	-	-
Externe Skripte	-	✓	✓

Tabelle 4.1: Vergleich der verwandten Ansätze im Bereich der Tuning-Sprachen.

Es fällt auf, dass keiner der genannten Ansätze das Setzen expliziter Messpunkte unterstützt. XLanguage und Orio messen implizit die Laufzeit des Programms, POET sieht weder Messpunkte noch Messungen vor. Um aber auch komplexere parallele Programme optimieren zu können, sind jedoch unter Umständen mehrere Messpunkte nötig. Insbesondere sollten Messpunkte auch innerhalb von Programmteilen gesetzt werden können, um Messungen einzelner Komponenten vornehmen zu können.

Des Weiteren fehlt die explizite Unterstützung zur Instrumentierung und Optimierung *paralleler* Programme.

4.2 Ansätze zu Entwurf und Implementierung konfigurierbarer Applikationen

Ein weiteres Teilkonzept der Dissertation befasst sich mit dem Entwurf paralleler optimierbarer Softwarearchitekturen. Das damit verbundene Forschungsgebiet ist gewiss

weit gefasst, weshalb selbst eine Auswahl relevanter Ansätze weit über den Rahmen dieses Kapitels hinausgehen würde. Im Kontext dieser Arbeit können wir jedoch den Architekturentwurf sowie die adaptierbare Implementierung des Entwurfs als konzeptionelle Schwerpunkte betrachten.

Dieser Überlegung folgend, lassen sich vier Bereiche als besonders relevant einstufen: parallele algorithmische Skelette, parallele musterbasierte Rahmenarchitekturen, modellbasierte Softwareadaption sowie dynamische Rekonfiguration.

Zu jedem dieser Bereiche wird im Folgenden eine zentrale Arbeit vorgestellt und an Hand der Zielsetzung dieser Arbeit bewertet. Ein Vergleich der Arbeiten ist kaum möglich, da Anwendungsbereiche und Entwurfsziele naturgemäß stark divergieren.

4.2.1 Parallele algorithmische Skelette

Algorithmische Skelette (engl. *algorithmic skeletons*) wurden erstmals von Cole [Cole89] im Kontext paralleler Programmierung vorgestellt. Cole beschreibt algorithmische Skelette im Wesentlichen als Funktionen höherer Ordnung, die jedoch nicht in funktionalen, sondern in imperativen Programmiersprachen angewendet werden können. Hier repräsentiert eine Funktion höherer Ordnung eine Schablone für ein Programm oder eine Prozedur, die die allgemeine Struktur der Berechnung spezifiziert und entsprechende Lücken für problemspezifische Deklarationen und Prozeduren bereitstellt. Schon bei Cole lag der Fokus auf parallelen Algorithmen.

Die Definition algorithmischer Skelette ähnelt daher der der parallelen Entwurfsmuster, wobei Skelette sich wesentlich näher an der Implementierung orientieren.

Basierend auf dem Konzept von Cole entstanden einige Arbeiten, die algorithmische Skelette als Bausteine für parallele Programme nutzen. Des Weiteren wurde versucht, Skelette durch Modellierung für unterschiedliche Hardware-Plattformen zu optimieren.

Ein derartiger Ansatz wird in [DFHK+93] verfolgt. Zunächst werden konkrete parallele Skelette funktional definiert. Hierzu zählen ein Fließband-Skelett (`PIPE`), ein Daten-Dekompositions-Skelett (`FARM`), ein Teile-und-Herrsche-Skelett (`DC`) sowie ein Map/Reduce-Skelett (`RaMP`). `RaMP` steht hierbei für *Reduce-and-Map-over-Pairs* und repräsentiert die erste Definition des heute noch gebräuchlichen Map/Reduce-Algorithmus.

Zu jedem Skelett wurden mehrere Modelle entwickelt, die dessen Leistung, bezogen auf eine konkrete Hardware-Plattform, beschreiben. Es existiert also ein Modell für jedes Skelett/Plattform-Paar.

Neben den Performanzmodellen verfügt das Verfahren über Transformationsregeln, die ein bestimmtes Skelett in ein oder mehrere andere überführen, wobei die ursprüngliche Funktionalität erhalten bleibt. Es stehen somit mehrere Möglichkeiten zu Verfügung, ein Programm oder einen Algorithmus auszudrücken. Beispielsweise kann ein auf dem RaMP-Skelett basierendes Programm auch durch ein Fließband bestimmter Länge ausgedrückt werden.

Diese Transformationsregeln macht sich das Verfahren zu Nutze, um ein Programm zu optimieren. Für ein gegebenes Problem werden alle zutreffenden Skelette und Kombinationen von Skeletten gemäß der Transformationsregeln automatisiert getestet. Die beste Implementierung wird schließlich beibehalten.

Eine weitere Arbeit untersucht die Performanz von verteilten Programmen, die auf einem Fließband-Skelett basieren [BCGH04]. Hierzu wurde zunächst ein umfangreiches Modell des Fließband-Skeletts entwickelt, das neben den Stufen auch die Prozessoren des Systems sowie Parameter des zu Grunde liegenden Netzwerks mit einbezieht. Die

Arbeit wurde an Hand eines numerischen Problems evaluiert, was jedoch keine Aussage über größere Programme oder andere Parallelisierungsstrategien zulässt.

Zusammenfassend kann festgehalten werden, dass algorithmische Skelette auf sehr feingranularer Ebene eingesetzt werden können und primär zur Lösung numerischer paralleler Probleme geeignet sind. Jedoch ist das Grundkonzept der Skelette sowohl bei parallelen Entwurfsmustern als auch bei Rahmenarchitekturen zur Entwicklung paralleler Programme wiederzufinden. Des Weiteren wurden erste Ansätze zur automatisierten Optimierung vordefinierter Programmbausteine entwickelt, allerdings mit Schwerpunkt auf Skelett-Transformation. Konzepte zum Entwurf paralleler Architekturen werden allerdings weitgehend außer Acht gelassen.

4.2.2 CO^2P^3S

CO^2P^3S [TSSA+03] (*Correct Object-Oriented Pattern-based Parallel Programming System*) ist ein Werkzeug, welches mit so genannten generativen Entwurfsmustern arbeitet. Zur Klasse der generativen Entwurfsmuster werden alle Entwurfsmuster gezählt, die eine konkrete Implementierungsanweisung beinhalten.

Das Ziel von CO^2P^3S ist die Unterstützung von Anwendungsentwicklern bei der Implementierung eines neuen parallelen Programms. Hierzu wird eine muster-basiertes Rahmenarchitektur generiert, das die entsprechende parallele Sektion des Programms implementiert.

Für den Entwurf und die Generierung der Rahmenarchitektur stellt CO^2P^3S einen intuitiven Prozess zur Verfügung: Der Benutzer wählt über eine grafische Oberfläche ein für die geplante Struktur des Programms passendes Entwurfsmuster aus. Danach generiert das System den entsprechenden Quelltext, der die Kommunikation und Synchronisation des parallelen Entwurfsmusters sicherstellt.

Der generierter Quelltext stellt Platzhaltermethoden (so genannte *hook methods*) zu Verfügung, mit deren Hilfe sequentieller programmspezifischer Quelltext eingebunden werden kann, der die eigentliche Funktionalität des Programms implementiert. Diese Platzhaltermethoden müssen vom Benutzer implementiert werden.

CO^2P^3S verwendet zur Erstellung der Rahmenarchitektur so genannte parametrisierte Musterschablonen, mit denen der Generierungsprozess gesteuert werden kann. Eine Musterschablone repräsentiert die grundlegende Struktur eines parallelen Entwurfsmusters und stellt Parameter zur Verfügung, mit denen die Struktur angepasst und eine Variante des Musters erzeugt werden kann.

Obwohl die Auswahl einer Mustervariante die Leistung des Programms durchaus beeinflussen kann, stellen die Parameter der Musterschablonen keine Tuning-Parameter im eigentlichen Sinne dar, da sie keine leistungskritischen Engpässe adressieren. Hinzu kommt, dass die Parameter nach der Quelltextgenerierung nicht direkt angepasst werden können (weder manuell noch automatisiert).

CO^2P^3S unterstützt keine automatische Performanzoptimierung und bietet keine Möglichkeit, architektonische Optimierungsentscheidungen während des Entwurfs einfließen zu lassen. Dennoch stellt das Verfahren einen richtungweisenden Ansatz für musterbasierte Entwicklung paralleler Programme dar.

Auf CO^2P^3S basieren eine ganze Reihe von Arbeiten, die sich insbesondere damit beschäftigen, neue parallele Entwurfsmuster in das Werkzeug einzubinden (beispielsweise [MSSB00, MSSA+02]).

4.2.3 Rainbow Framework

Ein wichtiger Aspekt für optimierbare Programme ist die Adaptierbarkeit derselben. Hierzu wurden bereits im Grundlagen-Kapitel in Abschnitt 3.7.2 einige Überlegungen angestellt.

Eine relevante Arbeit auf diesem Gebiet ist das *Rainbow Framework* [GCHS+04, GaSC09], ein Verfahren zur architekturbasierten Selbstadaption von Systemen. Ein System kann im Kontext von Rainbow eine komplexe Struktur mehrerer (auch verteilter) Komponenten darstellen, beispielsweise eine Client-Server-Applikation.

Rainbow basiert auf einem externen Architekturmodell sowie einer wieder verwendbaren Infrastruktur, womit das Verhalten eines zu Grunde liegenden Systems zur Laufzeit angepasst werden kann. Das Modell entspricht der Standarddefinition einer Architektur, wie sie im Rahmen dieser Arbeit bereits dargelegt wurde. Rainbow nutzt die Softwarearchitektur des zu adaptierenden Systems jedoch nicht auf herkömmliche Weise zur Unterstützung des Entwurfs, sondern verwendet ein Modell der Architektur zur Laufzeit. Mit Hilfe der Informationen aus dem Modell können das System überwacht und adaptionsrelevante Entscheidungen getroffen werden.

Ein zentrales Prinzip von Rainbow ist die Wiederverwendbarkeit seiner Komponenten. Um dies zu erreichen, besteht das Rainbow Framework aus einer mehrschichtigen Infrastruktur, bestehend aus den folgenden Ebenen:

- **System-Ebene:** Enthält die Zugriffsschnittstelle des System, Überwachungsmechanismen (*probes*) sowie die Komponente zur tatsächlichen Ausführung der Systemmodifikationen (*effector*).
- **Architektur-Ebene:** Steuert die Verarbeitung der Überwachungsdaten (*gauges*), aktualisiert entsprechende Eigenschaften im Architekturmodell und enthält eine Gültigkeitsüberprüfung (*constraint evaluator*) sowie die Komponente zur Durchführung der Modelladaption (*adaptation engine*).
- **Translations-Infrastruktur:** Steuert die Abbildung der Informationen vom Modell zum System und umgekehrt. Alle Abbildungen werden im so genannten *translation repository* abgelegt.

Diese Infrastruktur ermöglicht einerseits eine wohldefinierte Beziehung zwischen dem Architekturmodell des Systems und seiner Implementierung, andererseits die Wiederverwendung der Komponenten durch die konzeptionelle Trennung von Modell und Implementierung. Bevor eine bereits durchgeführte Modifikation am Modell auf das System übertragen wird, führt Rainbow eine Gültigkeitsüberprüfung durch.

Um das System passend adaptieren zu können, sammelt Rainbow sowohl die statischen als auch die dynamischen Attribute des Systems, woraus die Adaptionsspezifikation entsteht. Mittels Adaptionsoperatoren (engl. *adaptation operators*) sowie Adaptionsstrategien (engl. *adaptation strategies*) wird definiert, welche systemspezifischen Aktionen auf welche Weise zur Adaption verwendet werden können.

Rainbow zählt zu den klassischen Ansätzen auf der Basis dynamischer Architekturmodelle. Das Konzept ermöglicht eine flexible Spezifikation von Adaptionsmöglichkeiten eines Systems und stellt eine wieder verwendbare Infrastruktur zur Verfügung. Die systemspezifischen Aktionen, um Verhalten und Topologie des Systems zu beeinflussen, müssen jedoch manuell implementiert werden.

Rainbow bietet keine explizite Unterstützung für parallele Programme. Des Weiteren können die Adaptionen des Systems zwar durch Änderungen bestimmter Parameter beeinflusst werden, die jedoch nicht mit reinen performanzorientierten Tuning-Parametern zu vergleichen sind; insbesondere werden Aspekte der automatischen Performanzoptimierung nahezu vollständig ignoriert.

4.2.4 Dynamische Rekonfiguration

Arbeiten im Bereich der dynamischen Rekonfiguration von Programmen zählen sicherlich nicht zu den engsten verwandten Ansätzen dieser Arbeit, da durch die Fokussierung auf Offline-Tuning zur Laufzeit keine Änderungen durchgeführt werden müssen. Dennoch ist Konfigurierbarkeit im Allgemeinen für diese Arbeit von Interesse, da die Einstellung von Tuning-Parametern eine Rekonfiguration des Programms nach sich zieht.

Dynamische Rekonfiguration von Programmen wurde bereits in vielen Forschungsarbeiten untersucht und weist einige Parallelen zu selbstadaptierender Software auf. Einige Ansätze verfolgen die Entwicklung theoretischer Konzepte und beschreiben mögliche Verfahren zur Umsetzung.

Wir stellen jedoch eine Arbeit von Rasche et al. vor [RaPo03], die neben den konzeptionellen Überlegungen auch eine konkrete Implementierung der Methoden auf Basis des Microsoft .NET-Frameworks [Micr09a] vorschlägt. Der Schwerpunkt der Arbeit liegt auf der dynamischen Rekonfiguration verteilter komponentenbasierter Anwendungen.

Zur Beschreibung von Konfigurationsmöglichkeiten einer Anwendung wird ein so genanntes Konfigurationsprofil angelegt, welches später vom Rekonfigurationswerkzeug ausgelesen wird.

Für die Rekonfiguration des Programms hinsichtlich Verhalten und Topologie stehen drei elementare Operationen zur Verfügung: das Hinzufügen einer Komponente, das Löschen einer Komponente sowie die Modifikation von Attribute einer Komponente. In angegebener Literaturstelle werden keine konkreten Beispiele für Komponenten-Attribute erwähnt. Es können jedoch alle Attribute der Form `Bezeichner:Wert` verarbeitet werden. Grundsätzlich bezeichnet ein Attribut einen Parameter, der das Verhalten der Komponente beeinflusst. Diese Parameter müssen nicht unbedingt leistungsrelevant sein.

Komplexere Rekonfigurationen werden durch Kombination und Aneinanderreihung der Operationen erreicht. Ein zentraler Bestandteil des Werkzeugs ist der Rekonfigurationsalgorithmus (aus [Werm97]), der im Wesentlichen die Blockierung der Transaktionen zwischen einzelnen Komponenten während ihrer Rekonfiguration steuert. Hierbei gilt es, für jede betroffene Komponente eine möglichst kurze Ausfallzeit zu gewährleisten, Datenverlust zu vermeiden und Abhängigkeiten korrekt auszulösen.

Gemäß dem Konfigurationsprofil und der aktuellen Rekonfigurationsanweisungen lädt das Werkzeug dynamisch einen neue Konfiguration und wendet diese automatisch an.

Die Rekonfigurationslogik wurde nach dem Prinzip der aspektorientierten Programmierung implementiert (vgl. Kapitel 3, Abschnitt 3.7.2); für die Kommunikation zwischen den Komponenten sowie für die Steuerung der Transaktionen wurde das Kommunikations-Framework von .NET (*.NET Remoting*) verwendet.

Ähnlich wie Rainbow bietet auch die Arbeit von Rasche et al. einen flexiblen Ansatz zur Modifikation von Programmen zur Laufzeit, wobei die konkreten Implementierungstechniken von Vorteil sind. Allerdings werden parallele Programme und Auto-Tuning ebenfalls nicht berücksichtigt.

4.2.5 Adaptive MPI

Im Kontext konfigurierbarer Architekturen sei zuletzt *Adaptive MPI (AMPI)* [HuLK04] genannt, das eine Implementierung und Erweiterung des *Message Passing Interface (MPI)* [MPI 09] zur Unterstützung von Prozessorvirtualisierung darstellt. Der durch MPI gegebene Fokus auf verteilte Systeme klassifiziert AMPI nur als bedingt verwandt mit dieser Arbeit, jedoch beinhaltet AMPI einen Ansatz zur automatischen Anpassung von (verteilten) Architekturen, der hier erwähnt werden soll.

Die Grundidee von AMPI ist die strikte Trennung zwischen der Zuweisung von Aufgaben zu Prozessoren und der Identifikation von Aufgaben, die überhaupt parallel verarbeitet werden können. Herkömmliche MPI-Programme teilen eine Berechnung in p Prozesse – für jeden der verfügbaren p Prozessoren einen. Ein AMPI-Programmierer teilt die Berechnung in v virtuelle Prozesse, deren Anzahl unabhängig von der Zahl der physikalischen p Prozessoren ist. Letztere sind für den Programmierer nicht sichtbar. Die Zuweisung der v virtuellen Prozesse auf die p physikalischen Prozessoren wird von AMPI automatisch durchgeführt.

Hierfür implementiert AMPI virtuelle MPI-Prozesse, von denen mehrere auf einen physikalischen Prozessor abgebildet werden können. Eine spezielles Laufzeitsystem sorgt für eine möglichst performante Zuweisung von virtuellen MPI-Prozessen auf Prozessoren, so dass die verfügbaren Ressourcen optimal ausgenutzt werden. Neben automatischem Lastausgleich ermöglicht das Laufzeitsystem das automatische Setzen von Kontrollpunkten. Des Weiteren ist das Laufzeitsystem in der Lage, die Anzahl an Prozessoren, die ein parallel zu verarbeitender Job verwendet, dynamisch zu verkleinern oder zu vergrößern.

Es wurden einige wissenschaftliche Anwendungen mit AMPI entwickelt, wodurch bereits vielversprechende Ergebnisse hinsichtlich Performanz erzielt werden konnten.

4.3 Ansätze zur Beschreibung von Architekturen

Neben den genannten Ansätzen, die Konzepte für parallele und konfigurierbare Programme bereitstellen, sind im Kontext dieser Arbeit insbesondere Arbeiten von Interesse, die sich mit Beschreibungen paralleler Architekturen beschäftigen. Hierbei steht der Entwurf und die Entwicklung eines (parallelen) Programms auf der Basis allgemeiner architektonischer Idiome im Vordergrund. Der Fokus liegt daher auf grobgranularen architektonischen Elementen und nicht auf einzelnen Quelltextzeilen.

Auf Grund der höheren Abstraktionsebene eignen sich derartige Konzepte für die grobgranulare Parallelisierung von Programmen.

Im Folgenden werden die wichtigsten Vertreter in diesem Bereich vorgestellt und die Ansätze verglichen. Es sei angemerkt, dass nicht alle Ansätze zu den reinen Architekturbeschreibungssprachen gerechnet werden können. Jedoch dienen die vorgestellten Konzepte im weiteren Sinne zur Planung und zum Entwurf von (parallelen) Anwendungen. Für eine weiterführende Klassifikation und einen Vergleich von Architekturbeschreibungssprachen sei auf die angegebene Literatur verwiesen.

4.3.1 Parallel Pattern Language

Im Grundlagen-Kapitel wurde bereits die Eignung paralleler Entwurfsmuster für den Entwurf paralleler Applikationen diskutiert. Betrachtet man die Entwurfsmuster als Bausteine der Architektur, so können mit deren Hilfe schnell und effizient parallele Programme entworfen werden, da das Verhalten sowie die Intention der einzelnen Muster bekannt ist.

Von diesen Beobachtungen motiviert, stellen Mattson et al. eine Entwurfsmustersprache (engl. *Parallel Pattern Language*) vor [MaSM04]. Das Konzept repräsentiert keine Sprache im eigentlichen Sinne, sondern vielmehr einen Leitfaden, der den Programmierer durch den gesamten Entwurfs- und Entwicklungsprozess eines parallelen Programms führt.

Die Sprache ist in so genannte Entwurfsbereiche (engl. *design spaces*) eingeteilt, wobei jeder Entwurfsbereich einer Phase des Entwurfs- und Entwicklungsprozesses zugeordnet ist:

- Der *FindingConcurrency*-Entwurfsbereich beinhaltet Muster auf höchster Abstraktionsebene, die bei der Identifikation von Parallelität in Problemen helfen sollen. Es handelt sich daher um deskriptive Muster, die weniger eine konkrete Implementierung vorschlagen, sondern unterstützende Anleitungen und Checklisten zur Verfügung stellen. Die Muster bieten Hilfestellungen, um herauszufinden, (i) wie das Problem in einzelne Aufgaben zerlegt werden kann, (ii) welche Daten exklusiv und welche gemeinsam von den Aufgaben genutzt werden und (iii) welche (Daten-)Abhängigkeiten zwischen den Aufgaben bestehen.

- Der *AlgorithmStructure*-Entwurfsbereich beinhaltet Muster, die eine konkrete Parallelisierungsstrategie vorschlagen. Wurde mit Hilfe der Muster aus dem *FindingConcurrency*-Entwurfsbereich Parallelisierungspotential identifiziert, kann nun eine passende Strategie ausgewählt werden, mit der die Parallelität am effizientesten ausgenutzt werden kann. Der *AlgorithmStructure*-Entwurfsbereich umfasst also diejenigen Muster, die gemeinhin als parallele Entwurfsmuster bezeichnet werden. Hierzu zählen unter anderem das *EmbarrassinglyParallel*-Muster, das *DevideAndConquer*-Muster oder das *Pipeline*-Muster.

- Der *SupportingStructure*-Entwurfsbereich stellt Muster zur Verfügung, die eine Zwischenstufe zwischen den problemorientierten Mustern des *AlgorithmStructure*-Entwurfsbereichs und ihrer tatsächlichen Implementierung darstellen. Diese Muster repräsentieren konkrete Implementierungsanweisungen. Vertreter dieses Entwurfsbereichs sind beispielsweise das *SharedQueue*-Muster oder das *Barrier*-Muster.

Die Sprache besteht demnach aus klassifizierten Mustern, die als Sprachkonstrukte fungieren. Die Muster bieten dem Entwickler Lösungsvorschläge und Vorgehensweisen an und unterstützen durch einen Top-Down-Ansatz den Entwurfs- und Entwicklungsprozess. Ein Konzept zur musterbasierten Spezifikation bzw. Beschreibung eines konkreten Programms (z.B. in Form einer domänenspezifischen Sprache) existiert allerdings nicht.

Dennoch werden mit der Entwurfsmustersprache erstmals alle Aspekte der Parallelisierung in geordneter Form definiert und als Katalog zur Verfügung gestellt. Der Ansatz hat zudem gezeigt, dass sich parallele Entwurfsmuster bestens zur strukturierten Entwicklung paralleler Programme eignen. Einzig die Performanzoptimierung wird in keiner Weise betrachtet.

Abschließend ist erwähnenswert, dass eine Reihe weiterführender Arbeiten existieren, die die Sprache mit neuen Mustern erweitern und ergänzen (z.B. [MaMS99, MaMS00]).

4.3.2 Architekturbeschreibungssprachen

Klassische Architekturbeschreibungssprachen dienen der formalen Definition einer Softwarearchitektur auf abstrakter Ebene. Mittlerweile existieren eine nahezu unüberschaubare Menge an Sprachen, die auf unterschiedliche Architekturbereiche spezialisiert sind.

Eine ausführliche Diskussion aller aktuell verfügbaren Sprachen und deren Konzepte würde den Rahmen dieses Kapitels deutlich übersteigen. Aus diesem Grund werden wir uns an einer Studie von Medvidovic und Taylor [MeTa00] orientieren, in der die wichtigsten Vertreter von Architekturbeschreibungssprachen verglichen wurden.

Um die Eignung und den Anwendungsbereich gängiger Verfahren einschätzen zu können, wurden die folgenden Sprachen näher betrachtet: *Aesop* [GaAO94], *C2* [MORT96], *Darwin* [MDEK95], *Rapide* [LKAV+95], *UniCon* [SDKR+95], *Weaves* [GoRa91] und *Wright* [AlGa97]. Tabelle 4.2 zeigt die genannten Sprachen sowie deren primären Anwendungsbereich.

Sprache	**Anwendungsbereich**
Aesop	Spezifikation von Architekturen in verschiedenen Architektur-Stilen
C2	Beschreibung von Architekturen verteilter, dynamischer Systeme
Darwin	Beschreibung von Architekturen verteilter Systeme, deren Dynamik mittels strikter formaler Definitionen gesteuert wird
Rapide	Modellierung und Simulation von dynamischem Verhalten, das von einer Architektur beschrieben wird
UniCon	Verbindung bestehender Komponenten mittels allgemeiner Interaktionsprotokolle
Weaves	Beschreibung von Datenfluss-Architekturen
Wright	Modellierung und Deadlock-Analyse von nebenläufigen Systemen

Tabelle 4.2: Vergleich relevanter Architekturbeschreibungssprachen.

Beim Studium der einzelnen Ansätze zeigt sich, dass jede Sprache einen speziellen Anwendungsbereich abdeckt. Architekturbeschreibungssprachen repräsentieren somit keine universellen Werkzeuge, sondern beleuchten stets einen bestimmten Aspekt einer Architektur.

In diesem Zusammenhang fällt auf, dass keine der vorgestellten Sprachen explizit die Beschreibung paralleler Applikationen unterstützt. Ansätze wie C2 und Darwin dienen zwar zur Modellierung großer verteilter Systeme, bieten aber keine Möglichkeiten, Parallelität in Applikationen auszudrücken.

Selbiges gilt für den Aspekt der Performanzoptimierung. Keine der Sprachen verfügt über tuning-relevante Konstrukte oder integriert Verfahren zur Anbindung eines Auto-Tuners.

Unter Berücksichtigung der Zielsetzung dieser Arbeit eignen sich die klassischen Architekturbeschreibungssprachen offensichtlich nicht, um Architekturen im Hinblick auf Parallelisierung und Optimierung zu modellieren. Insbesondere fehlen Sprachkonzepte, die beide Aspekte in geschickter Weise kombinieren.

4.4 Ansätze zur automatischen Performanzoptimierung

Die letzte Gruppe verwandter Arbeiten umfasst Ansätze zur automatischen Performanzoptimierung. Auto-Tuning-Verfahren sind bereits seit geraumer Zeit Gegenstand der Forschung und finden heute in vielen Bereichen Anwendung. Über die Jahre haben sich unterschiedliche Konzepte und Ausprägungen von Performanzoptimierern entwickelt, wobei ein ständig wachsender Teil der Arbeiten auf die Optimierung paralleler Programme fokussiert ist.

Grundsätzlich kann zwischen domänen- bzw. problemspezifischen Ansätzen und allgemeiner einsetzbaren Verfahren unterschieden werden. Wir folgen dieser Unterscheidung und stellen zunächst die wichtigsten domänenspezifischen Arbeiten vor, um im Anschluss die Ansätzen mit einem erweiterten Einsatzgebiet zu diskutieren.

4.4.1 Domänenspezifische Auto-Tuning-Ansätze

In der Numerik werden seit langem Optimierungsstrategien für Algorithmen und rechenintensive Operationen entwickelt und auch erfolgreich eingesetzt. Die meisten der Arbeiten beschäftigen sich mit problemspezifischen Ansätzen. Das Ziel ist die Optimierung meist nur eines speziellen Problems oder einer Problemklasse, dies dafür aber mit großer Präzision und Effizienz. Auch wenn sich derartige Ansätze für die Zielsetzung dieser Arbeit als zu eingeschränkt und zu wenig flexibel erweisen, so bilden sie doch die Grundlage der Performanzoptimierung. Im Folgenden werden die wichtigsten Vertreter der problemspezifischen Performanzoptimierung vorgestellt, die alle auf suchbasiertem Tuning basieren.

4.4.1.1 FFTW

FFTW [FrJo98] (*Fastest Fourier Transform in the West*) ist ein domänenspezifischer Quelltextgenerator zur Erzeugung performanter Implementierungen der Fouriertransformation. Hierbei wird auf spezifisches Wissen und Erfahrung über die Fouriertransformation zurückgegriffen. Das Werkzeug besteht aus mehreren Implementierungen bekannter Algorithmen zum Lösen einzelner Schritte der Fouriertransformation sowie einem Ablaufplaner, dem so genannten *planner*, der auf einer gegebenen Hardware-Plattform die Implementierungen so lange durchprobiert, bis ein optimales Ergebnis erzielt wird.

Ausgehend von einer abstrakten mathematisch-formalen Beschreibung des Problems in *OCamL* für die Diskrete Fouriertransformation zerteilt der *planner* das Problem, bis es „ausreichend klein“ ist, um von einem einzigen Stück Quelltext bearbeitet zu werden. Diese so genannten *Codelets* werden von einem Quelltextgenerator erzeugt, der mehrere Implementierungen für die gleiche Operation generiert. Ein *Codelet* besteht aus C-Quelltext und kann auch manuell geschrieben werden. FFTW testet nun so lange *Codelet*-Kombinationen, bis die optimale Kombination von *Codelets* bezüglich des formulierten Problems gefunden ist.

Mit FFTW wurden auf diese Weise auch bislang unbekannte Algorithmen zur Berechnung der schnellen Fouriertransformation gefunden sowie bekannte Algorithmen in ihrer Komplexität reduziert.

4.4.1.2 ATLAS/AEOS

ATLAS [WhPD01] (*Automatically Tuned Linear Algebra Software*) ist eine Bibliothek für mathematische Operationen der linearen Algebra. Das Projekt verwendet ein Verfahren namens *AEOS* (*Automated Empirical Optimization of Software*), um die für eine gegebene Hardware-Plattform schnellste Variante eines Algorithmus zu finden. Im Gegensatz zu FFTW ist ATLAS/AEOS der Kategorie der Bibliotheksgeneratoren zuzuordnen. AEOS erzeugt mittels domänenspezifischem Wissen über das zu lösende Problem und in Abhängigkeit der zu Grunde liegenden Hardware-Plattform eine hochperformante Variante der ATLAS-Bibliothek. AEOS führt die Quelltextoptimierung innerhalb der ATLAS-Bibliothek durch iteratives Messen und Ändern der Parameterwerte durch. Die Verwendung von AEOS ist nicht unbedingt auf ATLAS beschränkt, jedoch gelten für das zu optimierende Programm bzw. die darunter liegende Plattform einige Voraussetzungen, die von ATLAS natürlich erfüllt werden.

Alternativ wurde auch ein modellbasierter Ansatz entwickelt, der auf Grund des sehr eng gefassten Anwendungsbereichs Erfolg versprechend ist. Statt der Rückführung der Messergebnisse in den Optimierungsprozess und der schrittweisen Anpassung der Parameterwerte wurde ein manuell auf den Anwendungsbereich angepasstes Modell erstellt, das direkt die optimale Parameterkonfiguration für die Quelltextgenerierung errechnet.

Ein weiteres Projekt auf der Basis von ATLAS beschäftigt sich mit der Kombination aus analytischen Modellen und dem suchbasierten Ansatz zur Performanzoptimierung von numerischen Anwendungen [EGDP+06]. Das Verfahren schränkt den Suchraum mittels eines analytischen Modells zunächst ein, bevor gesucht wird. Die Modelle werden durch Verfahren aus dem Bereich des Maschinenlernens erstellt. Die Ergebnisse sind viel versprechend – vorausgesetzt, man beschränkt sich auf numerische Anwendungen basierend auf ATLAS. Der Ansatz ist einer der wenigen, der gezielt die Größe des Suchraums angeht und versucht, durch eine geführte Suche den Optimierungsprozess zu beschleunigen.

Die Fokusierung des Tuning-Ansatzes auf die ATLAS-Bibliothek und die darin enthaltenen Operationen für die weitere Betrachtung im Rahmen dieser Arbeit allerdings ungeeignet.

4.4.1.3 SPIRAL

SPIRAL [PMJP+05] (*Signal Processing Implementation Research for Adaptable Libraries*) ist ebenfalls ein Projekt zur Generierung von optimiertem Quelltext. Mit der Fokussierung auf das Gebiet der digitalen Signalverarbeitung (engl. *Digital Signal Processing, DSP*) hat SPIRAL gegenüber FFTW eine etwas erweiterte Anwendungsdomäne. SPIRAL sucht nicht zur Lauf-, sondern zur Übersetzungszeit nach der optimalen Version eines in der Hochsprache SPL (*Signal Processing Language*) formulierten mathematischen Problems. SPIRAL erzeugt dafür verschiedene Versionen von maschinenspezifischem C-Quelltext für dasselbe Problem und testet diese auf Performanz. Hierbei wird das Ziel verfolgt, automatisiert schnelle Bibliotheken für eine bestimmte Hardware-Plattform zu erzeugen. Zur Optimierung wird dabei stark auf anwendungsspezifisches Wissen über DSP-Algorithmen zurückgegriffen.

SPIRAL kann keine parallelen Programme optimieren, da Mehrkernplattformen nicht unterstützt werden. Allerdings erreicht die in SPIRAL angewandte Technik die Performanz manuell optimierter Bibliotheken und übersteigt diese sogar.

4.4.2 Allgemeinere Auto-Tuning-Ansätze

Es folgt die Betrachtung von Ansätzen, deren Anwendungsdomäne deutlich weiter gefasst ist als die der oben vorgestellten Optimierer für numerische Probleme. Die Konzepte der folgenden verwandten Arbeiten erlauben es, Programme in einem allgemeineren Kontext zu optimieren. Jedoch liegt auch hier der Schwerpunkt meist auf wissenschaftlichen, langlaufenden Anwendungen, die bestimmte Voraussetzungen erfüllen müssen.

4.4.2.1 FIBER

FIBER [KKHY03] (*Framework of Install-time, Before Execute-time and Run-time optimization*) ist ein Rahmenwerk zur automatischen Optimierung mittels Leistungsmessung und anschließender Iteration. Hierzu werden Funktionsaufrufe sowie Annotationen im Quelltext der Anwendung eingebettet und zu verschiedenen Zeitpunkten ausgeführt. FIBER klassifiziert die Funktionen anhand des Optimierungszeitpunkts der Tuning-Parameter: bei der Installation der Anwendung (*install-time*), vor der Ausführung (*before execution-time*) und während der Ausführung (*run-time*).

Das Rahmenwerk besteht aus der Skriptsprache *ABCLibScript*, die als eine Art Tuning-Sprache verstanden werden kann, der Bibliothek *ABCLib* (*Automatically Blocking-and-Communication Adjustment Library*, [KaKKY06]) sowie den zugehörigen Optimierungsfunktionen, die die automatische Suche nach der optimalen Parameterkonfiguration (das Auto-Tuning) vornehmen. BCLibScript umfasst einen Sprachschatz, dessen Fokus auf dem parallelen Rechnen mit verteiltem Speicher liegt.

Obwohl FIBER als ein generisches Rahmenwerk zur Optimierung von (meist wissenschaftlichen) Programmen vorgestellt wird, wird in angegebener Literaturstelle im Wesentlichen die Sicht des Anwendungsentwicklers beschrieben. Details über die Arbeitsweise des Werkzeugs werden nicht erwähnt.

4.4.2.2 MATE

MATE [MCMS+04, MCSML07, MoML07] (*Monitoring, Analysis and Tuning Environment*) ist ein Optimierungswerkzeug, das Programme zur Laufzeit optimiert. Zu diesem Zweck wird das übersetzte Programm automatisch mit Messpunkten instrumentiert. Optional werden vom Programmierer im Quelltext explizit gesetzte Messpunkte betrachtet. Diese Messpunkte sowie ein Performanzmodell werden von MATE zur Ermittlung einer optimalen Parameterkonfiguration eingesetzt. Der Schwerpunkt von MATE liegt auf verteilten wissenschaftlichen Anwendungen. Um MATE nutzen zu können, müssen die Anwendungen auf dem Programmier-Modell *PVM* (*Parallel Virtual Machine*) basieren und nach dem Prinzip der verteilten Master/Worker-Strategie aufgebaut sein. Die Optimierungsstrategie orientiert sich an typischen Problemen von Systemen mit verteiltem Speicher. Bei jedem Start des Programms wird entschieden, welche Messpunkte in die Analyse mit aufgenommen und für die Optimierung herangezogen werden. Ein so genannter *application controller* wird auf jedem Knoten eines PVM-Clusters gestartet und übernimmt die Kontrolle über die Anwendung.

Im Zusammenhang mit MATE ist das Projekt POETRIES (*Performance Oriented Environment for Transparent Resource-management, Implementing End-user parallel/distributed applications*, [CMSL04, CéSL05]) zu erwähnen. POETRIES nutzt MATE als Tuning-Umgebung und wird in angegebener Literaturstelle als so genanntes *distributed-program development environment (DPSE)* bezeichnet, wodurch – wie bei MATE – das Einsatzgebiet definiert wird. Im Wesentlichen bietet POETRIES für das Master/Worker- und das Fließband-Entwurfsmuster in verteilten wissenschaftlichen Programmen analytische Modelle an, die für die Performanzoptimierung eingesetzt werden.

MATE stellt ein umfangreiches Optimierungswerkzeug dar, das eine PVM-Anwendung zur Laufzeit überwacht, deren Leistungsengpässe analysiert und geeignete Modifikationen durchführt. Die modellbasierte Vorhersage der Leistung führt zu guten Ergebnissen. MATE basiert auf einem Online-Tuning-Ansatz und benötigt zur Optimierung keine expliziten Angaben des Anwendungsentwicklers. Allerdings ist das Einsatzgebiet auf PVM-Programme, die damit verbundene Anwendungsstruktur und Hardware-Architektur beschränkt.

4.4.2.3 Active Harmony

Active Harmony [TaCH02] stellt ein Laufzeitsystem zur Verfügung, das Programme hinsichtlich Netzwerk- und Systemressourcen optimieren kann. Hierbei geht es aber weniger um einzelne Tuning-Parameter, sondern um Bibliotheken, die zur Laufzeit ausgetauscht werden können. So soll beispielsweise der Entwickler einer Algorithmen-Bibliothek unterstützt werden, indem er über eine vorgegebene Programmierschnittstelle (API) mehrere Versionen der Bibliothek und des darin enthaltenen Algorithmus erstellt.

Alle Bibliotheken, die austauschbar sein sollen, müssen auf der Active Harmony API aufbauen.

Active Harmony entscheidet dann mittels einer suchbasierten Methode, welche Implementierung am performantesten ist. Der verwendete Suchalgorithmus wird in einer weiterführenden Arbeit verbessert, angepasst sowie formal untersucht [TaTH05].

Active Harmony ist als einer der wenigen Ansätze in der Lage, von feingranularer Optimierung zu abstrahieren und auf höherer Ebene des Programms leistungsrelevante Änderungen durchzuführen. Auch wenn sich die Zielsetzungen von Active Harmony und dieser Arbeit unterscheiden und beide Arbeiten unterschiedliche Ansätze verfolgen, so sind manche der Grundideen durchaus zu vergleichen. Insbesondere ist Active Harmony nicht auf bestimmte Anwendungsfälle beschränkt.

Auch wenn der Ansatz vielversprechend scheint, so bietet er jedoch keine explizite Unterstützung zur Optimierung *paralleler* Programme.

4.4.3 Vergleich und Bewertung

Beim Vergleich der vorgestellten Arbeiten zeichnet sich ein breites Spektrum an Techniken und Methoden und damit verbundenen Einsatzzwecken ab.

Die klassischen problemorientierten Ansätze wie FFTW liefern zur Lösung eines konkreten (meist numerischen) Problems eine möglichst optimale Berechnung. Verfahren wie ATLAS oder SPIRAL bedienen zwar eine komplette Klasse von Problemen (algebraische Berechnungen bzw. digitalen Signalverarbeitung), sind jedoch trotzdem hochspezialisiert. Domänenspezifische Verfahren sind in der Lage, innerhalb ihres Anwendungsbereichs hervorragende Ergebnisse zu produzieren. Auf Grund des beschränkten Einsatzzwecks eignen sich derartige Konzepte jedoch kaum für die softwaretechnisch relevante Optimierung ganzer Programme. Eine explizite Unterstützung für parallele Probleme steht meist nicht zur Verfügung, es sei denn, die konkrete Anwendungsdomäne erfordert dies.

Bei Betrachtung der allgemeineren Auto-Tuning-Ansätze ist zu erkennen, dass nahezu alle Techniken zum Einsatz kommen, die mit Hilfe der Auto-Tuning-Taxonomie klassifiziert wurden (vgl. Kapitel 3, Abschnitt 3.4.1).

Neben Online- und Offline-Tuning wird sowohl modellbasierte als auch suchbasierte Optimierung eingesetzt. Erstere ist beispielsweise bei MATE zentraler Bestandteil, um die Leistung von PVM-Anwendungen zu optimieren. Die Leistungsvorhersagen der Modelle sind in der Regel gut und ermöglichen eine ungefähre Berechnung vordefinierter Tuning-Parameter. Im Vergleich zu reinen suchbasierten Ansätzen wie ATLAS oder Active Harmony werden keine Tuning-Iterationen benötigt, in denen verschiedene Parameterkonfigurationen getestet werden müssen. Daher stellt MATE auch einen Online-Tuner zur Verfügung, da direkt auf Änderungen der Umgebung reagiert werden kann.

Die Modelle setzen allerdings feste Rahmenbedingungen voraus, wie zum Beispiel eine bestimmte Hardware-Plattform, Programmiermodelle sowie eine vordefinierte Anwendungsstruktur. Möchte man Teile der Anwendung ändern oder sie auf einer anderen Hardware-Plattform ausführen, werden meist stark modifizierte oder gar neue Modelle benötigt.

Durch Modellierung paralleler Entwurfsmuster (wie beispielsweise in POETRIES) kann eine gewisse Entkopplung von konkreter Applikation und Modell erreicht werden, da die zu erwartende Leistung auf der Basis von bekannten Mustern errechnet wird. Die Ergebnisse zeigen, dass mit guten Modellen die Leistung eines Entwurfsmusters durchaus exakt vorausberechnet werden kann (z.B. Master/Worker [CMSL04] oder Fließband

[CéSL05]). Dennoch berücksichtigen auch musterbasierte Modelle meist nicht, dass eine parallele Anwendung aus vielen parallelen Sektionen bestehen kann, die wiederum unterschiedliche Entwurfsmuster implementieren oder gar geschachtelt sind. Es ist daher offensichtlich ein allgemeinerer Ansatz wünschenswert, der unter Umständen etwas an Präzision verliert, aber dafür mit einem wesentlich größeren Spektrum an Applikationen umgehen kann.

Active Harmony kann als eines der wenigen Verfahren gelten, das – obwohl auf wissenschaftliche Programme spezialisiert – ein flexibles Konzept zur Optimierung ganzer Programme bietet. Es vereint einen Online- und einen Offline-Tuning-Ansatz und basiert auf austauschbaren Bibliotheken, die vom Anwendungsentwickler bereitzustellen sind. Ein besonderes Augenmerk richten die Autoren auf die Perfektionierung des Suchalgorithmus. Während andere Verfahren die klassischen Suchverfahren nahezu unverändert übernehmen, nutzt Active Harmony spezielle Varianten des Simplex-Algorithmus nach Nelder und Mead (siehe Kapitel 3, Abschnitt 3.3.2.2), die eine besondere Robustheit gegenüber Leistungsschwankungen des Programms aufweisen. Vorverarbeitungsschritte zur Suchraumreduktion (wodurch vermutlich deutlich weniger komplexe Suchstrategien ähnlich gute Ergebnisse liefern würden) sowie eine explizite Unterstützung für parallele Programme fehlen allerdings.

Im Folgenden werden die vorgestellten Auto-Tuning-Ansätze noch einmal in tabellarischer Form gegenübergestellt und an Hand einiger zentraler Eigenschaften verglichen.

Neben den drei Dimensionen der Auto-Tuning-Taxonomie (*Tuning-Typ* (Offline-Tuning (*Off*) bzw. Online-Tuning (*On*)), *Tuning-Strategie* (suchbasiertes (S) bzw. modellbasiertes (M) Tuning) und *Tuning-Zeitpunkt* (implizites (I) bzw. explizites (E) Tuning)) führt Tabelle 4.3 den jeweiligen Einsatzzweck auf (problemspezifisch (*PS*) bzw. problemunabhängig (*PU*)) und zeigt, ob die Verfahren explizit parallele Programme unterstützen sowie eine Suchraumreduktion durchführen.

	FFTW	**ATLAS**	**SPIRAL**	**FIBER**	**MATE**	**Act. Ha.**
Tuning-Typ	Off	Off	Off	Off	On	Off/On
Tuning-Strategie	S	S/M	S	S	M	S
Tuning-Zeitpunkt	E	E	E	E	I	E/I
Einsatzzweck	PS	PS	PS	PS	PU	PU
Parallelität	-	-	-	-	✓	-
Suchraumred.	-	✓	-	-	-	-

Tabelle 4.3: Vergleich der Auto-Tuning-Ansätze.

Es zeigt sich, dass bis auf einen keiner der Ansätze explizit die Optimierung paralleler Programme unterstützt. Dasselbe gilt für die Durchführung einer Suchraumreduktion, obwohl die meisten Verfahren mit suchbasierter Optimierung arbeiten.

4.5 Ansätze zur Performanzanalyse

An dieser Stelle seinen Arbeiten erwähnt, die sich nicht mit dem aktiven Optimieren von Programmen beschäftigen, sondern mit deren Performanzanalyse [LCMS+00, ShMa06, LiMa06, LiMa07]. Das Ziel derartiger Ansätze besteht darin, dem Anwendungsentwickler Anhaltspunkte zu geben, an welchen Stellen das Programm nicht performant ist (also beispielsweise überproportional viel Rechenzeit verbraucht). Zum Teil werden Analyseverfahren auf spezielle Entwurfsmuster angewendet. Mittels Wissensdatenbanken,

in denen Leistungsdaten des Entwurfsmusters archiviert wurden, werden dann durch Schlussfolgern Hinweise zur Verbesserung der Performanz generiert [LiMa07].

Eine Arbeit aus dem Bereich der Leistungsvorhersage für Softwarearchitekturen wird im Folgenden näher betrachtet, da die Modellierung von Softwarearchitekturen zur anschließenden Vorhersage der Performanz durchaus verwandte Aspekte mit der vorliegenden Arbeit aufweist.

4.5.0.1 Palladio-Komponentenmodell

Das *Palladio-Komponentenmodell (engl. Palladio Component Model (PCM)* [BeKR09] ist ein Meta-Modell zur Beschreibung und Leistungsvorhersage komponentenbasierter Softwarearchitekturen und ermöglicht eine umfangreiche Evaluierung der Softwarequalität in der Entwurfsphase der Softwareentwicklung.

Gemäß dem komponentenbasierten Softwareentwicklungsprozess wird zwischen vier verschiedenen Entwicklerrollen unterschieden, die jeweils Teilmodelle eines Softwaresystems erstellen. Zu diesen Rollen zählen der Komponentenentwickler, der Softwarearchitekt, der Experte der Anwendungsdomäne sowie des Systemintegrators. PCM leitet das vollständige Modell des Systems von den einzelnen Teilmodellen der Rollen ab, die üblicherweise in unterschiedlichen domänenspezifischen Modellierungssprachen spezifiziert sind.

Das Meta-Modell erlaubt schließlich die Spezifikation leistungsrelevanter Informationen einer komponentenbasierten Architektur. Um die Leistung einer Softwarearchitektur vorherzusagen, verwendet und erweitert PCM existierende Vorhersagemodelle, wie beispielsweise Petrinetze und Markov-Modelle.

Im Vergleich zur tatsächlichen Optimierung eines Programms erlaubt die Modellierung der Softwarearchitektur und ihrer leistungsrelevanten Parameter zwar eine durchaus präzise Vorhersage der Performanz, bietet jedoch eine deutlich geringere Flexibilität. Während im Modell leistungsrelevante Informationen explizit spezifiziert und integriert werden müssen, können bei der suchbasierten Performanzoptimierung nahezu beliebige Parameter und Leistungsmerkmale in den Tuningprozess einbezogen werden.

PCM berücksichtigt jedoch im Gegensatz zu suchbasiertem Offline-Tuning die Eingabedaten eines Programms. Mittels Wahrscheinlichkeitsfunktionen kann spezifiziert werden, wie wahrscheinlich ein bestimmtes Eingabedatum ist und in welcher Weise dieses den Programmablauf beeinflusst. Beispielsweise kann modelliert werden, bei welcher Eingabe welcher Programmzweig abgearbeitet wird.

Weiterführende Arbeiten auf der Basis von PCM befassen sich mit der Leistungsvorhersage auf Mehrprozessor- bzw. Mehrkernrechnern. Hierbei steht der Einfluss des Ablaufplaners des Betriebssystems auf die Leistung von Programmen im Mittelpunkt des Interesses [HaGR09]. Es wird untersucht, welche Modifikationen des Ablaufplaners eine Leistungssteigerung von Programmen bewirken, indem Prozesse und Ausführungsfäden auf möglichst optimale Art und Weise den vorhandenen Prozessoren und Rechenkernen zugewiesen werden.

4.6 Zusammenfassung

In diesem Kapitel wurden die wichtigsten Arbeiten und Verfahren vorgestellt, die mit der Zielsetzung und den Konzepten dieser Arbeit verwandt sind oder deren Grundlage bilden.

Mit dem Bereich der Tuning-Sprachen wurden Ansätze vorgestellt, die auf unterschiedliche Weise die Spezifikation von Tuning-Instruktionen ermöglichen. Auf Grund der teilweise komplexen Syntax sind die meisten Arbeiten jedoch nicht zur Instrumentierung größerer paralleler Applikationen geeignet, sondern finden eher im algorithmischen und numerischen Bereich Anwendung.

Das umfangreiche Gebiet der Softwareadaption wurde durch wichtige Vertreter entsprechender Rahmenwerke repräsentiert, die jedoch meist keine direkte Unterstützung für die automatische Performanzoptimierung bieten.

Der Entwurf paralleler Architekturen stellt einen weiteren verwandten Bereich dieser Arbeit dar. Mit einer Entwurfsmustersprache sowie klassischen Architekturbeschreibungssprachen wurden unterschiedliche Ansätze präsentiert, die für sich genommen gute Konzepte beinhalten, jedoch für die Zielsetzung dieser Arbeit lediglich als Grundlage dienen können.

Schließlich wurde der Bereich der automatischen Performanzoptimierung betrachtet und es wurden Arbeiten vorgestellt, die sich insbesondere in ihrer Granularität und Anwendungsdomäne unterscheiden. Kaum ein Optimierungskonzept sieht allerdings konkrete Verfahren zur Suchraumreduktion vor.

5. Konzepte und Lösungsansätze

In diesem Kapitel werden Konzepte und Lösungsansätze erörtert, die im Rahmen dieser Arbeit entwickelt wurden.

Zunächst werden die grundlegenden konzeptionellen Erweiterungen beschrieben, um Auto-Tuning unabhängig von Anwendungsdomäne und Größe der zu optimierenden Programme effizient einsetzen zu können. Anschließend werden die einzelnen Teilkonzepte dieser Arbeit vorgestellt, die auf den grundlegenden Überlegungen aufbauen.

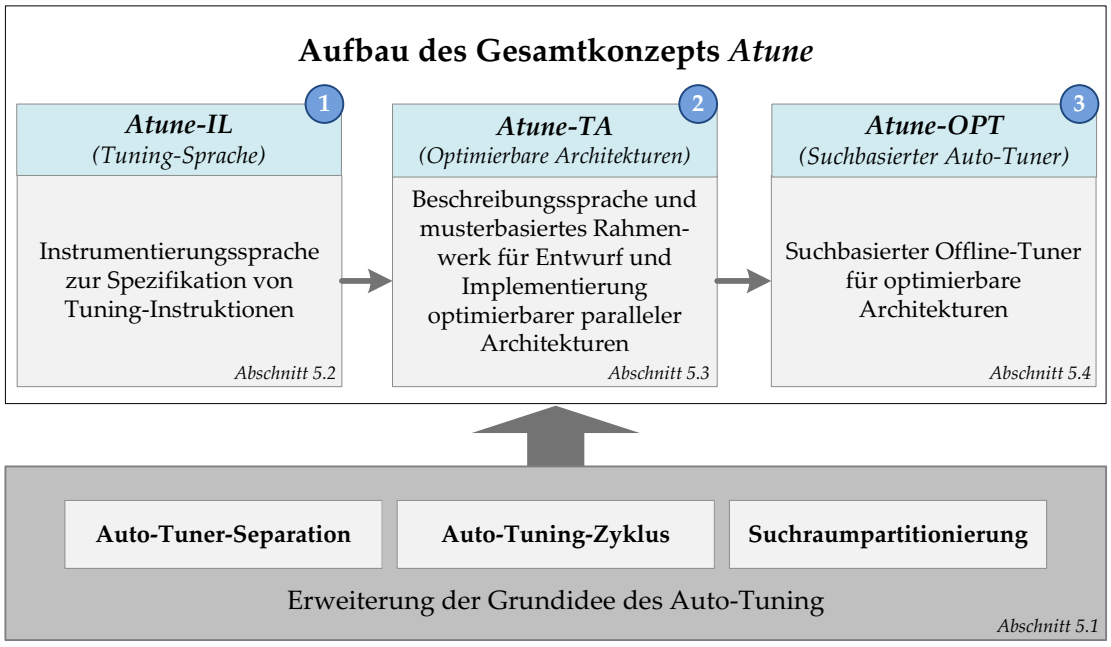


Abbildung 5.1: Gesamtkonzept *Atune* mit logisch aufeinander aufbauenenden Teilkonzepten.

Abbildung 5.1 zeigt den konzeptionellen Aufbau der Arbeit. Die Arbeit basiert auf der konzeptionellen Erweiterung des Auto-Tuning-Ansatzes. Hierfür werden die Grundkonzepte *Auto-Tuner-Separation*, *Auto-Tuning-Zyklus* und *Suchraumpartitionierung* eingeführt (Abschnitt 5.1).

Das auf der grundlegenden Erweiterung des Auto-Tuning-Ansatzes aufbauende Gesamtkonzept mit dem Namen *Atune* bildet den Rahmen um die einzelnen Teilkonzepte, die sich an den Zielsetzungen dieser Arbeit orientieren (vgl. Abbildung 2.1 in Kapitel 2):

- *Atune-IL* (*Tuning **I**nstrumentation **L**anguage*): Allgemein einsetzbare Tuning-Instrumentierungs-Sprache zur Spezifikation von Tuning-Instruktionen im Quelltext eines Programms (Abschnitt 5.2).
- *Atune-TA* (***T**unable **A**rchitectures for Parallel Applications*): Kombination aus einer Beschreibungssprache und einer musterbasierten Rahmenarchitektur für den Entwurf und die automatisierte Implementierung optimierbarer paralleler Architekturen (Abschnitt 5.3).
- *Atune-OPT* (*Automatic Architecture **OPT**imizer*): Suchbasierter Offline-Tuner mit Vorverarbeitungskonzepten zur Suchraumpartitionierung und kontextbasierter Suchraumreduktion (Abschnitt 5.4).

Eine wichtige Charakteristik der Teilkonzepte ist deren stufenweise Integration. Abbildung 5.2 veranschaulicht das Prinzip.

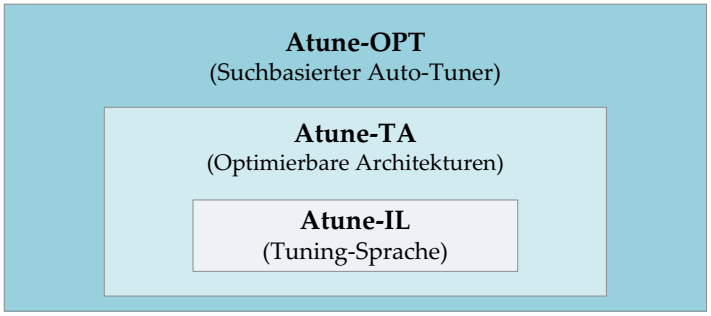


Abbildung 5.2: Integration der Teilkonzepte.

Die Tuning-Sprache Atune-IL ist Bestandteil der Umsetzung der optimierbaren Architekturen, Atune-OPT wiederum ist auf das automatische Tuning der optimierbaren Architekturen abgestimmt.

Bereits in Abschnitt 3.4 haben wir die Vor- und Nachteile von *Online-Tuning* und *Offline-Tuning* diskutiert. Hierbei wurde deutlich, dass Offline-Tuning hinsichtlich der Beschaffenheit des Programms sowie seiner Laufzeit keine Einschränkungen voraussetzt und somit einen flexiblen Ansatz darstellt. Diese Eigenschaften harmonieren mit dem Ziel dieser Arbeit, Konzepte zur Optimierung von parallelen Programmen im Allgemeinen zu entwickeln.

Aus diesem Grund fiel die Entscheidung zu Gunsten von Offline-Tuning, auf das wir uns im Folgenden stets beziehen werden. Es ist allerdings anzumerken, dass nahezu alle Konzepte dieser Arbeit – mit Ausnahme des Auto-Tuning-Ansatzes selbst – auch in Kombination mit Online-Tuning verwendet werden könnten und somit unabhängig vom Tuning-Typ sind.

5.1 Erweiterung der Grundidee des Auto-Tuning

Die Diskussion verwandter Arbeiten in Kapitel 4 zeigte, dass die meisten bisherigen Arbeiten zum Thema Auto-Tuning sich mit Ansätzen befassen, die ausschließlich in einem

speziellen Anwendungsbereich Verwendung finden. Neben der Beschränkung auf eine Anwendungsdomäne setzen existierende Ansätze meist weitere Randbedingungen voraus, wie zum Beispiel eine bestimmte Struktur des Programms oder eine bestimmte Art von Algorithmen, die Verwendung von bestimmten Programmiermodellen oder die Benutzung einer genau definierten Hardware-Plattform.

Ein wichtiges Ziel dieser Arbeit ist daher, die Grundidee des Auto-Tuning derart zu erweitern, dass parallele Architekturen mittels eines automatisierten Verfahrens optimiert werden können - unabhängig von Größe, Anwendungsgebiet oder Zielplattform der Applikation.

5.1.1 Auto-Tuner-Separation

Ein allgemeinerer Auto-Tuning-Ansatz erfordert eine Abstraktion vom zu optimierenden parallelen Programm und dessen Anwendungsbereich. Dies bedeutet, dass eine Trennung zwischen dem eigentlichen Programmablauf und dem Tuning-Vorgang vollzogen werden muss, um einerseits die Funktionalität und Übersichtlichkeit des Programms nicht durch tuning-relevante Programmteile zu beeinträchtigen, und andererseits die Tuning-Funktionalität nicht in Abhängigkeit des Programms zu setzen. Das zu Grunde liegende Konzept wird als *Separation der Interessen* (engl. *separation of concerns*) [Dijk82] bezeichnet. Die Auto-Tuning-Komponente (oder kurz der *Auto-Tuner*) sollte also nicht ein Teil des Programms oder vollständig in dieses integriert sein, sondern vielmehr den Tuning-Prozess von externer Position aus steuern.

Abbildung 5.3 veranschaulicht die Separierung von Programm und Tuning-Funktionalität.

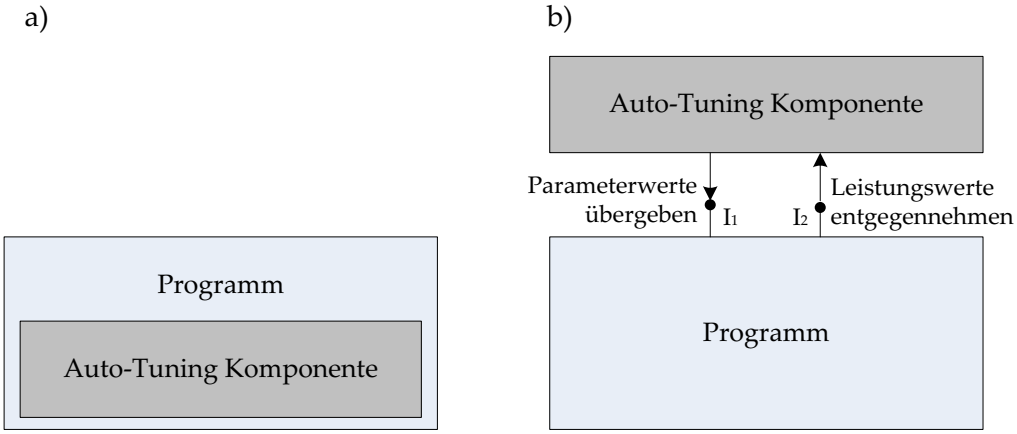


Abbildung 5.3: Darstellung der Separation von Auto-Tuning-Komponente und Programm.

Während in Abbildung 5.3 a) die Tuning-Komponente als Teil des Programms fungiert, ist sie in Abbildung 5.3 b) ausgelagert und kommuniziert mit dem Programm über zwei wesentliche Schnittstellen: Mittels I_1 werden die konkreten Werte für die Tuning-Parameter übergeben (in Form einer Parameterkonfiguration), über I_2 erhält die Auto-Tuning-Komponente den Leistungswert der gerade aktuellen Parameterkonfiguration, der zur Laufzeit des Programms gemessen wurde.

Das Übergeben einer konkreten Parameterkonfiguration und das Entgegennehmen des entsprechenden Leistungswertes repräsentiert den Prozess, mit dem der Auto-Tuner jeder Parameterkonfiguration einen Leistungswert zuordnet und dadurch in der Lage ist,

die Konfigurationen auf der Grundlage des Leistungskriteriums zu bewerten. Mit jeder Parameterkonfiguration werden Funktionswerte von $\omega_{\wp}(p_1, ..., p_n) : K \rightarrow \mathbb{R}$ (vgl. Definition 3.8) bezüglich eines Programms $\wp$ ermittelt und der Suchraum somit schrittweise erforscht.

5.1.2 Auto-Tuning-Zyklus

Der zyklische Vorgang, der Parameterkonfigurationen Leistungswerte zuweist, legt nahe, diesen Prozess allgemein durch einen Kreislauf zu beschreiben. Wir führen daher den *Auto-Tuning-Zyklus* ein, welcher in Abbildung 5.4 skizziert ist:

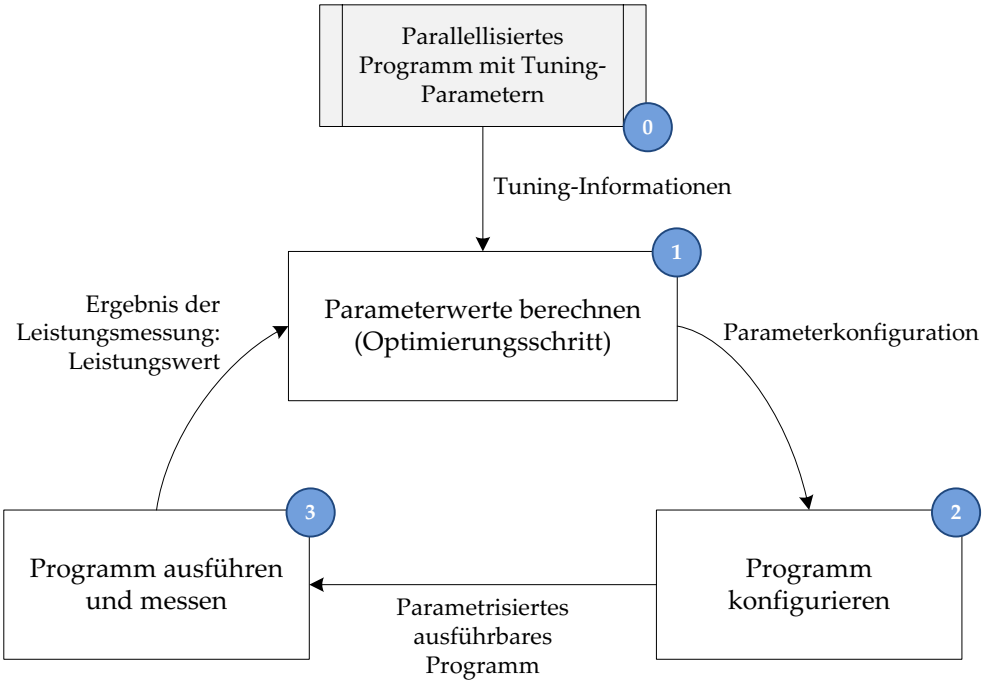


Abbildung 5.4: Darstellung des Auto-Tuning-Zyklus.

Den Ausgangspunkt (0) stellt ein parallelisiertes Programm dar, welches externalisierte Tuning-Parameter besitzt. Die Informationen über die Tuning-Parameter werden dem Auto-Tuner einmalig übermittelt. Danach beginnt der Auto-Tuning-Zyklus, der sich aus drei Schritten zusammensetzt:

1. Es werden für alle Tuning-Parameter konkrete Werte ausgewählt oder berechnet und in einer neuen Parameterkonfiguration zusammengefasst.
2. Die neue Parameterkonfiguration wird an das Programm übergeben und angewendet. Hieraus resultiert eine semantisch äquivalente Variante des Programms aus Punkt 0, welches durch die neue Parameterkonfiguration jedoch eine veränderte Performanz aufweist.
3. Das Programm wird ausgeführt und die Leistung gemäß dem Leistungskriterium gemessen. Als Ergebnis der Leistungsmessung wird der entsprechende Leistungswert zurückgegeben und der Auto-Tuning-Zyklus beginnt erneut.

Die Schritte 1 bis 3 bezeichnen wir als *Tuning-Iteration*. In einer Tuning-Iteration wird genau eine Parameterkonfiguration getestet. Nach einer Iteration kann der ermittelte Leis-

tungswert zur Berechnung einer neuen Parameterkonfiguration beitragen, indem im Optimierungsschritt bewertet wird, ob die letzte Parameterkonfiguration hinsichtlich des Leistungskriteriums gut oder schlecht war (engl. *performance feedback*).

Es ist zu beachten, dass der Auto-Tuning-Zyklus in der allgemeinen Form für alle Tuning-Verfahren (vgl. hierzu die Auto-Tuning-Taxonomie in Kapitel 3.4.1) gültig ist. Beim Online-Tuning beziehen sich die Schritte 2 und 3 nicht auf das gesamte Programm, sondern auf einen Teil des Programms, der parametrisiert ist und mehrmals innerhalb eines Programmlaufs ausgeführt wird (z.B. eine parametrisierte Schleife).

Das Konzept der Separierung von Auto-Tuner und Programm sowie die Einführung des Auto-Tuning-Zyklus tragen zur Abstraktion des Tuning-Prozesses vom Anwendungsfall und der Applikation selbst bei.

Eine begleitende Eigenschaft der Verallgemeinerung ist jedoch der Verlust von Information, wie beispielsweise Informationen über die Struktur des Programms, den Kontext von Tuning-Parametern oder das Wissen über konkrete Algorithmen. Dies führt dazu, dass sich der zu bearbeitende Suchraum zunächst vergrößert, da gilt: je weniger Wissen existiert, umso mehr muss empirisch überprüft werden. Es werden daher Methoden benötigt, die zur Reduktion des Suchraums vor dem eigentlichen Optimierungsprozess beitragen.

5.1.3 Suchraumpartitionierung

Der Suchraum wächst exponentiell mit der Anzahl der Tuning-Parameter („*Explosion des Suchraums*“), weshalb selbst ein äußerst effizienter Suchalgorithmus unter Umständen eine lange Zeit benötigt, bis er eine hinreichend gute Parameterkonfiguration gefunden hat. Hinzu kommt, dass ein nicht auf die wesentlichen Teile reduzierter Suchraum dazu führen kann, dass ein Suchalgorithmus sich in einem unwichtigen Teil des Suchraums „verfängt“ (vgl. Kapitel 3, Abschnitt 3.3.3).

Um dem Problem der Suchraumexplosion entgegenzuwirken, führen wir das Konzept der *Suchraumpartitionierung* ein. Hierbei wird der Suchraum *vor* der eigentlichen Suche in unabhängige Bereiche zerteilt, wodurch die Anzahl möglicher Parameterkonfigurationen deutlich verringert werden kann. Wie bereits im Grundlagen-Kapitel eingeführt, spricht man in diesem Zusammenhang von einer geführten Suche.

Im Folgenden werden die hierfür nötigen Überlegungen formal erörtert.

5.1.3.1 Analyse der Programmstruktur

Der grundlegende Ansatz zur Suchraumpartitionierung basiert auf Informationen über die Struktur des Programms, aus denen Wissen über Programmteile extrahiert werden kann, die *unabhängig im Sinne der Optimierung* sind.

Definition 5.1 *Zwei Programmteile P_1 und P_2 sind genau dann unabhängig im Sinne der Optimierung, wenn*

- *P_1 und P_2 auf jedem möglichen Ausführungspfad des Programms hintereinander ausgeführt werden, also das Ergebnis von P_1 stets vorliegen muss, bevor P_2 startet.*
- *in P_1 deklarierte Tuning-Parameter und Messpunkte nicht auch in P_2 verwendet werden und umgekehrt. Hierbei gilt insbesondere, dass die zu einem in P_1 deklarierten Tuning-Parameter p gehörige Programmvariable nicht außerhalb von P_1 in anderen Programmteilen verwendet werden darf.*

Zwei im Sinne der Optimierung unabhängige Programmteile sind also unabhängig bzgl. den Aspekten der Nebenläufigkeit, da sie niemals gleichzeitig ausgeführt werden können. Diese Form der Unabhängigkeit darf jedoch nicht mit der Unabhängigkeit hinsichtlich des Kontrollflusses verwechselt werden. Diese ist hier genau nicht gegeben, da die beiden Programmteile in einer vorgegebenen Reihenfolge ausgeführt werden müssen. Die Unabhängigkeit von Programmteilen bezieht sich daher im Folgenden stets auf die Unabhängigkeit im Sinne der Optimierung.

Nehmen wir nun an, ein Programm $\wp$ besitzt n voneinander unabhängige Programmteile $P_i, 1 \leq i \leq n$, die jeweils mit Tuning-Parametern ausgestattet sind. Aus der Unabhängigkeit der n Programmteile folgt, dass die Parameter eines Programmteils P_i diejenigen eines anderen Teils P_j nicht beeinflussen und umgekehrt, wobei $i \neq j$ und $1 \leq j \leq n$. Daraus wiederum folgt, dass parametrisierte unabhängige Programmteile nicht gemeinsam, also in Abhängigkeit von einander optimiert werden müssen; die Parameter aller P_i müssen nicht zu einem großen Suchraum zusammengefasst werden. Stattdessen kann jeder Programmteil P_i separat optimiert werden.

Durch die Partitionierung kann die Größe des Suchraums signifikant verkleinert werden, was im Folgenden gezeigt wird.

Sei T_i die Menge der Tuning-Parameter in Programmteil P_i, wobei $T_i \cap T_j = \emptyset$ gilt und $\overline{T}_\wp = \bigcup_{i=1}^{n} T_i$ die Menge aller Tuning-Parameter des Programms $\wp$ ist.

Sei weiterhin k_i die Anzahl der Tuning-Parametern in T_i, die gemäß den Definitionen 3.6 und 3.7 einen Suchraum C_i mit der Größe $|C_i|$ aufspannen.

Hieraus ergeben sich nun die Größen des gesamten Suchraums $\mathcal{S}_\wp$ sowie des partitionierten Suchraums $\mathcal{R}_\wp$ zu

$$|\mathcal{S}_\wp| = \prod_{i=1}^{n} |C_i|$$

bzw.

$$|\mathcal{R}_\wp| = \sum_{i=1}^{n} |C_i|$$

Somit gilt $\forall i \in \mathbb{N} > 1$ und $|C_i| > 1 : |\mathcal{R}_\wp| < |\mathcal{S}_\wp|$.

5.1.3.2 Abhängigkeitsanalyse paralleler Sektionen

Statt allgemein von Programmteilen zu sprechen, können im Kontext paralleler Applikationen auch deren *parallele Sektionen* betrachtet werden. Eine parallele Sektion beinhaltet Nebenläufigkeit, das heißt, es werden zwei oder mehrere Berechnungen parallel ausgeführt. Da parallele Sektionen ebenfalls Programmteile darstellen, gelten für sie obige Annahmen in gleicher Weise.

Hinsichtlich der Optimierung paralleler Programme sind offensichtlich deren parallele Sektionen von größtem Interesse, weswegen diese üblicherweise mit Tuning-Parametern versehen werden. Große parallele Programme können eine Vielzahl paralleler Sektionen beinhalten, so dass die Analyse der Abhängigkeitsverhältnisse lohnend erscheint.

Zur Erläuterung skizziert Abbildung 5.5 a) zwei parallele Sektionen eines Programms $\wp$, P_1 und P_2.

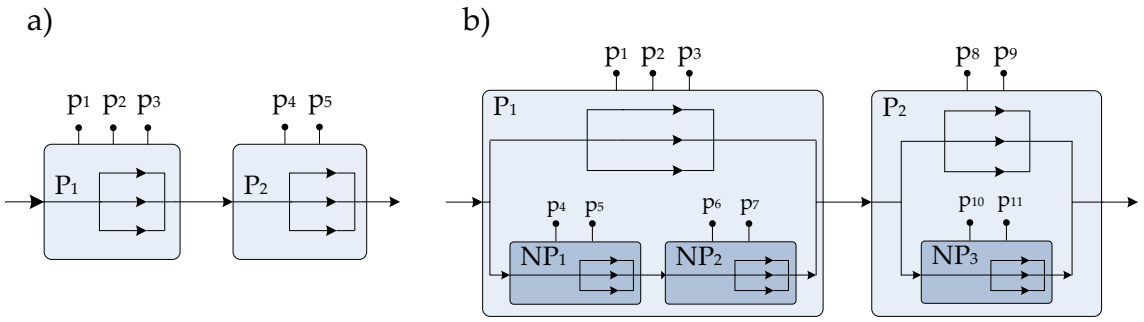


Abbildung 5.5: Darstellung zweier unabhängiger paralleler Sektionen: a) einfach; b) mit geschachtelten Sektionen und zusätzlichen Tuning-Parametern. Die Pfeile markieren den Ausführungspfad.

Der durch Pfeile dargestellte Ausführungspfad des Programms zeigt, dass die Sektionen unabhängig sind, da nie eine Nebenläufigkeit von P_1 und P_2 entstehen kann. Sektion P_1 beinhaltet drei Tuning-Parameter, $p_1, ..., p_3$, während Sektion P_2 die beiden Tuning-Parameter p_4 und p_5 besitzt. Der gesamte Suchraum ist daher definiert als $\mathcal{S}_\wp = V_{p_1} \times ... \times V_{p_5}$. Da aber bekannt ist, dass beide Sektionen unabhängig sind, kann $\mathcal{S}_\wp$ in die zwei wesentlich kleineren Suchraumpartitionen C_1 und C_2 von P_1 bzw. P_2 zerteilt werden, wobei $C_1 = V_{p_1} \times ... \times V_{p_3}$ und $C_2 = V_{p_4} \times V_{p_5}$ gilt. Der relevante Suchraum des Beispielprogramms ergibt sich dann durch $\mathcal{R}_\wp = C_1 \cup C_2$.

C_1 und C_2 bezeichnen wir als *Tuning-Einheiten*. Die beiden Tuning-Einheiten können nun nacheinander optimiert werden. Die Parameter der Tuning-Einheit, die gerade nicht optimiert wird, werden auf ihre Standardwerte gesetzt.

Abbildung 5.5 b) skizziert ebenfalls P_1 und P_2, jetzt allerdings mit geschachtelten parallelen Sektionen NP_1, NP_2 und NP_3. Bei geschachtelten Strukturen können ebenfalls separate Tuning-Einheiten gebildet werden, indem jeweils die am tiefsten geschachtelte Sektion mit all ihren Obersektionen zusammengefasst wird. Diese Zusammenfassung ist notwendig, da zunächst davon ausgegangen werden muss, dass die übergeordneten Sektionen und die geschachtelte Sektion potentiell gleichzeitig ausgeführt werden können, so dass keine Unabhängigkeit im Sinne der Optimierung gegeben ist. Die Parameter der übergeordneten Sektionen könnten daher die Parameter der geschachtelten Sektion beeinflussen oder umgekehrt.

Im Beispiel ergeben sich somit die Tuning-Einheiten $C_{1a} = V_{p_1} \times ... \times V_{p_5}$, $C_{1b} = V_{p_1} \times ... \times V_{p_3} \times V_{p_6} \times V_{p_7}$ und $C_2 = V_{p_8} \times ... \times V_{p_{11}}$.

Ist eine Sektion Teil mehrerer Tuning-Einheiten, so wird diese mehrfach optimiert, was dazu führen kann, dass die jeweils beste Parameterkonfiguration dieser Sektion je nach Tuning-Einheit unterschiedlich ist (im Beispiel ist p_1 Teil von C_{1a} und C_{1b}). In derartigen Fällen benötigt der Auto-Tuner eine Heuristik, um den Konflikt aufzulösen. Beispielsweise kann versucht werden, bei stark abweichenden optimalen Parameterkonfigurationen die Mittelwerte der divergierenden Parameter zu verwenden, um für alle Fälle eine annähernd gute Konfiguration zu erhalten.

5.2 *Atune-IL*: Tuning-Instrumentierungssprache

Der erste Schwerpunkt der Arbeit (vgl. Abbildung 5.1) beschäftigt sich mit dem Konzept der Tuning-Instrumentierungssprache *Atune-IL*. Wie in Abbildung 5.6 dargestellt, bildet Atune-IL die Grundlage für die weiteren Teilkonzepte der Arbeit.

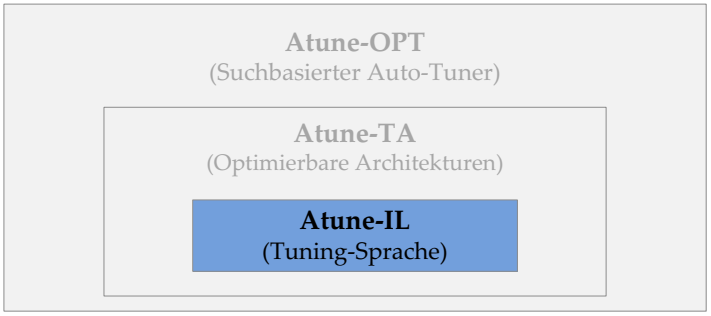


Abbildung 5.6: Atune-IL im Kontext des Gesamtkonzeptes.

Mit Atune-IL können Tuning-Instruktionen in einfacher und flexibler Art im Quelltext des parallelen Programms deklariert werden. Das Sprachkonzept sowie ein erster Prototyp konnte bereits veröffentlicht werden [ScPT09].

5.2.1 Sprachanforderungen

Im Folgenden werden die Anforderungen an eine Tuning-Instrumentierungssprache definiert, insbesondere im Hinblick auf deren flexiblen Einsatz in großen parallelen Applikationen.

5.2.1.1 Markieren von Tuning-Parametern

Die wichtigste Information für einen Auto-Tuner stellen offensichtlich die Tuning-Parameter selbst dar. Durch sie wird der initiale Suchraum definiert. Mit Hilfe der Instrumentierungssprache sollen beliebige Variablen im Programmquelltext als Tuning-Parameter markiert und mit entsprechenden Zusatzinformation versehen werden können (vgl. hierzu Kapitel 3, Abschnitt 3.6.2).

5.2.1.2 Definieren konditionaler Parameterabhängigkeiten

Eine konditionale Parameterabhängigkeiten tritt ein, falls ein Parameter q nur dann gültig ist (also in Parameterkonfigurationen aufgenommen werden muss), wenn ein anderer Parameter p bestimmte Werte annimmt. In diesem Fall ist q konditional abhängig von p.

Definition 5.2 *Eine konditionale Parameterabhängigkeit von q zu p ist definiert durch*

$$(q \rightarrow p|V'_p)$$

wobei $V'_p \subset V_p$ die möglichen Werte von p enthält, für die q gültig ist.

Mit Hilfe von Parameterabhängigkeiten kann ein exakteres Bild des Programms und der darin enthaltenen Tuning-Parameter spezifiziert werden, wodurch ein Auto-Tuner unnötige Parameterkonfigurationen ausschließen kann. Aus diesem Grund sollte die Instrumentierungssprache in der Lage sein, derartige Zusammenhänge zu beschreiben und abzubilden.

5.2.1.3 Setzen von Messpunkten

In Abschnitt 5.1.2 wurde der Auto-Tuning-Zyklus erklärt. Um ein Programm optimieren zu können, benötigt ein Auto-Tuner nach der Zuweisung und dem Testen einer neuen Parameterkonfiguration Rückmeldung über den erzielten Leistungswert, womit eine vollständige Tuning-Iteration durchlaufen ist.

Um den Leistungswert zu ermitteln, werden Messpunkte benötigt, die an relevanten Stellen im Programm (z.B. an einer parallelen Sektion) gemäß einer definierten Metrik die Leistung messen und protokollieren. Die während des Programmlaufs protokollierten Werte müssen nach Beendigung des Programms an den Auto-Tuner übergeben werden.

Viele der existierenden Sprachansätze (vgl. Kapitel 4, Abschnitt 4.1) beschränken sich auf die Deklaration von Tuning-Instruktionen sowie Transformationsregeln. Die Leistungsmessung muss mit separaten Werkzeugen durchgeführt werden. Es ist jedoch äußerst sinnvoll, sowohl das Deklarieren von Tuning-Informationen als auch das Setzen von Messpunkten mit ein und demselben Werkzeug erledigen zu können, da beides im Hinblick auf den Auto-Tuning-Zyklus zusammengehört.

5.2.1.4 Sprachkonzepte zur Unterstützung der Suchraumpartitionierung

Eine der wichtigsten Anforderungen an die Tuning-Instrumentierungssprache ist die Bereitstellung von Sprachkonstrukten, mit denen Suchraumpartitionen ausgedrückt werden können. Besonderes Augenmerk gilt hierbei der Markierung unabhängiger paralleler Sektionen, um den in Abschnitt 5.1.3.1 vorgestellten grundlegenden Ansatz zur Suchraumpartitionierung zu unterstützen.

5.2.1.5 Portabilität

Ein wichtiges Entwurfsziel von Atune-IL ist die universelle Einsetzbarkeit der Sprache hinsichtlich Programmgröße, Anwendungsdomäne und verwendeter Programmiersprache. Aus diesem Grund soll das Sprachkonzept eine hohe Portabilität gewährleisten.

Das aktuelle Konzept von Atune-IL wurde zur Verwendung auf Multikern-Plattformen mit verteiltem Speicher entworfen. Verteilte Architekturen und deren Programmiermodelle (z.B. MPI) werden nicht unterstützt.

5.2.1.6 Automatische Generierung

Große parallele Anwendungen benötigen meist eine große Menge an Instrumentierung, da sie viele parallele Sektionen enthalten, die Tuning-Parameter aufweisen. Hier sind Größenordnungen von 50 oder mehr Instrumentierungen keine Seltenheit. Das manuelle Instrumentieren ist zwar auch in diesem Fall möglich, erweist sich jedoch als zeitaufwendig.

Aus diesem Grund sollte eine automatische Generierung der Instrumentierungen durch möglichst einfache Syntax mit wenigen Sprachkonstrukten ermöglicht bzw. unterstützt werden.

5.2.2 Allgemeine Sprachkonzepte

In Abschnitt 3.6 sind bereits Möglichkeiten zur Anbindung des Auto-Tuners an ein Programm erörtert worden; allen voran die direkte Anbindung mittels Sprachkonstrukten der verwendeten Programmiersprache sowie die Anbindung durch Instrumentierung des Programms.

Um eine hohe Portabilität und Flexibilität sowie eine größtmögliche Unabhängigkeit von der zu Grunde liegenden Programmiersprache zu erreichen, wurde Atune-IL auf der Basis von Übersetzeranweisungen zur *Quelltext*-Instrumentierung entworfen. Eine ebenfalls denkbare *Binär*-Instrumentierung wäre deutlich komplexer und notwendigerweise von der jeweiligen Programmiersprache der Anwendung abhängig.

Der Quelltext des Programms wird direkt mit Atune-IL-spezifischen Übersetzerdirektiven instrumentiert. Dies bringt die folgenden Vorteile mit sich:

- **Übersetzerneutralität.** Die Übersetzerdirektiven, die mit dem Schlüsselwort *pragma* beginnen, garantieren eine Neutralität gegenüber Übersetzern, die mit den speziellen Anweisungen nicht umgehen können. Das bedeutet, dass ein mit Atune-IL instrumentiertes Programm ohne Fehler von einem herkömmlichen Übersetzer der verwendeten Programmiersprache übersetzt werden kann.
- **Plattformneutralität.** Mit Atune-IL instrumentierte Quelltexte bleiben unabhängig von der Hardware-Plattform, auf der die Anwendung zum Einsatz kommt. Es werden keine hardwarespezifischen Annahmen getroffen oder Voraussetzungen gestellt.

Die Atune-IL-Instrumentierungen innerhalb eines Programms stellen in kompakter Weise mehrere unterschiedliche, aber semantisch äquivalente Programmvarianten dar. Aus diesen Varianten wird später durch den Auto-Tuner diejenige ausgewählt, die hinsichtlich des gewählten Leistungskriteriums das beste Ergebnis erzielt. Die Generierung des nötigen Quelltextes, basierend auf den Atune-IL-Anweisungen, wird von einem Quelltextgenerator übernommen, der in Kapitel 6 behandelt wird.

Im folgenden Abschnitt werden alle Sprachkonstrukte von Atune-IL im Einzelnen vorgestellt und diskutiert.

5.2.3 Sprachkonstrukte

Die Sprachkonstrukte von Atune-IL orientieren sich an den Anforderungen, die wir weiter oben in diesem Kapitel spezifiziert haben. Der Sprachumfang setzt sich aus zwei Gruppen zusammen.

Die erste Gruppe beinhaltet Anweisungen, die direkt eine Quelltexttransformation bewirken, die also zur Definition der tuning-relevanten Programmvarianten beitragen. Zu dieser Gruppe zählt beispielsweise die Deklaration von Tuning-Parametern.

Die zweite Gruppe besteht aus Anweisungen, die für den Auto-Tuner Meta-Informationen über das Programm, dessen Struktur oder den Kontext von Parametern bereitstellen. Diese Anweisungen haben keine Auswirkungen auf die Generierung der Programmvarianten. Sie unterstützen lediglich den Optimierungsprozess. Hierzu zählen zum Beispiel alle Sprachkonstrukte, die zur Suchraumpartitionierung beitragen.

Um den Gebrauch der Sprachkonstrukte zu erklären, zeigt Listing 5.1 ein übergreifendes Beispiel eines instrumentierten C#-Programms. Das Programm sucht nach Zeichenketten in einem Text, speichert diese in einem Feld und sortiert dieses mittels paralleler Sortieralgorithmen. Schließlich wird die Anzahl an Zeichen im gesamten Feld ermittelt. Wir werden in den nächsten Abschnitten sukzessive auf das Beispiel zurückkommen. Es sei angemerkt, dass aus Gründen der Übersichtlichkeit die Atune-IL-Anweisungen der Einrückung des jeweiligen Wirtssprachenblocks folgen. Normalerweise müssen die Anweisungen jedoch stets am Zeilenanfang beginnen.

```
List<string> words = new List<string>(3);
void main()
{
    #pragma atune startblock fillBlock
    #pragma atune gauge mySortExecTime

    string text = "Auto-tuning has nothing to do with tuning cars."

    #pragma atune startpermutation fillOrder
    #pragma atune nextelem
    words.Add(text.Find("cars"));
    #pragma atune nextelem
    words.Add(text.Find("do"));
    #pragma atune nextelem
    words.Add(text.Find("Auto-tuning"));
    #pragma atune endpermutation

    sortParallel(words);

    #pragma atune gauge mySortExecTime
    #pragma atune endblock fillBlock

    countWords(words)
}

// Sortiert Liste mit Zeichenketten
void sortParallel(List<string> words)
{
    #pragma atune startblock sortBlock inside fillBlock

    IParallelSortingAlgorithm sortAlgo = new ParallelQuickSort();
    #pragma atune setvar depth values 1-4 scale ordinal
        depends sortAlgo values "new ParallelMergeSort(depth)"

    int depth = 1;
    #pragma atune setvar sortAlgo type generic
        values "new ParallelMergeSort(depth)","new ParallelQuickSort()" scale nominal

    sortAlgo.Run(words);

    #pragma atune endblock sortBlock
}

// Ermittelt die Gesamtzahl an Zeichen in der Liste
int countCharacters(List<string> words)
{
    #pragma atune startblock countBlock
    #pragma atune gauge myCountExecTime

    int numThreads = 2;
    #pragma atune setvar numThreads values 2-8 scale ordinal

    int total = countParallel(words, numThreads);

    #pragma atune gauge myCountExecTime
    return total;
    #pragma atune endblock countBlock
}
```

Listing 5.1: Mit Atune-IL instrumentiertes C#-Programm (aus [ScPT09])

Tabelle 5.1 gibt einen Überblick über die Sprachspezifikation von Atune-IL. In Anhang A.1 ist zudem die vollständige Grammatik aufgeführt, die Atune-IL beschreibt.

Anweisung	Beschreibung
Deklarieren von Tuning-Parametern	
`setvar PARAM_IDENTIFIER`	Deklariert einen Tuning-Parameter. `PARAM_IDENTIFIER`: Bezeichner, der den Parameter unter allen Parameter-Bezeichnern eindeutig identifizieren und der mit dem Namen der instrumentierten Programmvariable übereinstimmen muss.
`(type [int?|float?|bool?|string?|generic])?`	Gibt explizit den Typ des Parameters an (*optional*). *Anmerkungen*: Wird kein Typ angegeben, wird dieser durch Typinferenz von der zugehörigen Programmvariable ermittelt. Mit dem Typ `generic` kann eine beliebige rechte Seite einer Zuweisung als Quelltextfragment eingefügt werden.
`values VALUE_LIST`	Spezifiziert den Wertebereich des Tuning-Parameters. `VALUE_LIST`: Wertebereich, einzelne Werte oder eine Kombination aus beidem. Beispiele für gültige Wertebereiche: • Für numerischen Parametertyp: `values 2;3-5;7;9-15` • Für nicht-numerischen Parametertyp: `values "static";"dynamic"`
`(step INCREMENT)?`	Gibt bei numerischen Typen eine Schrittweite an, mit der der Wertebereich durchlaufen werden soll (*optional*). `INCREMENT`: Wert für die Schrittweite. *Standardwerte*: 1 bei Typ `integer`, $0,1$ bei Typ `float`.
`(scale [nominal|ordinal])?`	Spezifiziert das Skalenniveau des Parameters (*optional*). • `nominal`: Parameter ist nominalskaliert. • `ordinal`: Parameter ist ordinalskaliert. *Standardwerte*: `ordinal` bei numerischen Parametertypen, sonst `nominal`
`(default VALUE)?`	Legt den Standardwert des Parameters fest (*optional*). `VALUE`: Wert aus dem Wertebereich des Parameters. *Anmerkung*: Wird kein Standardwert angegeben, wird der erste Wert des Parameterwertebereiches (siehe Schlüsselwort `values`) als Standardwert verwendet.

Anweisung	**Beschreibung**
`(context [numthreads|lb|general])?`	Spezifiziert den Kontext bzw. Zweck des Parameters als Zusatzinformation für den Auto-Tuner (*optional*). • `numthreads`: Parameter steuert Anzahl an Ausführungsfäden. • `lb`: Parameter steuert Auswahl der Lastausgleichsstrategie. • `general`: Anderer nicht näher spezifizierter Kontext/Zweck. *Standardwert*: `general`
`(weight [0..10])?`	Spezifiziert die Gewichtung des Parameters (*optional*). Die Zahl informiert den Auto-Tuner über die Parametersensitivität hinsichtlich der Gesamtleistung des Programms (1: niedrig, 10: hoch). *Standardwert*: 10
`(depends (PARAM_IDENTIFIER VALUE_LIST)|CONDITION)?`	Definiert eine konditionale Parameterabhängigkeit (*optional*). • `PARAM_IDENTIFIER`: Bezeichner des übergeordneten Parameters. • `VALUE_LIST`: Teilmenge des Wertebereichs des übergeordneten Parameters • `CONDITION`: Algebraischer Ausdruck, der komplexere Abhängigkeiten zu mehreren Parametern definiert.
`(inside BLOCK_IDENTIFIER)?`	Weist den Parameter logisch einem Tuning-Block zu (*optional*). `BLOCK_IDENTIFIER`: Bezeichner des Tuning-Blocks.
Deklarieren von Permutationsbereichen	
`startpermutation PARAM_IDENTIFIER`	Öffnet einen Permutationsbereich, der eine beliebige Anzahl an Anweisungen der Wirtssprache enthalten kann. Ein Permutationsbereich entspricht semantisch einem Tuning-Parameter `PARAM_IDENTIFIER`: Bezeichner des Permutationsbereiches (muss unter allen Parameter-Bezeichnern eindeutig sein)
`nextelem`	Trennt die einzelnen Quelltextelemente der Wirtssprache innerhalb eines Permutationsbereiches.
`endpermutation`	Schließt einen Permutationsbereich. *Anmerkung*: Zusammengehörige `startpermutation`- und `endpermutation`-Anweisungen müssen sich innerhalb desselben Anweisungsblocks der Wirtssprache befinden.

Anweisung	Beschreibung
Deklarieren von Messpunkten	
`gauge GAUGE_IDENTIFIER`	Deklariert einen Messpunkt. `GAUGE_IDENTIFIER`: Bezeichner des Messpunktes (muss unter allen Messpunkt-Bezeichnern eindeutig sein).
`(metric [exectime|memory])?`	Gibt das Leistungskriterium an, das gemessen werden soll (*optional*). • `exectime`: Misst die Ausführungszeit. • `memory`: Misst den Speicherverbrauch. *Standardwert*: `exectime`.
Deklarieren von Tuning-Blöcken	
`startblock BLOCK_IDENTIFIER`	Öffnet einen Tuning-Block `BLOCK_IDENTIFIER`: Bezeichner des Blocks (muss unter allen Block-Bezeichnern eindeutig sein).
`(inside BLOCK_IDENTIFIER)?`	Schachtelt einen Block logisch innerhalb eines Oberblocks (*optional*). `BLOCK_IDENTIFIER`: Bezeichner des übergeordneten Tuning-Blocks. *Anmerkung*: Ein Block kann maximal einen Oberblock, aber beliebig viele Unterblöcke besitzen.
`(pattern [forkjoin|pipeline|replication|general])?`	Spezifiziert die im Tuning-Block enthaltene Parallelisierungsstrategie (Muster) als Kontextinformation für den Auto-Tuner (*optional*). • `forkjoin`: Block enthält eine Fork/Join-Sektion. • `pipeline`: Block enthält eine Fließband-Sektion. • `replication`: Block enthält eine datenparallele Sektion. • `general`: Block enthält einen nicht näher spezifizierten Programmteil. *Standardwert*: `general`. *Anmerkung*: Enthält ein Block eine geschachtelte parallele Sektion, so muss stets die äußerste Sektion spezifiziert werden.
`endblock`	Schließt einen Tuning-Block. *Anmerkung*: Zusammengehörige `startblock`- und `endblock`-Anweisungen müssen sich innerhalb desselben Anweisungsblocks der Wirtssprache befinden.

Tabelle 5.1: Sprachkonstrukte von Atune-IL

In den folgenden Abschnitten werden die Sprachkonstrukte im Einzelnen behandelt.

5.2.3.1 Tuning-Parameter

Als einen Tuning-Parameter bezeichnen wir eine Programmvariable, die zur Beeinflussung leistungsrelevanter Aspekte des Programmablaufs verwendet wird (vgl. hierzu Definition 3.1).

Mit Hilfe von Atune-IL können derartige Variablen in der Wirtssprache mit den nötigen Tuning-Anweisungen instrumentiert werden. Beispiele für leistungsrelevante Variablen sind die Anzahl der Ausführungsfäden in einer parallelen Sektion, die Auswahl der besten Implementierung eines Algorithmus oder die Größe von Datenpartitionen.

Um einen Tuning-Parameter zu definieren, stellt Atune-IL die `setvar`-Anweisung zur Verfügung:

```
#pragma atune setvar PARAM_IDENTIFIER values VALUE_LIST
```

Die `setvar`-Anweisung muss stets nach der Variablendeklaration erfolgen, die als Tuning-Parameter markiert werden soll. Der Bezeichner (`PARAM_IDENTIFIER`) der `setvar`-Anweisung muss dem Namen der Variable in der Deklaration entsprechen.

Atune-IL unterstützt Programmvariablen mit den Typen *integer*, *float*, *boolean* sowie *string*. Der jeweilige Typ der Variablen wird von Atune-IL automatisch durch Typinferenz festgestellt. Ist eine Typinferenz nicht möglich oder nicht erwünscht, kann der Typ des Parameters auch explizit mittels des Attributs `type` spezifiziert werden.

Außer beim Typ *boolean* (hier besteht der zulässige Wertebereich stets aus den Werten *wahr* und *falsch*) muss in jedem Fall ein für den Typ der Programmvariable gültiger Wertebereich angegeben werden. Bei den numerischen Typen *integer* und *float* kann mit dem Attribut `step` zusätzlich eine Schrittweite definiert werden. Standardwerte sind 1 bzw. $0,1$.

Gültige Definitionen von Wertebereichen für einen Parameter vom Type *integer* sind beispielsweise

```
... values 2-16
... values 2-8;9-20
... values 2;3-5;7;9-15
```

Die Instrumentierung der Variablen repräsentiert also eine dynamische Wertzuweisung basierend auf einem Wertebereich, wobei numerische und nicht-numerische Parameter definiert werden können. Ausdrücke, die einen dynamischen Wertebereich in Abhängigkeit anderer Parameter definieren, sind im Sprachkonzept nicht vorgesehen. Eine sinnvolle Einschränkung der Wertebereiche (z.B. in Abhängigkeit der verfügbaren Rechenkerne) wird dem Auto-Tuner überlassen.

Neben den oben genannten Typen, die von Atune-IL unterstützt werden, kann auch ein generischer Typ verwendet werden, indem der Typ als `generic` deklariert wird. Auf diese Weise kann eine beliebige Zuweisung realisiert werden; als Wertebereich können beliebige linke Seiten einer Zuweisung angegeben werden.

Die `setvar`-Anweisung erlaubt die Angabe einer Reihe optionaler Attribute:

- **Skalenniveau.** Das Attribut `scale` gibt optional das Skalenniveau des Parameters an. Mögliche Werte sind `ordinal` und `nominal`. Diese Information kann für einen Auto-Tuner wichtig sein, da nominalskalierte Parameter (deren Wertebereich also keine Metrik zu Grunde liegt) während des Optimierungsprozesses je nach Suchstrategie gesondert behandelt werden müssen.

  Wird kein Skalenniveau definiert, so wird für die numerischen Typen eine ordinale und für die nicht-numerischen Typen eine nominale Skalierung angenommen.

- **Parameterzweck.** Mit dem Attribut `context` kann ein Verwendungszweck des Parameters als Meta-Information für den Auto-Tuner angegeben werden. Konzeptionell akzeptiert das Attribut einen beliebiger Bezeichner, der einen Nutzungskontext des Parameters beschreibt. Fest im Sprachumfang verankert sind die Werte `numthreads` und `lb`, die angeben, dass der Parameter entweder die Anzahl an Ausführungsfäden oder eine Lastausgleichsstrategie beeinflusst. Diese beiden Verwendungszwecke gehören nach Erfahrungswerten zu den am häufigsten gebräuchlichen Parameterzwecken.

  Der Auto-Tuner kann, sofern ein Zweck angegeben ist, mit dieser Information Rückschlüsse auf die Verwendung des Parameters ziehen und seine Optimierungsstrategie entsprechend anpassen oder den Wertebereich des Parameters sinnvoll einschränken.

- **Parameterabhängigkeit.** Das Attribut `depends` bietet die Möglichkeit, konditionale Abhängigkeiten zu anderen Parametern anzugeben (siehe Definition 5.1). Dies kann auf zwei unterschiedliche Arten erfolgen.

  Weist ein Parameter q eine einfache Abhängigkeit zu einem Parameter p mit Wertebereich V_p auf, so kann neben einer Referenz auf p der Wertebereich $V'_p \subset V_p$ definiert werden, der die Werte enthält, bei denen q berücksichtigt werden muss:

  ```
  #pragma atune p values 2-10
  ...
  #pragma atune q values 4-8 depends p values 6-8
  ```

  In diesem Beispiel wird q nur dann in Parameterkonfigurationen aufgenommen, wenn p die Werte 6, 7 oder 8 annimmt.

  Erfordert die Abhängigkeitsbedingung eine komplexere Beschreibung (beispielsweise bei Abhängigkeiten zu mehreren Parametern), kann die Bedingung des `depends`-Attributs auch durch einen algebraischen Ausdruck spezifiziert werden:

  ```
  #pragma atune p values 2-10
  #pragma atune r values 1-5
  ...
  #pragma atune q values 4-8 depends (6<=p<=8 and r=3)
  ```

  Hier ist q nur dann zu beachten, wenn p die Werte 6, 7 oder 8 *und* r den Wert 3 annimmt.

- **Gewichtung.** Das Attribut `weight` erlaubt die optionale Angabe eines Gewichtungsfaktors, mit dem die Parametersensitivität gesetzt werden kann. Als gültige Werte sind $\{1, ..., 10\}$ definiert, wobei 1 die niedrigste und 10 die höchste Parametersensitivität repräsentiert. Die Werte dazwischen stellen entsprechende Abstufungen dar. Wird keine Gewichtung angegeben, so ist 10 der Standardwert.

Betrachten wir die Verwendung der `setvar`-Anweisung am Beispiel in Listing 5.1, wobei wir uns auf die Methode `sortParallel()` konzentrieren. Hier werden zwei Tuning-Parameter deklariert: `sortAlgo` und `depth`. Für beide Tuning-Parameter existieren entsprechende Variablendeklarationen, denen jeweils ein Standardwert zugewiesen wird.

Der Parameter `sortAlgo` bestimmt, welcher konkrete parallele Sortieralgorithmus verwendet werden soll. Zur Auswahl stehen `ParallelMergeSort` und `ParallelQuickSort`. Der Typ des Tuning-Parameters ist mit `generic` angegeben, da die möglichen Zuweisungen (`new ParallelMergeSort(depth)` und `new ParallelQuickSort()`) als Quelltextfragmente und nicht als Werte vom Typ *string* interpretiert werden sollen. In den Programmvarianten, die während des späteren Optimierungsprozesses auf Grund dieses Parameters generiert werden, wird also entweder eine Instanz vom Typ `ParallelMergeSort` oder `ParallelQuickSort` erzeugt.

Der Parameter `depth` ist konditional abhängig von `sortAlgo`. `depth` bestimmt die Rekursionstiefe von `ParallelMergeSort` und damit dessen Parallelitätsgrad. Offensichtlich ist der Parameter `depth` nur dann relevant, wenn `ParallelMergeSort` als Sortieralgorithmus gewählt wurde. Dieser Sachverhalt wird durch das Attribut `depends` ausgedrückt.

Anmerkungen. Unabhängig vom Typ müssen die angegebenen Werte bzw. der Wertebereich einer `setvar`-Anweisung gültige Zuweisungen bezüglich der instrumentierten Programmvariable darstellen. Zudem muss sichergestellt sein, dass die Variable in der Wirtssprache korrekt deklariert und mit einem Standardwert initialisiert ist.

5.2.3.2 Permutation von Anweisungen

Die Reihenfolge von aufeinanderfolgenden Anweisungen in einem bestimmten Kontext kann in vielen Fällen die Leistung des Programms oder einer parallelen Sektion beeinflussen, wie beispielsweise das Konstruieren und Befüllen einer Datenstruktur, auf der anschließend gearbeitet wird.

Aus diesem Grund bietet Atune-IL neben der Deklaration von Tuning-Parametern, die eine dynamische Wertzuweisung ermöglichen, ein Sprachkonstrukt zur Permutation von Anweisungen der Wirtssprache.

Um aufeinanderfolgende Anweisungen der Wirtssprache als permutierbar zu markieren, wird mit den Atune-IL-Konstrukten `startpermutation` und `endpermutation` ein Permutationsbereich definiert, in den die Anweisungen eingeschlossen werden. Mit der Anweisung `nextelem` werden die Permutationselemente definiert, wobei ein Permutationselement aus einer oder mehreren Anweisungen der Wirtssprache besteht.

```
#pragma atune startpermutation PARAM_IDENTIFIER
#pragma atune nextelem
... Anweisungen der Wirtssprache...
#pragma atune nextelem
... Anweisungen der Wirtssprache...
#pragma atune endpermutation
```

Der Bezeichner des Permutationsbereiches (`PARAM_IDENTIFIER`) kann beliebig (aber eindeutig unter allen Parameter-Bezeichnern im Programm) gewählt werden, da keine zugehörige Programmvariable existiert, zu der eine Referenz hergestellt werden muss. Ein Permutationsbereich gleicht semantisch einem Tuning-Parameter, dessen Wertebereich sich aus allen möglichen Permutationen der eingeschlossenen Permutationselemente ergibt.

Zur Verdeutlichung sei wieder auf das Beispiel in Listing 5.1 verwiesen. In der Methode `main()` ist ein Permutationsbereich `fillOrder` definiert. Die enthaltenen Anweisungen der Zielsprache suchen Wörter in einem gegebenen Text und fügen diese der Liste `words` hinzu, in der die Wörter in derselben Reihenfolge abgelegt werden, in der sie hinzugefügt werden.

Mit Hilfe des definierten Permutationsbereiches kann nun dynamisch die Reihenfolge bestimmt werden, in der die Wörter der Liste hinzugefügt werden. Da die Liste direkt nach dem Befüllen sortiert wird, übt die Reihenfolge der Elemente in der unsortierten Liste einen nicht unerheblichen Einfluss auf die Leistung des parallelen Sortieralgorithmus aus.

Obgleich in diesem einfachen Fall die optimale Reihenfolge schnell ohne die Hilfe eines Auto-Tuners erkennbar ist, so sind weitaus komplexere Beispiele denkbar, in denen die optimale Permutation von Anweisungen keineswegs intuitiv zu ermitteln ist, aber dennoch das Verhalten nachfolgender paralleler Sektionen signifikant beeinflusst.

Anmerkungen. Es liegt im Verantwortungsbereich des Anwendungsentwicklers, ausschließlich Anweisungen in einen Permutationsbereich einzubetten, deren Ausführungsreihenfolge keine Auswirkungen auf den korrekten Ablauf des Programms nach sich ziehen. Des Weiteren muss ein Permutationsbereich innerhalb eines Anweisungsblocks der Wirtssprache deklariert sein.

5.2.3.3 Messpunkte

Für die Ermittlung des Leistungswertes in Abhängigkeit einer konkreten Parameterkonfiguration werden Messpunkte benötigt. Wie in Abschnitt 3.1.5 definiert, bezeichnet ein Messpunkt eine bestimmte Position im Programm, an der gemäß eines Leistungskriteriums Daten erfasst werden, um die Leistung des Programms zu ermitteln.

Atune-IL stellt zu diesem Zweck die `gauge`-Anweisung zur Verfügung, mit der ein Messpunkt deklariert wird:

```
#pragma atune gauge GAUGE_IDENTIFIER
    (metric [exectime|memory])?
```

Der Bezeichner (`GAUGE_IDENTIFIER`) einer `gauge`-Anweisung kann frei gewählt werden, muss aber innerhalb des Programms eindeutig in Bezug auf die Bezeichner anderer `gauge`-Anweisungen sein.

Das Sprachkonzept von Atune-IL unterstützt die Messung der Laufzeit sowie des Speicherverbrauchs. Das entsprechende Leistungskriterium kann mit Hilfe des Attributs `metric` für jeden Messpunkt definiert werden.

Für einen Messpunkt, der die Laufzeit des Programms (oder eines Programmteils) bestimmen soll, wird offensichtlich eine Start- sowie eine Endzeit benötigt. Aus diesem Grund besteht ein Messpunkt zur Ermittlung der Laufzeit (`type exectime`) stets aus zwei aufeinanderfolgenden `gauge`-Anweisungen mit demselben Bezeichner. Jeder der beiden Messpunkte speichert nur einen Zeitstempel. Die Zeit, die zur Ausführung des Quelltextes zwischen den beiden `gauge`-Anweisungen benötigt wurde, entspricht schließlich der Differenz der beiden Zeitstempel und ergibt die Laufzeit. Zwei zusammengehörige `gauge`-Anweisungen zur Laufzeitmessung müssen sich im selben Anweisungsblock der Wirtssprache befinden.

Bei der Messung des Speicherverbrauchs ist nur eine `gauge`-Anweisungen erforderlich.

Die Spezifikation von Atune-IL lässt die Deklaration beliebig vieler Messpunkte in einem Programm zu. Existiert mehr als ein Messpunkt, so fällt die geeignete Aggregation der Leistungswerte (z.B. Aufsummierung, Durchschnittsbildung) in den Verantwortungsbereich des Auto-Tuners, der die Werte nach Abschluss einer Tuning-Iteration erhält.

Im Beispielprogramm sind zwei Messpunkte deklariert (`mySortExecTime` in der Methode `main()` und `myCountExecTime` in der Methode `countCharacters()`). Da in beiden Fällen die Laufzeit gemessen werden soll, existieren jeweils zwei gleichnamige `gauge`-Anweisungen, die jeweils die zu messenden Quelltextsegmente einschließen. Der Messpunkt `mySortExecTime` misst also die Laufzeit der Wortsuche und des parallelen Sortieralgorithmus, während der Messpunkt `myCountExecTime` die Zeit misst, die für das Zählen der Zeichen aller in der Liste gespeicherten Wörter benötigt wird.

Anmerkungen. Eine `gauge`-Anweisung wird bei der Verarbeitung der Atune-IL-Instrumentierung in einen Bibliotheksaufruf übersetzt. Daher muss eine `gauge`-Anweisung stets an einer Position im Quelltext deklariert werden, die von einem Ausführungspfad des Programms erreicht wird.

Die Ermittlung von Leistungswerten stellt einen essentiellen Schritt im Optimierungsprozess dar. Daher trägt die geeignete Platzierung von Messpunkten innerhalb des Programms in großem Maße zu einer korrekten und vergleichbaren Optimierung bei. Diese Verantwortung wird bewusst dem Anwendungsentwickler übertragen, da er die Struktur und die Eigenschaften des Programms am besten kennt und daher in der Lage ist, korrekte und sinnvolle Messpunkte zu deklarieren.

Dennoch wird später im Rahmen von Atune-TA ein Ansatz zur automatischen Generierung von Atune-IL-Instrumentierungen vorgestellt, der es ermöglicht, auch Messpunkte implizit zu setzen und somit dem Problem der Fehleranfälligkeit entgegenwirkt.

5.2.3.4 Tuning-Blöcke

In vielen modernen Programmiersprachen wie beispielsweise C/C++ findet sich das Konzept der Anwendungsblöcke (engl. *compound statements*) [Micr]. Ein Anwendungsblock stellt den Gültigkeitsbereich der in ihm deklarierten Variablen dar und definiert somit deren Sichtbarkeit.

Atune-IL bietet mit den *Tuning-Blöcken* ein vergleichbares Sprachkonstrukt, um innerhalb des Atune-IL-Kontextes den Gültigkeitsbereich von Tuning-Parametern sowie Messpunkten zu definieren. Ein Tuning-Parameter oder Messpunkt, der innerhalb eines Blocks B deklariert wird, ist nur innerhalb von B sichtbar und besitzt daher eine auf B beschränkte lokale Gültigkeit.

Das Konzept der Tuning-Blöcke basiert jedoch auf einer etwas eingeschränkteren Semantik als die aus Programmiersprachen bekannten Anwendungsblöcke. Die Intention von Tuning-Blöcken ist die Markierung und Abgrenzung von unabhängigen parallelen Programmabschnitten, die mit Tuning-Parametern oder Messpunkten versehen sind.

Hierbei wird das in Abschnitt 5.1.3 vorgestellte Konzept zur Partitionierung des Suchraums zu Grunde gelegt, das im Wesentlichen auf Architekturinformationen und der Abhängigkeitsanalyse der parallelen Sektionen des Programms beruht (siehe Definition 5.1). Parametrisierte parallele Sektionen, die nicht mit einander interagieren, können demnach separat optimiert werden.

Um einen Tuning-Block zu deklarieren, stehen die Anweisungen `startblock` und `endblock` zur Verfügung. Ein Tuning-Block kann sowohl Quelltext der Wirtssprache als auch Atune-IL-Anweisungen beinhalten, um beispielsweise Tuning-Parameter zu deklarieren:

```
#pragma atune startblock BLOCK_IDENTIFIER
    (pattern [forkjoin|pipeline|replication|general])?

...Quelltext der Wirtssprache und
   andere Atune-IL-Anweisungen...

#pragma atune endblock
```

Der Bezeichner (`BLOCK_IDENTIFIER`) eines Tuning-Blocks kann frei gewählt werden, muss aber innerhalb des Programms eindeutig in Bezug auf die Bezeichner anderer Tuning-Blöcke sein. Das optionale Attribut `pattern` ermöglicht die Spezifikation der Parallelisierungsstrategie (Muster), die der Block einschließt (z.B. `pipeline`), um dem Auto-Tuner Zusatzinformationen über die Art der markierten parallelen Sektion zur Verfügung zu stellen.

Um auch komplexe Architekturen abbilden zu können, wird die Schachtelung von Tuning-Blöcken unterstützt. Dies kann entweder lexikalisch – also durch Deklaration eines Tuning-Blocks innerhalb eines anderen – oder logisch geschehen. Für letzteren Fall wird das Attribut `inside` verwendet, womit ein existierender Tuning-Block innerhalb des Programms referenziert und auf diese Weise als Oberblock definiert werden kann:

```
#pragma atune startblock block1
...
#pragma atune endblock


#pragma atune startblock block2 inside block1
...
#pragma atune endblock
```

Der referenzierte Tuning-Block (im obigen Beispiel `block1`) muss lexikalisch nicht notwendigerweise vor dem referenzierenden Block (hier `block2`) deklariert werden. Referenzen zwischen Tuning-Blöcken sind selbst über Dateigrenzen hinweg möglich.

Ein Tuning-Block kann maximal einen Oberblock, jedoch beliebig viele Unterblöcke besitzen. Geschachtelte Tuning-Blöcke sind naturgemäß voneinander abhängig, so dass ein Unterblock nie getrennt von seinem Oberblock betrachtet werden kann.

Das Beispiel in Listing 5.1 beinhaltet die Tuning-Blöcke `sortBlock`, `fillBlock` und `countBlock`. `sortBlock` ist logisch in `fillBlock` geschachtelt, während `countBlock` als unabhängig von allen anderen Tuning-Blöcken betrachtet wird. Abbildung 5.7 veranschaulicht die Blockstruktur.

Gemäß der Tuning-Block-Semantik werden die Parameter `sortAlgo` und `depth` innerhalb von `sortBlock` unter Berücksichtigung des Parameters `fillOrder` in `fillBlock` optimiert, während `numThreads` in `countBlock` separat betrachtet werden kann. Die Abhängigkeit der Parameter `sortAlgo` und `depth` von `fillBlock` beruht auf der Tatsache, dass die Reihenfolge der Wörter in der Liste `words` das Sortieren beeinflusst.

Die aus der gegebenen Instrumentierung resultierende Tuning-Block-Struktur ermöglicht das Partitionieren des initialen Suchraums $\mathcal{S} = V_{fillOrder} \times V_{sortAlgo} \times V_{depth} \times V_{numThreads}$ in die beiden kleineren Suchraumpartitionen $C_1 = V_{fillOrder} \times V_{sortAlgo} \times V_{depth}$ und $C_2 = V_{numThreads}$, die separat optimiert werden können.

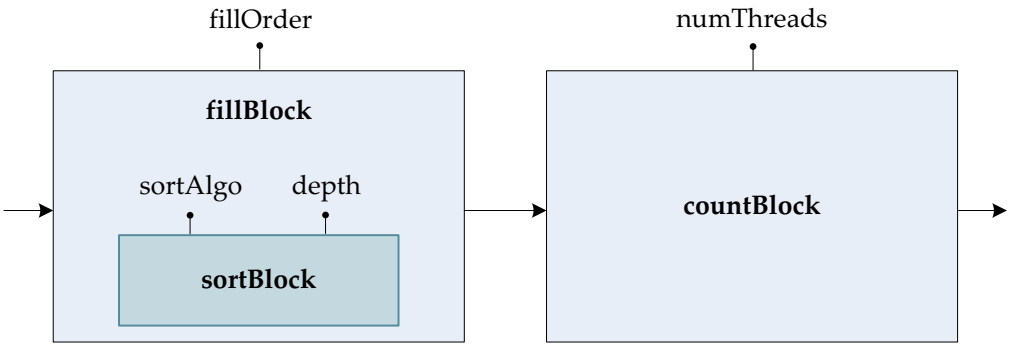


Abbildung 5.7: Darstellung der Tuning-Block-Struktur des Beispielprogramms aus Listing 5.1.

Anmerkungen. Im Quelltext des Programms muss ein Block innerhalb desselben Anweisungsblocks, ein lexikalisch geschachtelter Block vollständig innerhalb seines Oberblocks definiert sein. Blöcke, die dieser Konvention nicht folgen, sind ungültig und werden ignoriert. Es gelten demnach dieselben Regeln wie für herkömmliche Anweisungsblöcke in Programmiersprachen.

Atune-IL fügt der Struktur der Tuning-Blöcke eines Programms implizit einen globalen Tuning-Block hinzu, der das gesamte Programm umfasst und die Wurzel des auf diese Weise entstehenden Baumes bildet. Tuning-Parameter oder Messpunkte, die keinem Tuning-Block zugeordnet sind, werden automatisch dem globalen Tuning-Block zugeordnet. Daraus folgt, dass jeder Tuning-Parameter oder Messpunkt einem Block zugehörig ist – explizit oder implizit.

Grundsätzlich muss angemerkt werden, dass auch eine Quelltextanalyse Hinweise auf unabhängige Programmabschnitte liefern kann. Derartige Verfahren benötigen jedoch unter Umständen zusätzliche Programmläufe oder können zu unpräzisen Ergebnissen führen. Aus diesem Grund basiert Atune-IL in seinem Grundkonzept auf expliziten Instrumentierungen.

Jedoch kann auch an dieser Stelle bereits auf einen Ansatz zur automatischen Generierung der Atune-IL-Instrumentierungen verwiesen werden, der im weiteren Verlauf dieses Kapitels kurz sowie im nächsten Kapitel ausführlich behandelt wird.

5.2.4 Zusammenfassung und Vergleich

Atune-IL stellt im Vergleich zu den meisten der verwandten Arbeiten (vgl. Kapitel 4, Abschnitt 4.1) eine allgemein einsetzbare Tuning-Sprache dar. Die einfache und zugleich mächtige Sprache ermöglicht die Definition umfangreicher Tuning-Instruktionen.

Die meisten verwandten Arbeiten (z.B. XLanguage oder POET) sind auf kleine numerische Programme spezialisiert, die in Regel ein einziges konkretes Problem lösen, wie beispielsweise die Matrix-Multiplikation. Auf Grund der eng gefassten Anwendungsdomäne können derartige Sprachen offensichtlich zielgerichtetere Instrumentierungs-Konstrukte zur Verfügung stellen, die die Quelltext-Transformation wesentlich feingranularer steuern können.

Allerdings trägt die hohe Spezialisierung dazu bei, dass parallele Programme im Allgemeinen kaum im Fokus der Arbeiten liegen.

Atune-IL bietet zudem Konstrukte zur Definition von Messpunkten sowie zur Suchraumpartitionierung. Beide Eigenschaften bescheren Atune-IL ein Alleinstellungsmerkmal.

5.3 *Atune-TA*: Optimierbare parallele Architekturen

In Kapitel 3 wurden bereits die Vorzüge einer klar strukturierten Architektur für parallele Programme besprochen und es wurde die Verwendung paralleler Entwurfsmuster als wichtige vordefinierte Architekturbausteine erläutert.

Als zweites Teilkonzept dieser Arbeit führen wir daher *Atune-TA* ein, welches ein musterbasiertes Verfahren für den Architekturentwurf paralleler optimierbarer Programme darstellt. Das Ergebnis des Entwurfsprozesses ist eine *optimierbare Architektur*. Eine der zentralen Ideen ist die implizite Zusammenführung von Parallelisierung und Optimierung in Form eines grobgranularen Entwurfsprozesses.

Abbildung 5.8 ordnet Atune-TA in das Gesamtkonzept der Arbeit ein. Atune-IL stellt einen wesentlichen Bestandteil der Umsetzung von Atune-TA dar und kommt bei der prototypischen Implementierung zum Einsatz, die im nächsten Kapitel vorgestellt wird.

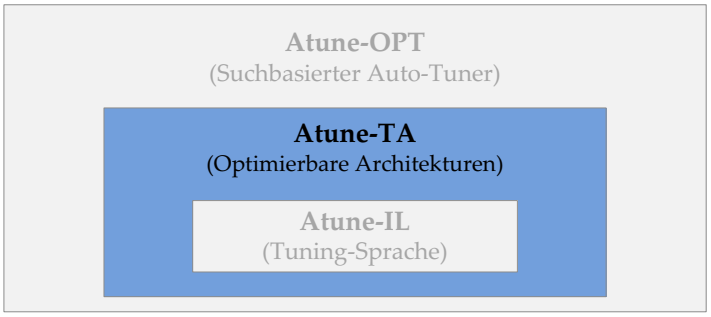


Abbildung 5.8: Atune-TA im Kontext des Gesamtkonzeptes.

5.3.1 Notwendigkeit *optimierbarer Architekturen*

Die Entwicklung paralleler Applikationen gestaltet sich oftmals zeitaufwendig und fehleranfällig. Die Ausnutzung vorhandener Parallelität bedarf insbesondere bei großen Anwendungen einer genauen und zeitintensiven Planung, da meist auf mehreren softwaretechnischen Ebenen des Programms Parallelisierungspotential vorhanden ist [PSJT08].

Jedoch nicht nur die Planung, sondern auch die Implementierung der parallelen Sektionen sowie die Interaktion zwischen grob- und feingranularer Parallelität verlangt ein hohes Maß an Expertenwissen und stellt selbst erfahrene Software-Entwickler vor große Herausforderungen, da Synchronisation, Verwaltung der Fäden sowie Ablaufplanung der einzelnen parallelen Sektionen von Hand umgesetzt werden müssen.

Die Komplexität der Parallelisierung auf mehreren Ebenen eines Programms war bereits Schwerpunkt einiger Arbeiten und Studien, in denen mögliche Parallelisierungsansätze auf ihre Effizienz untersucht wurden [PSJT08, PaJT09]. Grundsätzlich eignet sich demnach gerade für größere Anwendungen eine *Top-Down-Strategie*, bei der die gesamte Architektur des Programms von oben herab hinsichtlich potentieller Parallelität analysiert wird. Zunächst stehen also grobgranulare Parallelisierungen im Fokus, wodurch meist schon ein wesentlicher Teil des vorhandenen Parallelisierungspotentials ausgenutzt werden kann. Falls erforderlich, können anschließend auf tieferen Ebenen zusätzlich feingranulare Parallelisierungen vorgenommen werden.

Neben dem Einsatz von Parallelisierungsstrategien auf den verschiedenen softwaretechnischen Ebenen eines Programms darf die Optimierung nicht außer Acht gelassen werden. Programme, die parallele Sektionen an verschiedenen Stellen ihrer Architektur aufweisen, bieten ein enormes Optimierungspotential und damit eine kaum zu überschauende Zahl an Tuning-Parametern.

Abbildung 5.9 veranschaulicht die Situation an Hand eines Beispiels. Das Diagramm zeigt den schematischen Aufbau eines Programms, das Datenpakete verarbeitet und auf drei softwaretechnischen Ebenen parallelisiert wurde. Zunächst werden die übergebenen Datenpakete auf oberster Ebene mittels eines Fließbandes verarbeitet. Das Fließband gibt somit die Architektur der Anwendung vor. Innerhalb der Fließbandstufen (Stufe 1 bis 4) werden die Datenpakete von mehreren Algorithmen (M_1 bis M_{10}) analysiert. Manche der Algorithmen können unabhängig von einander ausgeführt werden und sind daher nebenläufig angeordnet. Von einigen dieser Algorithmen können wiederum mehrere Instanzen erzeugt werden, die die anfallenden Daten unter sich aufteilen und parallel verarbeiten (siehe die beispielhafte Darstellung für M_{10}).

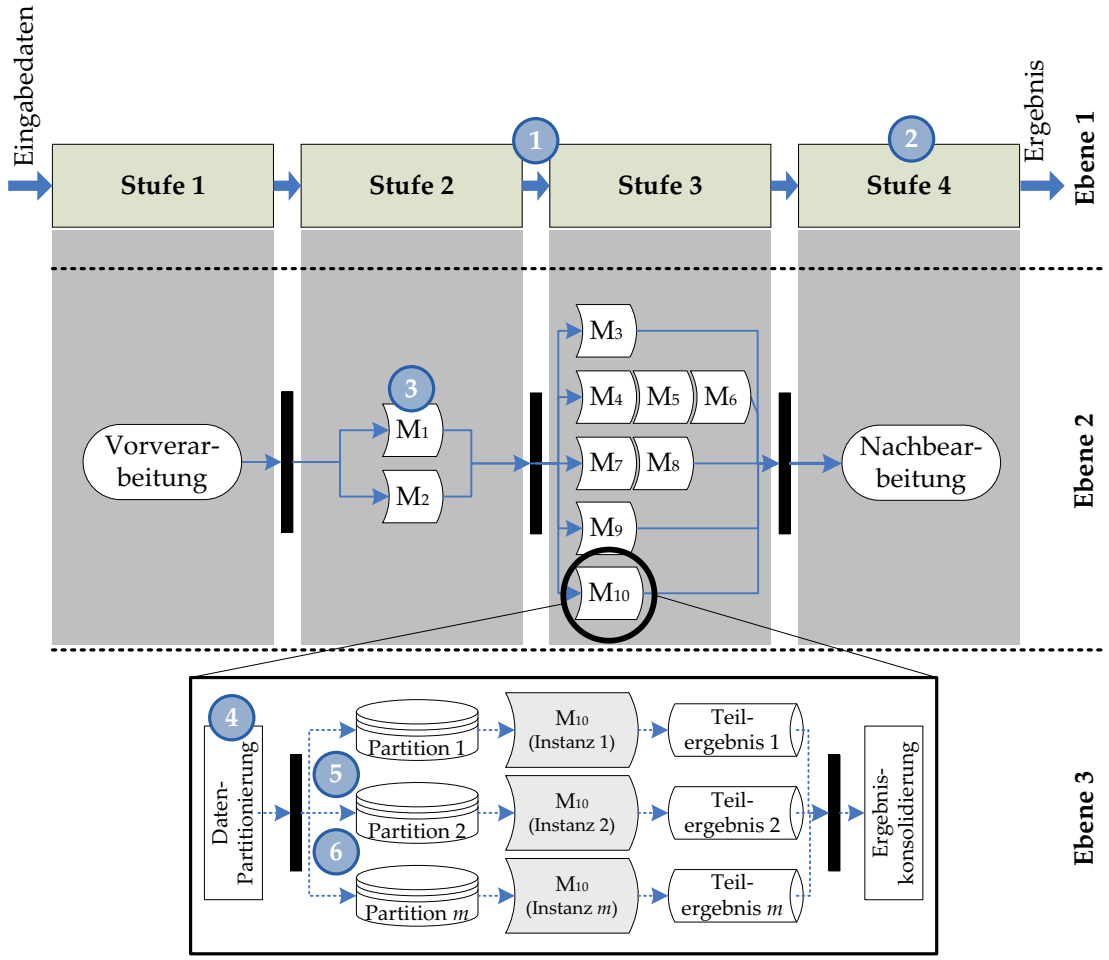


Abbildung 5.9: Darstellung eines Programms mit mehrstufiger Parallelität und beispielhaft markierten Tuning-Parametern (aus [PSJT08]).

Das Beispiel verdeutlicht zwei zentrale Beobachtungen, auf die wir uns konzentrieren. Zum einen wird ersichtlich, dass innerhalb einer einzigen Applikation alle Typen von Parallelität auf unterschiedlichen Ebenen auftreten können (Fließband-, Aufgaben- und

Datenparallelität; vgl. Kapitel 3, Abschnitt 3.7.3.1). Zum anderen ist an den exemplarisch markierten Tuning-Parametern zu erkennen, welch große Zahl sich an Möglichkeiten zur Konfiguration der gesamten Programmarchitektur ergibt, um dessen Leistung zu beeinflussen:

(1) Wie viele Fließbandstufen werden auf der höchsten Abstraktionsebene benötigt?

(2) Können die Stufen selbst repliziert werden, um grobgranulare datenparallele Verarbeitung zu erreichen?

(3) Existieren von einem Algorithmus mehrere Implementierungen? Welche passt dann am besten in die Architektur und kann somit zu optimaler Leistung verhelfen?

(4) Wie groß dürfen die Datenpartitionen sein?

(5) Welche Lastausgleichsstrategie eignet sich am besten, um die einzelnen Datenpartitionen zuzuteilen?

(6) Wie viele Instanzen eines Algorithmus müssen erzeugt werden, um die Daten möglichst schnell parallel zu verarbeiten?

Es gilt zu berücksichtigen, dass sich die Antworten auf obige Fragen grundsätzlich auf eine konkrete Hardware-Plattform beziehen.

Obwohl die Kombination aus Parallelisierung und Optimierung einer Applikation eine schwierige Aufgabe darstellt, sind die meisten Software-Entwickler gezwungen, ihre Programme ohne einen strukturierten Architekturentwurf zu parallelisieren und anschließend von Hand zu optimieren. Dieses Vorgehen ist mühsam, kostenintensiv und führt meist zu Quelltext, der nicht automatisch portiert werden kann und somit auf anderen Plattformen erneut optimiert werden muss [WKTi00, WhPD01, TaCH02, QaKMC06, ScPT09].

Darüber hinaus haben Studien [PSJT08, PaJT09, Scha09] gezeigt, dass die Architektur eines parallelen Programms an sich bereits großen Einfluss auf die Leistung des Programms haben kann und eine Architekturadaption sich wesentlich stärker auf die Leistung auswirkt als die alleinige Optimierung auf Instruktionsebene. Derartige Adaptionen werden jedoch selten durchgeführt, da sie invasive Änderungen am Quelltext erfordern.

Diese Erkenntnis hat entscheidend zur Entwicklung von Atune-TA beigetragen. Der Ansatz erlaubt die einfache Konfiguration der Programmarchitektur und nutzt gleichzeitig architekturinhärente Parallelität. Der Software-Entwickler implementiert atomare Programmkomponenten, die ausführbaren Quelltext enthalten, jedoch kein weiteres Parallelisierungspotential bieten. Mit einer neuartigen *Beschreibungssprache für optimierbare Architekturen* wird anschließend definiert, auf welche Weise die atomaren Komponenten interagieren und parallel verarbeitet werden sollen. Die hierfür zur Verfügung stehenden Operatoren werden jeweils durch eine parallele Verarbeitungssemantik geprägt, wobei die konkrete Parallelisierung sowie die notwendige Adaptierbarkeit gemäß vordefinierter Parameter implizit durchgeführt wird.

Das Konzept der optimierbaren Architekturen beinhaltet insbesondere die folgenden Aspekte:

- **Zusammenführung von Parallelisierung und Optimierung.** Die implizite Berücksichtigung von Parallelität und Optimierbarkeit bereits beim Entwurf der Anwendung stellt den Kern des Konzepts dar. Der Software-Entwickler bestimmt, welche Aufgaben des Programms parallel verarbeitet werden können. Die Umsetzung der Parallelisierung sowie die Implementierung der Tuning-Parameter sind auf einander abgestimmt und geschehen im Verborgenen.

- **Einheitliche Entwurfs- und Implementierungs-Richtlinien.** Der Software-Entwickler wird an Hand konzeptioneller Vorgaben durch einen automatisierten Prozess geführt, dessen Ergebnis eine optimierbare parallele Anwendung darstellt. Hierbei unterstützt das Konzept nicht nur eine für die grobgranulare Parallelisierung geeignete Herangehensweise, sondern vermittelt auch eine Denkweise, die dem Entwickler grundsätzlich hilft, Parallelität zu entdecken und auszudrücken.

- **Modularer Entwurf.** Das Konzept bietet eine modulare Entwurfstechnik, die es ermöglicht, einzelne Architekturelemente oder vollständige Architekturalternativen auszutauschen und somit als Optimierungsoption zu deklarieren.

- **Automatisierte Umsetzung des Entwurfs.** Der Architekturentwurf des parallelen Programms wird in einem automatisierten Prozess auf ein lauffähiges Programm abgebildet.

- **Unterstützung für automatische Performanzoptimierung.** Durch die Integration von *Atune-IL* (vgl. Abschnitt 5.2) zur Deklaration von Tuning-Instruktionen können optimierbare Architekturen von einem Auto-Tuner optimiert werden.

Im Folgenden wird als zentraler konzeptioneller Baustein die Beschreibungssprache für optimierbare Architekturen eingeführt und deren Anwendung an Hand konkreter Beispiele erklärt. Die Sprache sowie das Konzept der optimierbaren Architekturen konnten bereits veröffentlicht werden [ScPT10].Die dort vorgestellten Quelltextbeispiele werden daher auch in dieser Arbeit verwendet.

5.3.2 Beschreibungssprache für optimierbare Architekturen

Atune-TA basiert auf der neuartigen Beschreibungssprache *TADL* (*Tunable Architecture Description Language*). Mit Hilfe dieser Sprache können alle Architekturvarianten eines parallelen Programms beschrieben werden, die für die spätere Optimierung relevant sind. Die formale Grammatik der Sprache ist im Anhang A.2 zu finden.

Eine Architekturbeschreibung wird in Form eines so genannten TADL-Skriptes entworfen. Ein TADL-Übersetzer transformiert das TADL-Skript in ausführbaren und optimierbaren Quelltext. Der Übersetzer ist Teil der prototypischen Implementierung und wird daher im nächsten Kapitel vorgestellt.

Im Wesentlichen setzt sich TADL aus zwei Sprachkonstrukten zusammen. Die *atomaren Komponenten* stellen die Operanden der Sprache dar und definieren ausführbare sequentielle Programmabschnitte. Die Gruppe der Operatoren bilden die *Konnektoren*, die – angewendet auf atomare Komponenten – deren parallele Verarbeitung definieren. Die jeweilige Verarbeitungssemantik orientiert sich an bekannten parallelen Entwurfsmustern, wie Fließband-, Erzeuger/Verbraucher- oder Fork-Join-Muster (siehe Kapitel 3, Abschnitt 3.7.3.1). Das leistungsrelevante Verhalten der Konnektoren wird durch vordefinierte Tuning-Parameter beeinflusst, die bei der späteren Optimierung berücksichtigt werden.

5.3.2.1 Verwandschaft zu Architekturbeschreibungssprachen

TADL stellt keine Architekturbeschreibungssprache (engl. *Architecture Description Language (ADL)*) im klassischen Sinn dar, wobei man in der Literatur vergeblich nach einer einheitlichen Definition für Architekturbeschreibungssprachen sucht. Um TADL dennoch konzeptionell einzuordnen, ziehen wir die Studie von Medvidovic und Taylor heran, die bereits im Kapitel über die verwandten Arbeiten zitiert wurde [MeTa00]. Danach besitzt eine Architekturbeschreibungssprache die folgenden Modellierungskonstrukte:

- **Komponenten (*engl. components*).** Eine Komponente ist ein Architekturelement, das eine Berechnung durchführt oder Daten speichert:

  "Components are loci of computation and state." [MeTa00]

  Hierbei muss eine Komponente eine definierte Schnittstelle zur Verfügung stellen, über die Daten ausgetauscht werden können.

  Eine Komponente kann entweder eigene Daten sowie einen separaten Ausführungsbereich erfordern, oder beides mit anderen Komponenten teilen. Eine Komponente wird grundsätzlich auf ein Element im implementierten Programm abgebildet.

- **Konnektoren (*engl. connectors*).** Konnektoren repräsentieren die Bausteine der Architektur, indem sie die Interaktionen zwischen Komponenten modellieren und Regeln definieren, die die Interaktionen steuern. Im Gegensatz zu Komponenten müssen Konnektoren nicht unbedingt einem Element im implementierten Programm entsprechen.

- **Konfigurationen (*engl. configurations*).** Architekturkonfigurationen (auch Topologien genannt) sind Graphen aus Komponenten und Konnektoren, welche die architektonische Struktur beschreiben. Mit Hilfe dieser Information kann beispielsweise ermittelt werden, ob Komponenten mit geeigneten Konnektoren verbunden sind, ob die Schnittstellen der verbundenen Komponenten zueinander passen oder ob die Semantik der kombinierten Komponenten dem gewünschten Verhalten entspricht.

Die atomaren Komponenten von TADL entsprechen in ihrer Semantik der Definition von Medvidovic und Taylor. Die Definition der TADL-Konnektoren weicht hingegen etwas von der gebräuchlichen Konnektor-Definition in Architekturbeschreibungssprachen ab. Während Konnektoren üblicherweise die Kommunikation zwischen genau zwei Komponenten steuern, definiert ein TADL-Konnektor eine vollständige Verarbeitungsstrategie sowie die dazugehörigen Interaktionen für mehrere Komponenten sowie untergeordnete TADL-Konnektoren.

Des Weiteren wird die explizite Modellierung von Architekturkonfigurationen von TADL nicht unterstützt. Die Gründe hierfür liegen in der Zielsetzung, die Sprache möglichst kompakt und einfach zu halten. Da TADL ausschließlich zur Beschreibung paralleler Verarbeitung von Komponenten entwickelt wurde, kann durch eine klare Semantik der Konnektoren sowie durch einen einheitlichen Typ atomarer Komponenten zunächst auf explizite Konfigurationen verzichtet werden.

5.3.2.2 Atomare Komponenten

Eine *atomare Komponente* bezeichnet einen elementaren Arbeitsschritt, dessen Instruktionen sequentiell ablaufen. Atomare Komponenten besitzen also keine interne Parallelität, sind aber für die parallele Ausführung mit anderen atomaren Komponenten vorgesehen. Atomare Komponenten stellen demnach die kleinsten sequentiellen Einheiten des parallelen Programms dar und bilden daher die Grundlage einer optimierbaren Architektur.

Eine atomare Komponente wird durch eine Methode des Programms implementiert. Eine typische Aufgabe der assoziierten Methode kann beispielsweise die Verarbeitung eines einzelnen Datenelements sein.

Die Deklaration einer atomaren Komponente in TADL besteht aus dem Namen der assoziierten Methode im Quelltext des Programms:

```
MeinMethodenName
```

Die Assoziation von Bezeichner einer atomaren Komponente in TADL und implementierender Methode wird über den Namen der Methode hergestellt. Der TADL-Übersetzer sucht mittels Reflexion nach der entsprechenden öffentlichen Methodendeklaration im benutzerdefinierten Namensraum des Programms (hierzu mehr in Kapitel 6). Ist die Methode überladen, wird der Entwickler vor dem Übersetzungsvorgang des TADL-Skriptes gefragt, welche Methodenimplementierung als atomare Komponente verwendet werden soll. Gleiches gilt für den Fall, dass die Methode mit demselben Namen in unterschiedlichen Klassen deklariert wurde.

Die mit einer atomaren Komponente assoziierte Methode kann beliebigen Quelltext der verwendeten Programmiersprache enthalten. Es ist jedoch zu beachten, dass die grundsätzliche Intention atomarer Komponenten die parallele Ausführung derselben vorsieht. Es liegt daher in der Verantwortung des Entwicklers, korrekten Quelltext zu produzieren, der fehlerfrei ausgeführt werden kann. Besondere Aufmerksam gilt hierbei Datenstrukturen, die von zwei oder mehreren atomaren Komponenten gemeinsam genutzt werden. Hier sollten Vorkehrungen zur Synchronisation getroffen werden, um das gewünschte Verhalten des Programms zu gewährleisten.

Auf Grund der methodenbasierten Struktur bieten sich zur Identifikation von Fehlverhalten Modultests an, die alle Methoden, die atomare Komponenten implementieren, auf ihre korrekte Funktion hin überprüfen.

Das Konzept der atomaren Komponente lässt sich wie folgt zusammenfassen. Atomare Komponenten bilden die *Operanden* der Architekturbeschreibung und werden auf Elemente funktionaler Logik des Programms in Form von implementierten Methoden abgebildet. Jede atomare Komponente stellt daher eine essentielle Aufgabe des Programms dar, die parallel zu anderen Aufgaben ausgeführt werden kann. Ein Entwickler ist nicht mehr gezwungen, in Ausführungsfäden zu denken, sondern in einzelnen Arbeitsvorgängen, wodurch wesentlich intuitiver beschrieben werden kann, *was* parallelisiert werden soll.

Im nächsten Abschnitt werden die Operatoren behandelt, welche die Strategie zur parallelen Ausführung atomarer Komponenten definieren und diese somit zu einer parallelen Sektion verbinden.

Anmerkung: Wird ein Programm von Grund auf neu geschrieben, so müssen zunächst alle Methoden geschrieben werden, die atomare Komponenten implementieren. Zumindest muss eine Schnittstelle existieren, welche die Methodensignaturen definiert.

5.3.2.3 Konnektoren

TADL-Konnektoren bilden die Bindeglieder zwischen atomaren Komponenten. Folglich stellen die TADL-Konnektoren Operatoren auf atomaren Komponenten oder geschachtelten Konnektoren (zusammengefasst als *Kind-Elemente*) dar.

Durch die beliebige Schachtelung von Konnektoren entspricht eine optimierbare Architektur einer Baumstruktur, wobei die Knoten durch Konnektoren und die Blätter durch atomare Komponenten repräsentiert werden. Abbildung 5.10 skizziert die Struktur einer beispielhaften Architektur, bestehend aus geschachtelten Konnektoren und atomaren Komponenten. Die Wurzel der Baumstruktur bezeichnen wir als *Wurzel-Konnektor*.

Ein Konnektor definiert eine Verarbeitungsstrategie sowie die hierfür notwendigen Interaktionen für seine Kind-Elemente.

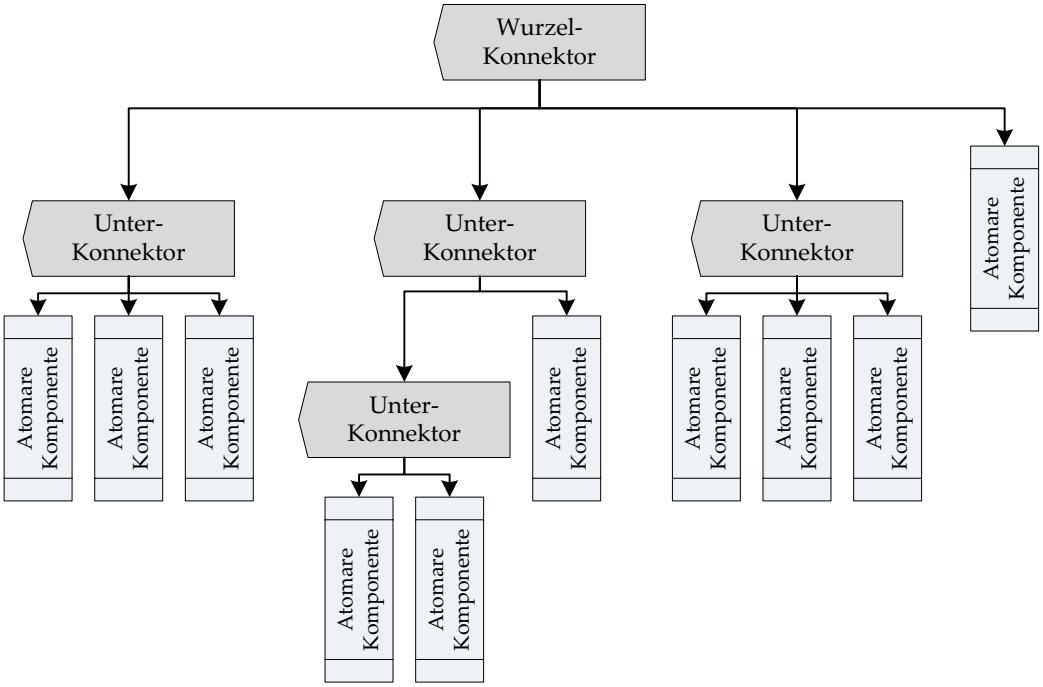


Abbildung 5.10: Exemplarische Darstellung einer optimierbaren Architektur aus geschachtelten Konnektoren und atomaren Komponenten.

In Kapitel 3 wurde dargelegt, dass parallele Entwurfsmuster großes Optimierungspotential bieten können und sich daher der Einsatz parametrisierter paralleler Muster empfiehlt. Wir greifen dieses Konzept auf und wenden es auf die TADL-Konnektoren an. Die Verarbeitungsstrategien der Konnektoren beeinflussen wesentlich die Leistung des Programms und stellen somit leistungskritischen Elemente der Architektur dar. Nahezu alle TADL-Konnektoren besitzen daher implizite Tuning-Parameter, die die Leistung des implementierten Musters beeinflussen können. Um von der eigentlichen Optimierung zu abstrahieren, müssen diese Parameter nicht explizit in TADL definiert werden; die Parameter werden vielmehr bei der Umsetzung des TADL-Skripts in eine Implementierung der Architektur automatisch integriert.

Tabelle 5.2 listet alle vordefinierten Tuning-Parameter der optimierbaren Architekturen auf – gruppiert nach Konnektoren.

Tuning-Parameter	**Beschreibung**
Tunable Alternative	
`SelectAlternative`	Bestimmt, welche Alternative ausgeführt wird
Tunable Fork/Join	
`NumThreads`	Anzahl an Arbeiterfäden, die der Konnektor verwenden kann.
Tunable Producer/Consumer	
`BufferSize`	Größe des zentralen Puffers zwischen Erzeuger und Verbraucher.

Tuning-Parameter	**Beschreibung**
`BatchSize`	Anzahl an Datenelementen, die der Verbraucher auf einmal aus dem zentralen Puffer liest.
Tunable Pipeline	
`StageFusion`	Schaltet für jede Fließbandstufe die Fusionierung (engl. *stage fusion*) mit der nachfolgenden Stufe ein oder aus.
Tunable Replication	
`NumInstances`	Anzahl replizierter Instanzen der atomaren Komponente.
`LoadBalancing`	Strategie, mit der die Datenelemente den Instanzen zugewiesen werden (*statisch*, *dynamisch*).
`BatchSize`	Anzahl an Datenelementen, die einer Instanz auf einmal zugeteilt werden (auch *Blockgröße* genannt). Abhängigkeit zu Parameter `LoadBalancing`, da die Blockgröße nur bei statischer Lastausgleichsstrategie eingestellt werden kann.

Tabelle 5.2: Liste aller implizit integrierten Tuning-Parameter der optimierbaren Architekturen.

Die tabellarische Übersicht über die impliziten Tuning-Parameter der optimierbaren Architekturen dient an dieser Stelle lediglich dem Verständnis, da TADL von konkreten Tuning-Parametern abstrahiert. Im Kapitel 6 wird schließlich die automatisierte Implementierung der Tuning-Parameter behandelt.

Im Folgenden werden die TADL-Konnektoren im Einzelnen vorgestellt.

Sequential Composition

Mit dem *Sequential Composition*-Konnektor steht für den Architekturentwurf ein universeller Baustein zur Verfügung, der eine sequentielle Ausführungssemantik für zwei oder mehrere Kind-Elemente bietet. Der Sequential Composition-Konnektor beschreibt somit Programmteile, die nicht parallel ausgeführt werden sollen. Daher eignet sich dieser Konnektor auch als Überbrückung zwischen mehreren parallelen Programmteilen. Alle Kind-Elemente müssen kompatible Ein- und Ausgabedatentypen besitzen, um sequentiell verarbeitet werden zu können.

Da bei einer sequentiellen Hintereinanderschaltung der Kind-Elemente in der Regel kein Optimierungspotential besteht, besitzt der Sequential Composition-Konnektor als einziger der TADL-Konnektoren keine impliziten Tuning-Parameter.

Tunable Alternative

Der *Tunable Alternative*-Konnektor drückt eine exklusive Auswahl zwischen zwei oder mehreren Kind-Elementen aus. Bei der späteren automatischen Optimierung der Archi-

tektur wird in jeder Tuning-Iteration eine bestimmte Alternative gewählt, mit der das Programm ausgeführt wird.

Es ist anzumerken, dass der Tunable Alternative-Konnektor ein sehr mächtiges Konstrukt darstellt. Da der Konnektor auf jeder Ebene der architektonischen Baumstruktur eingesetzt werden kann, bietet er die Möglichkeit, atomare Komponenten, untergeordnete Konnektoren (also einzelne Verarbeitungsstrategien), oder aber ganze (Teil-)Architekturen dynamisch auszutauschen. Gerade aus Sicht der Optimierung stellt diese Funktionalität enormes Potential dar.

Tunable Fork/Join

Der *Tunable Fork/Join*-Konnektor ermöglicht die Beschreibung von Aufgabenparallelität. Die Semantik entspricht der des Fork/Join-Musters, das in Kapitel 3, Abschnitt 3.7.3.1 erklärt wurde. Der zunächst sequentielle Kontrollfluss gabelt sich (*fork*), um alle Kind-Elemente parallel auszuführen. Nachdem alle Aufgaben beendet wurden, wird der Kontrollfluss wieder zusammengeführt (*join*). Da die Kind-Elemente des Fork/Join-Konnektors gemäß der Semantik nicht direkt miteinander interagieren, existieren keine Einschränkungen bzgl. deren Ein- und Ausgabedatentypen.

Tunable Producer/Consumer

Der *Tunable Producer/Consumer*-Konnektor besitzt die Semantik des Erzeuger/Verbraucher-Musters (siehe Kapitel 3, Abschnitt 3.7.3.1) und erwartet daher exakt zwei Kind-Elemente: den *Erzeuger* und den *Verbraucher*.

Dieser Konnektor besitzt eine datenstromorientierte Semantik: Der Erzeuger erhält Datenelemente von einer enumerierbaren Datenstruktur. Nachdem er ein Element verarbeitet hat, gibt er dieses sofort an den zentralen Puffer weiter, aus dem der Verbraucher es dann zur weiteren Verarbeitung wieder herausnimmt. Diese Semantik erfordert kompatible Datentypen für die Ausgabe des Erzeugers und die Eingabe des Verbrauchers.

Obwohl das Erzeuger/Verbraucher-Muster auch als zweistufiges Fließband aufgefasst und entsprechend umgesetzt werden kann, bietet TADL hierfür dennoch einen separaten Konnektor, da beim Erzeuger/Verbraucher-Musters eher die Synchronisierung als die Fließbandverarbeitung im Vordergrund steht. Die dynamische Fusion von Erzeuger und Verbraucher durch Entfernen des Puffers als Möglichkeit zur Performanzoptimierung ist aber in Anbetracht der Verwendung des Konnektors als Synchronisierungsmechanismus nicht vorgesehen.

Tunable Pipeline

Der *Tunable Pipeline*-Konnektor stellt ein Konstrukt zur Beschreibung von Fließbandparallelität dar; die Semantik entspricht dem Fließband-Muster (siehe ebenfalls Kapitel 3, Abschnitt 3.7.3.1). Der Konnektor kann zwei oder mehr Kind-Elemente verarbeiten, die jeweils einer Fließbandstufe entsprechen.

Wie der Tunable Producer/Consumer-Konnektor besitzt auch der Tunable Pipeline-Konnektor eine datenstromorientierte Semantik, die durch das Konzept des Fließbandes vorgegeben ist. Die erste Stufe des Fließbandes erhält die Eingabedaten von einer enumerierbaren Datenquelle, die letzte Stufe übergibt alle verarbeiteten Datenelemente der Reihe nach einer Datensenke. Eine Stufe, die ein Datenelement verarbeitet hat, übergibt dieses der nächsten Stufe.

Tunable Replication

Der *Tunable Replication*-Konnektor nimmt eine Sonderrolle ein, da dieser nicht auf mehrere Kind-Elemente, sondern ausschließlich auf genau eine atomare Komponente angewendet werden kann und somit einen unären Operator darstellt. Der Tunable Replication-Konnektor kann nicht auf einen anderen Konnektor angewendet werden.

Die Anwendung des Tunable Replication-Konnektors auf eine atomare Komponente bewirkt deren Replizierung. Bei der Umsetzung der Architektur werden mehrere Instanzen der atomaren Komponente erzeugt, so dass die anfallenden Daten von den verfügbaren Instanzen parallel verarbeitet werden können. Der Tunable Replication-Konnektor stellt somit ein Konstrukt zur Beschreibung von Datenparallelität dar und besitzt die Semantik des Musters der Gebietszerlegung (vgl. Kapitel 3, Abschnitt 3.7.3.1).

Um eine atomare Komponente als replizierbar markieren zu können, muss sichergestellt sein, dass die von der Komponente durchgeführte Berechnung zustandslos ist (d.h. die Verarbeitung eines Datenelements muss unabhängig von der Verarbeitung eines anderen sein).

Syntax für Konnektoren

Alle TADL-Konnektoren mit Ausnahme des Tunable Replication-Konnektors werden mit derselben Syntax spezifiziert, die an Hand eines Tunable Pipeline-Konnektors mit zwei Stufen exemplarisch erläutert wird:

```
TunablePipeline MeinFliessband {
    TunableAlternative {
        Berechne1,
        Berechne1a
    },
    Berechnung2
}
```

Die erste Stufe des Fließbandes beinhaltet einen geschachtelten Konnektor in Form einer Alternative (Tunable Alternative-Konnektor) aus den atomaren Komponenten `Berechne1` und `Berechne1a`, die zweite Stufe besteht aus der atomaren Komponente `Berechne2`. Die Baumstruktur der Architektur wird durch Anweisungsblöcke in geschweiften Klammern ausgedrückt. Ein Konnektor kann mit einem optionalen Bezeichner versehen werden. Wird kein Bezeichner angegeben, so verwendet der TADL-Übersetzer bei der Implementierung der Architektur einen generierten Bezeichner.

Die Alternative im obigen Beispiel bewirkt nun, dass bei der späteren automatischen Optimierung zwei architektonische Varianten des Fließbandes getestet werden: Zunächst beinhaltet die erste Stufe `Berechne1`, in einem zweiten Programmlauf `Berechne1a`. Die zweite Stufe bleibt davon unberührt und enthält stets `Berechne2`.

Die Syntax des Tunable Replication-Konnektors weicht von der oben erwähnten Syntax ab. Der Tunable Replication-Konnektor wird mit dem Attribut `replicable` auf eine atomare Komponente angewendet. Im Folgenden wurde das obige Beispiel nochmals aufgegriffen, wobei nun alle atomaren Komponenten durch den Tunable Replication-Konnektor als replizierbar markiert wurden:

```
TunablePipeline MeinFliessband {
    TunableAlternative {
        Berechne1[replicable],
        Berechne1a[replicable]
    },
    Berechnung2[replicable]
}
```

5.3.2.4 Behandlung von Ein- und Ausgabedaten

Der Datenfluss in einen Konnektor und aus einem Konnektor heraus wird über das Datenflussattribut des Konnektors gesteuert. Mit Hilfe dieses Attributs kann beschrieben werden, wie die Kind-Elemente des Konnektors mit Daten versorgt werden und auf welche Weise die Daten zurückgegeben werden. Der Datenfluss innerhalb des Konnektors bleibt davon unberührt, da dieser durch die Semantik der jeweiligen Verarbeitungsstrategie definiert wird.

Im Datenflussattribut eines Konnektors werden so genannte Ein- und Ausgabekomponenten definiert. Eine Eingabekomponente stellt Daten zur Verfügung (beispielsweise aus einer Datenquelle oder von der Kommandozeile), eine Ausgabekomponente nimmt Daten zur Weiterverarbeitung entgegen (z.B. zur Speicherung auf der Festplatte, zur Ausgabe auf dem Bildschirm oder zur Aktualisierung einer Datenstruktur). Die Semantik von Ein- und Ausgabekomponenten entspricht im Wesentlichen der atomarer Komponenten und werden daher mit einer Methode im implementierten Programm assoziiert, die Eingabedaten bereitstellt bzw. Ausgabedaten entgegennimmt.

Die Definition der Ein- und Ausgabekomponenten sowie die Assoziation mit den Kind-Elementen des Konnektors ergibt sich aus den Datenflussattributen der Kind-Konnektoren. Weist ein Konnektor beispielsweise für all seine Kind-Elemente entsprechende Definitionen für Ein- und Ausgabekomponenten auf, so müssen im übergeordneten Konnektor keine Ein- und Ausgabekomponente für den geschachtelten Konnektor definiert werden.

Ein- und Ausgabe bei nicht-datenstromorientierten Konnektoren

Die nicht-datenstromorientierten Konnektoren (Sequential Composition-, Tunable Alternative- und Tunable Fork/Join-Konnektor) erwarten für alle Kind-Elemente die Definition jeweils einer Ein- und Ausgabekomponente. Hierfür werden die Schlüsselwörter `input` und `output` verwendet. Die Bezeichner der Ein- und Ausgabekomponenten eines Konnektors werden im Datenflussattribut in Form einer Liste als Argumente für `input` bzw. `output` angegeben. Der erste Bezeichner der Liste entspricht der Ein- bzw. Ausgabekomponente des ersten Kind-Elements, der zweite Bezeichner der des zweiten Elements usw.

Das folgende TADL-Beispiel veranschaulicht den Sachverhalt. `Berechne1` erhält die zu verarbeitenden Daten von `EingabeMethode1` und übergibt die Ergebnisse der Berechnung an `AusgabeMethode1` zur weiteren Verarbeitung. Für `Berechne2` gilt der analoge Fall.

```
TunableForkJoin MeinForkJoinKonnektor {
    [input:EingabeMethode1,Eingabemethode2;
     output:AusgabeMethode1,Ausgabemethode2]
    Berechne1,
    Berechne2
}
```

Der Rückgabetyp der Eingabekomponente muss jeweils kompatibel zum Eingabetyp des entsprechenden Kind-Elements sein. Gleiches gilt in umgekehrter Weise für die Ausgabekomponente.

Sei beispielsweise die Methode, die die atomare Komponente `Berechne1` implementiert, wie folgt definiert:

```
public float Berechne1(int i) {...}
```

Dann sind die folgenden Methoden gültige Implementierungen für Ein- und Ausgabekomponenten:

```
public int EingabeMethode1() {...}
public void AusgabeMethode1(float f) {...}
```

Eine Ausnahme bilden atomare Komponenten als Kind-Elemente, die mittels des Tunable Replication-Konnektors als replizierbar markiert wurden. In diesem Fall muss die zugehörige Eingabekomponente eine enumerierbare Datenstruktur zurückgeben, deren Elementtyp mit dem Eingabetyp der atomaren Komponente kompatibel ist. Wiederum gilt das Gleiche für die entsprechende Ausgabekomponente.

Zum Verständnis betrachten wir nochmals das obige Beispiel und nehmen zusätzlich an, dass `Berechne1` als replizierbar markiert ist:

```
TunableForkJoin MeinForkJoinKonnektor {
    [input:EingabeMethode1,Eingabemethode2;
     output:AusgabeMethode1,Ausgabemethode2]
    Berechne1[replicable],
    Berechne2
}
```

Somit stellen die folgenden Methoden gültige Implementierungen der Ein- und Ausgabekomponenten für `Berechne1` dar:

```
public List<int> EingabeMethode1() {...}
public void AusgabeMethode1(List<float> fList) {...}
```

Ein- und Ausgabe bei datenstromorientierten Konnektoren

Tunable Pipeline- und Tunable Producer/Consumer-Konnektoren besitzen auf Grund ihrer datenstromorientierten Semantik nur eine einzige Datenquelle bzw. -senke, weshalb für diese Konnektoren nur eine Ein- und Ausgabekomponenten benötigt wird. Entsprechend stellt TADL zwei spezielle Schlüsselwörter zur Verfügung. Im Datenflussattribut der beiden datenstromorientierten Konnektoren werden anstelle von `input` und `output` die Schlüsselwörter `source` und `sink` verwendet, die jeweils nur genau eine Ein- bzw. Ausgabekomponente als Argument erwarten.

Das folgende TADL-Beispiel verdeutlicht die Verwendung des Datenflussattributs bei datenstromorientierten Konnektoren:

```
TunablePipeline MeinFliessband {
    [source:EingabeMethode;
     sink:AusgabeMethode]
    Berechnung1
    Berechnung2
}
```

Das Datenflussattribut definiert als Quelle die Eingabekomponente `EingabeMethode` und als Senke die Ausgabekomponente `AusgabeMethode`.

Da die datenstromorientierten Konnektoren auf eine enumerierbare Datenquelle angewiesen sind, muss die implementierende Methode der Eingabekomponente eine entsprechende Datenstruktur zurückliefern, deren Elementtyp mit dem Eingabetyp des ersten Kind-Elements kompatibel ist. Die implementierende Methode der Ausgabekomponente muss wiederum eine enumerierbare Datenstruktur als Senke entgegennehmen, deren Elementtyp mit dem Rückgabetyp des letzten Kind-Elements des Konnektors kompatibel ist.

Für obiges TADL-Beispiel seien die Methoden, die die atomaren Komponenten `Berechne1` und `Berechne2` implementieren, wie folgt definiert:

```
public float Berechne1(int i) {...}
public float Berechne2(float f) {...}
```

Die folgenden Methoden sind dann gültige Implementierungen für die Ein- und Ausgabekomponente des beispielhaften Tunable Pipeline-Konnektors:

```
public List<int> EingabeMethode1() {...}
public void AusgabeMethode1(List<float> fList) {...}
```

Die erste Stufe des Fließbandes (`Berechne1`) erhält nun von der Quelle `EingabeMethode` eine enumerierbare Liste mit Datenelementen vom Type `integer`, die elementweise abgearbeitet wird. Des Weiteren übergibt die letzte Stufe des Fließbandes (`Berechne2`) jedes verarbeitete Datenelement an die Senke des Fließbandes (`AusgabeMethode`).

Anmerkung: Bedarf ein Kind-Element keiner Ein- bzw. Ausgabe, so wird dies durch das `null`-Schlüsselwort an entsprechender Stelle im Datenflussattribut markiert. Werden für einen Konnektor überhaupt keine Ein- oder Ausgabekomponenten benötigt, so kann das Datenflussattribut vollständig wegfallen.

5.3.3 Integration von *Atune-IL*

Wie eingangs dieses Kapitels erwähnt, werden die in TADL beschriebenen parallelen Architekturen vom TADL-Übersetzer in ausführbaren Quelltext transformiert. Diese Programmteile sind zwar parametrisiert und daher adaptierbar, jedoch nicht automatisch optimierbar. Es fehlen Tuning-Instruktionen, über die ein Auto-Tuning-Werkzeug an die Architektur angebunden werden kann, um Parameter-Eigenschaften auszulesen, Parameterwerte zu setzen sowie Messdaten zu sammeln.

Doch genau diesen Zweck erfüllt die Tuning-Instrumentierungs-Sprache Atune-IL. Bei der automatisierten Umsetzung eine Architektur erzeugt der TADL-Übersetzer daher neben ausführbarem Quelltext auch entsprechende Atune-IL-Anweisungen, die den Quelltext um alle nötigen Tuning-Instruktionen ergänzen. Darüber hinaus besitzt der TADL-Übersetzer ein umfangreiches Kontextwissen über die gerade zu verarbeitenden Konnektoren und die damit einhergehenden Parallelisierungsstrategien, das eine kontextbezogene Generierung der Tuning-Instruktionen erlaubt. Das erzeugte parallele und instrumentierte Programm kann auf diese Weise von einem Auto-Tuning-Werkzeug effizient optimiert werden.

Neben der automatisierten Generierung von Atune-IL-Anweisungen kann der Software-Entwickler bei Bedarf auch manuell weitere Instrumentierungen in den Quelltext integrieren. Diese werden dann ebenfalls beim Optimierungsprozess berücksichtigt.

Die Funktionsweise des TADL-Übersetzers und damit die Generierung von Atune-IL-Anweisungen ist der Umsetzung der Konzepte dieser Arbeit zugeordnet, weshalb die entsprechenden Abläufe und Zusammenhänge in Kapitel 6 ausführlich diskutiert werden.

5.3.4 Beispiele optimierbarer Architekturen

Das Konzept der Optimierbaren Architekturen bietet nicht zuletzt auf Grund der Beschreibungssprache TADL ein breites Einsatzspektrum sowie eine umfangreiche Menge an optimierbaren Architekturvarianten für parallele Applikationen.

Im Folgenden wird daher an Hand zweier Beispiele gezeigt, wie parallele optimierbare Architekturen von Programmen aus der realen Welt in einem Top-Down-Ansatz entworfen werden. Beide Beispiele sind der genannten Veröffentlichung entliehen [ScPT10].

5.3.4.1 Entwurf einer parallelen Anwendung zur Videoverarbeitung

Das erste Beispiel ist übersichtlich und befasst sich mit einer Anwendung zur Verarbeitung von Videos. Das Programm wendet eine Reihe von Filtern an, um die einzelnen Bilder (engl. *frames*) eines Videos zu bearbeiten. Offensichtlich eignet sich dieser Anwendungsfall, um die einzelnen Filter in Form eines Fließbandes zu verbinden und auf diese Weise die Parallelität auszunutzen.

Wir beginnen jedoch zunächst mit der Implementierung der Methoden für die Filter sowie für das Laden und Ausgeben des Videos. Listing 5.2 zeigt die entsprechenden Methodensignaturen. `LoadVideo()` lädt das Video von der Festplatte, die Filtermethoden `Crop(),Oilpaint(),Resize()` und `Sharpen()` implementieren jeweils einen Algorithmus zur Bearbeitung eines Bildes des Videos. `ConsumeVideo()` zeigt das modifizierte Video schließlich auf dem Bildschirm an.

```
public IEnumerable<Bitmap> LoadVideo() {...}
public Bitmap Crop(Bitmap bmp) {...}
public Bitmap OilPaint(Bitmap bmp) {...}
public Bitmap Resize(Bitmap bmp) {...}
public Bitmap Sharpen(Bitmap bmp) {...}
public void ConsumeVideo(IEnumerable<Bitmap> bmps) {...}
```

Listing 5.2: Methoden atomarer Komponenten des Videoverarbeitungsprogramms (aus [ScPT10]).

Die Filtermethoden repräsentieren die Implementierungen der atomaren Komponenten, `LoadVideo()` die Eingabekomponente und `ConsumeVideo()` die Ausgabekomponente.

Es ist zu beachten, dass alle Filtermethoden denselben Ein- und Ausgabedatentyp besitzen. Des Weiteren erwartet `LoadVideo()` als Implementierung der Eingabekomponente keinen Parameter, sondern gibt nur eine enumerierbare Datenstruktur zurück. Für `ConsumeVideo()` als Implementierung der Ausgabekomponente gilt der umgekehrte Fall.

Es wird deutlich, dass eine korrekte und geeignete Implementierung der Komponenten äußerst relevant ist und die Grundlage der optimierbaren Architektur bildet.

Im nächsten Schritt kann die gewünschte Architektur mit Hilfe eines TADL-Skripts beschrieben werden, welches in Listing 5.3 aufgeführt ist.

Zur Verbindung der atomaren Komponenten haben wir den Tunable Pipeline-Konnektor gewählt, da die Verarbeitung eines Videos am besten durch ein Fließband beschrieben werden kann. Die vier Stufen des Fließbandes repräsentieren also die vier Filtermethoden. Ferner wurde als Quelle und Senke des Fließbandes `LoadVideo` bzw. `ConsumeVideo` definiert.

```
TunablePipeline MeineVideoVerarbeitung
    [source:LoadVideo;sink:ConsumeVideo] {
    Crop[replicable],
    OilPaint[replicable],
    Resize[replicable],
    Sharpen[replicable]
}
```

Listing 5.3: Architekturbeschreibung des parallelen Videoverarbeitungsprogramms (aus [ScPT10])

Da alle Filtermethoden zustandslos sind (d.h., die Verarbeitung eines Bildes ist unabhängig von der Verarbeitung eines anderen), haben wir die entsprechenden atomaren Komponenten als replizierbar markiert.

Damit ist der Entwurf der parallelen optimierbaren Architektur des Videoverarbeitungsprogramms vollständig. Durch ein 6-zeiliges TADL-Skript wurde eine Architektur beschrieben, die sowohl Fließband- als auch Datenparallelität auf zwei Anwendungsebenen ausnutzt und die automatisiert implementiert und optimiert werden kann.

5.3.4.2 Entwurf einer Parallelen Desktopsuche

Das zweite Beispiel fällt etwas umfangreicher aus und soll die architektonischen Variationsmöglichkeiten aufzeigen, die mit TADL beschrieben werden können.

Zu diesem Zweck entwerfen wir ein paralleles Programm zur Desktopsuche. Das Programm durchsucht zunächst das Dateisystem auf der Festplatte nach indizierbaren Dokumenten. Anschließend erstellt das Programm eine invertierte Indexdatenstruktur, indem es alle Wörter in den Dokumenten den jeweiligen Dateien zuordnet. Ein Benutzer kann schließlich eine aus einzelnen Wörtern bestehende Suchanfragen absetzen, worauf das Programm eine Liste mit Dokumenten zurückgibt, in denen die Wörter vorkommen.

Auch hier wird mit der Implementierung der Methoden begonnen, die wir später als atomare Komponenten definieren. Listing 5.4 zeigt die entsprechenden Signaturen.

```
public List<string> Crawl() {...}
public SearchResult StringSearch1(string path) {...}
public SearchResult StringSearch2(string path) {...}
public void UpdateIndex(SearchResult pr) {...}
public void CreateIndexFile() {...}
public List<string> Query(string[] keywords) {...}
public string[] GetKeywords() {...}
public void ShowResults(List<string> results) {...}
```

Listing 5.4: Methoden atomarer Komponenten der Desktopsuche (aus [ScPT10]).

Auf Grund der größeren Zahl an benötigten Methoden werden die atomaren Komponenten im Einzelnen erklärt:

- `Crawl`: Durchsucht das Dateisystem nach indizierbaren Dokumenten und liefert eine Liste mit Dateipfaden.
- `AStringSearch1`: Zerlegt unter Verwendung von Delimiterzeichen einen Text in einzelne Wörter. Die so gewonnenen Wörter werden als Schlüsselworte zum Aufbau des Indexes verwendet.
- `StringSearch2`: Alternative Implementierung des Suchalgorithmus auf Basis des String-Matching-Algorithmus von Knuth Morris und Pratt [CLRS01] (KMP-Algorithmus) . Der KMP-Algorthmus sucht in allen Dokumenten nach einer vorgegeben Menge an Schlüsselwörtern (engl. `white list`).

  Im Unterschied zum ersten Algorithmus, der zwar naiv vorgeht, aber wenig Overhead verursacht, wendet der KMP-Algorithmus eine geschicktere Suchstrategie an, die jedoch etwas mehr Overhead kostet (z.B. um die Wort-Indizes zu speichern).

  Da im Vorhinein bekannt ist, welcher Algorithmus in einer parallelen Architektur auf bestimmten Hardware-Plattformen schneller ist, soll diese Entscheidung dem Auto-Tuner überlassen werden.
- `UpdateIndex`: Aktualisiert die Indexdatenstruktur im Speicher.
- `CreateIndexFile`: Schreibt die Indexdatenstruktur auf die Festplatte.
- `Query`: Führt eine Suche auf dem Index durch, um alle Dokumente zu finden, die die Wörter der Suchanfrage enthalten.
- `GetKeywords`: Erfasst eine Liste von Suchworten von der Kommandozeile.
- `ShowResults`: Zeigt die Suchergebnisse auf der Kommandozeile an.

Listing 5.5 stellt die parallele Architektur der Desktopsuche in Form des entsprechenden TADL-Skripts dar.

Die Indizierung der gefundenen Dokumente erfolgt in zwei Schritten. Zunächst wird ein Schlüsselwort durch einen der beiden String-Matching-Algorithmen gefunden. Anschließend wird dieses Wort in den Index aufgenommen und mit dem Dokument assoziiert, in dem es gerade gefunden wurde. Da dieser Prozess datenstromorientiert abläuft (d.h. jedes gefundene Schlüsselwort wird separat verarbeitet), lässt sich dieser Prozess entweder mit einem Fließband oder mit einem Erzeuger/Verbraucher-Konstrukt beschreiben. Die entsprechenden atomaren Komponenten können demnach entweder mit einem Tunable Pipeline-Konnektor oder mit einem Producer/Consumer-Konnektor verbunden werden.

Allerdings kann nicht abgeschätzt werden, welche Verarbeitungsstrategie in diesem Anwendungsfall performanter ist. Gemäß der Semantik der TADL-Konnektoren verarbeitet das Fließband die Datenelemente eins zu eins (d.h., jede Stufe verarbeitet genau ein Datenelement, übergibt dieses der nächste Stufe und fordert dann ein neues Element an), wohingegen bei der Erzeuger/Verbraucher-Strategie das Anfordern mehrerer Datenelemente auf einmal durch den Verbraucher möglich ist. Daraus folgt, dass eine Fließbandstufe nach der Verarbeitung eines Datenelements mit weniger Wartezeit zu rechnen hat. Der Verbraucher hingegen kann mehrere Datenelemente nach einander verarbeiten, ohne nach jedem Element zu synchronisieren.

```
TunableAlternative DesktopSearchAlternatives {
    SequentialComposition DesktopSearch1
        [input: null, GetKeywords;
        output: null, ShowResults] {
```

```
        TunablePipeline
            [source:Crawl;
            sink:CreateIndexFile] {
            TunableAlternative {
                StringSearch1[replicable],
                StringSearch2[replicable]
            },
            UpdateIndex[replicable]
        },
        Query
    },

    SequentialComposition DesktopSearch2
        [input:null,GetKeywords;
        output:null,ShowResults] {
        TunableProducerConsumer
            [source:Crawl;
            sink:CreateIndexFile] {
           TunableAlternative {
                StringSearch1[replicable],
                StringSearch2[replicable]
            },
            UpdateIndex[replicable]
        },
        Query
    }
}
```

Listing 5.5: Architekturbeschreibung der parallelen Desktopsuche (aus [ScPT10]).

Um beide Architekturvarianten zu definieren, verwenden wir einen Tunable Alternative-Konnektor als äußersten Operator des TADL-Skripts. Somit überlassen wir die Entscheidung dem Auto-Tuner, der später beide Varianten testet.

In beiden Alternativen beginnen wir jeweils mit einem Sequential Composition-Konnektor. Wir stellen somit sicher, dass die Indizierung der Dokumente stets abgeschlossen ist, bevor das Programm Suchanfragen entgegennimmt.

Im weiteren Verlauf setzen wir in der ersten Variante einen Tunable Pipeline-Konnektor ein, der als Quelle `Crawl` aufruft, um eine Liste mit Dateipfaden zu erhalten und seine Ergebnisse der Senke `CreateIndexFile` übergibt, die den Index auf der Festplatte aktualisiert. Das Fließband enthält zwei Stufen, wovon die erste wiederum ein Tunable Alternative-Konnektor ist, der die Auswahl eines geeigneten String-Matching-Algorithmus (`StringSearch1` oder `StringSearch2`) wie gewünscht in den Verantwortungsbereich des Auto-Tuners übergibt. Die zweite Stufe aktualisiert die Indexdatenstruktur im Speicher (`UpdateIndex`). `Query` repräsentiert die zweite Komponente des Sequential Composition-Konnektors.

Die zweite Architekturvariante entspricht der ersten, außer dass der Tunable Pipeline-Konnektor den Tunable Producer/Consumer-Konnektor ersetzt.

Durch die konsistente Syntax und Semantik der TADL-Konnektoren erlaubt das Konzept die Austauschbarkeit von Architekturelementen mittels einfacher Änderung der Beschreibung. Gleichzeitig können auf allen Ebenen der Architektur Alternativen definiert werden, die bei der späteren Optimierung automatisch getestet werden.

In Kapitel 6 zeigen wir, wie eine durch ein TADL-Skript beschriebe optimierbare Architektur automatisch in eine lauffähige Implementierung überführt wird und mit welchen Techniken Atune-OPT sich der Optimierung der gesamten Architektur annimmt.

5.3.5 Zusammenfassung und Vergleich

Atune-TA unterstützt den Entwurf und die Entwicklung einer parallelen optimierbaren Programmarchitektur, indem Parallelität sowie Optimierungspotential vordefinierter Architekturelemente implizit genutzt werden. Mit TADL stellt Atune-TA eine Beschreibungssprache zur Verfügung, mit der die Interaktion atomarer Programmkomponenten sowie deren parallele Verarbeitung definiert werden kann. TADL kombiniert im Kontext des Architekturentwurfs erstmals Aspekte der Parallelisierung sowie der automatischen Performanzoptimierung und repräsentiert somit ein Modell paralleler optimierbarer Architekturen.

Im Kapitel über verwandte Arbeiten wurden einige Ansätze vorgestellt, die mit Atune-TA zumindest im weiteren Sinne verwandt sind.

Betrachten wir zunächst die Arbeiten, die sich mit dem Entwurf und der Implementierung konfigurierbarer Programme beschäftigen. Hier zeigt sich, dass die klassischen Verfahren im Bereich der Softwareadaption sowie der dynamischen Rekonfiguration (z.B. das Rainbow Framework [GCHS+04, GaSC09]) zwar auf Architekturmodellen basieren, jedoch den Entwurf paralleler Applikationen etwa durch Modellierung entsprechender Parallelisierungsstrategien nicht unterstützen.

Des Weiteren beschränkt sich die Adaptivität der vorgestellten Ansätze auf bestimmte Umgebungsparameter, die – obwohl sie das Verhalten einer Komponente beeinflussen können – nicht uneingeschränkt als leistungsrelevante Tuning-Parameter zu verstehen sind. In vielen Fällen stellt die topologische Adaption der Architektur den Schwerpunkt der Verfahren dar [RaPo03]. Hinzu kommt, dass in der Regel keine automatische Performanzoptimierung vorgesehen ist.

Im Bereich der Architekturbeschreibungssprachen bietet keiner der relevanten Ansätze die Möglichkeit, parallele Architekturen unter Berücksichtigung automatischer Performanzoptimierung zu entwerfen. Sprachen wie *C2* [MORT96] und *Darwin* [MDEK95] sind auf die Modellierung verteilter Systeme spezialisiert und können somit eine gewisse Form von Parallelität ausdrücken. Die Beschreibung optimierbarer paralleler Programmstrukturen ist jedoch nicht möglich.

Die vorgestellte Entwurfsmustersprache [MaSM04, MaMS99, MaMS00] kann in gewissem Maße als Grundlage für TADL verstanden werden. Auch wenn der Ansatz keine Beschreibungssprache im eigentlichen Sinne darstellt, so bietet das Konzept dennoch einen Leitfaden, der einen musterbasierten Prozess für Entwurf und Entwicklung paralleler Programme beschreibt.

Zusammenfassend repräsentiert Atune-TA einen neuartigen Ansatz, der Teile existierender Verfahren sowie neue Ideen zu einem Konzept vereint, das die Entwicklung und die Optimierung paralleler Anwendungen entscheidend vereinfacht.

5.4 *Atune-OPT*: Optimierung paralleler Architekturen

Der dritte und in der logischen Aneinanderreihung der Teilkonzepte letzte Schwerpunkt der Arbeit behandelt die tatsächliche Optimierung einer parallelen Applikation und das damit verbundene Konzept eines adäquaten Auto-Tuners für parallele Architekturen.

Diese Aufgabe erfüllt *Atune-OPT*, das in den folgenden Abschnitten vorgestellt wird. Atune-OPT ist ein Auto-Tuning-Konzept, das konzeptionell auf der Basis von Atune-TA

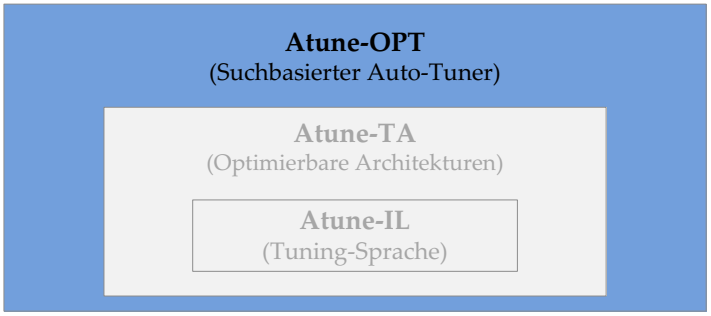


Abbildung 5.11: Atune-OPT im Kontext des Gesamtkonzeptes.

arbeitet und daher aus technischer Sicht die Funktion einer Auto-Tuning-Komponente für optimierbaren Architekturen einnimmt. Abbildung 5.11 skizziert diesen Sachverhalt.

Atune-OPT stellt Techniken zur Verfügung, um optimierbare Architekturen und damit komplexe parallele Applikationen effizient zu optimieren. Hierfür nutzt Atune-OPT wiederum Informationen, die bei der Implementierung einer optimierbaren Architektur in Form von Atune-IL-Instrumentierungen erzeugt werden.

Die technischen Zusammenhänge sowie die Implementierung sind im nächsten Kapitel Gegenstand der Diskussion. Im Folgenden erörtern wir zunächst die Herausforderungen für einen Auto-Tuner zur Optimierung ganzer Architekturen – im Gegensatz zu domänenspezifischen Tunern für numerische Programme oder spezielle Algorithmen. Anschließend erläutern wir im Einzelnen die Kernaspekte von Atune-OPT, insbesondere die Verfahren zur Generierung von möglichst kleinen Suchräumen.

5.4.1 Einordnung in die Auto-Tuning-Taxonomie

Zunächst ordnen wir Atune-OPT in die Auto-Tuning-Taxonomie ein (vgl. hierzu Kapitel 3, Abschnitt 3.4.1).

Wie bereits eingangs des Kapitels erwähnt, basiert Atune-OPT auf suchbasiertem Offline-Tuning. Die Kombination aus einem suchbasierten Ansatz und Offline-Tuning bietet die größtmögliche Flexibilität hinsichtlich optimierbarer Programmstruktur, Anwendungsdomäne sowie Hardware-Plattform.

Wenngleich modellbasierte Optimierung ein durchaus effizientes Verfahren darstellt und bezogen auf konkrete Anwendungsfälle gute Ergebnisse erzielen kann, so gilt es jedoch im Allgemeinen als schwer bis unmöglich, ganze Architekturen durch flexible analytische Performanzmodelle zu beschreiben, die zur Optimierung herangezogen werden können und die präzise den Leistungswert einer bestimmten Parameterkonfiguration vorhersagen können [EGDP+06]. Atune-OPT kombiniert den such-basierten Ansatz mit neuen Techniken, um diesen zur Optimierung paralleler Architekturen zu befähigen. Hierzu zählen insbesondere Verfahren zur Partitionierung und kontextbasierten Reduktion des Suchraums.

Der Ansatz des Offline-Tuning wurde gewählt, da dieses Verfahren – abgesehen von der Laufzeit – keine Voraussetzungen an die zu optimierende Applikation stellt (vgl. hierzu die Diskussion in Kapitel 3, Abschnitt 3.4.1.1). Im Gegensatz hierzu erfordert Online-Tuning eine Programmstruktur, die es erlaubt, zur Laufzeit wiederkehrende Messungen sowie Änderungen entsprechender Parameterwerte durchzuführen. Diese Einschränkung ist jedoch bei einem allgemeinen Ansatz, der für ein breites Spektrum paralleler Architekturen verwendbar sein soll, eher kontraproduktiv.

5.4.2 Vorverarbeitung des Suchraums

Wie in Kapitel 4 gezeigt wurde, kommt Auto-Tuning bereits in vielen Bereichen zum Einsatz. Die meisten der existierenden Auto-Tuning-Ansätze konzentrieren sich auf die Optimierung konkreter Algorithmen (z.B. [FrJo98]), kleiner Programme einer bestimmten Anwendungsdomäne (z.B. [WhPD01, KKHY03]), Programme, die auf bestimmten Programmiermodellen basieren(z.B. [MCSML07]) oder das Interesse liegt auf der Optimierung von Bibliotheken (z.B. [TaCH02]). In der Regel werden derartige Ansätze zur Optimierung numerischer und wissenschaftliche Anwendungen verwendet.

Alle genannten Ansätze bieten für ihre jeweiligen Anwendungsdomänen und Zielplattformen vielversprechende Ergebnisse. Begibt man sich jedoch aus dem Bereich der wissenschaftlichen Anwendungen heraus und betrachtet allgemeine parallele Applikationen, auf welche die bisherigen Konzepte dieser Arbeit abgestimmt sind, so zeigen sich einige Unzulänglichkeiten existierender Tuning-Ansätze. Der Fokus auf nicht-numerische parallele Programme mit breitem Einsatzgebiet erfordert erweiterte Optimierungskonzepte, da die Programme, die von einem Auto-Tuner optimiert werden sollen, unterschiedlichen Anwendungsdomänen angehören und meist komplexere parallele Strukturen aufweisen, die zudem je nach Programm sehr verschieden ausfallen können. Für ein generalisiertes Auto-Tuning-Konzept ist also die Abstraktion von Art und Zweck des zu optimierenden Programms von großer Bedeutung, um größtmögliche Portabilität des Konzeptes selbst zu erreichen.

Des Weiteren zeichnen sich unter Berücksichtigung eines breiten Spektrums an parallelen Programmen diverse Optimierungsmöglichkeiten ab, die mit hochspezialisierten Methoden kaum abzudecken sind.

Nicht zuletzt ergibt sich durch den meist großen Umfang allgemein verwendbarer Programme eine zusätzliche Herausforderung für ein Auto-Tuning-Konzept, da Ansätze zur Strukturierung des Programms sowie zur Partitionierung des Suchraums benötigt werden.

Atune-OPT soll den genannten Anforderungen durch eine starke Integration mit den bisherigen Teilkonzepten gerecht werden, indem die Vorzüge der Verfahren für eine effiziente Optimierung genutzt werden. Hierzu sind zusätzliche Verarbeitungsschritte nötig, die der eigentlichen Optimierung vorgelagert sind. Abbildung 5.12 skizziert die Phase der Vorverarbeitung, die aus der *Suchraumpartitionierung* sowie der *kontextbasierten Suchraumreduktion* besteht. Beide Verfahren dienen der Generierung kleiner, aber relevanter Suchräume, die für die üblichen Suchalgorithmen handhabbar sind. Erst nachdem der Suchraum hinreichend verkleinert wurde, beginnt der eigentliche suchbasierte Optimierungsvorgang.

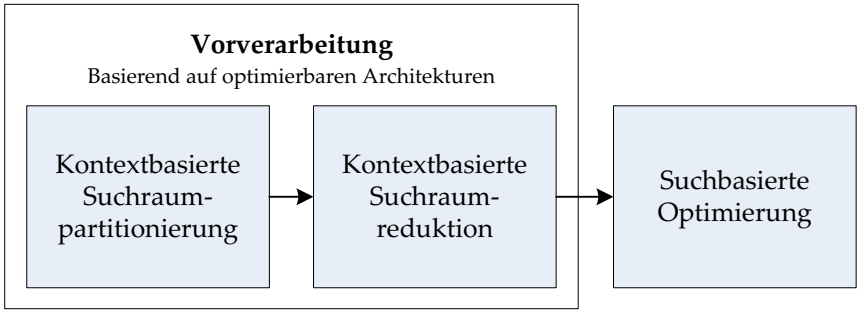


Abbildung 5.12: Vorverarbeitungsschritte von Atune-OPT zur Suchraumreduktion.

In den folgenden Abschnitten werden die Vorverarbeitungsschritte im Einzelnen erörtert.

5.4.3 Kontextbasierte Suchraumpartitionierung: Identifikation von Tuning-Einheiten

Der kritischste Faktor der suchbasierten Optimierung ist der Suchraum selbst. Auf Grund seiner Definition, dem kartesischen Produkt der Wertebereiche aller Parameter des Programms (vgl. Definition 3.6), wächst der Suchraum exponentiell zur Anzahl der Parameter. In diesem Zusammenhang spricht man auch von *Zustandsexplosion*.

Je umfangreicher ein paralleles Programm ist und je mehr parallele Sektionen es besitzt, umso größer ist die Zahl seiner Tuning-Parameter. Es liegt daher nahe, den Suchraum gemäß der Struktur des zu optimierenden Programms in kleinere Partitionen zu zerlegen, indem man sich die Unabhängigkeit bestimmter Programmteile zu Nutze macht.

5.4.3.1 Analyse der Programmstruktur

In Abschnitt 5.1.3 dieses Kapitels haben wir ein grundlegendes Konzept zur Partitionierung des Suchraums abstrakt definiert: Indem wir unabhängige Programmteile separat von einander optimieren, wird der gesamte Suchraum in mehrere kleinere Suchräume zerteilt.

Das Prinzip soll hier noch einmal rekapituliert werden (siehe Definition 5.1). Zwei parametrisierte Programmteile P_1 und P_2 eines Programms $\wp$ gelten im Sinne der Optimierung als unabhängig, wenn sie auf jedem möglichen Ausführungspfad des Programms nach einander ausgeführt werden. Das Ergebnis von P_1 muss also vorliegen, bevor P_2 startet. Aus der Unabhängigkeit von P_1 und P_2 folgt, dass die Parameter von P_1 diejenigen von P_2 nicht beeinflussen und umgekehrt. P_1 und P_2 können daher separat optimiert werden. Somit ergibt sich anstelle des gesamten Suchraums $\mathcal{S}_\wp$ ein wesentlich kleinerer Suchraum $\mathcal{R}_\wp$, der aus den separaten Suchraumpartitionen besteht, die jeweils von einem der Programmteile aufgespannt werden: $\mathcal{R}_\wp = C_1 \cup C_2$. Es gilt $|\mathcal{R}_\wp| \leq |\mathcal{S}_\wp|$.

5.4.3.2 Ausnutzung der TADL-Semantik

Die Analyse unabhängiger Programmteile zur kontextbasierten Suchraumpartitionierung setzt jedoch voraus, dass genaue Informationen über die architektonische Programmstruktur vorliegen.

Zu diesem Zweck nutzt Atune-OPT die Semantik der optimierbaren Architekturen, sofern das Programm auf Atune-TA basiert. Hierbei sind insbesondere der Sequential Composition- sowie der Tunable Alternative-Konnektor von Interesse. Beide Konnektoren stellen sicher, dass ihre Kind-Elemente niemals gleichzeitig ausgeführt werden. Daraus folgt, dass die Tuning-Parameter eines Kind-Elements und seines untergeordneten Teilbaums nicht mit Parametern anderer Kind-Elemente interferieren. Aus Sicht der Optimierung stellen daher die Kind-Elemente von Sequential Composition- und Tunable Alternative-Konnektor zusammen mit ihren jeweiligen untergeordneten Teilbäumen unabhängige Programmteile dar, die separat optimiert werden können.

Nehmen wir an, $\mathcal{S}_{Kon}$ sei der Suchraum, der durch alle Tuning-Parameter unterhalb eines Sequential Composition- oder Tunable Alternative-Konnektors aufgespannt wird. Weiterhin sei C_i der Suchraum, der durch die Tuning-Parameter des i-ten Kind-Elementes und dessen untergeordneten Teilbaum aufgespannt wird. Auf Grund der Semantik des jeweiligen Konnektors sowie der Definition der Suchraumpartitionierung kann der Suchraum für alle n Kind-Elemente eines Sequential Composition- oder Tunable Alternative-Konnektors mit

$$\mathcal{R}_{Kon} = C_1 \cup C_2 \cup ... \cup C_n$$

beschrieben werden. Gemäß obiger Definition gilt auch hier $|\mathcal{R}_{Kon}| \leq |\mathcal{S}_{Kon}|$.

Die Einteilung des Suchraums in Tuning-Einheiten erfolgt an Hand der architektonischen Struktur. Wurden für das zu optimierende Programm keine entsprechenden Konnektoren verwendet, besteht der Suchraum aus einer einzigen impliziten Tuning-Einheit, die das gesamte Programm umfasst.

Abbildung 5.13 veranschaulicht den Sachverhalt.

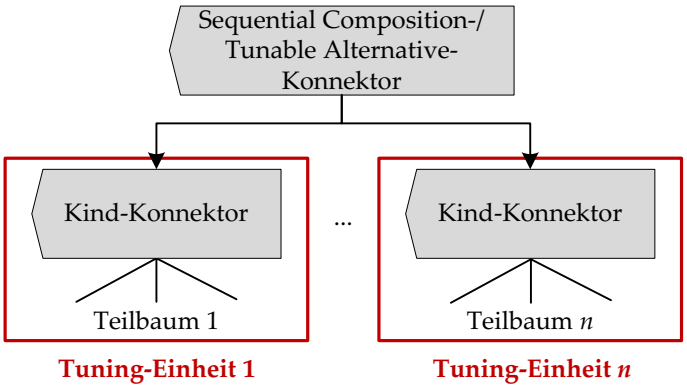


Abbildung 5.13: Generierung von Tuning-Einheiten an Hand der Semantik von TADL-Konnektoren.

Taucht an einer beliebigen Stelle in der Baumstruktur einer optimierbaren Architektur ein Sequential Composition- oder Tunable Alternative-Konnektor auf, so kann jeder seiner untergeordneten Teilbäume einer separaten Tuning-Einheit zugeordnet werden. Anstatt nun die Tuning-Parameter aller Kind-Elemente unterhalb des Konnektors gemeinsam auf einander abzustimmen, kann jede Tuning-Einheit getrennt von den anderen optimiert werden, ohne die Ergebnisse zu verfälschen.

Das beschriebene Verfahren kann innerhalb des Architekturbaums rekursiv angewendet werden, um auf allen Ebenen die Semantik der beiden Konnektoren entsprechend auszunutzen.

Der Identifikation von Tuning-Einheiten stellt den ersten Schritt in der Verarbeitung eines auf Atune-TA basierenden Programms dar. Das Ergebnis der Suchraumpartitionierung sind $n \geq 1$ Tuning-Einheiten des zu optimierenden Programms. Alle weiteren Schritte werden dementsprechend für jede identifizierte Tuning-Einheit separat durchgeführt.

Formal betrachtet repräsentiert eine Tuning-Einheit eine maskierte Parameterkonfiguration über der Parameter-Menge U, wobei U alle Tuning-Parameter der Tuning-Einheit enthält (vgl. Definition 3.5).

5.4.4 Kontextbasierten Suchraumreduktion: Analyse der Tuning-Einheiten

Nach der Partitionierung des Suchraums in einzelne Tuning-Einheiten werden diese im nächsten Verarbeitungsschritt hinsichtlich der enthaltenen Tuning-Parameter sowie der verwendeten Parallelisierungsstrategien analysiert. Es wird versucht, aus jeder Tuning-

Einheit in möglichst großem Maße Kontextwissen zu extrahieren, mit dessen Hilfe der Optimierungsvorgang deutlich beschleunigt werden kann.

Ohne Kontextwissen wäre die effiziente Optimierung einer parallelen Architektur nur schwer möglich. Aufgrund der Abstraktion des Auto-Tuning-Prozesses von der eigentlichen Anwendung gehen Informationen über die Tuning-Parameter und deren Zweck verloren. Ein Tuning-Parameter stellt für einen suchbasierten Auto-Tuner daher lediglich eine Menge an Werten dar, aus denen er auswählen kann. Dieser Vorgang wird *Black Box Tuning* genannt. Dies kann dazu führen, dass unnötige oder gar sinnlose Parameterkonfigurationen getestet werden. Die Spezifikation des Suchraums ist also zu ungenau.

Zur exakteren Spezifizierung des Suchraums nutzt Atune-OPT Kontextwissen aus den einzelnen Tuning-Einheiten. Das Ziel besteht darin, der eigentlichen suchbasierten Optimierung eine möglichst geringe Anzahl an Parameterkonfigurationen zu überlassen.

5.4.4.1 Anwenden von Tuning-Heuristiken

Um Tuning-Parameter in einen für die Optimierung relevanten Kontext zu setzen, behilft sich Atune-OPT auch hier mit der Semantik der Konnektoren, die mit TADL spezifiziert wurden.

Ein Konnektor beschreibt eine konkrete Parallelisierungsstrategie und wird basierend auf einem entsprechenden parallelen optimierbaren Entwurfsmuster implementiert. Alle Tuning-Parameter, die durch einen Konnektor in die spätere Implementierung der Architektur eingebunden werden, können dem Konnektor und damit einer konkreten Parallelisierungsstrategie zugeordnet werden. Aus Sicht des Auto-Tuners stellt ein Parameter nun nicht mehr nur eine anonyme Menge an Werten dar, sondern eine kontextbezogene Konfigurationsmöglichkeit.

Mit diesem Zusatzwissen müssen die Tuning-Parameter innerhalb einer Tuning-Einheit nicht unmittelbar zu einem Suchraum zusammengefasst werden, der anschließend von einem Suchalgorithmus verarbeitet wird. Vielmehr können für jeden Konnektor innerhalb der Tuning-Einheit so genannte *Tuning-Heuristiken* angewendet werden. Eine Tuning-Heuristik beschreibt, welche Parameter eines Konnektors in erster Linie konfiguriert werden müssen, welche Abhängigkeiten existieren, bei welchen Parametern der Wertebereich eingeschränkt werden kann, welche Parameter von externen Umgebungsparametern (z.B. Anzahl der Rechenkerne) abhängen und welche Parameterkonfigurationen schließlich für die nachfolgende suchbasierte Optimierung übrig bleiben. Eine Tuning-Heuristik kann als Beschreibung eines Optimierungsziels für einen bestimmten Konnektor verstanden werden.

Hierbei ist eine Tuning-Heuristik nicht mit einem analytischen Modell zu verwechseln. Letzteres beschreibt mittels einer mathematischen Formel die Leistung eines Programms (oder eines Programmteils) und wird zur Vorhersage der Leistung herangezogen. Tuning-Heuristiken hingegen basieren nicht auf einem Modell, sondern auf einer Beschreibung eines effizienten Optimierungsvorgangs.

Tuning-Heuristiken können auch historische Daten mit in den Optimierungsvorgang einbeziehen, die über mehrere Tuning-Läufe hinweg gesammelt werden und als Erfahrungswerte für den aktuellen Optimierungsvorgang dienen.

Das Konzept von Atune-OPT umfasst Tuning-Heuristiken für den Tunable Pipeline-Konnektor mit replizierbaren Kind-Elementen sowie für den Tunable Fork/Join-Konnektor. Beide Heuristiken werden im Folgenden behandelt. Zusätzlich sei an dieser Stelle angemerkt, dass das Konzept der musterbasierten Tuning-Heuristiken bereits veröffentlicht wurde [Scha09].

Heuristik für den Tunable Pipeline-Konnektor mit replizierbaren Kind-Elementen

Die Leistung eines Fließbandes wird an Hand seines Durchsatzes gemessen. Entscheidend ist demnach, wie viele Datenelemente pro Zeiteinheit das Fließband durchlaufen und nach der letzten Stufe als Ergebnis vorliegen.

Aus dieser Überlegung folgt, dass alle Stufen des Fließbandes etwa die gleiche Laufzeit haben sollten, da die Stufe mit der längsten Laufzeit den Durchsatz bestimmt. Das Optimierungsziel für einen Tunable Pipeline-Konnektor ist daher ein Fließband mit ausgeglichenen Stufen.

Bei einem Fließband mit einfachen Stufen kann die Laufzeit kaum beeinflusst werden, da die Verarbeitungsgeschwindigkeit einer Stufe einzig von der durchzuführenden Berechnung pro Datenelement abhängt.

Wesentlich größeres Optimierungspotential bietet ein Fließband mit replizierbaren Stufen. Dies entspricht einem Tunable Pipeline-Konnektor, dessen Stufen replizierbare atomare Komponenten darstellen (Anwendung des *Tunable Replication*-Konnektors auf jede Stufe). Durch Hinzufügen von parallelen Instanzen kann die Laufzeit einer Stufe positiv beeinflusst werden.

Abbildung 5.14 skizziert den schematischen Aufbau. Gemäß den impliziten Tuning-Parametern des Tunable Pipeline-Konnektors sowie der Tunable Replication (siehe Tabelle 5.2 in Abschnitt 5.3.2.3) besitzt ein datenparalleles Fließband jedoch eine große Anzahl an Tuning-Parametern: Besteht das Fließband aus s Stufen, so sind $3s - 1$ Parameter zu berücksichtigen.

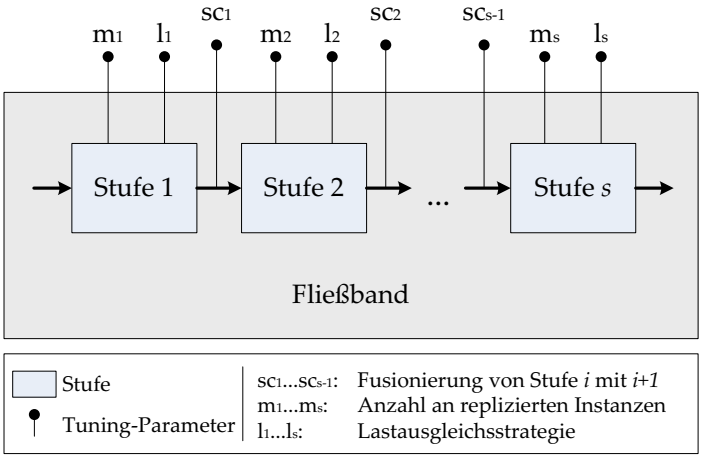


Abbildung 5.14: Darstellung eines Tunable Pipeline-Konnektors mit datenparallelen Stufen.

Normalerweise müsste ein Auto-Tuner alle Tuning-Parameter gemeinsam optimieren, wodurch sich alleine für die Optimierung eines Konnektors dieser Art ein Suchraum signifikanter Größe ergäbe.

Da jedoch bekannt ist, dass die Parameter ein Fließband konfigurieren, kann eine geeignete Tuning-Heuristik angewendet werden. Hierfür müssen die Laufzeiten der schnellsten sowie der langsamsten Stufe schrittweise aneinander angepasst werden, um das Fließband zu balancieren.

Abbildung 5.15 zeigt beispielhaft ein Fließband, welches zunächst nicht balanciert ist. Es fällt auf, dass Stufe 2 für die Verarbeitung der Datenelemente deutlich mehr Zeit benötigt

als die Stufen 1 und 3. Da wir annehmen, dass Stufe 2 replizierbar ist, können mehrere Instanzen der Stufe erzeugt und auf diese Weise mehr Datenelemente pro Zeiteinheit von dieser Stufe verarbeitet werden. Das Fließband ist balanciert und damit seine Gesamtlaufzeit verkürzt.

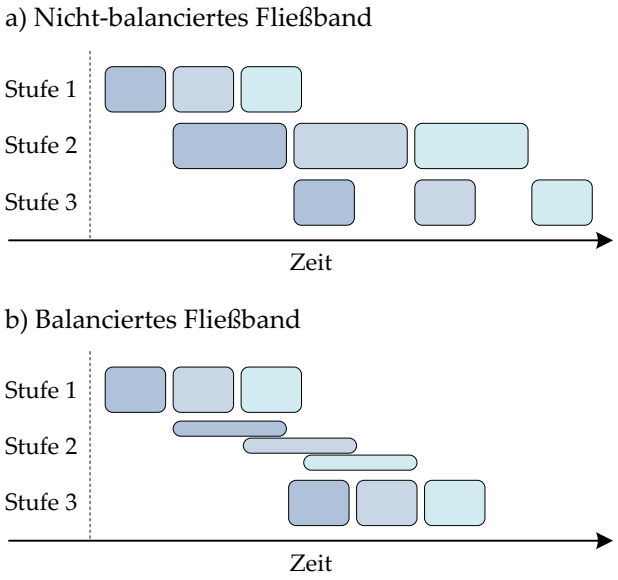


Abbildung 5.15: Beispiel eines Fließbandes mit a) nicht-balancierten und b) balancierten Stufen.

Um ein balanciertes Fließband zu erhalten, wird zunächst die Anzahl paralleler Instanzen für die langsamste ($m_{langsam}$) und die schnellste Stufe ($m_{schnell}$) konfiguriert, indem Parameterkonfigurationen aus der resultierenden Suchraumpartition $V_{m_{langsam}} \times V_{m_{schnell}}$ getestet werden. Die Suche ist beendet, sobald beide Stufen etwa die gleiche Laufzeit aufweisen oder sich die Laufzeit beider Stufen nicht mehr signifikant verändert. Für die übrigen Stufen werden die jeweiligen Werte für m wie folgt gesetzt: Zwei Stufen i und j weisen wir einen Wert für m_i und m_j zu, wobei $m_{schnell} \leq m_i, m_j \leq m_{langsam}$ gilt. Die Zuweisung der Werte erfolgt proportional zur Laufzeit der Stufe: eine langsamere Stufe erhält also mehr Instanzen als eine schnelle.

Nachdem die Anzahl paralleler Instanzen für jede Stufe konfiguriert wurde, können die Laufzeiten ggf. mit Hilfe der jeweiligen Lastausgleichsstrategie (Parameter $l_1, ..., l_s$) feinabgestimmt werden.

Der Prozess der Balancierung kann, falls nötig, für die übrigen datenparallelen Stufen wiederholt werden.

Ebenfalls leistungsrelevant ist die Fusionierung von Stufen (engl. *stage fusion*). Hierbei wird für jedes Stufenpaar entschieden, ob die beiden Stufen verbunden und zu einer einzigen Stufe zusammengefasst werden (vgl. Abbildung 5.14, Parameter $sc_1, ..., sc_{s-1}$). Insbesondere bei replizierbaren Stufen hat dies den Vorteil, dass die aufwendige Synchronisation zwischen den beiden Stufen entfällt und die zur Replikation erforderlichen Ausführungsfäden nur einmal erzeugt werden müssen.

Da das Fusionieren von replizierbaren Stufen in den meisten Fällen lohnenswert ist, muss dies nicht für jedes Stufenpaar ausprobiert werden. Vielmehr können aufeinander folgende datenparallele Stufen pauschal zu einer Gruppe zusammengefasst werden, wie in Abbildung 5.16 dargestellt. Hierbei wird die Anzahl der parallelen Instanzen auf den

höchsten Wert der Gruppe gesetzt, die Lastausgleichsstrategie hingegen von der ersten Stufe der Gruppe übernommen.

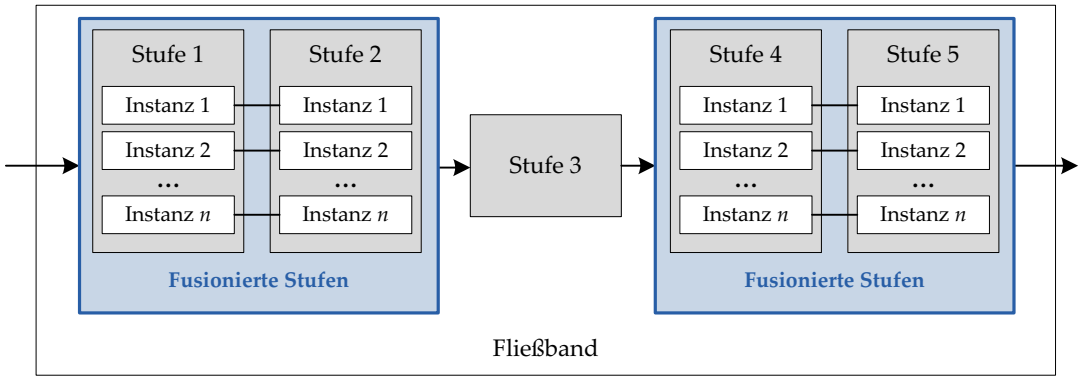


Abbildung 5.16: Beispiel eines Fließbandes mit fusionierten replizierbaren Stufen.

Betrachten wir die erläuterte Heuristik, so wird deutlich, dass durch gezielte Analyse sowie durch Verwendung von Kontextwissen eine signifikante Einsparung an Parameterkonfigurationen erreicht werden kann. Der Anteil der suchbasierten Optimierung für einen Tunable Pipeline-Konnektor wird damit auf ein notwendiges Minimum reduziert.

Heuristik für den Tunable Fork/Join-Konnektor

Die Leistung des Tunable Fork/Join-Konnektors hängt im wesentlichen von der Anzahl der Arbeiterfäden w, auf die die Aufgaben verteilt werden. Abbildung 5.17 skizziert den entsprechenden schematischen Aufbau.

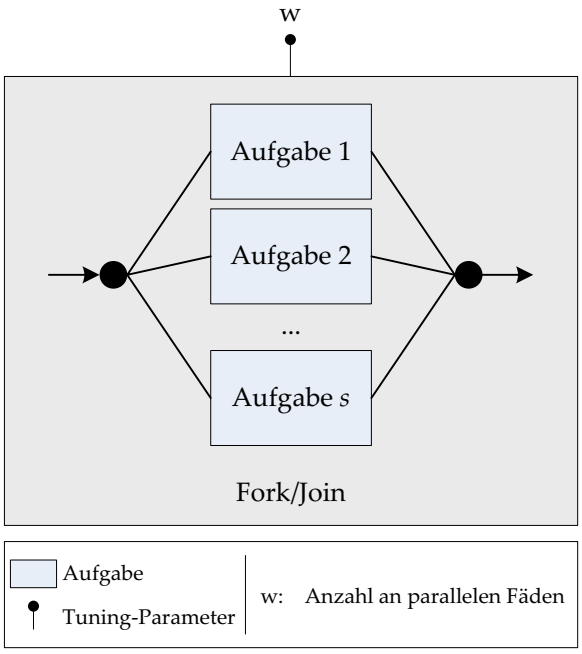


Abbildung 5.17: Darstellung eines optimierbaren Fork/Join-Konnektors.

Die optimale Anzahl an Arbeiterfäden hängt offenbar mit der verfügbaren Anzahl an Rechenkernen c der zu Grunde liegenden Hardware-Plattform sowie mit der Anzahl an Aufgaben t zusammen. Die genaue Korrelation kennen wir jedoch nicht. Ohne Kontextwissen über den verwendeten Konnektor besäße der Suchraum die Größe V_w.

Für die Suchraumreduktion beginnen wir mit der Einschränkung des Wertebereichs von w auf eine Umgebung um c. Hierfür nutzen wir historische Daten in Form zweier Parameter α und β, so dass $\lfloor \alpha \cdot c \rfloor \leq w \leq \lfloor \beta \cdot c \rfloor$ gilt, wobei $0 < \alpha \leq 1$ und $\beta > 1$. Die historischen Daten für α und β hängen von konkreten Werten für c ab. Wurde w beispielsweise auf einer 8-Kern-Maschine $c = 8$ bei allen vergangenen Tuning-Läufen auf Werte zwischen 6 und 10 gesetzt und in einer Datenbank gespeichert, so liefern α und β die Werte $0,75$ bzw. $1,25$.

Die Korrelation von w mit t wird berücksichtigt, indem eine fixe Umgebung um t zum (bereits reduzierten) Wertebereich von w hinzugefügt wird. Es hat sich gezeigt, dass der Bereich mit $\lfloor 0,9t \rfloor \leq w \leq \lfloor 1,1t \rfloor$ hinreichend gut abgeschätzt werden kann. t kann offensichtlich nur dann in die Heuristik aufgenommen werden, falls die Anzahl an Aufgaben im Programm bereits vor der Ausführung feststeht und sich nicht dynamisch zur Laufzeit ergibt.

Nach Anwendung der Heuristik setzt sich der Suchraum nur noch aus dem eingeschränkten Wertebereich von w zusammen. Das Ziel, der anschließenden suchbasierten Optimierung einen möglichst kleinen Suchraum zu überlassen, ist damit erreicht.

Die hier vorgestellte Heuristik ist nahezu unverändert auch für die Optimierung des Tunable Replication-Konnektors verwendbar. Die zusätzliche Wahl der besten Lastausgleichsstrategie kann unabhängig von der Anzahl der Fäden bzw. Instanzen bestimmt werden.

Es ist anzumerken, dass sich die parallele Architektur eines Programms üblicherweise aus mehreren Konnektoren zusammensetzt, so dass die Einsparung an Parameterkonfigurationen für das gesamte Programm beachtliche Ausmaße erreichen kann.

5.4.4.2 Klassifikation der Tuning-Parameter

Die Anwendung von Tuning-Heuristiken kann die Suche nach einer optimalen Parameterkonfiguration wesentlich vereinfachen. Jedoch ist nicht garantiert, dass alle Parameter eines Programms durch Heuristiken erfasst werden. Beispielsweise können Parallelisierungsstrategien verwendet werden, zu denen noch keine passende Heuristik existiert.

Atune-OPT analysiert jedoch auch diejenigen Parameter, die nicht von einer Heuristik abgedeckt sind. Hierzu zählt die Klassifizierung der Tuning-Parameter an Hand ihres Skalenniveaus.

Wie bereits im Grundlagen-Kapitel erläutert, besitzen Tuning-Parameter wie alle Variablen ein Skalenniveau. Im Hinblick auf die suchbasierte Optimierung ist die Unterscheidung zwischen der Nominal- und der Ordinalskala relevant.

Diese Unterscheidung ist wichtig, da für nominalskalierte Parameter keinerlei Aussage hinsichtlich ihres Verhaltens und der Ordnung der Werte gemacht werden kann. Daraus folgt, dass meta-heuristische Suchverfahren, die auf bestimmten Annahmen über die Parameterwerte basieren, nominalskalierten Parameter nicht korrekt verarbeiten können. Oftmals werden beispielsweise von einem Suchverfahren im Umkreis einer hinreichend guten Parameterkonfiguration weitere gute Konfigurationen vermutet. Diese Vermutung ist aber nur dann gültig, wenn die Parameter eine höhere Skalierung aufweisen, so dass auf deren Wertebereichen eine natürliche Rangordnung definiert werden kann.

Hiervon nicht betroffen sind die vollständige Enumeration sowie das zufällige Testen, da beide Verfahren nicht auf Annahmen über den Suchraum und die Parameterwerte basieren. Atune-OPT nutzt diese Eigenschaft und führt alle nominalskalierten Parameter einem separaten Verarbeitungsprozess zu, der auf zufälligem Testen basiert und in Abschnitt 5.4.5 beschrieben wird. Alle ordinalskalierten Parameter, die durch keine der bisherigen Vorverarbeitungsschritte ausgeschlossen werden konnten, werden später der suchbasierten Optimierung zugeführt.

Atune-OPT stellt aktuell eines der sehr wenigen Verfahren dar, das die Skalierung von Parametern überhaupt berücksichtigt.

5.4.4.3 Berücksichtigung konditionaler Parameterabhängigkeiten

Den letzten Analyseschritt, den Atune-OPT zur Suchraumreduktion durchführt, ist die Berücksichtigung von konditionalen Parameterabhängigkeiten. Ähnlich wie die Klassifizierung der Parameter wird auch die Berechnung der Abhängigkeit nur auf diejenigen Parameter angewendet, die nicht bereits von einer Tuning-Heuristik verarbeitet wurden.

Gemäß Definition 5.2 im Rahmen der Spezifikation von Atune-IL tritt eine konditionale Parameterabhängigkeiten genau dann ein, wenn ein Parameter q nur für bestimmte Werte eines anderen Parameters p gültig ist: $(q \rightarrow p|V'_p)$.

Indem eine Parameterabhängigkeit bei der Berechnung des Suchraums berücksichtigt wird, können einige Parameterkonfigurationen eingespart werden. Die folgende Überlegung verdeutlicht den Sachverhalt.

Es sei

$$\mathcal{S} = V_p \times V_q$$

der Suchraum, der von p und q aufgespannt wird. Da q jedoch nur dann in den Suchraum aufgenommen werden muss, wenn p einen Wert $v \in V'_p \subset V_p$ annimmt, ergibt sich ein reduzierter Suchraum $\mathcal{R}'$ mit

$$\mathcal{R}' = V'_p \times V_q$$

Aus $V'_p \subset V_p$ folgt dann $|\mathcal{R}'| < |\mathcal{S}|$.

Atune-OPT analysiert in jeder Tuning-Einheit die vorhandenen Parameterabhängigkeiten und passt den Suchraum der Einheit entsprechend an. Existieren konditionale Parameterabhängigkeiten, müssen alle abhängigen Parameter (und damit der durch diese Parameter aufgespannte Teil des Suchraums) nur dann berücksichtigt werden, wenn der übergeordnete Parameter bestimmte Werte annimmt.

Steuert ein Tuning-Parameter beispielsweise die Auswahl zweier alternativer Algorithmen-Implementierungen, die jeweils weitere Tuning-Parameter enthalten, so müssen die Tuning-Parameter der gerade nicht ausgewählten Alternative bei der Suchraumberechnung nicht berücksichtigt werden.

5.4.5 Optimierung der Tuning-Einheiten

Im letzten Verarbeitungsschritt werden alle n identifizierten und vorverarbeiteten Tuning-Einheiten nacheinander mittels suchbasiertem Offline-Tuning optimiert. Hierzu ordnet Atune-OPT zunächst jeder Tuning-Einheit i, $1 \leq i \leq n$ die entsprechende Suchraumpartition C_i zu, die nur noch diejenigen Parameterkonfigurationen enthält, die nicht durch einen der Vorverarbeitungsschritte bereits ausgeschlossen werden konnten.

Die Parameter der Tuning-Einheiten, die gerade nicht bearbeitet werden, werden auf ihre jeweiligen Standardwerte gesetzt. Formal beschreibt eine Tuning-Einheit i daher die

Menge der maskierten Parameterkonfigurationen über der Menge T_i, die alle Parameter der Tuning-Einheit enthält ($\bar{\mathcal{K}}_{T_i}$).

Das Tuning-Prozessmodell von Atune-OPT basiert auf dem in Abschnitt 5.1.2 eingeführten Auto-Tuning-Zyklus.

5.4.5.1 Messpunkte

Wie durch den Auto-Tuning-Zyklus beschrieben, ist Atune-OPT in jeder Tuning-Iteration auf die Ergebnisse der Leistungsmessung angewiesen, um daraus Schlussfolgerungen für die Berechnung einer neuen Parameterkonfiguration zu ziehen.

Um für jede Tuning-Einheit Aussagen über die Leistungswerte der jeweils zu Grunde liegenden maskierten Parameterkonfigurationen treffen zu können, muss jede Tuning-Einheit mindestens einen für das Leistungskriterium passenden Messpunkt besitzen.

Befindet sich in einer Tuning-Einheit kein Messpunkt, so kann sie nicht separat optimiert werden. Atune-OPT ordnet daher eine Tuning-Einheit ohne Messpunkt der übergeordneten Tuning-Einheit zu. Existiert keine übergeordnete Tuning-Einheit, so wird die Tuning-Einheit aufgelöst und deren Tuning-Parameter damit der impliziten Tuning-Einheit, die das gesamte Programm umfasst, zugewiesen.

Es sei angemerkt, dass in der Regel alle Tuning-Einheiten über Messpunkte verfügen, da die Messpunkte bei der Implementierung der TADL-Konnektoren automatisch in das Programm integriert werden.

Besitzt eine Tuning-Einheit mehr als einen Messpunkt, so aggregiert Atune-OPT die Messergebnisse. Hierbei kann in den zentralen Einstellungen von Atune-OPT spezifiziert werden, auf welche Weise die Ergebnisse zusammengefasst werden sollen (z.B. Aufsummierung, Bildung des Durchschnitts, usw.).

5.4.5.2 Konfiguration der nominalskalierten Tuning-Parameter

Die zuvor separierten Tuning-Parameter mit Nominalskalierung (*NP*) werden in einem vorgelagerten Prozess konfiguriert. Wir sprechen in diesem Zusammenhang nicht von Optimierung, da kein Näherungsverfahren im klassischen Sinn angewendet werden kann.

Atune-OPT nutzt eine Kombination aus Zufallsverfahren und vollständiger Enumeration, um eine möglichst gute Konfiguration der nominalskalierten Parameter zu erzielen. Zunächst werden alle ordinalskalierten Parameter der Tuning-Einheit auf ihren Standardwert fixiert. Anschließend wird eine feste Zahl an maskierten Parameterkonfigurationen über der Menge der nominalskalierten Parameter U_{NP} zufällig ausgewählt.

In einem weiteren Schritt wird aus allen zufällig ausgewählten maskierten Parameterkonfiguration das kartesische Produkt gebildet, so dass ein Suchraum entsteht, der von den nominalskalierten Parametern aufgespannt wird. Schließlich testet Atune-OPT jede Parameterkonfiguration dieses Suchraums und konfiguriert die nominalskalierten Parameter gemäß der besten Parameterkonfiguration.

Während der nachfolgenden Optimierung der ordinalskalierten Parameter werden die nominalskalierten Parameter nicht mehr berücksichtigt und auf ihrem besten gefundenen Wert fixiert. Diese Werte fließen somit auch in die beste Parameterkonfiguration des gesamten Programms ein.

Es ist zu beachten, dass die zuvor bestimmte beste Konfiguration der nominalskalierten Parametern in Kombination mit der anschließend besten Konfiguration der ordinalskalierten Parameter nicht unbedingt die optimale Gesamtkonfiguration ergibt, da die separate Wahl der ordinalskalierten Parameterwerte die nominalskalierten Parameter beeinflussen könnte. Dennoch hat sich gezeigt, dass dieses Verfahren zu hinreichend guten Lösungen führt.

5.4.5.3 Suchstrategien für die ordinalskalierten Tuning-Parameter

Zur Lösung kombinatorischer Optimierungsprobleme wie das des Auto-Tuning existieren eine große Zahl an brauchbaren Verfahren. Bereits im Grundlagen-Kapitel (Abschnitt 3.3) wurden die wichtigsten Vertreter der Suchverfahren vorgestellt.

Es ist allerdings kaum möglich, aus der Menge der Suchverfahren eines auszuwählen, das für Optimierungsproblem des Auto-Tuning am besten geeignet ist. Jedes Verfahren weist gleichermaßen Vor- und Nachteile auf.

Diese Tatsache ist auch bei existierenden Arbeiten zu beobachten. Einige Ansätze basieren auf dem Simplex-Algorithmus nach Nelder und Mead (z.B. [TaCH02]), wohingegen andere Arbeiten dem Simplex-Algorithmus Unzulänglichkeiten bei einem mehrdimensionalen Suchraum bescheinigen [McKi98]. Wieder andere Ansätze verwenden den Algorithmus der simulierten Abkühlung oder Modifikationen desselben (z.B. [HaPo09]).

Hinzu kommt, dass sich die meisten der existierenden Ansätze im Wesentlichen mit der Optimierung wissenschaftlicher Anwendungen und numerischen Algorithmen im Kontext des Hochleistungsrechnens beschäftigen. Dieser Anwendungsbereich stellt jedoch spezielle Anforderungen an die Suchalgorithmen, wie beispielsweise möglichst kurze Laufzeit oder Robustheit gegenüber Systemveränderungen.

Da sich kein einzelnes Verfahren als allgemein besonders geeignet herausstellt und der Schwerpunkt dieser Arbeit in diesem Zusammenhang nicht auf der mathematischen Analyse und Modifikation eines Suchverfahrens basiert, verwendet Atune-OPT für die suchbasierte Optimierung aller ordinalskalierten Tuning-Parameter drei unterschiedliche Suchstrategien, die je nach Größe und Beschaffenheit der zu optimierenden Tuning-Einheit ausgewählt und eingesetzt werden. Die drei in das Konzept integrierten Strategien wurden mit Bedacht gewählt: Mit einem lokalen und einem globalen Suchverfahren sowie einem Zufallsverfahren werden drei wichtige Klassen der Metaheuristiken abgedeckt. Die dynamische Auswahl einer geeigneten Suchstrategie an Hand von Suchraumcharakteristiken wird mit Atune-OPT in dieser Form erstmals angewendet.

Darüber hinaus sieht das Konzept von Atune-OPT den Einsatz beliebiger weiterer metaheuristischer Suchverfahren vor. Zu diesem Zweck stellen die Suchalgorithmen austauschbare Elemente des Auto-Tuning-Zyklus dar.

Im Folgenden werden die drei Suchstrategien erklärt.

Zufälliges Testen

Als einfaches Zufallsverfahren verwendet Atune-OPT das zufällige Testen (engl. *Random Sampling*). Die Strategie entspricht der Zufallsmethode, wie sie im Grundlagen-Kapitel, Abschnitt 3.3.2 beschrieben wurde.

Hierbei wird aus allen Parameterkonfigurationen zufällig eine Menge an Konfigurationen ausgewählt, die Atune-OPT schließlich testet. Die Parameterkonfiguration mit dem besten Leistungswert der ausgewählten Menge wird als optimale Konfiguration zurückgegeben. Zur Generierung der benötigten Zufallszahlen wird das *Mersenne Twister*-Verfahren [MaNi98] verwendet, das hochgradig gleichverteilte Sequenzen liefert.

Entgegen allgemeiner Bedenken können derartigen Zufallsverfahren durchaus brauchbare Optimierungsergebnisse mit wenig Aufwand erzielen. Dies ist insbesondere dann der Fall, wenn der Suchraum der zu optimierenden Tuning-Einheit sehr klein ist oder wenn die Leistungswerte der Tuning-Einheit eng beisammen liegen (die Parametersensitivität aller Tuning-Parameter der Einheit also gering ist).

Hillclimbing mit Check-Back-Schritt

Die zweite Suchstrategie basiert auf dem klassischen Bergsteigeralgorithmus und damit auf dem Verfahren der lokalen Suche (siehe Kapitel 3, Abschnitt 3.3.2).

Um das Konzept des Bergsteigeralgorithmus an die Anforderungen des Auto-Tuning anzupassen, haben wir das Verfahren modifiziert und um einen so genannten *Check-Back-Schritt* erweitert.

Listing 5.6 zeigt den Pseudo-Code des Algorithmus.

```
// INITIALISIERUNG
bestParamConfig = GetInitialParamConfig();
bestPerfValue = MAXVALUE;
threshold = SetThreshold();

// SUCHE
// Suche wird für jede Dimension separat durchgeführt
foreach (param in searchspace)
{
   // Suchrichtung ermitteln
   leftParamConfig = righParamConfig = bestParamConfig;
   leftParamConfig.Modify(param.NextValue, direction.left);
   rightParamConfig.Modify(param.NextValue, direction.right);
   leftPerfValue = EVAL(leftParamConfig);
   rightPerfValue = EVAL(rightParamConfig);
   currentParamConfig =
        leftPerfValue < rightPerfValue ? leftParamConfig : rightParamConfig;
   climbDirection =
        leftPerfValue < rightPerfValue ? direction.left : direction.right;

   // Bergsteiger starten
   while (param.HasValues)
   {
      currentParamConfig.Modify(param.NextValue, climbDirection);
      perfValue = EVAL(currentParamConfig);
      if (perfValue < bestPerfValue)
      {
         bestPerfValue = perfValue;
         bestParamConfig = currentParamConfig;
      }
      if (bestPerfValue - perfValue < threshold)
         break;
   }

   // Check-Back
   checkBackParamConfigs = GenerateCheckBackParamConfigs(prevParams, bestParamConfig);
   foreach (checkBackParamConfig in checkBackParamConfigs)
   {
      checkBackPerfValue = EVAL(checkBackParamConfig);
      if (checkBackPerfValue < bestPerfValue)
      {
         bestPerfValue = checkBackPerfValue;
         bestParamConfig = checkBackParamConfig;
      }
   }
}
return bestParamConfig;
```

Listing 5.6: Pseudo-Code für Hillclimbing-Verfahren mit Check-Back.

Zunächst wird eine initiale Parameterkonfiguration gewählt. Hierfür stehen mehrere Methoden zur Verfügung. Entweder werden alle Parameter auf ihren Standardwert oder auf den kleinsten Wert ihrer Wertemenge gesetzt, oder es wird eine zufällige Startkonfiguration gewählt.

Anschließend wird für jede Dimension des Suchraums (also für jeden Parameter) das Bergsteigerprinzip angewendet. Die aktuelle Konfiguration besteht hierbei aus den je-

weils besten Werten für die Parameter, die bereits getestet wurden, sowie aus den initial gesetzten Werten der Parameter, die noch zu testen sind.

Es werden zwei Nachbarkonfigurationen (`leftParamConfig`, `rightParamConfig`) generiert, die sich von der aktuellen Konfiguration dadurch unterscheiden, dass der Wert des gerade untersuchten Parameters durch seine jeweiligen Nachbarwerte ersetzt wird. Die Nachbarkonfiguration mit dem besseren Leistungswert legt die Richtung fest, in der auf der aktuellen Dimension gesucht wird.

Der Wertebereich des aktuellen Parameters wird in der vorgegebenen Richtung auf der Grundlage der aktuellen Konfiguration so lange untersucht, bis der Rand des Wertebereichs erreicht ist oder bis die Modifikation des Parameterwertes und damit der aktuellen Konfiguration keine signifikante Verbesserung des Leistungswertes mehr liefert.

Verglichen mit dem klassischen Bergsteigeralgorithmus ist dies eine der wesentliche Änderungen. Während sich bei letzterem der Leistungswert nicht *verschlechtern* darf, muss dieser sich hier mindestens um einen definierten Schwellwert (`threshold`) *verbessern*. Der Verlust an Genauigkeit kann in Kauf genommen werden, da die Modifikation verhindert, dass das Verfahren auf Grund stetiger minimaler Verbesserungen des Leistungswertes eine große Anzahl an Iterationen benötigt.

Abschließend wird der Check-Back-Schritt durchgeführt. Mit diesem Zusatzschritt soll herausgefunden werden, ob bereits getestete Parameter auf Grund der Wertemodifikation des aktuellen Parameters beeinflusst werden. Der Check-Back-Schritt wird wie folgt durchgeführt

1. Die Wertebereiche V_{p_i} aller Parameter p_i, die bereits getestet wurden, werden jeweils auf die Randwerte reduziert: $V_{p_i} = \{min(V_{p_i}), max(V_{p_i})\}$.
2. Der aktuelle Parameter wird auf den besten gefundenen Wert gesetzt.
3. Alle Parameter, die noch nicht getestet wurden, werden auf ihren Standardwert gesetzt.
4. Es wird ein Suchraum $\mathcal{S}_{CB}$ definiert, der von den reduzierten Wertebereichen der schon getesteten Parametern sowie den jeweils festen Werten des aktuellen und der noch zu testenden Parameter aufgespannt wird.
5. $\mathcal{S}_{CB}$ wird vollständig durchsucht.
6. Erzielt eine der Parameterkonfigurationen aus $\mathcal{S}_{CB}$ einen besseren Leistungswert als die aktuelle Konfiguration, wird die aktuelle Konfiguration durch diese ersetzt und für den Test des nächsten Parameters verwendet.

Partikel-Schwarm-Optimierung

Die Partikel-Schwarm-Optimierung (PSO) ist ein globales Suchverfahren, das erstmals von Kennedy und Eberhart [KeEb95] beschrieben wurde.

PSO basiert auf der Ausnutzung von Schwarmintelligenz. Zur Lösung eines gegebenen Problems in Form einer Fitnessfunktion wird mittels eines Zufallsverfahrens ein so genannter Partikel-Schwarm definiert. Jedes Partikel des Schwarms ist eine potentielle Lösung des Problems. Anschließend beginnt ein iterativer Prozess mit dem Ziel, die potentiellen Lösungen zu verbessern. Hierzu evaluieren die Partikel jeweils ihre aktuelle potentielle Lösung. Jedes Partikel merkt sich stets die beste seiner bisherigen Lösungen (*lokale beste Lösung*) und kennt die aktuell beste Lösung aller Partikel (*globale beste Lösung*).

In jeder Iteration errechnet jedes Partikel in Abhängigkeit der lokalen und globalen besten Lösung eine neue potentielle Lösung, die in der nächsten Iteration evaluiert wird. Auf diese Weise bewegt sich der Schwarm durch den Suchraum und konvergiert schließlich bei der globalen besten Lösung.

Üblicherweise wird der Schwarm in Form von Partikeln in einem mehrdimensionalen Raum modelliert. Jedes Partikel weist eine Position (die gerade aktuelle Lösung des Partikels) sowie eine Geschwindigkeit (engl. *velocity*) auf, mit der es sich durch den Raum bewegt.

Mittels der gespeicherten besten lokalen Lösung sowie dem Wissen über die gerade aktuelle globale beste Lösung passt ein Partikel in jeder Iteration seine Position und Geschwindigkeit an.

Im Kontext des Auto-Tuning besteht ein Partikel aus einer aktuellen Parameterkonfiguration sowie aus derjenigen Parameterkonfiguration, mit der das Partikel den bisher besten lokalen Leistungswert erzielt hat.

Die Gleichungen zur Aktualisierung der Position und Geschwindigkeit eines Partikel im Kontext des Auto-Tuning bei n Dimensionen (Suchraum wird durch n Tuning-Parameter aufgespannt) und einer Schwarmgröße m sind gegeben durch

$$(g_i)_j = c_1(g_i)_j + c_1 r_1(globalBest - (x_i)_j) + c_2 r_2(localBest_j - (x_i)_j), \\ i \in \{1, ..., n\}, j \in \{1, ..., m\}$$

bzw.

$$(\overline{x}_i)_j = (x_i)_j + (g_i)_j$$

Hierbei repräsentiert das n-Tupel $(x_i)_j$ die aktuelle Parameterkonfiguration des Partikels j, $(g_i)_j$ das n-Tupel, welches für jeden Parameterwert der Konfiguration die Änderung angibt und $(\overline{x}_i)_j$ die neue Parameterkonfiguration, die aus der elementweisen Addition von $(g_i)_j$ und $(x_i)_j$ entsteht. $globalBest$ ist die globale beste Lösung und $localBest_j$ die lokale beste Lösung des Partikels j.

Des Weiteren sind c_1 und c_2 Konstanten, die bestimmen, wie stark die lokale beste Lösung des Partikels sowie die globale beste Lösung die Bewegung des Partikels beeinflussen. Allgemein ist $c_1, c_2 \approx 2$ ein guter Wert, jedoch können die Konstanten auch mit Zufallszahlen für jedes Partikel initialisiert werden.

r_1 und r_2 sind Vektoren, wobei jede Komponente eine Zufallszahl zwischen 0 und 1 darstellt.

Das Prinzip des PSO-Algorithmus wurde für die Verwendung als Atune-OPT-Suchstrategie nicht verändert. Listing 5.7 zeigt den Algorithmus im Pseudo-Code.

```
// INITIALISIERUNG
swarm = List<Particle>(numParticles);
bestPerfValue_global = MAXVALUE;
bestParamConfig_global = null

for k in (1, numParticles)
   particle = CreateNewParticle();
   particle.CurrentParamConfig = GetRandomParamConfig();
   particle.SetVelocity(0);
   particle.BestLocalParamConfig = null;
   particle.BestLocalPerfValue = MAXVALUE;
   swarm.Add(particle);

// SUCHE
for i in (1, iterations)
```

```
{
   foreach (particle in swarm)
   {
      currentPerfValue = EVAL(particle.CurrentParamConfig);
      if (currentPerfValue < bestPerfValue_global)
         bestPerfValue_global = currentPerfValue;
         bestParamConfig_global = particle.CurrentParamConfig;

      if (currentPerfValue < particle.BestLocalPerfValue)
         particle.BestLocalPerfValue = currentPerfValue;
         particle.BestLocalParamConfig = particle.CurrentParamConfig;

      newParamConfig;
      foreach (param in particle.CurrentParamConfig)
      {
         param.SetValue = UpdateVelocity(param.CurrentValue);
         newParamConfig.AddParam(param);
      }
      particle.CurrentParamConfig = newParamConfig;
   }
}
return bestParamConfig_global;
```

Listing 5.7: Pseudo-Code für Partikel-Schwarm-Optimierung.

Zunächst wird der Schwarm initialisiert, indem alle Partikel erzeugt und mit einer zufällig gewählten Parameterkonfiguration versehen werden. Des Weiteren wird die initiale Geschwindigkeit jedes Partikels auf 0 gesetzt.

Die Anzahl an Iterationen wird über einen vordefinierten Wert festgelegt (`iterations`). In jeder Iteration werden nun für jedes Partikel die folgenden vier Schritte durchgeführt:

1. Testen der eigenen aktuellen Parameterkonfiguration (`particle.CurrentParamConfig`) als potentielle Lösung des Problems. Die Fitnessfunktion entspricht gemäß dem Tuning-Suchproblem der Minimierung bzw. Maximierung des Leistungswertes des Programms.

2. Wurde ein besserer Leistungswert als globale beste Leistungswert erreicht, wird die globale beste Parameterkonfiguration (`bestParamConfig_global`) durch die aktuelle Konfiguration (`particle.CurrentParamConfig`) ersetzt.

3. Wird ein besserer Leistungswert als mit der gespeicherten lokalen besten Parameterkonfiguration (`particle.BestLocalParamConfig`) erreicht, wird die lokale beste durch die aktuelle Konfiguration ersetzt.

4. Schließlich wird für jeden Parameter der aktuellen Konfiguration (also für jede Lösungsdimension) ein neuer Wert berechnet, indem die Geschwindigkeit des Partikels aktualisiert und auf den entsprechenden Parameterwert addiert wird (vgl. obige Formeln).

Nach Durchführung aller Iterationen wird die globale beste Parameterkonfiguration als Ergebnis zurückgegeben.

5.4.5.4 Auswahl der Suchstrategie

Atune-OPT wählt für die Optimierung einer Tuning-Einheit automatisch eine der drei integrierten Suchstrategien. Die Entscheidung, welche Strategie für die zu optimierende Tuning-Einheit am ehesten geeignet ist, fällt Atune-OPT mit Hilfe der Analyse bestimmter Charakteristiken der Tuning-Einheit.

Die Bedingungen, die für die Wahl einer konkreten Suchstrategie zutreffen müssen, können als heuristische Entscheidungsgrundlagen formuliert werden und basieren auf Beobachtungen und Erfahrungswerten:

- Das zufällige Testen eignet sich insbesondere für Suchraumpartitionen, die durch die Vorverarbeitungsschritte bereits deutlich reduziert wurden.

  Atune-OPT wendet daher für kleine Suchraumpartitionen mit weniger als 100 Parameterkonfiguration das Zufallsverfahren an, dessen Präzision bei wenigen Parameterkonfigurationen in etwa mit der eines iterativen Verfahrens vergleichbar ist. In jedem Fall bietet das Zufallsverfahren eine triviale Implementierung, geringe Komplexität und eine sehr kurze Laufzeit. Diese Eigenschaften sprechen dafür, das Zufallsverfahren für kleine Suchräume vorzuziehen, bei denen meist ohnehin nur noch die exakte Anpassung einiger Parameter vorgenommen werden muss.

- Der klassische Bergsteigeralgorithmus erzielt insbesondere dann gute Ergebnisse, wenn die gedachte Leistungskurve (gebildet aus allen Leistungswerten) monoton steigt oder fällt. Abbildung 5.18 zeigt zwei typische Beispiele geeigneter Leistungskurven – zur Vereinfachung für nur einen Tuning-Parameter.

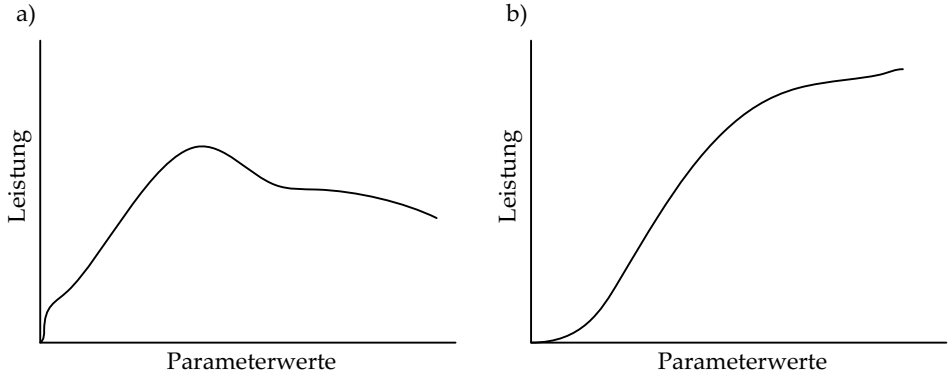


Abbildung 5.18: Beispiele günstiger Leistungskurven für den Bergsteigeralgorithmus.

  Die erste Kurve (a) besitzt nur einen Extremwert, so dass die Gefahr lokaler Optima nicht gegeben ist. Das Hillclimbing-Verfahren mit Check-Back-Schritt würde diesen Punkt als Ergebnis zurückgeben. Die zweite Kurve (b) ist monoton steigend; das Suchverfahren würde im flachen Teil der Kurve konvergieren. Somit erlaubt das Hillclimbing-Verfahren mit Check-Back-Schritt in beiden Fällen mit hoher Wahrscheinlichkeit das Auffinden des jeweils optimalen Parameterwertes.

  Beobachtungen haben gezeigt, dass Tuning-Parameter, die die Anzahl an Ausführungsfäden bestimmen, in der Regel Leistungskurven ähnlich den gezeigten verursachen. Aus diesem Grund wählt Atune-OPT das Hillclimbing-Verfahren mit Check-Back-Schritt zur Optimierung von Tuning-Einheiten, die ausschließlich Tuning-Parameter zur Anpassung der Faden-Anzahl beinhalten. Dies ist beispielsweise immer dann der Fall, wenn eine Tuning-Einheit einen Tunable Fork/Join-Konnektor beinhaltet und die entsprechende Tuning-Heuristik angewendet wurde (vgl. Abschnitt 5.4.4.1).

- Treffen keine der obigen Bedingungen zu, so nutzt Atune-OPT das PSO-Verfahren. Mit Hilfe des globalen Suchverfahrens können auch große Suchräume mit verschie-

densten Tuning-Parametern verarbeitet werden. PSO erzielt meist gute Ergebnisse mit beachtlicher Präzision, benötigt hierfür jedoch eine große Zahl an Iterationen.

Es sei angemerkt, dass die Wahl der Suchstrategie weniger entscheidend ist als die vor dem Starten der Suche durchgeführten Vorverarbeitungsschritte zur Suchraumreduktion. Die automatische Auswahl der Strategie stellt Atune-OPT als Hilfestellung zur Verfügung. Für jeden Optimierungsprozess kann die passende Suchstrategie auch manuell festgelegt werden.

Die Untersuchung der Korrelation zwischen bestimmten Suchraumcharakteristiken und passender Suchstrategie erfordert weiterführende Forschungsarbeiten und bietet noch ausreichend Raum für Verbesserungen. Die hier vorgestellten Ideen sollen als erster Schritt in Richtung intelligenter Strategieauswahl verstanden werden.

5.4.6 Zusammenführen der Ergebnisse

Nachdem für alle Tuning-Einheiten eine optimale Parameterkonfiguration gefunden wurde, fügt Atune-OPT diese zusammen und generiert eine optimale Parameterkonfiguration für das gesamte Programm.

Die optimale Parameterkonfiguration einer Tuning-Einheit entspricht einer konkreten maskierten Parameterkonfiguration über der Menge der Parameter in der Tuning-Einheit. Die optimale Parameterkonfiguration für das gesamte Programm erhält man, indem jedem Tuning-Parameter des Programms jeweils der entsprechende Wert aus den optimalen Parameterkonfigurationen der einzelnen Tuning-Einheiten zugewiesen wird.

5.4.6.1 Validierung der Tuning-Einheiten

Obwohl Atune-OPT unabhängige Programmteile zuverlässig analysiert und aus ihnen entsprechende Tuning-Einheiten erzeugt, kann es in seltenen Fällen vorkommen, dass zwei Tuning-Einheiten dennoch eine Abhängigkeit aufweisen, die nicht erkannt wurde oder sich erst zur Laufzeit äußert.

Um solche Fälle gänzlich auszuschließen, überprüft Atune-OPT die Unabhängigkeit der Tuning-Einheiten und gibt erst nach positivem Ergebnis den daraufhin gültigen Optimierungsprozess frei.

Die Überprüfung erfolgt an Hand des Vergleichs von Messwerten. Da jede Tuning-Einheit mindestens einen eigenen Messpunkt besitzen muss, um als solche fungieren zu können, werden auch die Leistungswerte der gerade nicht am Optimierungsprozess beteiligten Tuning-Einheiten mitprotokolliert. Der Leistungswert einer unbeteiligten Tuning-Einheit darf sich während der Optimierung anderer Tuning-Einheiten nicht ändern, da alle enthaltenen Tuning-Parameter auf ihrem Standardwert festgehalten werden.

Ändert sich allerdings der Leistungswert einer unbeteiligten Tuning-Einheit während der Optimierung einer anderen, so besteht offenbar eine Abhängigkeit, da sich die Änderung von Parameterwerten der gerade optimierten Tuning-Einheit auf die eigentlich unbeteiligte Tuning-Einheit auswirkt.

Atune-OPT kontrolliert während des gesamten Optimierungsprozesses fortlaufend die Leistungswerte und erkennt auf diese Weise Differenzen, die eine Abhängigkeit indizieren.

Wurde eine Abhängigkeit zwischen zwei Tuning-Einheiten identifiziert, so wird der Optimierungsprozess als ungültig markiert und abgebrochen. Für einen erneuten Optimierungsprozess vereinigt Atune-OPT die beiden abhängigen zu einer übergeordneten Tuning-Einheit.

5.4.7 Berücksichtigung der Eingabedaten

Im Grundlagen-Kapitel wurde bereits die Abhängigkeit des Optimierungsprozesses von den jeweiligen Eingabedaten diskutiert. Es wurde dargelegt, dass insbesondere die Problem-Komplexität der Eingabedaten die Leistungswerte der Parameterkonfigurationen beeinflussen kann.

Experimente haben allerdings gezeigt, dass bei vielen Programmen eine vereinfachte Klassifizierung von Eingabedaten vorgenommen werden kann [PSJT08, Scha09, ScPT09]. In den meisten Fällen lässt sich eine repräsentative Menge an Eingabedaten ermitteln. Das bedeutet, dass das Programm bei der Mehrzahl seiner Läufe Eingabedaten übergeben bekommt, die eine ähnliche Problemgröße und -komplexität aufweisen. Diese Tatsache kann man nutzen, indem man das Programm basierend auf der repräsentativen Menge an Eingabedaten optimiert.

Zu beachten sind anschließend jedoch die Randfälle, bei denen Problemgröße und Komplexität der Eingabedaten deutlich von denen der repräsentativen Menge abweichen. Für eine Auswahl an Randfällen müssen daher separate Tuning-Läufe durchgeführt werden.

Formal wird also die Menge aller möglichen Probleminstanzen (Eingabedaten) D in zwei Untermengen $D^* \subset D$ und $D' \subset D$ aufgeteilt, wobei D^* alle repräsentativen Probleminstanzen enthält, und D' alle Probleminstanzen, die in Problemgröße und -komplexität abweichen. Es gilt $D^* \cap D' = \emptyset$.

Zunächst wird ein Tuning-Lauf mit einer Probleminstanz $d \in D^*$ durchgeführt, um für alle üblicherweise auftretenden Eingabedaten eine möglichst optimale Parameterkonfiguration zu finden. Danach werden eine oder mehrere Probleminstanzen aus D' ausgewählt, für die jeweils einen weiteren Tuning-Lauf durchgeführt wird.

Sodann kann entschieden werden, wie das Programm zu konfigurieren ist. Weichen die jeweils ermittelten optimalen Parameterkonfigurationen und insbesondere die zugehörigen Leistungswerte stark voneinander ab, so empfiehlt es sich, zwei oder mehrere Parameterkonfiguration mit dem Programm auszuliefern, die je nach Zugehörigkeit der Eingabedaten zu einer der Mengen D^* oder D' ausgewählt werden. Bewegen sich die Leistungswerte des Programms trotz unterschiedlicher Problemgröße und Komplexität in ähnlichen Größenordnungen, so genügt eine einzige Parameterkonfiguration.

Das Konzept von Atune-OPT basiert daher auf der Annahme, dass das Programm mit einer repräsentativen Menge an Eingabedaten hinsichtlich Problemgröße und -komplexität optimiert wird. Die daraus resultierende optimale Parameterkonfiguration kann als Standard-Konfiguration des Programms für eine bestimmte Hardware-Plattform verwendet werden.

Atune-OPT archiviert zu jedem Programm und den zugehörigen Klassen der Eingabedaten die Leistungswerte bestimmter Parameterkonfigurationen sowie die Parametersensitivitäten, um bei einem erneuten Optimierungsvorgang des Programms die Informationen sofort zur Verfügung zu haben.

5.4.8 Parallelisierung des Optimierungsprozesses

Ein wichtiges Ziel bei der automatischen Performanzoptimierung ist eine möglichst kurze Laufzeit des gesamten Optimierungsprozesses. Bei Offline-Tuning-Verfahren repräsentiert die Anzahl an Tuning-Iterationen die kritische Größe, bei Online-Tuning-Verfahren spielt meist noch die Laufzeit des eigentlichen Suchalgorithmus eine wesentliche Rolle.

Aus diesem Grund wurde bereits in mehreren Arbeiten die Idee verfolgt, das Optimierungsverfahren selbst zu parallelisieren [DeTo91]. Grundsätzlich wird die parallele Verarbeitung von Parameterkonfigurationen in verschiedenen Bereichen des Suchraums angestrebt, um die Konvergenz des Suchverfahrens zu beschleunigen.

Aktuelle Ansätze (z.B. [TiTH09]) beschäftigen sich daher mit der Untersuchung parallelisierbarer Suchalgorithmen und der entsprechenden Umsetzung.

Das Konzept von Atune-OPT beinhaltet ebenfalls einen Ansatz zur parallelen Ausführung von Tuning-Iterationen, jedoch auf höherer Ebene. Hierzu machen wir uns die Eigenschaften der Tuning-Einheiten zu Nutze. Gemäß ihrer Definition stellen Tuning-Einheiten unabhängige Programmteile dar, die separat von anderen Tuning-Einheiten optimiert werden können. Insbesondere repräsentiert eine Tuning-Einheit i, $1 \leq i \leq n$ die Menge aller maskierten Parameterkonfigurationen über der Menge T_i, die alle Parameter der Tuning-Einheit enthält ($\bar{\mathcal{K}}_{T_i}$). Hieraus folgt, dass alle nicht in der Tuning-Einheit enthaltenen Parameter bei der Optimierung der Einheit ignoriert werden. Man erhält ein datenparalleles Problem.

Die Idee besteht nun in der parallelen Optimierung der n Tuning-Einheiten auf mehreren identischen Testplattformen. Abbildung 5.19 skizziert den Ablauf.

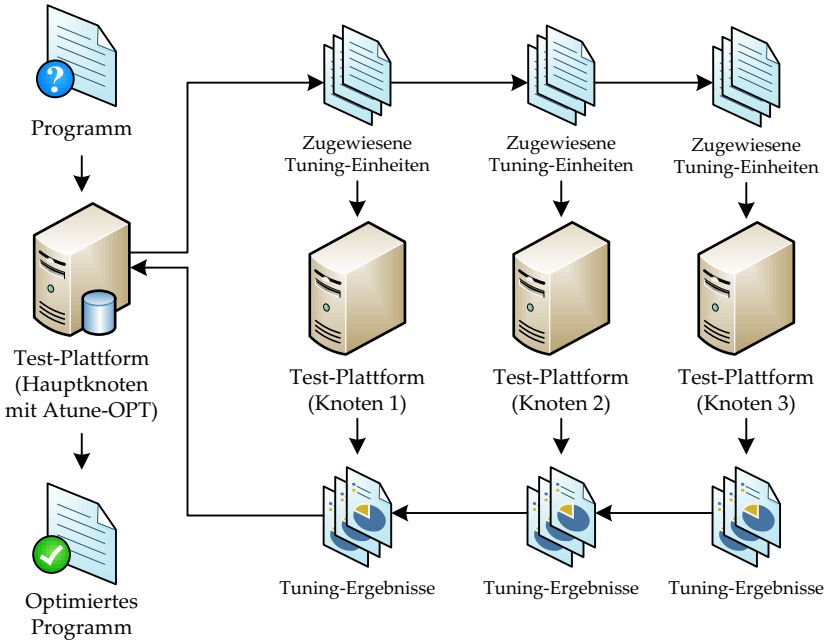


Abbildung 5.19: Schema der parallelen Optimierung von Tuning-Einheiten.

Die Identifikation der Tuning-Einheiten sowie alle weiteren Vorverarbeitungsschritte werden von Atune-OPT auf dem Hauptknoten durchgeführt. Anschließend verteilt Atune-OPT Kopien des Programms auf die Testplattformen. Auf jeder Testplattform werden nun ein oder mehrere Tuning-Einheiten optimiert. Anschließend sammelt Atune-OPT die Ergebnisse ein und konfiguriert das Programm entsprechend.

Auf diese Weise ist Atune-OPT in der Lage, den Optimierungsprozess enorm zu beschleunigen, was insbesondere bei umfangreichen parallelen Programmen, deren Suchraum in viele Tuning-Einheiten zerlegt werden kann, einen großen Vorteil mit sich bringt.

5.4.9 Zusammenfassung und Vergleich

Atune-OPT zeichnet sich gegenüber existierenden Auto-Tuning-Ansätzen insbesondere durch sein mehrstufiges Optimierungskonzept aus, das es ermöglicht, auch große parallele Applikationen durch Suchraumpartitionierung und kontextbasierter Suchraumreduktion automatisch zu optimieren.

Die meisten der verwandten Ansätze beschränken sich auf die Optimierung kleiner numerischer Programme [WhPD01, FrJo98, PMJP+05], wobei hier durchaus beachtliche Ergebnisse erzielt werden. Für den Einsatz im allgemeineren Umfeld paralleler Anwendungen ist Atune-OPT jedoch auf Grund des breiten Spektrums an möglichen Tuning-Instruktionen besser geeignet.

Allgemeinere Ansätze wie beispielsweise Active Harmony [TaCH02] basieren im Rahmen der eigentlichen Suche auf ähnlichen Verfahren wie Atune-OPT. Jedoch liegt hier der Fokus auf der Entwicklung eines geeigneten Suchalgorithmus, während Suchraumreduktion und spezielle Kontext-Analyse für parallele Programmkonstrukte fehlen.

Das Online-Tuning-System MATE [MCMS+04, MCSML07, MoML07] verbindet zusammen mit den weiterführenden Arbeiten um POETRIES [CMSL04, CéSL05] als einziger relevanter Ansatz parallele Entwurfsmuster und Optimierungsaspekte. Allerdings basiert die Optimierung auf analytischen Modellen und kann nur für verteilte Anwendungen auf der Basis von PVM eingesetzt werden. Dennoch gilt MATE als eine der ersten Arbeiten, die musterbasierte Optimierung in umfangreicher Weise untersucht und einsetzt.

Atune-OPT erweitert die Idee der musterbasierten Optimierung für suchbasiertes Auto-Tuning und stellt mit der kontextbasierten Suchraumreduktion einen vielversprechenden Ansatz vor.

Mit Atune-OPT wird das Gesamtkonzept dieser Arbeit abgerundet, so dass der Prozess vom Entwurf über die Implementierung und Instrumentierung bis hin zur automatischen Optimierung vollständig abgedeckt ist.

6. Implementierung

Alle Teilkonzepte dieser Arbeit (*Atune-IL*, *Atune-TA* und *Atune-OPT*) wurden in Form einer prototypischen Implementierung umgesetzt, um deren Funktionalität und Machbarkeit zu belegen. In diesem Kapitel werden für jedes der Teilkonzepte die entsprechenden Implementierungstechniken sowie die damit verbundenen Abläufe dargelegt.

Zu Beginn wird die Implementierung der Atune-IL-Funktionalität erörtert, die einen wichtigen Teil des Auto-Tuning-Zyklus umsetzt. Anschließend werden die Interna des TADL-Übersetzers sowie die Implementierungstechniken der optimierbaren Architekturen erläutert, um daraufhin den Prototyp von Atune-OPT vorzustellen. Das Kapitel schließt mit einer Prozessbeschreibung, in der die Schritte vom Entwurf bis hin zum optimierten parallelen Programm aus Sicht des Software-Entwicklers beschrieben werden.

6.1 Implementierung von Atune-IL

Die Umsetzung der Funktionalität von Atune-IL basiert auf einem *Präprozessor* und der *Anwendungssteuerung*. Diese beiden Komponenten implementieren wesentliche Teile des Auto-Tuning-Zyklus und stellen somit die Verbindung zwischen einer mit Atune-IL instrumentierten Anwendung und einem Auto-Tuner her.

Abbildung 6.2 zeigt den Auto-Tuning-Zyklus, wie er bereits in Kapitel 5, Abschnitt 5.1.2 veranschaulicht wurde. Zusätzlich sind die Zuständigkeitsbereiche von Präprozessor und Anwendungssteuerung der Atune-IL-Implementierung eingezeichnet.

Der Präprozessor übernimmt demnach das Auslesen der Tuning-Instruktionen sowie die Vorbereitung des Suchraums. Die Anwendungssteuerung wendet eine neue Parameterkonfiguration auf das Programm an, generiert daraus eine neue Programmvariante und führt die Leistungsmessung durch. Der Auto-Tuner selbst wird durch den Optimierungsschritt („*Parameterwerte berechnen*“) repräsentiert.

Die Darstellung verdeutlicht, dass sämtliche Kommunikation zwischen Auto-Tuner und zu optimierender Anwendung durch die beiden Komponenten der Atune-IL-Implementierung gesteuert wird. Auf diese Weise wird die angestrebte Separation von Tuner und Anwendung vollständig umgesetzt (vgl. Kapitel 5, Abschnitt 5.1.1).

6.1.1 Präprozessor

Gemäß des Auto-Tuning-Zyklus wird das Zerteilen (engl. *parsing*) der Atune-IL-Instrumentierungen im Programmquelltext und die anschließende Extraktion der Tuning-In-

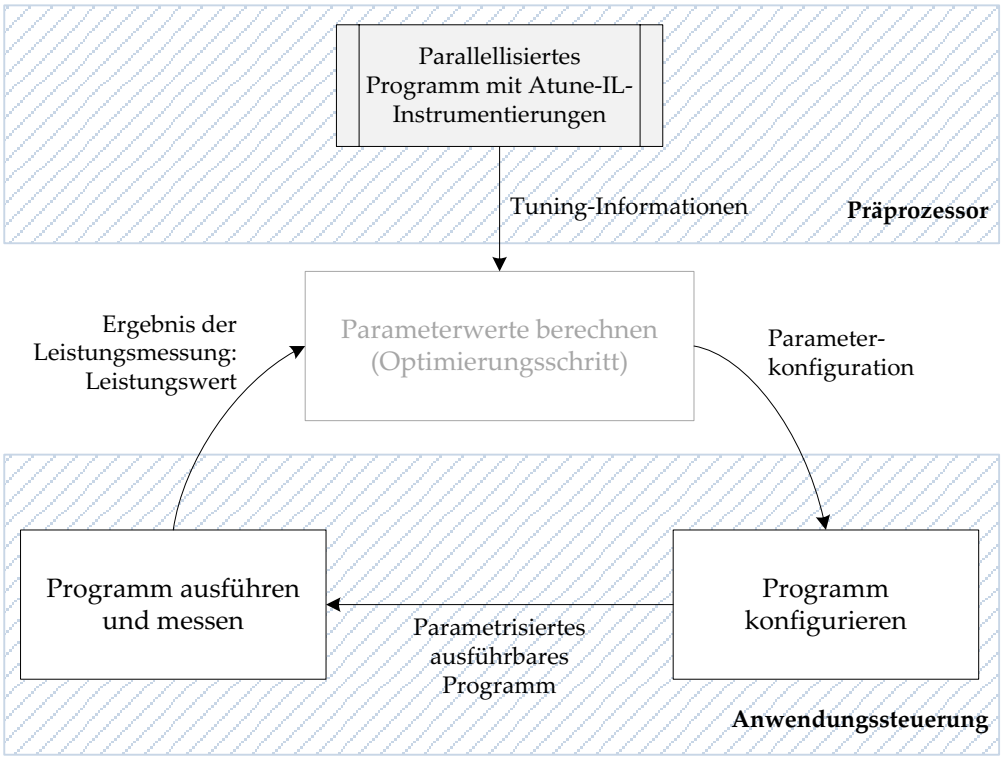


Abbildung 6.1: Zuordnung der Schritte des Auto-Tuning-Zyklus zu den Komponenten des Atune-IL-Implementierung.

struktionen in einem Vorverarbeitungsschritt einmal ausgeführt. Das Auslesen der Instruktionen bewerkstelligt der *Präprozessor*, dessen Zerteiler (engl. *parser*) mit Hilfe des *ANTLR-Parsergenerators* [antl09] erstellt wurde.

Zunächst betrachtet der Präprozessor nur die Direktiven der Atune-IL-Instrumentierungen in einem Programm und generiert daraus einen abstrakten Syntaxbaum (engl. *abstract syntax tree*). Der Syntaxbaum wird später verwendet, um beispielsweise die Zuordnung von Tuning-Parametern zu Tuning-Blöcken zu koordinieren. Alle nicht-interpretierten Quelltextzeilen der Wirtssprache werden für die spätere Transformation als Zeichenketten gespeichert.

Das Auslesen der Instrumentierung erfolgt ausgehend von einem Verzeichnis (typischerweise das Projektverzeichnis), in dem sich alle Quelltextdateien des Programms befinden. Eventuell vorhandene Unterverzeichnisse werden rekursiv verarbeitet. Alle Atune-IL-Instrumentierungen werden projektweit betrachtet, d. h. die Namen aller Parameter und Blöcke sind dateiübergreifend gültig.

Während des Auslesens überprüft der Präprozessor alle Instrumentierungen auf syntaktische und semantische Korrektheit.

6.1.2 Anwendungssteuerung

Die *Anwendungssteuerung* setzt die vom Auto-Tuner geforderten Parameterkonfigurationen um und versorgt den Tuner mit den aktuellen Leistungswerten des Programms.

Hierzu erzeugt die Anwendungssteuerung mittels Quelltexttransformationen die erforderlichen Programmvarianten (*Postprozessor*) und bereitet die Leistungswerte auf, die von den einzelnen Messpunkten ermittelt werden (*Monitor*).

6.1.2.1 Postprozessor

Der Postprozessor generiert aus dem gespeicherten (aber nicht interpretierten) Quelltext der Wirtssprache, den ausgelesenen Tuning-Instruktionen sowie der vom Auto-Tuner geforderten Parameterkonfiguration eine neue Programmvariante. Die Generierung einer Programmvariante entspricht einer Quelltext-zu-Quelltext-Transformation.

Für jede unterstützte Wirtssprache (C#, C++ und Java) werden Schablonen eingesetzt, um die Quelltextfragmente zu definieren. Hierzu zählt auch die von Sprache zu Sprache unterschiedliche Syntax der Übersetzerdirektiven. Dieses Verfahren ermöglicht die Trennung von Transformationslogik und den eigentlichen Transformationsregeln. Die Schablonen für eine konkrete Wirtssprache werden in einer separaten Datei spezifiziert, deren Aufbau vorgegeben ist. Dies ermöglicht das einfache Erstellen und Hinzufügen einer neuen Schablonendatei für eine weitere Wirtssprache. Somit wird die Portabilität zwischen mehreren Wirtssprachen garantiert.

Die einzelnen Atune-IL-Anweisungen werden auf unterschiedliche Weise verarbeitet. Hierbei unterscheidet der Postprozessor zwischen Anweisungen, die Programmvarianten definieren (z.B. Deklarationen von Tuning-Parametern), und solchen, die Meta-Informationen liefern, aber keine Quelltexttransformation bewirken (z.B. Tuning-Blöcke).

Deklarationen von Tuning-Parametern (hierzu zählen neben `setvar`-Anweisungen auch Permutationsbereiche, vgl. Kapitel 5, Abschnitt 5.2.3.2) werden durch entsprechende Zuweisungen ersetzt.

Abbildung 6.2 zeigt beispielhaft die Programmvarianten, die eine `setvar`-Anweisung spezfiziert. Die Programmvariable `sortAlgo` vom Typ `IParallelSortingAlgorithm` ist durch eine `setvar`-Anweisung als Tuning-Parameter markiert und mit den nötigen Tuning-Instruktionen versehen. Die Instrumentierung definiert zwei mögliche Zuweisungen, `new ParallelMergeSort()` und `new ParallelQuickSort()`. Folglich erzeugt der Postprozessor zwei Programmvarianten, die jeweils eine der beiden gültigen Zuweisungen enthalten. Die Instrumentierung selbst wird entfernt.

Die `gauge`-Anweisungen werden durch passende Aufrufe zu einer Messbibliothek ersetzt. Die Bibliothek stellt für jedes unterstützte Leistungskriterium Routinen zur Messung bereit und ist in der Wirtssprache des Programms verfasst. Dem Programmierer wird so ermöglicht, Routinen für nahezu beliebige Leistungskriterien zu implementieren. Der Prototyp unterstützt initial die Messung von Ausführungszeiten und Speicherverbrauch.

Tuning-Block-Anweisungen entfernt der Postprozessor ersatzlos, da diese bereits durch den Präprozessor verarbeitet wurden und für die Generierung der Programmvarianten nicht benötigt werden.

Nachdem der Postprozessor eine Programmvariante generiert hat, startet er den Übersetzungsvorgang, um ein ausführbares Programm zu erhalten. Es können je nach Wirtssprache und Zielplattform beliebige Übersetzer eingebunden und verwendet werden, beispielsweise *msbuild* von Microsoft oder *GNU make*.

Nach erfolgreicher Übersetzung steht eine ausführbare Programmvariante zur Verfügung, welche die angeforderte Parameterkonfiguration implementiert. Hierbei ist anzumerken, dass durch die strikte Trennung von Auto-Tuner und Programm sowie durch die Generierung der Programmvarianten kein zusätzlicher Aufwand im Programm entsteht (z.B. durch Einbinden von Bibliotheken o. ä.). Einzig die im nächsten Abschnitt erläuterte Implementierung der Messpunkte bindet in geringer Form zusätzlichen Quelltext in das Programm ein.

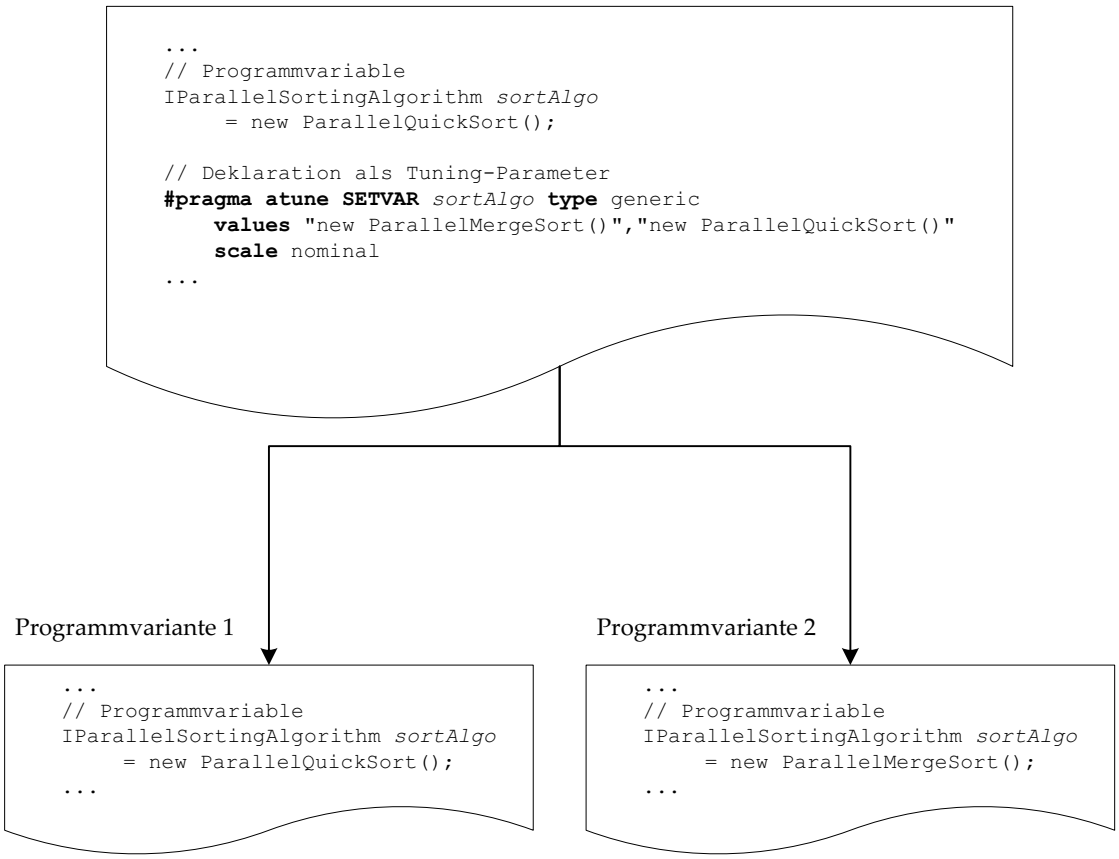


Abbildung 6.2: Darstellung der auf einer `setvar`-Anweisung basierenden Programmvarianten.

6.1.2.2 Monitor

Der *Monitor* ist für die Ausführung der Programmvarianten und das Erfassen der Leistungswerte verantwortlich. Hierzu startet der Monitor in jeder Tuning-Iteration eine Programmvariante und überwacht den Programmlauf.

Beim Passieren eines Messpunkts wird die Messfunktion aufgerufen. Innerhalb der Messbibliothek wird die Messung von einer statischen Methode durchgeführt, die in einer Schnittstelle definiert ist. Diese Methode ermittelt zur Laufzeit des Programms an jedem Messpunkt den entsprechenden Messwert und speichert diesen. Je nach gewähltem Leistungskriterium können zur Ermittlung eines Messwertes mehrere Messpunkte nötig sein. Die Bestimmung der Laufzeit eines Quelltextabschnittes benötigt beispielsweise zwei Messpunkte (die die jeweils aktuelle Zeit vor und nach dem zu messenden Quelltextabschnitt bestimmen), aus denen die Differenz gebildet wird. Die hierfür erforderliche Logik wird durch die entsprechende Messmethode zur Verfügung gestellt. Nach dem Programmlauf werden die gesammelten Messwerte in eine Protokolldatei geschrieben.

Alle Messmethoden sind in der Lage, Messungen im Kontext unterschiedlicher Ausführungsfäden zu trennen und separat zu speichern. Die Messmethoden sorgen außerdem dafür, dass nach dem Programmlauf für jeden Messpunkt genau ein Messwert in die Protokolldatei geschrieben wird. Die Messmethoden erlauben daher zur Laufzeit die Reduktion mehrerer gemessener Werte eines bestimmten Messpunktes auf einen einzigen.

Es ist zu beachten, dass die Messungen zur Laufzeit ausgeführt werden und daher Teil der Anwendung sind. Werden neue Messmethoden in die Bibliothek integriert, muss dafür Sorge getragen werden, dass die Routinen minimalen Einfluss auf die Ausführung

des Programms haben, um Verfälschungen der Messwerte zu vermeiden. Dies bedeutet, dass beispielsweise eine Messmethode zur Bestimmung der Laufzeit eine möglichst kurze Ausführungszeit besitzen sollte, damit diese beider Messung nicht ins Gewicht fällt. Ist dies nicht möglich, sollte die Laufzeit der Messmethode von der Gesamtlaufzeit des Programms abgezogen werden.

6.2 Implementierung von Atune-TA

Die Implementierung des Atune-TA-Konzepts basiert auf dem Microsoft .NET Frameworks [Micr09a], die durchgängig verwendete Programmiersprache ist C#. In der aktuellen Implementierung werden daher ausschließlich C#-Programme unterstützt.

Die beiden Hauptelemente des Systems sind die *Tunable Architecture Library* sowie der *TADL-Übersetzer*, auf die im Folgenden eingegangen wird. Abschließend wird mit der *Tunable Architecture Toolbox* ein Erweiterungspaket für die Entwicklungsumgebung *Microsoft Visual Studio 2008* [Micr09b] vorgestellt, das die gesamte Funktionalität der optimierbaren Architekturen in Visual Studio integriert.

6.2.1 Tunable Architecture Library

Die *Tunable Architecture Library (TALib)* bildet das Rückgrat der optimierbaren Architekturen. Der Aufbau der TALib orientiert sich im Wesentlichen am Sprachkonzept von TADL. Das Kernmodul `TACore` stellt alle erforderlichen Schnittstellen für Konnektoren und atomare Komponenten zur Verfügung.

Im Folgenden wird die softwaretechnische Umsetzung der atomaren Komponenten sowie das Implementierungsschema der TADL-Konnektoren dargelegt.

6.2.1.1 Implementierung Atomarer Komponenten

Die interne Repräsentation der atomaren Komponenten ist zunächst Gegenstand der Diskussion. Abbildung 6.3 zeigt ein vereinfachtes Klassendiagramm der Implementierung atomarer Komponenten. Demnach wird eine atomare Komponente durch die Schnittstelle `IProcessableItem` repräsentiert.

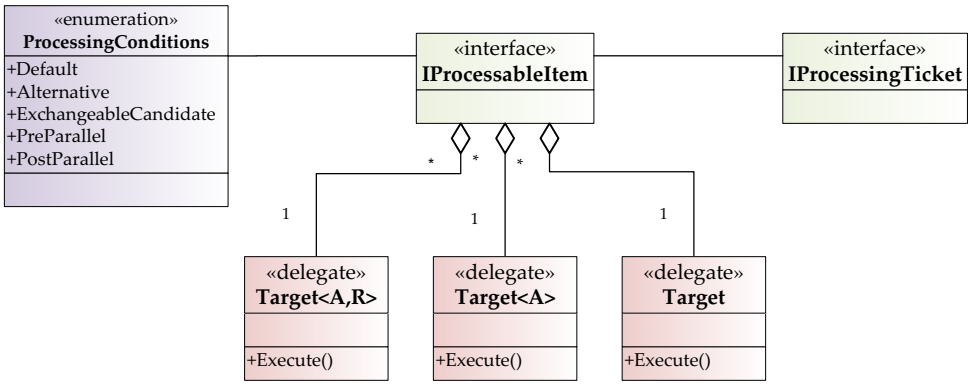


Abbildung 6.3: Vereinfachtes Klassendiagramm der Implementierung atomarer Komponenten im Kernmodul `TACore`.

Die Bindung an eine Methode des Programms wird über den generischen Delegierer `Target` (erweiterter, typsicherer Methodenzeiger in .NET) realisiert, den die `IProcessableItem`-Schnittstelle in drei Ausprägungen akzeptiert. `Target` kann an Methoden ohne Eingabeparameter und ohne Rückgabewert gebunden werden, `Target<A>` erwartet eine Methode ohne Rückgabewert, aber mit einem Eingabeparameter vom Typ `A`,

während `Target<A,R>` auf eine Methode mit Eingabeparameter vom Typ `A` und einem Rückgabewert vom Typ `R` verweist. Eine atomare Komponente kann also an Methoden mit unterschiedlichen Signaturen gebunden werden.

Neben der Bindung an eine Methode des Programms werden in `IProcessableItem` noch weitere Eigenschaften einer atomaren Komponente definiert. Beispielsweise wird über die Enumeration `ProcessingConditions` spezifiziert, ob die atomare Komponente eine Alternative zu einer anderen darstellt. Des Weiteren erhält jede atomare Komponente ein so genanntes *Processing Ticket*, das die Komponente innerhalb der optimierbaren Architektur eindeutig identifiziert. Dies ist insbesondere dann von Vorteil, wenn eine atomare Komponente in Form eines `IProcessableItem` zwischen Konnektoren ausgetauscht oder weitergereicht wird.

Der verarbeitende Konnektor einer atomaren Komponente liest die Eigenschaften von `IProcessableItem` aus und steuert dementsprechend den Verarbeitungsprozess. An Hand der Ein- und Ausgabetypen eines `IProcessableItem`, die durch den enthaltene `Target`-Delegierer definiert sind, führen Konnektoren auch zur Laufzeit Typüberprüfungen durch, um einen typsicheren Ablauf zu garantieren.

6.2.1.2 Implementierung der Konnektoren

Die TALib beinhaltet Module, die jeweils einen der TADL-Konnektoren implementieren. Beispielsweise stellt die TALib für den Tunable Pipeline-Konnektor ein entsprechendes Modul bereit, das die erforderliche Logik für ein optimierbares Fließband enthält. Zusätzlich implementiert jedes dieser Konnektormodule die Tuning-Parameter, die für den zugehörigen TADL-Konnektor definiert wurden (vgl. hierzu Tabelle 5.2 in Kapitel 5).

In Abbildung 6.4 ist stellvertretend für alle Konnektormodule der softwaretechnische Aufbau des `TunablePipelineConnector`-Moduls an Hand eines vereinfachten Klassendiagramms dargestellt. Es sei angemerkt, dass sich die Klassennamen in den einzelnen Konnektormodulen nur durch das jeweilige Präfix der Strategie unterscheiden (in diesem Fall `Pipeline`).

Um eine einheitliche Struktur der Konnektormodule und insbesondere deren Austauschbarkeit zu garantieren, implementiert ein Konnektormodul zentrale Schnittstellen, die im Kernmodul `TACore` definiert sind.

Die Klasse `Pipeline`, die die Schnittstelle `ITunableConnector` implementiert, fungiert als Einstiegspunkt des Konnektors. Die Methoden der Klasse registrieren die atomaren Komponenten in Form von `IProcessableItem`-Implementierungen, führen Typüberprüfungen durch und initialisieren die Verarbeitungsstrategie.

Die Klasse `PipelineStrategy`, eine Implementierung der Schnittstelle `IStrategy`, stellt die grundlegende Funktionalität zur Verarbeitung der atomaren Komponenten zur Verfügung. Im konkreten Fall des Tunable Pipeline-Konnektors werden die Fließbandstufen initialisiert sowie die Eingabedaten aufbereitet und in einer Startwarteschlange abgelegt. Nach Abschluss der Vorbereitungen wird die erste Fließbandstufe gestartet. Des Weiteren werden die in der Klasse `PipelinePolicy` deklarierten Tuning-Parameter geladen und deren konkrete Werte an entsprechenden Stellen des Moduls gesetzt.

Die gesamte Kommunikations- und Synchronisationslogik zur Steuerung der Fließbandstufen (oder allgemeiner zur Steuerung der Parallelisierungsstrategie) ist in der Klasse `PipelineScheduler` zusammengefasst, deren Definition die Schnittstelle `IScheduler` zur Verfügung stellt. Die Klasse beinhaltet die Logik zur feingranularen Verwaltung der parallelen Ausführungsfäden.

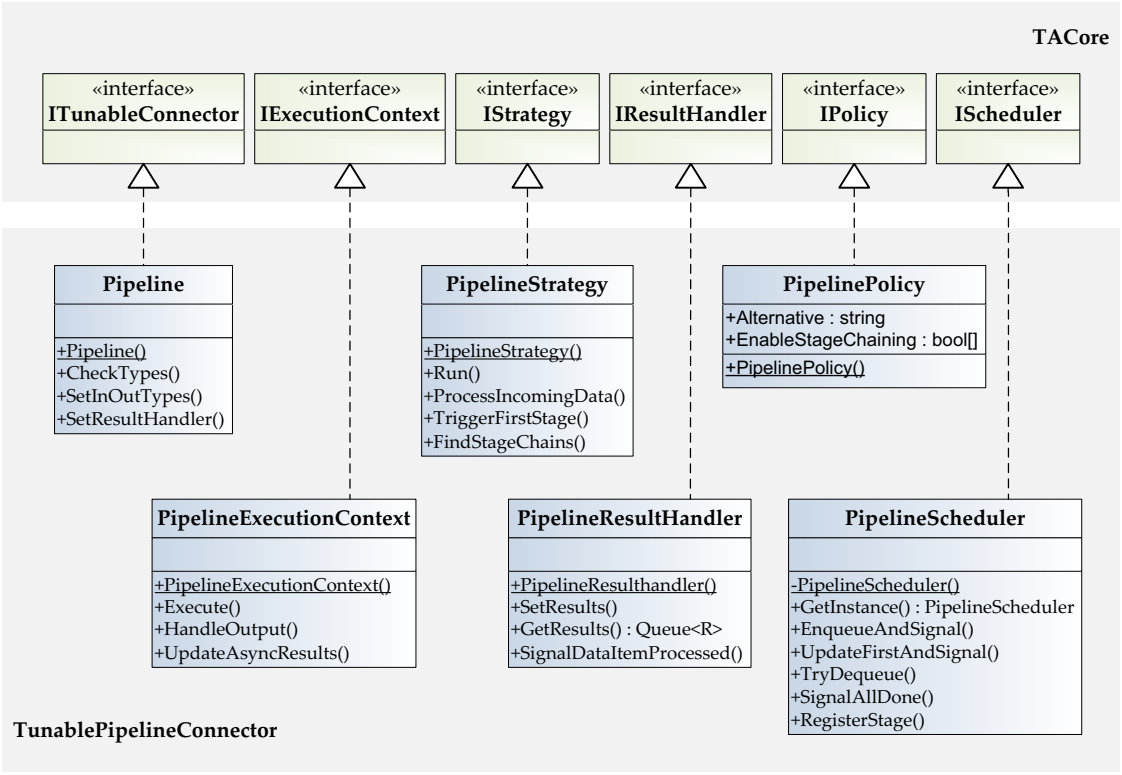


Abbildung 6.4: Vereinfachtes Klassendiagramm des Tunable Pipeline-Konnektors mit entsprechenden `TACore`-Schnittstellen.

Die Klasse `PipelineExecutionContext` bildet gewissermaßen den Rahmen zur tatsächlichen Ausführung der atomaren Komponenten. Hierzu gehört der kontrollierte Aufruf der Methoden des `Target`-Delegierers in den `IProcessableItem`-Implementierungen, die Übergabe der Argumente sowie die Entgegennahme der Rückgabedaten.

Die Implementierungen der übrigen Konnektoren entsprechen dem hier gezeigten `TunablePipelineConnector`-Modul.

6.2.1.3 Schachtelung von Konnektoren

Die `IProcessableItem`-Schnittstelle ermöglicht die Schachtelung von Konnektoren auf einfache Weise.

An den `Target`-Delegierer der `IProcessableItem`-Schnittstelle kann nicht nur eine Methode gebunden werden, die eine atomare Komponente implementiert, sondern auch eine Methode, die die Instanz eines Konnektoren startet.

Für ein bestimmtes Konnektormodul stellt somit ein geschachteltes Konnektormodul nichts anderes als eine konkrete Implementierung der `IProcessableItem`-Schnittstelle dar, die wie eine atomare Komponente ausgeführt wird.

6.2.1.4 Fehlerbehandlung

Ein Konnektormodul behandelt automatisch alle Fehler und Ausnahmen, die bei der Ausführung der atomaren Komponenten auftreten können. Ein Konnektormodul bleibt somit während der gesamten Laufzeit stabil und in einem konsistenten Zustand. Die Ausnahmen aller untergeordneter atomarer Komponenten werden abgefangen, gesammelt und als aggregierte kontrollierte Fehlermeldung ausgegeben.

Bei schwerwiegenden Fehlern wird das Programm abgebrochen. Tritt ein Fehler nur ein einziges Mal auf (d.h., arbeitet die atomare Komponente nach der Ausnahme normal weiter), wird die Verarbeitung fortgeführt, der Fehler protokolliert und nach Beendigung des Programms ausgegeben.

6.2.2 TADL-Übersetzer

Der TADL-Übersetzer erzeugt aus TADL-Skripts den nötigen Quelltext, um die sequentiellen Teile des Programms mit der TALib zu verbinden und somit das Programm gemäß der Architekturbeschreibung zu parallelisieren. Wie der Atune-IL-Zerteiler wurde auch der TADL-Zerteiler sowie Teile der semantischen Überprüfung mittels des ANTLR-Parsergenerators [antl09] realisiert.

Der Übersetzungsprozess erfolgt in drei Schritten:

1. Zunächst werden die Namen aller atomaren Komponenten im TADL-Skript mit den entsprechenden Methoden im Programm assoziiert. Dies geschieht durch Reflexion des kompilierten Programms.

   Hierbei werden alle Klassen berücksichtigt, die im benutzerdefinierten Namensraum des Programm deklariert sind. Schließlich werden die Namen der atomaren Komponenten mit denen aller öffentlichen Methoden in den reflektierten Klassen verglichen. Bei Übereinstimmung wird die Verbindung zwischen atomarer Komponente und ihrer implementierenden Methode hergestellt.

   Wurde die Methode mit demselben Namen in unterschiedlichen Klassen deklariert oder ist die Methode überladen, wird der Entwickler vor dem Übersetzungsvorgang des TADL-Skriptes gefragt, welche Methodenimplementierung als atomare Komponente verwendet werden soll.

2. Der Übersetzer transformiert das TADL-Skript in eine interne Zwischendarstellung und überprüft die Typkonsistenz der atomaren Komponenten hinsichtlich ihrer übergeordneten Konnektoren sowie die semantische Korrektheit der Architektur.

3. Schließlich generiert der TADL-Übersetzer den erforderlichen Quelltext und integriert diesen in das fertige ausführbare Programm.

Das Implementierungskonzept der Quelltextgenerierung wird im Folgenden erklärt.

6.2.2.1 Tuning-Hüllklassen

Der generierte Quelltext, der die nötige Funktionalität zur Verbindung der atomaren Komponenten mit der TALib bereitstellt, ist in Form so genannter *Tuning-Hüllklassen* organisiert. Eine Tuning-Hüllklasse repräsentiert eine Klasse, die eine Instanz eines Konnektormoduls der TALib initialisiert sowie Zugriffsmethoden zu dieser Instanz implementiert.

Der TADL-Übersetzer generiert für jeden Konnektor im TADL-Skript eine entsprechenden Tuning-Hüllklasse. Beispielsweise wird ein `TunablePipeline`-Konstrukt in eine Tuning-Hüllklasse übersetzt, die den Zugriff auf eine Instanz des `TunablePipeline-Connector`-Moduls der TALib steuert.

Hierzu implementiert eine Tuning-Hüllklasse Methoden zur Bewältigung von Ein- und Ausgabedaten und stellt eine `Run()`-Methode zur Verfügung, die die umschlossene Instanz des Konnektormoduls ausführt. Den Einstiegspunkt zur Ausführung der gesamten

optimierbaren Architektur repräsentiert die `Run()`-Methode der Tuning-Hüllklasse, der den Wurzel-Konnektor der Architektur umschließt.

Neben der Zugriffslogik sowie der Anbindung der atomaren Komponenten an die TA-Lib deklariert eine Tuning-Hüllklasse alle vom umschlossenen Konnektormodul bereitgestellten Tuning-Parameter.

Des Weiteren enthält eine Tuning-Hüllklasse alle erforderlichen Tuning-Instruktionen in Form von Atune-IL-Instrumentierungen. Der TADL-Übersetzer erzeugt also nicht nur C#-Quelltext, sondern auch entsprechende Atune-IL-Anweisungen. Der TADL-Übersetzer generiert somit ein paralleles, auto-tuning-fähiges Programm. Atune-IL fungiert in diesem Zusammenhang als vollintegriertes Werkzeug und repräsentiert eine Laufzeitumgebung für den Auto-Tuner. Dies begründet die Entwurfsentscheidung, Atune-IL als universelle Tuning-Sprache mit möglichst einfacher Syntax zu entwickeln, um die automatische Generierung der Instrumentierungen zu ermöglichen.

Zu den automatisch generierten Atune-IL-Instrumentierungen zählen insbesondere:

- Jeder Tuning-Parameter wird mit einer `setvar`-Anweisungen instrumentiert, der Wertebereich (`values`), ggf. Abhängigkeiten (`depends`), Skalierung (`scale`) sowie Parameterkontext (`context`) definiert.
- Die `Run()`-Methode jeder Tuning-Hüllklasse wird mittels `startblock`- und `endblock`-Anweisungen in einen Tuning-Block gehüllt, um die Hierarchie der Architektur nachzubilden. Hierbei wird über das `type`-Schlüsselwort der `startblock`-Anweisung der Typ des zu Grunde liegenden Konnektors angegeben, um dem Auto-Tuner die erforderlichen Kontextinformationen zu liefern.
- Der konkrete Aufruf des Konnektormoduls wird mit Messpunkten versehen, die die Ausführungszeit bestimmen.

In Anhang B wird an Hand eines Quelltextbeispiels der Aufbau einer Tuning-Hüllklasse detailiert erklärt.

6.2.2.2 Ablauf der Quelltextgenerierung

Eine Tuning-Hüllklasse stellt für ein Konnektormodul der TALib die gesamte Tuning-Infrastruktur bereit und bindet die Instanz in die übrige Architektur ein. Die für den Programmierer sichtbare Implementierung einer optimierbaren Architektur besteht also aus einer Menge assoziierter Tuning-Hüllklassen.

Abbildung 6.5 fasst den Ablauf der Quelltextgenerierung durch den TADL-Übersetzer noch einmal zusammen.

Für jeden TADL-Konnektor wird eine Tuning-Hüllklasse generiert, der diesen im Rahmen der Architekturimplementierung repräsentiert. Die Tuning-Hüllklasse initialisiert eine konkrete Instanz des geforderten Konnektormoduls der TALib und stellt die Zugriffsmethoden bereit. Der TADL-Übersetzer bindet die Deklarationen der atomaren Komponenten an Programm-Methoden (im Schaubild als *AC-Methoden* bezeichnet) und definiert diese Bindungen in der Tuning-Hüllklasse. Schließlich wird die Tuning-Hüllklasse mit Atune-IL-Anweisungen instrumentiert.

6.2.3 Tunable Architecture Toolbox

Das gesamte Konzept der optimierbaren Architekturen, insbesondere die Funktionalität des TADL-Übersetzers, wurde im Rahmen der prototypischen Implementierung vollständig in *Microsoft Visual Studio 2008* integriert.

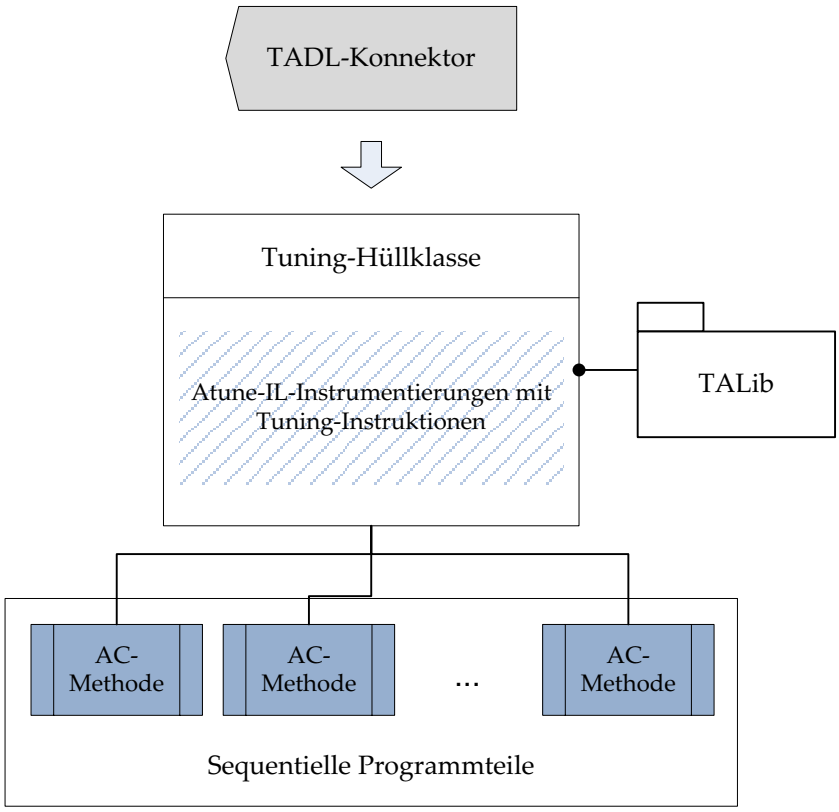


Abbildung 6.5: Schaubild der Quelltextgenerierung des TADL-Übersetzers.

Die unter dem Namen *Tunable Architecture Toolbox* entwickelte Erweiterung ermöglicht den Entwurf sowie die Implementierung einer optimierbaren Architektur aus dem gerade aktuellen Visual Studio-Projekt heraus. Ein Programm, das unter den konzeptionellen Vorgaben der atomaren Komponenten entworfen wurde, kann mit Hilfe der Tunable Architecture Toolbox nach Anfertigung eines TADL-Skriptes automatisiert in ein paralleles und optimierbares Programm transformiert werden.

Das TADL-Skript muss als Projektdatei mit der Endung `.tadl` abgespeichert werden. Anschließend kann die Tunable Architecture Toolbox gestartet und das TADL-Skript verarbeitet werden. Hierbei stehen dem Entwickler zwei Modi zur Verfügung. Im *Vorschau-Modus* wird die zu implementierende Architektur graphisch darstellt sowie Änderungen an den Tuning-Parametern und anderen Einstellungen zugelassen. Der *Implementierungs-Modus* hingegen erzeugt umgehend die erforderlichen Tuning-Hüllklassen, speichert den Quelltext in separaten Dateien und bindet diese in das aktuelle Projekt ein. Das Programm ist auch nach der Transformation sofort lauffähig.

Abbildung 6.6 zeigt einen Screenshot der Tunable Architecture Toolbox im Vorschau-Modus nach der Verarbeitung eines TADL-Skriptes. Die grafische Darstellung der Architektur ist im linken Bereich des Fensters zu sehen, der rechte Teil zeigt das Verarbeitungsprotokoll.

In Abbildung 6.7 ist der *Solution Explorer* von Visual Studio vergrößert dargestellt. Der TADL-Übersetzer hat im separaten Unterverzeichnis `TuningWrappers` die Quelltextdateien der Tuning-Hüllklassen (in diesem Fall bestand die Architekturbeschreibung aus einem einzigen Konnektor) angelegt und in das Projekt `DesktopSearch` eingebunden.

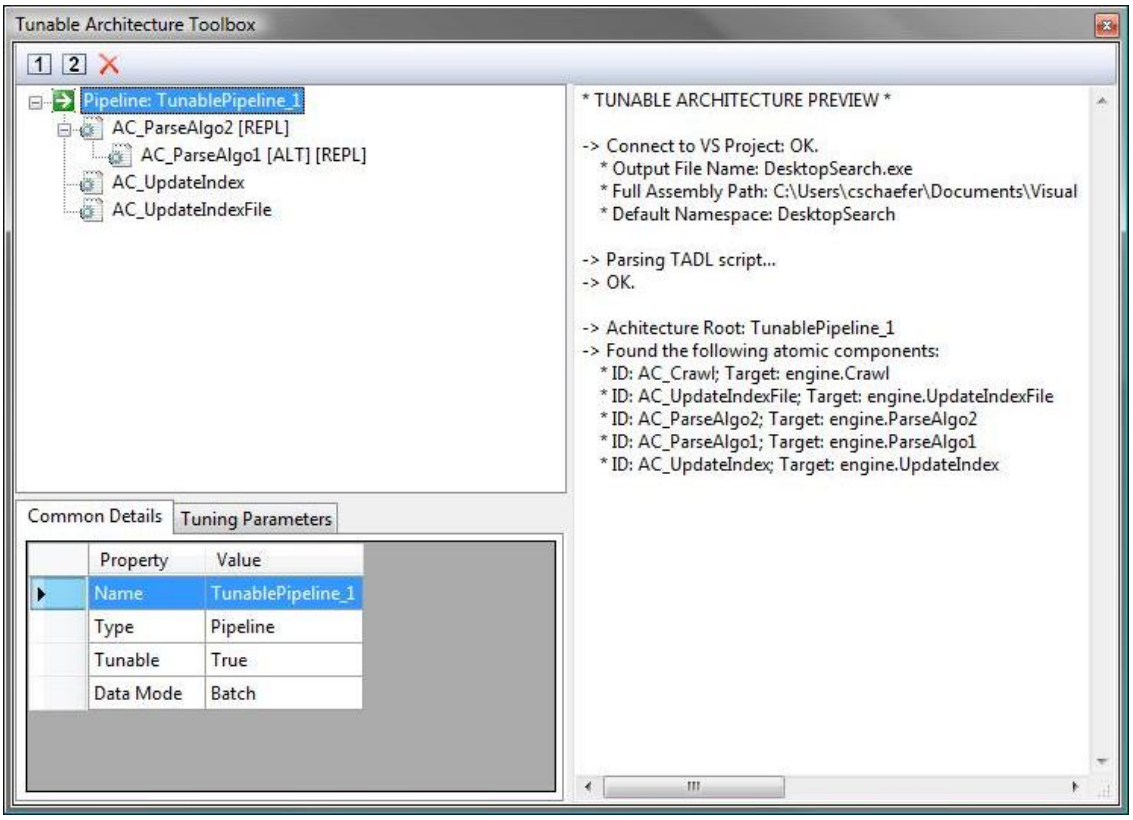


Abbildung 6.6: Screenshot der Tunable Architecture Toolbox im Vorschau-Modus nach der Analyse eines TADL-Skriptes.

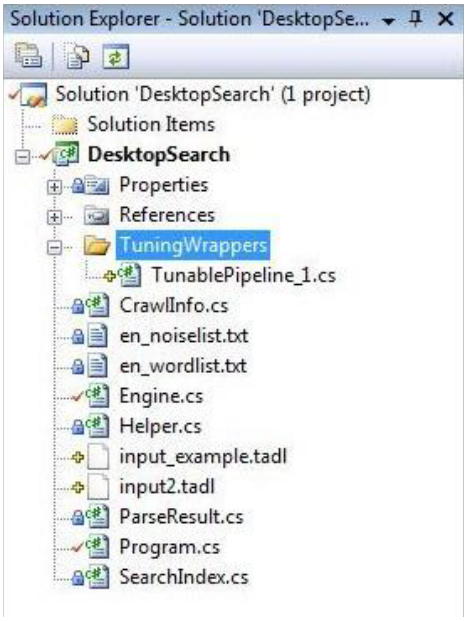


Abbildung 6.7: Screenshot des *Solution Explorer* von Visual Studio nach dem Hinzufügen Quelltextdateien für die Tuning-Hüllklassen.

6.3 Implementierung von Atune-OPT

Atune-OPT wurde als Einzelanwendung entworfen und in C# entwickelt. Charakteristisch für den Prototypen ist seine Modularität, die es ermöglicht, unterschiedliche Komponenten auszutauschen.

Atune-OPT fungiert als ein Rahmenwerk, das die grundlegende Funktionalität eines Auto-Tuners sowie die Modularisierungslogik bereitstellt und für austauschbare Komponenten entsprechende Anbindungsmöglichkeiten bietet. Abbildung 6.8 verdeutlicht den Sachverhalt.

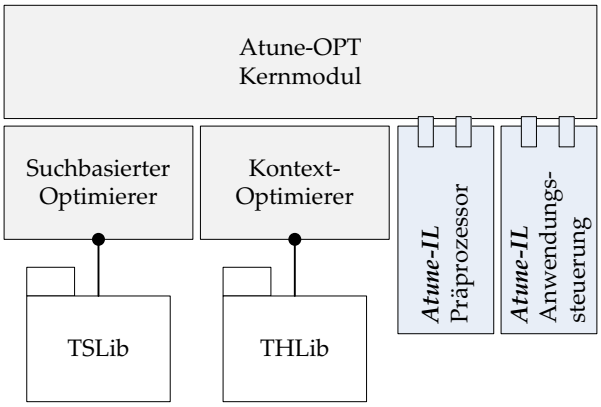


Abbildung 6.8: Modularer Aufbau des Atune-OPT-Prototyps.

Die Programmanbindung erfolgt über die Ankopplung des Atune-IL-Präprozessors sowie der Atune-IL-Anwendungssteuerung (siehe Abschnitt 6.1.1 bzw. Abschnitt 6.1.2) in Form externer Komponenten. Die Module für suchbasierte sowie kontextbasierte Optimierung können die erforderlichen Suchalgorithmen bzw. Tuning-Heuristiken zur Laufzeit aus externen Bibliotheken laden (*Tuning Strategy Library (TSLib)* und *Tuning Heuristics Library (THLib)*).

Die dynamische Austauschbarkeit wird durch so genannte .NET-Assemblies realisiert, die zur Laufzeit mittels später Bindung geladen werden können. Jede austauschbare Komponente – beispielsweise jeder Suchalgorithmus – muss daher in einer separaten .NET-Assembly implementiert werden. Um einheitliche Komponenten und damit deren Austauschbarkeit zu garantieren, stellt das Atune-OPT-Kernmodul entsprechende Schnittstellen zur Verfügung.

6.3.1 Optimierungsprozess

Der Optimierungsprozess des Prototyps von Atune-OPT entspricht im Wesentlichen dem konzeptionellen Prozess, den in Kapitel 5, Abschnitt 5.4 vorgestellt wurde.

Abbildung 6.9 zeigt den Ablauf der Optimierung eines parallelen Programms aus Sicht der Implementierung.

Der Prozess startet mit einem parallelen Programm, das über Atune-IL-Instrumentierungen verfügt. Während der Initialisierung lädt Atune-OPT alle erforderlichen Einstellungen aus einer zentralen Konfigurationsdatei und übergibt den Quelltext des Programms an den Atune-IL-Präprozessor. Dieser analysiert den Quelltext des Programms sowie die Instrumentierungen (vgl. hierzu Abschnitt 6.1.1) und übergibt der Initialisierungskomponente von Atune-OPT schließlich die aufbereiteten Tuning-Instruktionen.

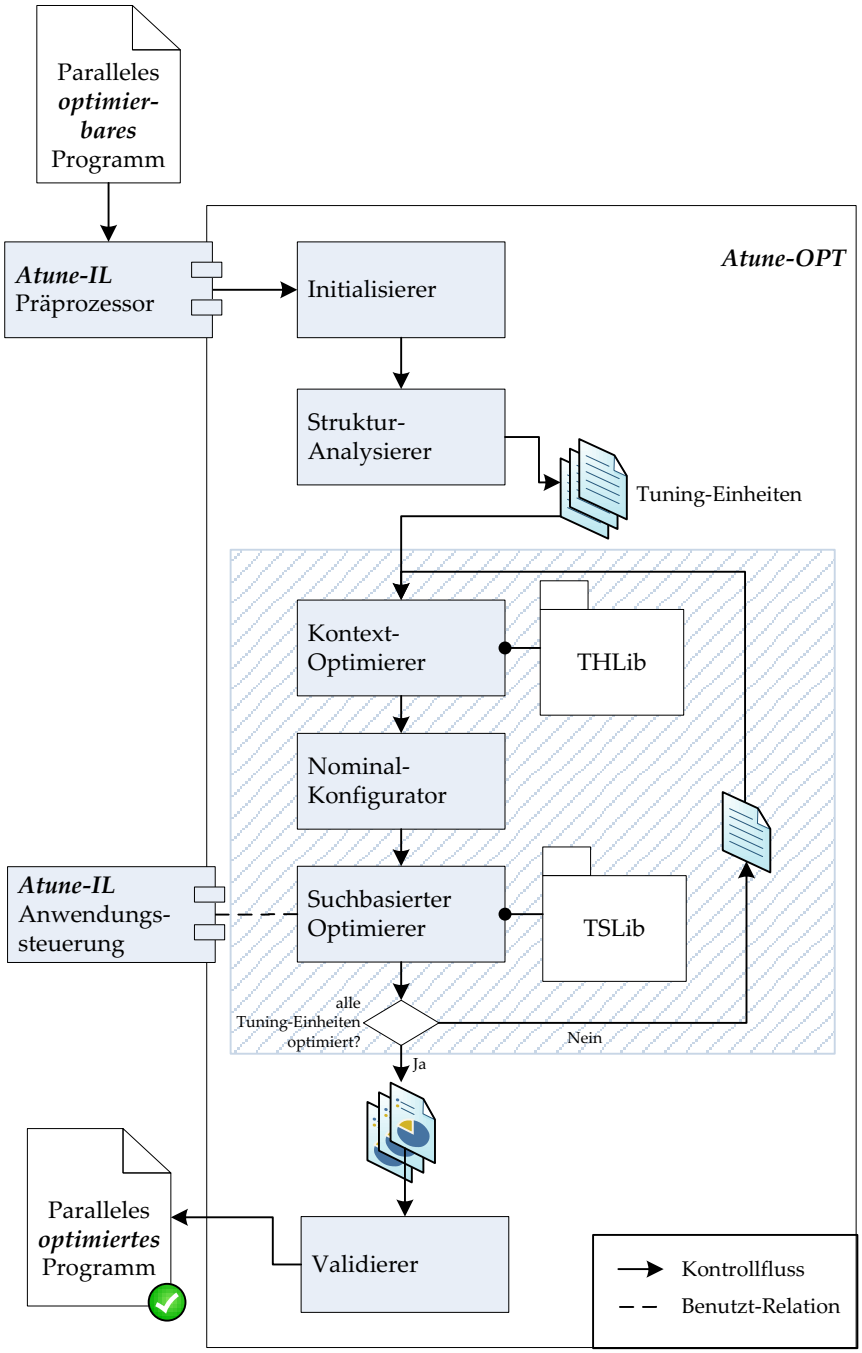


Abbildung 6.9: Ablauf des Optimierungsprozesses des Atune-OPT-Prototyps.

Im nächsten Schritt startet der *Struktur-Analysierer* den Prozess Suchraumpartitionierung. Der Analysierer identifiziert mögliche Tuning-Einheiten an Hand der Struktur der Tuning-Blöcke und übergibt die Tuning-Einheiten an die folgenden Komponenten, die von jeder Tuning-Einheit einmal durchlaufen werden.

Der *Kontext-Optimierer* überprüft eine Tuning-Einheit auf bekannte Konnektoren und wendet ggf. eine entsprechende Tuning-Heuristik an, die zur Laufzeit aus der THLib geladen wird. Die THLib ermöglicht das Einbinden zusätzlicher Tuning-Heuristiken für weitere Konnektoren oder andere Parallelisierungsstrategien. Die Aktivierung neuer Heuristiken erfolgt über die zentrale Konfigurationsdatei, während die Verwendung automatisch geschieht, sobald ein entsprechender Konnektor identifiziert wurde.

Der *Nominal-Konfigurator* ermittelt für alle nominal-skalierten Parameter der Tuning-Einheit, die nicht durch eine Tuning-Heuristik ausgeschlossen werden konnten, einen möglichst guten Wert. Wie im Konzept-Kapitel erläutert, erfolgt die Konfiguration der nominal-skalierten Parameter durch ein zufallsbasiertes Verfahren.

Der *suchbasierte Optimierer* nimmt sich schließlich all jener Parameter an, die in keinem der vorherigen Schritte verarbeitet werden konnten. Er wählt je nach Charakteristik der Tuning-Einheit einen Suchalgorithmus aus (siehe hierzu die konzeptionellen Grundlagen in Kapitel 5, Abschnitt 5.4.5.4). Alle zur Verfügung stehenden Suchalgorithmen sind als austauschbare .NET-Assemblies in der TSLib zusammengefasst. Der benötigte Suchalgorithmus wird zur Laufzeit geladen und eingebunden.

Es können auch weitere Suchalgorithmen der TSLib hinzugefügt werden. Über die zentrale Konfigurationsdatei kann in diesem Fall festgelegt werden, welche Suchalgorithmen Atune-OPT verwenden soll. Hinzugefügte Algorithmen sind von der automatischen Auswahl ausgeschlossen, da Atune-OPT über die Eigenschaften benutzerdefinierter Suchstrategien keine Aussage treffen kann.

Nach der Auswahl eines Suchalgorithmus startet der Auto-Tuning-Zyklus, wie wir ihn bereits im Zusammenhang mit Atune-IL beschrieben haben. Der suchbasierte Optimierer errechnet eine Parameterkonfiguration und übergibt diese an den Programm-Controller des Atune-IL-Backend. Der Programm-Controller erzeugt eine neue Programmvariante, führt diese aus, erfasst die Leistungswerte und übergibt diese als Feedback an den Optimierer. Diese Iteration wird so lange durchgeführt, bis der Suchalgorithmus konvergiert.

Nachdem alle Tuning-Einheiten optimiert wurden, überprüft der *Validierer* an Hand der Leistungswerte, ob sich Tuning-Einheiten gegenseitig beeinflusst haben. Wurden alle Tuning-Einheiten für gültig befunden, aggregiert Atune-OPT die Ergebnisse und ermittelt daraus eine optimale Parameterkonfiguration für das gesamte Programm.

6.4 Entwicklung eines parallelen optimierten Programms

Aus der prototypischen Implementierung aller drei Teilkonzepte dieser Arbeit sind Werkzeuge entstanden, die den Entwurfs- und Entwicklungsprozess eines parallelen Programms mit anschließender automatischer Performanzoptimierung signifikant unterstützen. In diesem Abschnitt werden die Schritte dargelegt, die ein Software-Entwickler durchführen muss, um ein paralleles optimiertes Programm zu erhalten.

Abbildung 6.10 skizziert den Prozess und stellt die nötigen Schritte aus der Sicht des Entwicklers dar, an denen wir uns im Folgenden orientieren.

1. Die funktionalen sequentiellen Aufgaben des Programms, die später als atomare Komponenten deklariert werden sollen, müssen zunächst in Form einzelner Methoden implementiert werden (im Schaubild als *AC-Methoden* bezeichnet).
2. In den Schritten 2 bis 4 kommt Atune-TA zum Einsatz. Zunächst wird die optimierbare Architektur mit Hilfe eines TADL-Skripts beschrieben. Durch die Deklaration atomarer Komponenten wird die Bindung zu den implementierten Methoden hergestellt.
3. Der TADL-Übersetzer wird gestartet und erhält 1 und 2 als Eingabe.
4. Nach Verarbeitung des TADL-Skripts erzeugt der TADL-Übersetzer die erforderlichen Tuning-Hüllklassen, die in separaten Quelltextdateien gespeichert werden. Jede Tuning-Hüllklasse verwaltet den Zugriff auf das entsprechende Konnektormodul der TALib und beinhaltet die automatisch generierten Atune-IL-Anweisungen.

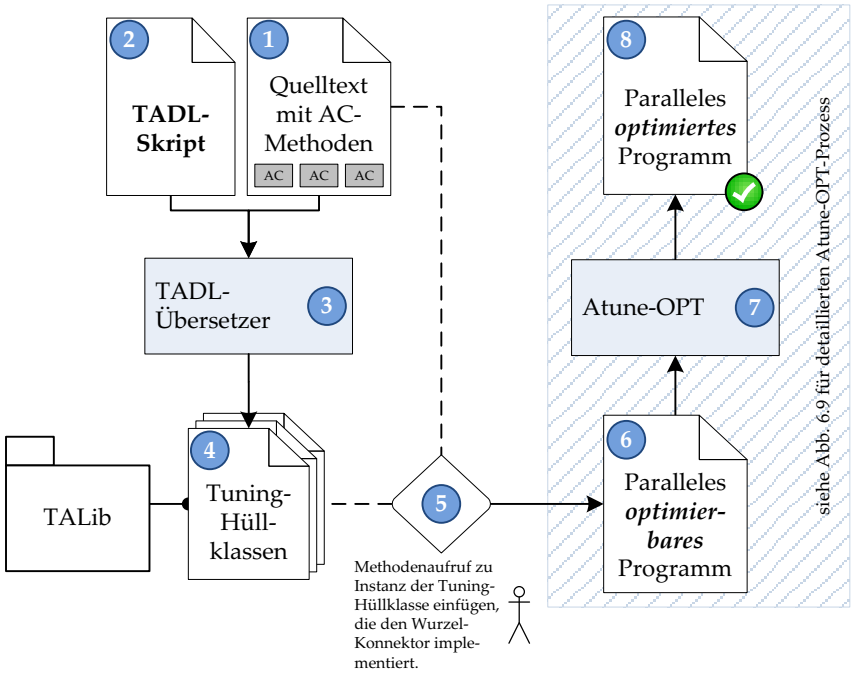


Abbildung 6.10: Entwurfs- und Entwicklungsprozess eines parallelen optimierbaren Programms.

5. Der Software-Entwickler fügt in das Programm einen Aufruf der Instanz derjenigen Tuning-Hüllklasse ein, der den Wurzel-Konnektor der Architektur implementiert. Dieser Aufruf wird üblicherweise in der `main()`-Methode des Programms platziert.

6. Das Ergebnis von Schritt 1 bis 5 ist ein ausführbares paralleles Programm, das zwar auto-tuning-fähig, jedoch noch nicht optimiert ist. Diesen Zustand des parallelen Programms bezeichnen wir als Rohversion, die nun durch den Optimierungsprozess für die Verwendung auf einer bestimmten Hardware-Plattform geprägt und konfiguriert wird.

7. Atune-OPT optimiert das parallele Programm hinsichtlich eines Leistungsmerkmals. Hierzu koppelt sich der Auto-Tuner über die Atune-IL-Instrumentierungen der Tuning-Hüllklassen an das Programm und testet die errechneten Parameterkonfigurationen.

   Es ist zu beachten, dass dieser Schritt den gesamten Optimierungsprozess von Atune-OPT zusammenfasst, der in Abbildung 6.9 dargestellt ist.

8. Das Ergebnis von Schritt 7 ist ein paralleles Programm, das für eine bestimmte Zielplattform optimiert ist. Wird das Programm auf eine andere Plattform portiert, müssen lediglich die Schritte 6 und 7 wiederholt werden.

Die obige Prozessbeschreibung verdeutlicht die Verzahnung der Konzepte dieser Arbeit zu einem umfassenden Ansatz. Der Entwickler wird beim Entwurf, der Implementierung sowie der Optimierung eines parallelen Programms unterstützt.

6.5 Zusammenfassung

In diesem Kapitel wurde die Implementierung der drei Teilkonzepte dieser Arbeit vorgestellt und die zugehörigen Prototypen im Einzelnen besprochen. Die Werkzeuge setzen einen Großteil der konzeptionellen Arbeit um und demonstrieren, dass die Kombination der einzelnen Aspekte auch im praktischen Einsatz einen wesentlichen Beitrag zur Vereinfachung des Entwurfs- und Entwicklungsprozesses paralleler Programme sowie deren Optimierung leistet.

7. Evaluation

Dieses Kapitel befasst sich mit der Bewertung der in dieser Arbeit Arbeit vorgestellten Konzepte. Hierzu haben wir verschiedene Fallstudien durchgeführt.

Mit Hilfe der Ergebnisse der Fallstudien soll zum Einen gezeigt werden, dass die Ansätze die Erwartungen hinsichtlich Funktionalität und Nutzen erfüllen. Zum Anderen werden die in Kapitel 2, Abschnitt 2.3 aufgestellten Thesen belegt.

7.1 Fallstudien

Zunächst werden die einzelnen Fallstudien vorgestellt sowie die jeweilige Testkonfiguration beschrieben. Jede Fallstudie befasst sich mit der Parallelisierung und Optimierung einer Anwendung, wobei als Ausgangspunkt sowohl neu entwickelte als auch bereits existierende sequentielle Anwendungen betrachtet werden.

Bei der Wahl geeigneter Anwendungen wurde darauf geachtet, dass der Umfang der Programme von softwaretechnischer Relevanz ist, dass das Parallelisierungspotential im Allgemeinen für eine Reihe von Applikationen repräsentativ ist und dass die Anwendungsbereiche der Programme ein möglichst breites Spektrum abdecken. Da der Fokus dieser Arbeit auf meist größeren parallelen Applikationen liegt, kommen einzelne Algorithmen oder kleine numerische Programme als Gegenstände einer Evaluation eher am Rande in Frage, werden aber dennoch in zwei Fallbeispielen betrachtet.

Es sei angemerkt, dass die Auswahl der Anwendungen keinesfalls einen Anspruch auf Vollständigkeit erheben soll und kann. Es sind gewiss weitere Applikationen aus anderen Anwendungsbereichen denkbar, die als Gegenstand für Studien geeignet wären. Für eine Arbeit wie diese sollte jedoch eine überschaubare Auswahl getroffen werden.

Für die Spezifikation der parallelen Architekturen der Applikationen verwenden wir naheliegenderweise die Beschreibungssprache TADL. Die auf diese Weise beschriebenen Programmstrukturen liegen im Wesentlichen allen Experimenten zu Grunde. Bei Abweichungen wird an gegebener Stelle darauf hingewiesen.

7.1.1 MetaboliteID (MID)

Die *Masshunter MetaboliteID Software (MID)* der Firma *Agilent Technologies* [Agil08] repräsentiert eine große kommerzielle Anwendung zur Analyse biologischer Daten.

MetaboliteID identifiziert Metaboliten in Massenspektrogrammen. Unter Metabolismus wird eine Folge chemischer Reaktionen verstanden, die in den Zellen lebender Organismen ablaufen. Metaboliten stellen die Zwischen- und Endprodukte dieser Reaktionsfolge dar. Der Prozess der Metaboliten-Identifikation wird beispielsweise als essentielle Methode bei der Erprobung neuer Medikamente eingesetzt. Hierzu werden Proben von Körperflüssigkeiten mit Hilfe der Massenspektroskopie verglichen. Die Kontrollprobe wird vor der Einnahme des Medikaments entnommen, die Vergleichsproben zu bestimmten Zeitpunkten nach der Einnahme.

MetaboliteID vergleicht die Massenspektrogramme unter Verwendung einer Sequenz unterschiedlicher Algorithmen (die abgekürzten Bezeichnungen lauten *SC*, *EIC*, *ADC*, *UV*, *BTL*, *IPM*, *PCS*, *FMSC*, *FPM* und *MFA*). Jeder Algorithmus führt den Identifikationsprozess auf eine bestimmte Weise und basierend auf unterschiedlichen Charakteristiken durch, so dass möglichst viele der vorhandenen Metaboliten entdeckt werden.

MetaboliteID existierte bereits als sequentielles Programm, das wir für die Fallstudie zunächst manuell parallelisiert haben. Später wurde dann mittels TADL eine identische parallele Architektur entworfen und durch Atune-TA implementiert. Das entsprechende TADL-Skript ist in Listing 7.1 aufgeführt.

```
SequentialComposition MID {
    RunPreProcessing,
    TunableForkJoin {
        RunSC,
        RunEIC
    },
    TunableForkJoin {
        RunADC,
        RunUV,
        RunMDF,
        RunBTL_IPM,
        SequentialComposition {
            RunPCS_FMSC,
            TunableForkJoin
                [input:FPMInput,MFAInput;
                output:FPMOutput,MFAOutput] {
                RunFPM[replicable],
                RunMFA[replicable]
            }
        }
    },
    RunPostProcessing
}
```

Listing 7.1: TADL-Skript der parallelen MetaboliteID-Version.

Alle Methoden, die die oben genannten Algorithmen implementieren, wurden als atomare Komponenten deklariert. Die Architektur besteht aus einer Sequenz von zwei atomaren Komponenten und zwei Fork/Join-Sektionen, von denen die letzte geschachtelt ist. Zwei der Algorithmen sind replizierbar. Diese kompakte Darstellung definiert fünf parallele Sektionen und nutzt damit Parallelität auf drei unterschiedlichen Programmebenen aus.

Die parallele Struktur von MetaboliteID haben wir gemeinsam mit den verantwortlichen Software-Entwicklern von Agilent Technologies in Deutschland erarbeitet, um auf deren

Wissen und Erfahrung zurückgreifen zu können und um schließlich sicherzustellen, dass die Funktionalität des Programms trotz der strukturellen Eingriffe erhalten bleibt.

Die für die Fallstudie verwendeten Eingabedaten wurden in einer 1 GB großen Datei mit proprietärem Format zusammengefasst.

7.1.2 GrGen.NET (GrGen)

GrGen.NET ist das aktuell schnellste verfügbare Graphersetzungssystem [GeBl08]. Bei der Graphersetzung handelt es sich um die regelbasierte Veränderung eines Graphen. Der Graph, auf dem diese Veränderung stattfindet, wird als Arbeitsgraph bezeichnet. Die Regeln bestehen aus einer linken und einer rechten Seite, wobei die linke Seite ein Muster vorgibt, das im Arbeitsgraphen gefunden werden soll. Wurde die linke Seite im Arbeitsgraphen gefunden, werden entsprechend der rechten Seite Änderungen am Graphen vorgenommen.

Für die Fallstudie haben wir mit GrGen.NET den biologischen Prozess der Genexpression an Hand der DNA des Escherichia Coli-Bakteriums (E.coli) simuliert [ScGS09]. Genexpression bezeichnet den Prozess der Bildung eines Proteins aus einem Gen. Der Prozess besteht aus zwei wesentlichen Schritten: Transkription und Translation. Um ein Protein zu synthetisieren, muss die Nucleotidsequenz in Form eines RNA-Moleküls vorliegen. Das Erzeugen dieser RNA-Moleküle durch Umschreiben der DNA wird Transkription genannt. Die erste von drei erzeugten RNA-Arten ist die *messenger-RNA (mRNA)*. Sie ist die Abschrift eines Gens und stellt somit den Bauplan für ein Protein dar. Der Vorgang, bei dem an Hand der mRNA ein entsprechendes Protein synthetisiert wird, heißt Translation.

Die Fragestellungen bei der Parallelisierung sind nun, unter welchen Bedingungen Suchvorgänge gleichzeitig durchgeführt und wie Ersetzungsschritte innerhalb des Graphen mit mehreren Ausführungsfäden beschleunigt werden können.

Zunächst haben wir unter Berücksichtigung graphentheoretischer Konzepte manuell eine parallele Version von GrGen.NET entwickelt [Schi08]. Hierbei wurden zwei wesentliche Leistungsengpässe identifiziert, bei denen jedoch massive Datenparallelität ausgenutzt werden kann.

Listing 7.2 zeigt das entsprechende TADL-Skript. Die beiden leistungskritischen Methoden müssen zwar nach einander ausgeführt werden, können aber als replizierbare atomare Komponenten `PromoterSearch` und `RnaPoly` deklariert werden.

```
SequentialComposition GrGen
    [input:PromoterInput,RnaPolyInput;
    output:PromoterOutput,RnaPolyOutput] {
    PromoterSearch[replicable],
    RnaPoly[replicable]
}
```

Listing 7.2: TADL-Skript für parallele GrGen.NET-Version.

Das entsprechende DNA-Modell zur Simulation der Genexpression wurde durch einen Arbeitsgraphen mit über 9 Millionen Knoten repräsentiert.

7.1.3 Desktopsuche (DS)

Das Programm zur Realisierung einer Desktopsuche wurde bereits in Kapitel 5, Abschnitt 5.3.4.2 als Entwurfsbeispiel für eine mit TADL beschriebe optimierbare Architektur vorgestellt.

Die Anwendung wurde von Grund auf implementiert. Das Programm indiziert an Hand einer Liste von Schlüsselwörtern eine Menge an Textdateien. Anschließend können Anfragen gegen den Index abgesetzt werden, um alle Dateien zu erhalten, in denen die Wörter der Anfrage enthalten sind. Für die Indizierung stehen zwei unterschiedliche String-Matching-Algorithmen zur Verfügung (herkömmlicher String-Vergleicher sowie KMP-Algorithmus, vgl. Kapitel 5, Abschnitt 5.3.4.2), deren geeignete Wahl durch einen Tunable Alternative-Konnektor dem Auto-Tuner überlassen wird.

Listing 7.3 führt noch einmal das zugehörige TADL-Skript aus Kapitel 5 auf.

```
TunableAlternative DesktopSearchAlternatives {
    SequentialComposition DesktopSearch1
        [input:null,GetKeywords;
        output:null,ShowResults] {
        TunablePipeline
            [source:Crawl;
            sink:CreateIndexFile] {
            TunableAlternative {
                StringSearch1[replicable],
                StringSearch2[replicable]
            },
            UpdateIndex[replicable]
        },
        Query
    },

    SequentialComposition DesktopSearch2
        [input:null,GetKeywords;
        output:null,ShowResults] {
        TunableProducerConsumer
            [source:Crawl;
            sink:CreateIndexFile] {
           TunableAlternative {
                StringSearch1[replicable],
                StringSearch2[replicable]
            },
            UpdateIndex[replicable]
        },
        Query
    }
}
```

Listing 7.3: Architekturbeschreibung der parallelen Desktopsuche (aus [ScPT10]).

Speziell für die Leistungsmessungen (siehe Abschnitt 7.2.4) wurde neben den beiden oben genannten Varianten eine dritte separate Variante der Desktopsuche implementiert, bei der der String-Matching-Algorithmus durch einen einfachen Scan-Algorithmus ausgetauscht wurde, der auf dem Zeichenstrom der geöffneten Datei arbeitet. Listing 7.4 zeigt die entsprechende Architekturbeschreibung.

Bei dieser Variante wurde bewusst auf `Query` und `CreateIndexFile` verzichtet, da es darauf ankam, das Parallelisierungspotential des sehr schnellen Scan-Algorithmus zu ermitteln. Da bei dieser Implementierung jede Instanz von `UpdateIndex` auf einer eigenen Indexdatenstruktur arbeitete, um Synchronisationsaufwand zu sparen, musste anschließend die atomare Komponente `MergeIndex` eingefügt werden, um aus den einzelnen Teilen den gesamten Index aufzubauen.

```
SequentialComposition Indexer
    TunableProducerConsumer
        [source:Crawl;
         sink:null] {
         Scan[replicable]
         UpdateIndex[replicable]
        },
    MergeIndex
    }
}
```

Listing 7.4: Architekturbeschreibung des parallelen Indizierers.

Da sich die beiden ersten sowie die letzte separate Variante hinsichtlich ihrer parallelen Architektur sehr ähnlich sind, wird die dritte Variante nur bei den Leistungsmessungen erwähnt, da sich hier interessante Unterschiede zeigten.

Die Eingabedaten für den Leistungsbenchmark dieser Fallstudie bestanden für alle Varianten aus 10.700 ASCII-Textdateien mit einer Größe zwischen 9 und 613 KB.

7.1.4 Videoverarbeitung (Video)

Wie die Desktopsuche wurde auch das Programm zur Verarbeitung von Videos bereits in Kapitel 5 als Entwurfsbeispiel eingeführt und für die Verwendung als Fallstudie von Grund auf neu entwickelt.

Das Programm bearbeitet ein AVI-Video (*Audio Video Interleave*-Format), indem es vier Filter (in Form der Filter-Methoden `Crop()`,`Oilpaint()`,`Resize()` und `Sharpen()`) nach einander auf jedes einzelne Bild des Videos anwendet.

Listing 7.5 wiederholt zur Erinnerung das TADL-Skript aus Kapitel 5. Die parallele Verarbeitungsstrategie der Filter-Methoden wurde als Fließband modelliert, die Filtermethoden selbst stellen die atomaren Komponenten der Architektur dar. Da die Verarbeitung der einzelnen Videobilder unabhängig und zustandslos ist, kann durch Replizieren der atomaren Komponenten die vorhandene Datenparallelität ausgenutzt werden.

```
TunablePipeline MeineVideoVerarbeitung
    [source:LoadVideo;sink:ConsumeVideo] {
    Crop[replicable],
    OilPaint[replicable],
    Resize[replicable],
    Sharpen[replicable]
}
```

Listing 7.5: Architekturbeschreibung des parallelen Videoverarbeitungsprogramms (aus [ScPT10])

Für den Benchmark wurde ein AVI-Video mit 800 Bildern und einer Auflösung von 800x600 Bildpunkten verwendet.

7.1.5 Berechnung der Mandelbrot-Menge (MB)

Die beiden letzten Fallstudien befassen sich mit algorithmischen Problemen und sollen zeigen, dass die Konzepte dieser Arbeit auch für kleine Programme anwendbar sind, obwohl sie für die Verwendung in großen parallelen Applikationen konzipiert wurden.

Die erste der beiden algorithmischen Fallstudien befasst sich mit der Berechnung der Mandelbrotmenge für eine Bildauflösung von 4096 x 4096 Punkten. Die maximale Anzahl an Berechnungsschritten pro Bildpunkt wurde dabei auf 1000 Iterationen beschränkt. Listing 7.6 führt die TADL-Beschreibung des Programms auf. Die Anwendung besteht aus einer einzigen replizierbaren atomaren Komponente, welche die Logik zur Berechnung der Mandelbrotmenge enthält. Um die Datenparallelität ausnutzen zu können, bearbeitet die Methode, welche die atomare Komponente `CalculateMandelbrot` implementiert, bei jedem Aufruf eine Zeile des Bildes. Somit können durch Replizieren der atomaren Komponente mehrere Bildzeilen parallel verarbeitet werden. Die Auswahl einer Bildzeile aus der Menge aller Zeilen sowie deren Übergaben an eine konkrete Instanz der atomaren Komponente `CalculateMandelbrot` übernimmt das TALib-Modul, welches die Replizierungslogik bereitstellt (vgl. Kapitel 6, Abschnitt 6.2.1).

```
SequentialComposition MeineMandelbrotMenge {
    CalculateMandelbrot[replicable]
}
```

Listing 7.6: Architekturbeschreibung des Programms zur Berechnung der Mandelbrotmenge

7.1.6 Matrix-Matrix-Multiplikation (Matrix)

Die zweite algorithmische Fallstudie befasst sich mit einem Programm zur Multiplikation zweier quadratischer Matrizen mit Integer-Werten. Listing 7.7 zeigt die zugehörige Architekturbeschreibung. Wie schon das Programm zur Berechnung der Mandelbrotmenge besteht auch diese Anwendung aus einer einzige replizierbaren atomaren Komponente, die den Algorithmus zur Multiplikation der Matrizen enthält. Eine Instanz der replizierbaren atomaren Komponente berechnet jeweils eine Zeile in der Ergebnismatrix. Auch hier erfolgt die Zuweisung der Zeilen der Eingabematrizen zu den jeweiligen Instanzen durch die Replizierungslogik.

```
SequentialComposition MeineMatrixMultiplikation {
    MatrixMultiply[replicable]
}
```

Listing 7.7: Architekturbeschreibung des Programms zur Matrix-Matrix-Multiplikation

Für den Benchmark wurden zwei Matrizen der Größe 1024 x 1024 Elemente multipliziert.

7.1.7 Zusammenfassung und Vergleich der Anwendungen

Dieser Abschnitt fasst die wichtigsten Eigenschaften der oben vorgestellten Programme noch einmal zusammen.

Alle Anwendungen basieren auf dem Microsoft .NET Framework und sind in C# geschrieben. Dies begründet sich insbesondere damit, dass die prototypische Implementierung von Atune-TA ebenfalls .NET-basiert entwickelt wurde und der Quelltextgenerator des TADL-Übersetzers in seiner momentanen Ausprägung ausschließlich C#-Code erzeugt.

Tabelle 7.1 zeigt eine Übersicht allgemeiner Eigenschaften der Programme. Neben der durchschnittlichen Laufzeit der sequentiellen Programmversionen (bezogen auf die in den vorherigen Abschnitten beschriebenen Eingabedaten) und dem Umfang, den wir mit

der Anzahl an Quelltextzeilen (engl. *lines of code (LOC)*) beschreiben, sind auch die potentiellen Parallelitätstypen aufgeführt. Wie im Grundlagen-Kapitel in Abschnitt 3.7.3.1 bereits erläutert, wird zwischen Fließbandparallelität (F), Aufgabenparallelität (A) und Datenparallelität (D) unterschieden.

	MID	**GrGen**	**DS**	**Video**	**MB**	**Matrix**
ø Laufzeit	85s	45s	14h35m*	60s	30s	38s
Umfang (LOC)	100.000	80.000	285	750	97	75
Parallelitätstypen	A/D	A/D	F/D	F/D	D	D

* Zusätzliche Variante der Desktopsuche benötigte nur 239 Sekunden.

Tabelle 7.1: Eigenschaften der Fallstudien-Programme.

7.2 Experimentelle Ergebnisse

Für die Evaluation dieser Arbeit haben wir verschiedene Experimente durchgeführt, um jedes der Teilkonzepte, aber auch deren Kombination adäquat bewerten zu können.

Zunächst werden die Experimente zu Atune-TA vorgestellt, bei denen die Architekturbeschreibung mittels TADL sowie die automatisierte Umsetzung der Architektur im Mittelpunkt stehen. Wir betrachten hier vor allem die Einsparung an Implementierungsaufwand.

Anschließend wird die Effizienz der Konzepte zur Suchraumpartitionierung bewertet. Die Grundlage der Suchraumpartitionierung bilden die Atune-IL-Anweisungen (insbesondere die Sprachkonstrukte zur Deklaration von Tuning-Blöcken und Parameterabhängigkeiten), die von Atune-OPT zur Identifikation von Tuning-Einheiten verwendet werden. Aus diesem Grund liegt der Schwerpunkt der entsprechenden Experimente auf Atune-IL.

Schließlich werden die Effizienz der Tuning-Heuristiken von Atune-OPT sowie die Optimierung der sechs Fallbeispiele an Hand von Leistungsmessungen analysiert.

Alle Experimente wurden auf einem 8-Kern-Rechner durchgeführt (2x Intel Xeon E5320 QuadCore, 1,86 GHz/Core, 8 GB RAM, Microsoft Windows 2003 R2 32bit). Daher basieren auch alle Annahmen (z.B. Standardwerte von Tuning-Parametern) auf den verfügbaren 8 Rechenkernen.

7.2.1 Reduktion des Parallelisierungssaufwandes durch Atune-TA

Eingangs dieses Kapitels wurden bereits die parallelen Architekturen der sechs Programme dargelegt und durch TADL-Skripte beschrieben. Ein wichtiger Vorteil von Atune-TA ist die signifikante Reduktion des Parallelisierungssaufwandes (Aufwand für die Implementierung der parallelen konfigurierbaren Konstrukte), die durch die automatisierte Umsetzung der Architekturbeschreibung erreicht wird. Die folgenden Experimente sollen die Einsparung aufzeigen.

7.2.1.1 Evaluationsmetriken

Um die Reduktion des Parallelisierungssaufwandes bestimmen zu können, wird im Folgenden der Atune-TA-Ansatz mit einer Vorgehensweise verglichen, bei der dieselbe Funktionalität (Parallelisierung, Adaptierbarkeit sowie Auto-Tuning-Instrumentierung) manuell implementiert wird.

Hierbei wird der Parallelisierungssaufwand sowie die Fehleranfälligkeit hinsichtlich der parallelen Programmierung mit Hilfe der folgenden Metriken verglichen:

- **Anzahl an Quelltextzeilen (engl. *lines of code (LOC)*.** Die Anzahl der erforderlichen Quelltextzeilen (LOC) zur Implementierung der parallelen Konstrukte korreliert grob mit dem Implementierungsaufwand und kann daher als Metrik für den Vergleich zwischen Atune-TA-basierter und manueller Implementierung herangezogen werden.

- **Anzahl an Synchronisationsanweisungen (Syncs).** Die Synchronisation von Ausführungsfäden ist eine der schwierigsten und fehleranfälligsten Aufgaben bei der Entwicklung paralleler Programme. Die Anzahl an Synchronisationsanweisungen (wie `lock`, `wait`, `notify`, `join`, usw.), die im Rahmen der Generierung der optimierbaren Architekturen automatisch bereitgestellt wird, gilt daher als Indikator für reduzierte Fehleranfälligkeit durch die Verwendung von Atune-TA.

- **Anzahl an Instrumentierungen (Inst).** Atune-IL-Instrumentierungen sind für die automatische Performanzoptimierung eines Programms unerlässlich. Die Anzahl an automatisch generierten Instrumentierungen zeigt die Vereinfachung des Tuning-Prozesses für den Software-Entwickler.

Für die Abschätzung des manuellen Aufwandes nehmen wir basierend auf Erfahrungswerten die folgenden Pauschalwerte für die Implementierungen der einzelnen Konnektoren an:

- **Tunable Pipeline-Konnektor:** 180 LOC, 10 Syncs.

- **Tunable Replication-Konnektor:** 120 LOC, 8 Syncs.

- **Tunable Fork/Join-Konnektor:** 170 LOC, 10 Syncs

- **Tunable Producer/Consumer-Konnektor:** 150 LOC, 9 Syncs.

- **Tunable Alternative-Konnektor:** 15 LOC, 0 Syncs.

Der Sequential Composition-Konnektor wird nicht betrachtet, da dieser weder Parallelität noch Tuning-Optionen beinhaltet und der nötige Quelltext nur die Methodenaufrufe zu den Kind-Elementen enthalten muss.

Da die Implementierung eines Konnektors wiederverwendet werden kann, wird für jedes Programm jeder vorkommende Konnektor-Typ nur einmal gezählt.

7.2.1.2 Ergebnisse im Überblick

Tabelle 7.2 führt die konkreten Zahlen zur Reduktion des Parallelisierungssaufwandes (*LOC*), der Synchronisationsanweisungen (*Syncs*) sowie der Instrumentierungen (*Inst*) auf.

Es sei angemerkt, dass die 3 Quelltextzeilen, die bei jedem Fallbeispiel manuell hinzugefügt werden müssen, für die Instanziierung und den Aufruf der optimierbaren Architektur benötigt werden. Diese Konstante wird bei den folgenden Einzelbetrachtungen zwar berücksichtigt, jedoch nicht erneut erwähnt.

		MID	GrGen	DS	Video	MB	Matrix
LOC	Manuell	290	120	465	300	120	120
	Atune-TA	3	3	3	3	3	3
	Reduktion	287 (99%)	117 (98%)	462 (99%)	297 (99%)	117 (97%)	117 (97%)
Syncs	Manuell	18	8	27	18	8	8
	Atune-TA	2	0	1	0	0	0
	Reduktion	16 (89%)	8 (100%)	26 (96%)	18 (100%)	8 (100%)	8 (100%)
Inst	Manuell	39	16	30	16	5	5
	Atune-TA	0	2	0	0	0	0
	Reduktion	39 (100%)	14 (87%)	30 (100%)	16 (100%)	5 (100%)	5 (100%)

Tabelle 7.2: Ergebnisse der Reduktion des Parallelisierungssaufwandes durch Atune-TA.

7.2.1.3 MetaboliteID

Atune-TA übernimmt die Implementierung von sechs optimierbaren parallelen Sektionen (3x Fork/Join und 2x Daten-Dekomposition) und damit 99% des Parallelisierungssaufwandes ($170 + 120 - 3 = 287$ LOC).

Bis auf 2 Synchronisationsanweisungen, die die Kommunikation von zwei atomaren Komponenten untereinander steuern, konnten durch Atune-TA alle parallelisierungs-relevanten Aufgaben (89%) automatisiert umgesetzt und gekapselt werden.

Des Weiteren hat Atune-TA alle erforderlichen Atune-IL-Instrumentierungen innerhalb der Tuning-Hüllklassen automatisch generiert. Zusätzliche Tuning-Instruktionen wurden nicht benötigt.

7.2.1.4 GrGen.NET

Für die Parallelisierung von GrGen.NET erzeugte der TADL-Übersetzer zwei Tuning-Hüllklassen für die beiden datenparallelen Sektionen und band die entsprechenden Konnektormodule der TALib ein. Damit wurden 98% des Parallelisierungssaufwand abgedeckt ($120 - 3 = 117$ LOC).

Die generierte optimierbare Architektur musste nicht um zusätzliche Synchronisationsanweisungen ergänzt werden, so dass Atune-TA 100% der Parallelisierung übernehmen konnte.

Um die Tuning-Instruktionen zu vervollständigen, musste lediglich der bereits erwähnte Permutationsbereich manuell definiert werden. 87% der Instrumentierungen wurden jedoch durch die generierten Tuning-Hüllklassen bereitgestellt.

7.2.1.5 Desktopsuche

Die Parallelisierung der Desktopsuche war hinsichtlich der LOC-Metrik am aufwendigsten, da zwei vollständige Architekturvarianten mit unterschiedlichen Parallelisierungsstrategien nebst Auswahllogik implementiert werden mussten. Jedoch erzeugte auch in diesem Fallbeispiel der TADL-Übersetzer mit Hilfe der TALib 99% der erforderlichen Programmlogik ($180 + 150 + 120 + 15 - 3 = 462$ LOC).

Des Weiteren wurde zur korrekten Parallelisierung nur eine manuelle Synchronisation benötigt, um den gemeinsamen Zugriff auf die Indexdatenstruktur zu steuern. Die übrigen 26 Synchronisationsanweisungen für die parallelen Sektionen wurden implizit bereitgestellt.

Die erforderlichen Tuning-Instruktionen in Form von Atune-IL-Anweisungen wurden vollständig erzeugt und bedurften keiner manuellen Zusätze.

7.2.1.6 Videoverarbeitung

Die Reduktion des Parallelisierungssaufwandes betrug auch bei diesem Fallbeispiel 99% und sparte somit $180 + 120 - 3 = 297$ LOC ein. Des Weiteren übernahm der TADL-Übersetzer durch Einbindung der entsprechenden TALib-Module alle erforderlichen Synchronisierungen; manuelle Ergänzungen wurden nicht benötigt.

Gleiches gilt für die Tuning-Instruktionen. Alle nötigen Atune-IL-Anweisungen wurden im Rahmen der Tuning-Hüllklassen generiert.

7.2.1.7 Algorithmische Fallbeispiele

Auf Grund der identischen parallelen Architektur der beiden algorithmischen Fallbeispiele fallen die Ergebnisse zur Reduktion des Parallelisierungssaufwandes gleich aus, weshalb wir diese gemeinsam betrachten können.

Bei beiden Fallbeispielen betrug die Reduktion des Parallelisierungssaufwandes $97,5\%$ und sparte somit $120 - 3 = 117$ LOC ein, die normalerweise jeweils für die Implementierung der konfigurierbaren datenparallelen Sektion benötigt würden. Sowohl für die Berechnung der Mandelbrotmenge als auch für die Matrix-Matrix-Multiplikation wurden alle nötigen Synchronisierungen automatisch durchgeführt, so dass manuelle Eingriffe nicht erforderlich waren.

Auch wurden alle Atune-IL-Anweisungen automatisch in die jeweiligen Tuning-Hüllklassen integriert.

7.2.1.8 Programmkomplexität

Neben der Reduktion von Parallelisierungssaufwand und Fehleranfälligkeit darf die Programmkomplexität nicht überdurchschnittlich zunehmen, damit die Lesbarkeit des Quelltextes sowie das Programmverständnis nicht erschwert werden.

Die Komplexität des Programms wurde mit Hilfe der Klassenkopplung (engl. *class coupling*) [StMC79], der Gesamtzahl an Klassen sowie der Zunahme an LOC abgeschätzt. Maßgeblich ist zu diesem Zweck der Vergleich zwischen der Programmversion, die nur die Methoden der atomaren Komponenten sowie anderweitige Programmlogik enthält, und der Version, die mit einer vollständigen optimierbaren Architektur ausgestattet ist.

Die Ergebnisse sind vielversprechend. Durchschnittlich nimmt der Faktor der Klassenkopplung eines Programm bei Verwendung optimierbarer Architekturen um lediglich 18% zu, die Gesamtzahl an Klassen um 13% sowie die Gesamtzahl an Quelltextzeilen um nur 3%.

Es wird deutlich, dass die Implementierung einer optimierbaren Architektur den Quelltext des Programms kaum vergrößert, während die Erhöhung der Komplexität in einem überschaubaren Rahmen bleibt. Hinzu kommt, dass durch die Organisation der Quelltextgenerierung in Form von Tuning-Hüllklassen der Umfang der Programmerweiterung stets vorhersagbar bleibt und der erzeugte Quelltext übersichtlich strukturiert ist.

7.2.1.9 Fazit

Die obigen Vergleiche zeigen, dass Atune-TA den Aufwand für die Implementierung der parallelen konfigurierbaren Konstrukte wesentlich reduziert und somit den gesamten Entwicklungsprozess paralleler Programme drastisch beschleunigt. Des Weiteren sind

nahezu keine manuellen Synchronisierungen erforderlich, wodurch die Fehleranfälligkeit verringert wird; Ausnahmen bilden atomare Komponenten, die auf eine gemeinsame Datenstruktur zugreifen, die nicht Teil der optimierbaren Architektur ist.

Schließlich werden bis auf wenige Ausnahmen alle benötigten Atune-IL-Anweisungen automatisch generiert.

Obwohl der TADL-Übersetzer nahezu allen erforderlichen Quelltext für die Implementierung der Parallelisierungs- und Adaptionslogik automatisiert erzeugt, nimmt die Komplexität der Programme nur in moderatem Maße zu. Durch die Organisation des generierten Quelltextes in Tuning-Hüllklassen bleiben Lesbarkeit und Verständlichkeit gewahrt.

Die hier vorgestellten Experimente zur Evaluierung von Atune-TA wurden bereits in ähnlicher Form veröffentlicht [ScPT10].

7.2.2 Effizienz der Suchraumpartitionierung

Die Suchraumpartitionierung stellt ein wesentliches Konzept zur Verringerung der Anzahl möglicher Parameterkonfigurationen dar. Die Atune-IL-Instrumentierungen im Programmquelltext bilden die Grundlage zur Umsetzung der Suchraumpartitionierung und damit der Identifikation von unabhängigen Tuning-Einheiten. Aus diesem Grund verwendet Atune-TA die Tuning-Instrumentierungssprache als integriertes Werkzeug, indem alle erforderlichen Atune-IL-Anweisungen für eine parallele Architektur automatisch erzeugt werden (vgl. vorherigen Abschnitt 7.2.1).

Um jedoch die Berechnung der Suchraumpartitionierung an Hand der Atune-IL-Sprachkonstrukte genau zu beschreiben und deren Effizienz zu beurteilen, betrachten wir im Folgenden die Instrumentierung mit Atune-IL als separaten Evaluationsschritt. Dies wird zusätzlich dadurch begründet, dass Atune-IL einen eigenständigen Ansatz darstellt, der auch unabhängig von den anderen Teilkonzepten dieser Arbeit verwendet werden kann. Des Weiteren besteht die Möglichkeit, eine durch den TADL-Übersetzer generierte optimierbare Architektur um zusätzliche Atune-IL-Instrumentierungen zu erweitern oder bereits existierende Teile des Programm mit Tuning-Instruktionen zu versehen, die Atune-OPT dann automatisch mit verarbeitet.

Zur Durchführung der Experimente wurden zunächst alle sechs Programme unter Verwendung der TALib parallelisiert. Anschließend wurden alle notwendigen Atune-IL-Anweisungen manuell hinzugefügt, um an Hand der Beispiele detailiert zu zeigen, welche Instrumentierungen erforderlich sind und welchen Beitrag diese im Einzelnen für die Suchraumpartitionierung leisten.

Für die gezielte manuelle Instrumentierung wurde die Programmlogik, die normalerweise in den Tuning-Hüllklassen gekapselt ist, von Hand implementiert. Die Instrumentierung selbst orientierte sich im Wesentlichen an den Tuning-Parametern der verwendeten TADL-Konnektoren. Tuning-Blöcke und Messpunkte wurden an geeigneten Stellen zur Einfassung der TALib-Aufrufe zu den Implementierungen der Konnektoren deklariert.

7.2.2.1 Evaluationsmetriken

Die Bewertung der Effizienz der Suchraumpartitionierung erfolgte an Hand der jeweils resultierenden Reduktion des Suchraums. Hierzu wurde auf Basis der verwendeten Atune-IL-Anweisungen jeweils die Größe des anfänglichen ($|\mathcal{S}|$) sowie die des partitionierten Suchraums ($|\mathcal{R}|$) berechnet. Für die Ermittlung der Suchraumgröße wurde gemäß Defi-

nition 3.7 das kartesische Produkt aus den Wertemengen der instrumentierten Tuning-Parameter gebildet. Die Reduktion ergibt sich dann durch die Ausnutzung der Tuning-Block-Semantik sowie den Parameterabhängigkeiten.

Die Wertebereiche der Tuning-Parameter aller Konnektoren wurden vorab definiert, um für alle Fallstudien dieselbe Ausgangssituation herzustellen. Tabelle 7.3 führt die definierten Wertebereiche jedes Tuning-Parameters auf.

Tuning-Parameter	**Wertebereich**
Tunable Alternative (TA)	
`SelectAlternative`	$< \#Alternativen >$
Tunable Fork/Join (TFJ)	
`NumThreads`	$\{2, ..., 16\}$ (bzgl. 8-Kern-Rechner)
Tunable Producer/Consumer (TPC)	
`BufferSize`	$\{0, 10, 20, 30\}$ Elemente (0: dynamisch)
`BatchSize`	$\{0, 5, 10, 15, 20\}$ Elemente (0: Element sofort lesen)
Tunable Pipeline (TP)	
`StageFusion`	pro Stufenpaar: $\{true, false\}$.
Tunable Replication (TR)	
`NumInstances`	$\{2, ..., 16\}$ (bzgl. 8-Kern-Rechner)
`LoadBalancing`	$\{static, dynamic\}$
`BatchSize`	$\{0, 5, 10, 15, 20\}$ (0: Element sofort lesen)

Tabelle 7.3: Wertebereiche aller Tuning-Parameter zur Verwendung in den Fallstudien (vgl. Tabelle 5.2).

Es sei angemerkt, dass mit obiger Evaluationsmetrik neben Atune-IL gleichzeitig auch die von Atune-OPT durchgeführte *Umsetzung* der Suchraumpartitionierung in Tuning-Einheiten bewertet wird.

7.2.2.2 Ergebnisse im Überblick

Tabelle 7.4 fasst die Ergebnisse der Experimente zusammen. Im oberen Teil der Tabelle sind die einzelnen Instrumentierungen aufgeführt, die pro Anwendung durchgeführt wurden. Der untere Teil zeigt die Ergebnisse der Suchraumpartitionierung, die mit Hilfe der speziellen Sprachkonstrukte von Atune-IL erreicht konnte. $|\mathcal{S}|$ entspricht der Größe des anfänglichen Suchraums ohne Berücksichtigung entsprechender Atune-IL-Instruktionen, $|\mathcal{R}|$ repräsentiert die Größe des *reduzierten* Suchraums nach Berücksichtigung aller Atune-IL-Instruktionen.

Im Folgenden werden die Einzelbetrachtungen der Fallstudien dargelegt.

	MID	GrGen	DS	Video	MB	Matrix
			Atune-IL-Anweisungen			
# Tuning-Blöcke	5	3	2	0	0	0
# Parameter	9	7	24	15	3	3
# Parameterabh.	2	2	6	4	1	1
# Messpunkte	3	2	2	1	1	1
			Suchraumreduktion durch Partitionierung			
$\|\mathcal{S}\|$	$76 \cdot 10^6$	$2,7 \cdot 10^6$	$2,7 \cdot 10^{15}$	$4 \cdot 10^9$	150	150
$\|\mathcal{R}\|$	40.515	300	275.400	$524 \cdot 10^6$	90	90
Reduktion	$> 99\%$	99%	$> 99\%$	83%	40%	40%

Tabelle 7.4: Experimentelle Ergebnisse der Evaluation von Atune-IL.

7.2.2.3 MetaboliteID

Die parallele Architektur von MetaboliteID (vgl. Listing 7.1) setzt sich aus den folgenden Sektionen zusammen:

- 3 Fork/Join-Sektionen (*TFJ*)
- 2 datenparallelen Sektionen (*TR*)

Die fünf parallelen Sektionen haben wir jeweils in Tuning-Blöcke eingefasst. Des Weiteren wurden die Tuning-Parameter für die Anzahl an Ausführungsfäden sowie zur Auswahl der Lastausgleichsstrategie mittels entsprechender `setvar`-Anweisungen markiert. Die beiden datenparallelen Sektionen wurden zusätzlich mit einer Instrumentierung für den Parameter der Blockgröße versehen. Da die Blockgröße nur bei statischer Lastausgleichsstrategie konfiguriert werden kann, konnten entsprechende Parameterabhängigkeiten definiert werden.

Zur Messung der Laufzeit haben wir in den beiden äußeren Fork/Join-Sektionen je zwei `gauge`-Anweisungen definiert.

Die Größe des initialen Suchraums $\mathcal{S}$ belief sich auf etwa $76 \cdot 10^6$ Parameterkonfigurationen. Unter Verwendung der Sprachkonstrukte von Atune-IL (insbesondere Tuning-Blöcke und Parameterabhängigkeiten) konnten 3 Tuning-Einheiten (*TE*) identifiziert werden:

- **TE 1:** 1. *TFJ*-Sektion.
- **TE 2:** 2. *TFJ*-Sektion mit geschachtelter *TFJ*-Sektion und 1. *TR*-Sektion.
- **TE 3:** 2. *TFJ*-Sektion mit geschachtelter *TFJ*-Sektion und 2. *TR*-Sektion.

Die Suchraumpartitionierung in 3 Tuning-Einheiten sowie die Berücksichtigung der Parameterabhängigkeiten in den datenparallelen Sektionen [1] ergab einen reduzierten Suchraum $\mathcal{R}$ mit 40.515 Parameterkonfigurationen. Tabelle 7.5 skizziert die Berechnung.

[1] Für alle datenparallelen Sektionen (*Tunable Replication (TR)*) gilt: Der Parameter `BatchSize` muss nur dann berücksichtigt werden, falls eine dynamische Lastverteilung gewählt wurde. Anstatt 2*5*15=150 Parameterkonfigurationen pro Tunable Replication ergeben sich somit 15+5*15=90 Parameterkonfigurationen.

	# Tuning-Parameter	Suchraumgröße
$\mathcal{S}$	$3 \cdot 1 + 2 \cdot 3 = 9$	$\overbrace{15 \cdot 15 \cdot 15}^{3 \cdot TFJ} \cdot \overbrace{150 \cdot 150}^{2 \cdot TR} \approx 76 \cdot 10^6$
$\mathcal{R}$	$\underbrace{3 \cdot 1}_{3 \cdot TFJ} + \underbrace{2 \cdot 3}_{2 \cdot TR} = 9$	$\underbrace{15}_{TE1} + \underbrace{(15 \cdot 15 \cdot 90^{(1)})}_{TE2} + \underbrace{(15 \cdot 15 \cdot 90^{(1)})}_{TE3} = 40.515$

Tabelle 7.5: Berechnung des Suchraums für MetaboliteID.

7.2.2.4 GrGen.NET

Die parallele Architektur von GrGen.NET (vgl. Listing 7.2) besteht aus 2 datenparallelen Sektion (*TR*). Beide *TR*-Sektionen haben wir jeweils als Tuning-Block markiert, da sie nacheinander ausgeführt werden und daher im Sinne der Optimierung unabhängig sind. Die Deklarationen der Tuning-Parameter, die die Anzahl an parallelen Instanzen, die Blockgröße sowie Lastausgleichsstrategie definieren, haben wir entsprechend instrumentiert.

Neben den parallelen Sektionen ist die Abarbeitungsreihenfolge der Ersetzungsregeln von Bedeutung. Eine günstige Reihenfolge kann die Suche nach den Graphmustern erheblich beschleunigen. Aus diesem Grund haben wir mit Atune-IL einen Permutationsbereich definiert, der die Deklaration sowie die Ausführung von 5 Ersetzungsregeln umfasst.

Da die beiden parallelen Sektionen vollständig entkoppelt sind, soll die Laufzeit jeder Sektion separat ermittelt werden. Aus diesem Grund haben wir in beiden Sektionen je ein Paar an `gauge`-Anweisungen platziert.

	# Tuning-Parameter	Suchraumgröße
$\mathcal{S}$	$2 \cdot 3 + 1 = 7$	$\overbrace{150 \cdot 150}^{2 \cdot TR} \cdot \overbrace{120}^{Perm.} \approx 2,7 \cdot 10^6$
$\mathcal{R}$	$\underbrace{2 \cdot 3}_{2 \cdot TR} + \underbrace{1}_{Perm.} = 7$	$\underbrace{90}_{TE1} + \underbrace{90}_{TE2} + \underbrace{120}_{TE3} = 300$

Tabelle 7.6: Berechnung des Suchraums für GrGen.NET.

Der initiale Suchraum $\mathcal{S}$ umfasste $2,7 \cdot 10^6$ Parameterkonfigurationen. Mit Hilfe von Atune-IL konnte der Suchraum zunächst in zwei Partitionen zerteilt werden, so dass jede der datenparallelen Sektionen eine Tuning-Einheit repräsentierte. Da die Permutation der Ersetzungsregeln unabhängig von den datenparallelen Sektionen ist, konnte der Permutationsparameter ebenfalls einer eigenen Tuning-Einheit zugewiesen werden. Somit ergaben sich 3 Tuning-Einheiten (*TE*):

- ***TE* 1:** 1. *TR*-Sektion.
- ***TE* 2:** 2. *TR*-Sektion.
- ***TE* 3:** Permutation.

Nach anschließender Berücksichtigung der Parameterabhängigkeiten in den datenparallelen Sektionen blieb ein reduzierter Suchraum $\mathcal{R}$ mit nur noch 300 Parameterkonfigurationen übrig. Tabelle 7.6 zeigt die konkreten Zahlen.

7.2.2.5 Desktopsuche

Die Architektur der Desktopsuche (vgl. Listing 7.3) setzt sich aus den folgenden parallelen Sektionen zusammen:

- 1 äußeren Alternative (*TA*)
- 1 Fließband-Sektion (*TP*)
- 1 Erzeuger/Verbraucher-Sektion (*TPC*)
- 3 datenparallele Sektionen (*TR*) in *TP*, wobei 2 als Alternative (*TA*) markiert sind
- 3 datenparallele Sektionen (*TR*) in *TPC*, wobei 2 als Alternative (*TA*) markiert sind

Der äußere Tunable Alternative-Konnektor stellt mit Fließband- oder Erzeuger/Verbraucher-Muster zwei unterschiedliche parallele Verarbeitungsstrategien zur Auswahl und prägt somit die Architektur. Die beiden voneinander unabhängigen parallelen Sektionen haben wir daher jeweils in einen Tuning-Block eingefasst.

Zur Steuerung der beiden inneren Alternativen (je eine im Fließband und in der Erzeuger/Verbraucher-Sektion) haben wir entsprechende Tuning-Parameter definiert sowie die alternativen datenparallelen Sektionen der String-Matching-Algorithmen jeweils als einen separaten Tuning-Block markiert.

Die insgesamt 6 datenparallelen Sektionen wurden mit Instrumentierungen für die bereits bekannten Tuning-Parameter (Anzahl an Instanzen, Lastausgleichsstrategie und Größe der Datenblöcke) versehen. Zudem wurde das Fließband mit Parametern zur Steuerung der Stufenverkettung sowie die Erzeuger/Verbraucher-Sektion mit Parametern für Puffer- und Blockgröße ausgestattet und instrumentiert.

	# Tuning-Parameter	**Suchraumgröße**
$\mathcal{S}$	$3 \cdot 1 + 1 + 2 + 6 \cdot 3 = 24$	$\overbrace{2^3}^{3 \cdot TA} \cdot \overbrace{2}^{TP} \cdot \overbrace{15}^{TPC} \cdot \overbrace{150^6}^{6 \cdot TR} \approx 2.7 \cdot 10^{15}$
$\mathcal{R}$	$\underbrace{3 \cdot 1}_{3 \cdot TA} + \underbrace{1}_{TP} + \underbrace{2}_{TPC} + \underbrace{6 \cdot 3}_{6 \cdot TR} = 24$	$\underbrace{(2 \cdot 90^2) \cdot 2}_{TE1,TE2} + \underbrace{(15 \cdot 90^2) \cdot 2}_{TE3,TE4} = 275.400$

Tabelle 7.7: Berechnung des Suchraums für die Desktopsuche.

Um die insgesamt 4 Architekturvarianten von einem Auto-Tuner auf ihre Leistung hin überprüfen zu lassen, wurden alle Varianten mit entsprechenden Messpunkten versehen.

Der anfängliche Suchraum $\mathcal{S}$ umfasste etwa $2.7 \cdot 10^{15}$ Parameterkonfigurationen. Die Tuning-Block-Struktur verhalf zu einer Partitionierung des Suchraums in die folgenden Tuning-Einheiten (*TE*):

- ***TE* 1:** *TP* mit 2x *TR* (innere Alternative 1)
- ***TE* 2:** *TP* mit 2x *TR* (innere Alternative 2)
- ***TE* 3:** *TPC* mit 2x *TR* (innere Alternative 1)
- ***TE* 4:** *TPC* mit 2x *TR* (innere Alternative 2)

so dass ein reduzierter Suchraum $\mathcal{R}$ mit noch 275.400 Parameterkonfigurationen errechnet werden konnte. In Tabelle 7.7 wird die Berechnung des Suchraums und dessen Reduktion hergeleitet.

7.2.2.6 Videoverarbeitung

Die Architektur des Programms zur Videoverarbeitung (vgl. 7.5) besteht aus:

- 1 Fließband-Sektion (*TP*)
- 4 datenparallelen Sektionen (*TR*) als replizierbare Fließbandstufen

Da keine weiteren unabhängigen Sektionen existieren, wird kein expliziter Tuning-Block benötigt.

Wir haben die Tuning-Parameter des Fließbandes sowie der datenparallelen Sektionen in den Stufen mit entsprechenden `setvar`-Instrumentierungen versehen und die Parameterabhängigkeiten definiert.

Die beiden Messpunkte zur Bestimmung der Laufzeit wurden vor und nach der Fließbandausführung platziert.

Da dieses Fallbeispiel nur eine geschachtelte parallele Sektion ausweist, ist eine Partitionierung des Suchraums nicht möglich. Durch Berücksichtigung der Parameterabhängigkeiten in den datenparallelen Sektionen kann der anfängliche Suchraum $\mathcal{S}$ mit etwa $4 \cdot 10^9$ Parameterkonfigurationen dennoch auf einen reduzierten Suchraum $\mathcal{R}$ mit etwa $524 \cdot 10^6$ Konfigurationen verkleinert werden. Tabelle 7.8 zeigt die Berechnung des Suchraums.

	# Tuning-Parameter	**Suchraumgröße**
$\mathcal{S}$	$3 + 4 \cdot 3 = 15$	$\overbrace{2^3}^{TP} \cdot \overbrace{150^4}^{4 \cdot TR} \approx 4 \cdot 10^9$
$\mathcal{R}$	$\underbrace{3}_{P} + \underbrace{4 \cdot 3}_{4 \cdot TR} = 15$	$\underbrace{2^3}_{TP} \cdot \underbrace{(90^4)}_{4 \cdot TR} \approx 524 \cdot 10^6$

Tabelle 7.8: Berechnung des Suchraums für das Videoverarbeitungsprogramm.

7.2.2.7 Algorithmische Fallbeispiele

Die beiden algorithmischen Fallbeispiele weisen eine identische parallele Architektur auf, so dass sie gemeinsam betrachtet werden können. Sowohl das Programm zur Berechnung der Mandelbrotmenge als auch jenes zur Matrix-Matrix-Multiplikation beinhaltet eine datenparallelen Sektion (*TR*), bei der – analog zu den datenparallelen Sektionen der anderen Fallstudien – die Anzahl an Instanzen, die Lastausgleichsstrategie sowie die davon abhängige Blockgröße konfiguriert werden kann.

Die Tuning-Parameter wurden mit einer entsprechenden `setvar`-Anweisung versehen. Des Weiteren wurde in den beiden Programmen jeweils ein Messpunkt zur Bestimmung der Laufzeit deklariert.

Für jedes der beiden Programme bestand der anfängliche Suchraum $\mathcal{S}$ aus $5 \cdot 5 \cdot 15 = 150$ Parameterkonfigurationen der datenparallelen Sektion, die in nur einer Tuning-Einheit zusammengefasst wurden.

Unter Berücksichtigung der Parameterabhängigkeit innerhalb der datenparallelen Sektion konnte jeweils ein um 40% reduzierter Suchraum $\mathcal{R}$ mit $15 + 5 \cdot 15 = 90$ Parameterkonfigurationen errechnet werden (vgl. Fußnote auf Seite 151). Tabelle 7.9 führt die Berechnung auf.

	# Tuning-Parameter	Suchraumgröße
$\mathcal{S}$	3	150
$\mathcal{R}$	3	90

Tabelle 7.9: Berechnung des Suchraums für die algorithmischen Fallbeispiele.

7.2.2.8 Fazit

Die Fallstudien zeigen, dass

1. Atune-IL zu einer signifikanten Reduktion des Suchraums durch Partitionierung beitragen kann,
2. die Sprachkonstrukte von Atune-IL trotz ihrer Einfachheit mächtig genug sind, um die nötigen Tuning-Instruktionen innerhalb eines breiten Spektrums an parallelen Programmen auszudrücken,
3. Atune-IL in Programmen unterschiedlicher Größe und Anwendungsdomäne eingesetzt werden kann.

Es lässt sich jedoch feststellen, dass insbesondere bei der Verwendung eines Fließbandes mit replizierbaren Stufen der anfängliche Suchraum eine enorme Größe aufweist (bei dem Programm zur Videoverarbeitung beinhaltete der initiale Suchraum beispielsweise mehr als 4 Milliarden Parameterkonfigurationen). Diese hohen Zahlen entstehen aus zwei Gründen.

Zum Einen stellt ein Fließband mit replizierbaren Stufen ein komplexes Konstrukt dar, das verhältnismäßig vieler Tuning-Parameter bedarf. Allein für eine replizierbare Stufe sind gemäß der Spezifikation der optimierbaren Architekturen 3 Tuning-Parameter erforderlich, wodurch unter Berücksichtigung der in Tabelle 7.3 definierten Wertebereiche bereits 150 Parameterkonfigurationen entstehen.

Zum Anderen können in einem Fließband naturgemäß keine separaten Tuning-Einheiten definiert werden, da durch die Datenstrom-Semantik alle parallelen Sektion in den Stufen als von einander abhängig zu betrachten sind. Daher muss die ohnehin schon große Zahl an Tuning-Parametern zusammenhängend optimiert werden.

Obwohl durch die Sprachkonstrukte von Atune-IL auch in derartigen Extremfällen eine Suchraumreduktion von teilweise über 80% erreicht wurde, blieb dennoch ein umfangreicher Suchraum übrig, der für herkömmliche Suchalgorithmen nur schwer zu bewältigen ist.

Diese Erkenntnisse zeigen, dass die Partitionierung des Suchraums für bestimmte parallele Architekturen zwar eine signifikante Reduktion bewirken kann, aber dennoch nicht ausreichend ist. Dies begründet den Einsatz der kontextbasierten Suchraumreduktion, die auf jeder identifizierten Tuning-Einheit (d. h. auf jeder Partition des Suchraums) durchgeführt wird, wodurch die einzelnen Suchraumpartitionen nochmals wesentlich verkleinert werden können. Die entsprechenden Studien werden im nächsten Abschnitt gezeigt.

7.2.3 Effizienz der kontextbasierten Suchraumreduktion

Atune-OPT führt neben der Partitionierung des Suchraums auf Basis der Atune-IL-Instruktionen auch eine kontextbasierte Reduktion des Suchraums unter Verwendung von Tuning-Heuristiken durch (vgl. Kapitel 5, Abschnitt 5.4.4).

Eine Tuning-Heuristik definiert einen effizienten Optimierungsvorgang für einen TADL-Konnektor (also für eine Parallelisierungsstrategie). Die Implementierung von Atune-OPT unterstützt die kontextbasierte Suchraumreduktion für den Tunable Pipeline-Konnektor mit replizierbaren Stufen sowie für den Tunable Fork/Join- und den Tunable Replication-Konnektor.

Mit den folgenden Fallstudien soll gezeigt werden, dass die Tuning-Heuristiken deutlich zur Suchraumreduktion beitragen, ohne dass sich die Präzision der Suchverfahren wesentlich verschlechtert.

7.2.3.1 Evaluationsmetriken

Um die Qualität der Tuning-Heuristiken im Einzelnen, aber auch die der kontextbasierten Suchraumreduktion im Allgemeinen evaluieren zu können, haben wir die Effizienz der drei von Atune-OPT eingesetzten Suchverfahren (*Zufälliges Testen*, *Hillclimbing mit Check-Back-Schritt* und *Partikel-Schwarm-Optimierung*, vgl. Kapitel 5, Abschnitt 5.4.5.3) sowohl mit als auch ohne Anwendung der Tuning-Heuristiken bestimmt. Die folgenden Metriken wurden für den Vergleich herangezogen:

- **Anzahl an Tuning-Iterationen.** Die Metrik gibt an, nach wie vielen Iterationen das Suchverfahren konvergiert hat und erlaubt somit eine Aussage über die Laufzeit des Optimierungsprozesses.
- **Mittlere Fehlerrate.** Hiermit wird die Präzision des Optimierungsprozesses beschrieben. Die Zahl der mittleren Fehlerrate entspricht der durchschnittlichen Abweichung des gefundenen besten Ergebnisses zum tatsächlichen Optimum.

Anmerkungen zur Bestimmung der optimalen Parameterkonfiguration. Die jeweils optimale Parameterkonfiguration der Fallbeispiele kann nur durch vollständige Enumeration des Suchraums und dem anschließenden Testen aller Parameterkonfigurationen exakt ermittelt werden. Auf Grund der initialen Suchraumgrößen von bis zu mehreren Milliarden ist eine vollständige Enumeration jedoch nicht durchführbar, da der Test jeder einzelnen Parameterkonfiguration einen Programmdurchlauf erfordert. Alleine das Testen aller Parameterkonfigurationen des Videoverarbeitungsprogramms hätte mehr als 2.400 Jahre in Anspruch genommen. Daher wurde bei allen Fallbeispielen die vollständige Enumeration des *partitionierten* Suchraums $\mathcal{R}$ zu Grunde gelegt. Für die Programme MetaboliteID und GrGen sowie für die beiden algorithmischen Fallbeispiele konnte somit das tatsächliche Leistungsoptimum exakt bestimmt werden. Bei der Desktopsuche und dem Videoverarbeitungsprogramm stellte sich jedoch selbst der partitionierte Suchraum als noch zu groß heraus, um die enthaltenen Parameterkonfigurationen vollständig zu testen.

Der Grund hierfür liegt in der bereits im vorherigen Abschnitt angesprochenen Komplexität eines Fließbandes mit replizierbaren Stufen. Die große Zahl an Tuning-Parametern kann in diesem Fall nicht auf mehrere Partitionen bzw. Tuning-Einheiten verteilt werden, da sich alle Parameter gegenseitig beeinflussen können. Somit war bei diesen beiden Fallbeispielen eine Abschätzung des tatsächlichen Leistungsoptimums erforderlich.

Zu diesem Zweck wurde bei der Desktopsuche und dem Videoverarbeitungsprogramm der Suchraum manuell verkleinert, indem nur diejenigen Tuning-Parameter bei der Enumeration berücksichtigt wurden, die eine hohe Sensitivität aufweisen. Alle Parameter mit geringer Auswirkung auf die Gesamtleistung des Programms wurden nicht berücksichtigt, da deren optimale Konfiguration nur einen verhältnismäßig kleinen Leistungsgewinn ermöglichen kann.

Konkret wurden bei den replizierbaren Fließbändern für jede Stufe nur der Parameter `NumInstances` mit reduzierter Wertemenge $\{5, ..., 14\}$ sowie der Parameter `StageFusion` berücksichtigt. Die Parameter `LoadBalancing` und `BatchSize` wurden auf Grund ihrer geringeren Sensitivität von der Enumeration ausgeschlossen. Die beiden Alternativen der Desktopsuche wurden zudem mit Hilfe des Auto-Tuners stichprobenartig untersucht, um letztlich nur die relevanten Tuning-Parameter der besten Alternative vollständig enummerieren zu müssen. Die reduzierten Suchräume für Desktopsuche und Videoverarbeitung wiesen schließlich eine Größe von 450 bzw. 80.000 Parameterkonfigurationen auf.

Anmerkung zu den Suchalgorithmen. Das Verfahren *Zufälliges Testen* konvergiert nicht, sondern benötigt eine feste Anzahl an durchzuführenden Iterationen als Vorgabe. Aus diesem Grund haben wir bei den Experimenten das Hillclimbing-Verfahren (welches konvergiert) als Referenz zur Bestimmung der Anzahl an Tuning-Iterationen verwendet und anschließend dem Zufallsverfahren jeweils die entsprechende Zahl vorgegeben.

Um die statistische Relevanz zu erhöhen, wurde jeder Tuning-Lauf (pro Suchverfahren und Programm) 20 Mal wiederholt. Die Ergebnisse repräsentieren daher die Mittelwerte der einzelnen Tuning-Läufe.

7.2.3.2 Ergebnisse im Überblick

Die Diagramme in Abbildung 7.1 fassen die konkreten Zahlen zusammen. Für jedes der sechs Programme sind die Ergebnisse des Experimente mit (markiert mit einem *H*) und ohne Verwendung der Tuning-Heuristiken aufgetragen.

Da die Experimente auf einem 8-Kern-Rechner durchgeführt wurden ($p = 8$), beinhaltete auch die Tuning-Datenbank die Ergebnisse früherer Tuning-Läufe auf 8-Kern-Rechnern, um entsprechende historische Daten bereitzustellen, die insbesondere für die Tuning-Heuristik des Tunable For/Join-Konnektors und des Tunable Replication-Konnektors benötigt werden: gemäß der historischen Daten setzte Atune-OPT die Werte für α und β auf $0,9 \cdot p$ bzw. $1,7 \cdot p$, um die Wertebereiche der Parameter `NumThreads` bzw. `NumInstances` zu begrenzen (vgl. Kapitel 5, Abschnitt 5.4.4.1).

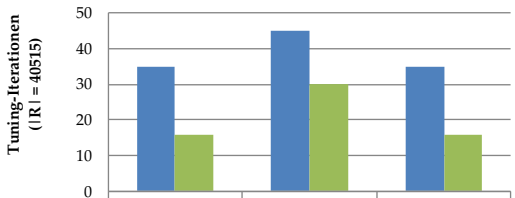
7.2.3.3 Diskussion der Ergebnisse

Grundsätzlich ist festzustellen, dass alle verwendeten Suchverfahren auch ohne kontextbasierte Suchraumreduktion nur einen kleinen Teil des partitionierten Suchraums $\mathcal{R}$ durchsuchen müssen, um zu konvergieren. Werden zudem entsprechende Tuning-Heuristiken verwendet, reduziert sich die Anzahl erforderlicher Iterationen nochmals – je nach Programm und Suchalgorithmus bis zu 50%.

Des Weiteren kann beobachtet werden, dass die durchschnittliche Fehlerrate sinkt, je kleiner die Programme sind und je weniger parallele Sektionen pro Tuning-Einheit enthalten sind. Insbesondere bei den algorithmischen Fallbeispielen ist die Fehlerrate mit maximal 2% sehr gering. Bei derart kleinen Suchräumen spielt die Wahl der Suchalgorithmus eine untergeordnete Rolle, da alle Verfahren im durchschnitt gleich gut abgeschnitten haben. Daraus folgt, dass bei sehr kleinen Suchräumen ein einfaches Verfahren wie das zufällige Testen ausreicht, um hinreichend gute Ergebnisse zu erzielen.

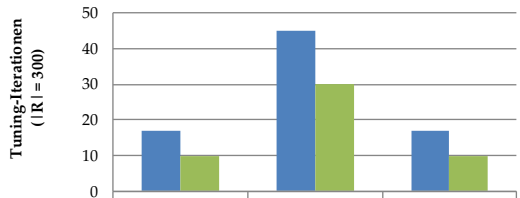
Beim Vergleich der Optimierung mit und ohne kontextbasierter Suchraumreduktion zeigt sich ebenfalls ein einheitliches Bild. Alle Suchverfahren profitierten von den kleineren Suchräumen, was sich in der zum Teil deutlich verringerten Anzahl an Tuning-Iterationen äußerte. Gleichzeitig blieb die Präzision nahezu gleich bzw. verbesserte sich in manchen Fällen. Letzteres ist darauf zurückzuführen, dass die Suchalgorithmen durch

MID



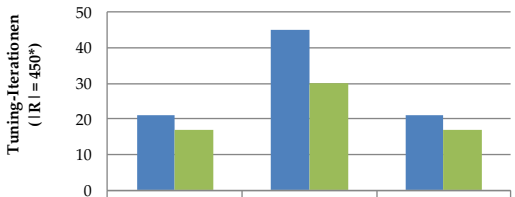
	Hillclimbing	Partikel-Schwarm	Zufälliges Testen
# Iterationen	35	45	35
# Iterationen (H)	16	30	16
Fehlerrate	18%	1%	15%
Fehlerrate (H)	6%	0%	4%

GrGen



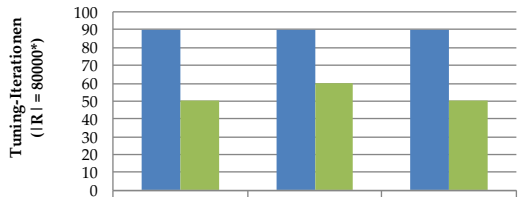
	Hillclimbing	Partikel-Schwarm	Zufälliges Testen
# Iterationen	17	45	17
# Iterationen (H)	10	30	10
Fehlerrate	10%	8%	19%
Fehlerrate (H)	9%	7%	19%

DS



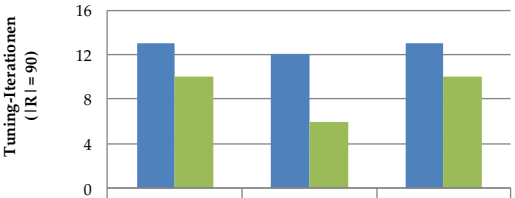
	Hillclimbing	Partikel-Schwarm	Zufälliges Testen
# Iterationen	21	45	21
# Iterationen (H)	17	30	17
Fehlerrate	10%	11%	25%
Fehlerrate (H)	6%	7%	17%

Video



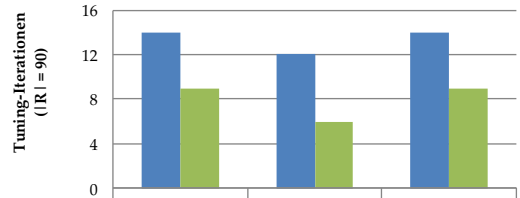
	Hillclimbing	Partikel-Schwarm	Zufälliges Testen
# Iterationen	90	90	90
# Iterationen (H)	50	60	50
Fehlerrate	25%	18%	37%
Fehlerrate (H)	10%	18%	16%

MB



	Hillclimbing	Partikel-Schwarm	Zufälliges Testen
# Iterationen	13	12	13
# Iterationen (H)	10	6	10
Fehlerrate	1%	1%	2%
Fehlerrate (H)	0%	1%	0%

Matrix



	Hillclimbing	Partikel-Schwarm	Zufälliges Testen
# Iterationen	14	12	14
# Iterationen (H)	9	6	9
Fehlerrate	1%	1%	1%
Fehlerrate (H)	1%	2%	1%

* Manuell reduzierter Suchraum zur Bestimmung eines möglichst guten Referenzwertes

Abbildung 7.1: Ergebnisse der Effizienzbewertung der kontextbasierten Suchraumreduktion.

die geführte Suche nahezu ausschließlich relevante Parameterkonfigurationen überprüfen und somit die verringerte Genauigkeit der Tuning-Heuristiken ausgleichen können. Die Suchalgorithmen laufen weniger Gefahr, in nicht relevanten Teilen des Suchraums zu konvergieren und damit eine schlechtere Präzision zu liefern.

Das Hillclimbing-Verfahren mit Check-Back-Schritt benötigte zur Optimierung der Programme mit Hilfe der kontextbasierten Suchraumreduktion stets weniger Tuning-Iterationen als ohne die Verwendung von Tuning-Heuristiken und konnte die Präzision meist verbessern. Je umfangreicher das zu optimierende Programm, desto größer fiel die Reduktion der Tuning-Iterationen aus.

Gleiches gilt für das Zufallsverfahren, wenngleich naturgemäß die Fehlerrate gegenüber dem Hillclimbing-Verfahren erhöht ist. Dennoch zeigt sich, dass auch mit einem einfachen Suchverfahren wie dem zufälligen Testen durchaus brauchbare Ergebnisse erzielt werden können, sofern der Suchraum zuvor entsprechend reduziert wurde. Es gilt daher abzuwägen, ob auch etwas weniger Präzision ausreicht, man dafür aber ein Verfahren einsetzen kann, welches nahezu keine Berechnungszeit benötigt sowie einen geringen Implementierungsaufwand erfordert.

Die Partikel-Schwarm-Optimierung profitierte von der Eigenschaft, bei kleinen Suchräumen initial deutlich weniger Partikel (d.h. Startkonfigurationen) verwenden zu müssen, was die Zahl der erforderlichen Tuning-Iterationen signifikant reduziert (wie beispielsweise bei den algorithmischen Fallbeispielen). Die Zahl der durchzuführenden Tuning-Iterationen ergibt sich aus der (festen) Größe des Schwarms (Anzahl der Partikel) sowie der Anzahl an Algorithmen-Iterationen (d.h., wie oft testen alle Partikel des Schwarms synchron eine neue Parameterkonfiguration). An dieser Stelle sei angemerkt, dass AtuneOPT die Konfigurationsparameter des Partikel-Schwarm-Algorithmus nicht automatisch setzt. Das optimale Verhältnis zwischen Suchraumgröße und Anzahl an Tuning-Iterationen bedarf jedoch einer umfangreichen Analyse und konnte im Rahmen dieser Arbeit nicht allgemeingültig festgelegt werden. Es wurde daher für jede Fallstudie empirisch eine möglichst geringe Anzahl an Startkonfigurationen ermittelt, so dass die Präzision des Verfahrens sich nicht wesentlich verschlechtert.

Bei genauer Betrachtung fällt außerdem auf, dass die Präzision der Suchalgorithmen je nach zu optimierendem Programm im Verhältnis zu den anderen Verfahren nicht immer gleich gut ist. In den meisten Fällen schneidet das Hillclimbing-Verfahren mit Check-Back-Schritt etwas besser ab als die Partikel-Schwarm-Optimierung. Bei MetaboliteID stellt sich der Sachverhalt jedoch umgekehrt dar. Hier schneidet selbst das zufällige Testen besser ab. Der Grund hierfür liegt in der Verteilung der Leistungswerte. Im Falle von MetaboliteID liefern eine große Zahl an Parameterkonfigurationen ähnlich Leistungswerte, die nahe am optimalen Leistungswert liegen (über 2.000 Konfiguration erzielen einen Leistungswert mit einer Fehlerrate von weniger als 10%). Somit trifft ein einfaches Verfahren wie das zufälligen Testen mit höherer Wahrscheinlichkeit eine gute Parameterkonfiguration. Gleiches gilt für die Partikel-Schwarm-Optimierung, die als globales Suchverfahren nahezu immer eine Region findet, die nahe dem Optimum liegt und dieses dann ausfindig machen kann.

Im Gegensatz zu obiger Betrachtung erzielt das Hillclimbing-Verfahren als Vertreter der lokalen Suche insbesondere dann gute Ergebnisse, wenn das Optimum stark exponiert ist und wenig andere Parameterkonfigurationen ähnlich gute Leistungswerte erzielen.

Aus diesen Beobachtungen folgt, dass ein Suchraum mit Tuning-Parametern der gleichen oder ähnlichen Sensitivität eher mit einem globalen Suchverfahren bearbeitet werden sollte, wohingegen ein Suchraum, der unter allen Tuning-Parameter einige wenige mit sehr hoher Sensitivität aufweist, mit einem lokalen Suchverfahren durchlaufen werden kann.

7.2.3.4 Fazit

Die Ergebnisse der obigen Experimente zeigen, dass kontextbasierte Suchraumreduktion nicht nur den Suchraum auf seine wesentlichen Teile beschränken, sondern auch zu einer wesentlich effizienteren und damit kürzeren Suche verhelfen kann. Die Qualität der Tuning-Heuristiken wird durch die gleichbleibende, oftmals sogar bessere Präzision der Suchverfahren belegt.

Des Weiteren wird deutlich, dass es lohnenswert sein kann, eine etwas geringere Präzision in Kauf zu nehmen, dafür aber eine deutlich reduzierte Laufzeit des Optimie-

rungsprozesses zu erreichen. Nicht immer ist es zwingend erforderlich, das tatsächliche Leistungsoptimum zu finden, da der zusätzliche Leistungsgewinn im Vergleich zu einer wesentlich schneller gefundenen semi-optimalen Parameterkonfiguration zwar messbar, aber oft kaum spürbar ausfällt.

Zusammenfassend sei festgehalten, dass durch Suchraumpartitionierung und kontextbasierte Suchraumreduktion das Offline-Tuning überhaupt erst konkurrenzfähig wird und als allgemeines Optimierungskonzept für parallele Applikationen eingesetzt werden kann.

7.2.4 Allgemeine Leistungsmessungen

Nicht zuletzt stellt sich die Frage, welche Leistung die parallelisierten und optimierten Programme schließlich erreichen. Es gilt demnach herauszufinden, wie gut die jeweiligen Implementierungen der Parallelisierungsstrategien von Atune-TA die Programme beschleunigen und welche Verbesserung der anschließende Optimierungsprozess von Atune-OPT erwirkt.

7.2.4.1 Evaluationsmetriken

Um die Leistung der parallelen und optimierten Programmversionen beurteilen zu können, wird der Geschwindigkeitszuwachs (engl. *speedup*) ermittelt, den die parallelen Programmversion im Vergleich zur sequentiellen Variante erreicht. Im Rahmen dieser Evaluation sind zwei unterschiedliche Maßzahlen für den Geschwindigkeitsgewinn der Programme von Interesse.

Zum einen soll die bestmögliche Leistung der Programme ermittelt werden, die durch Parallelisierung sowie anschließender Optimierung erreicht wird. Zum anderen soll jedoch auch der alleinige Einfluss der Optimierung auf die Leistung der Programme bestimmt werden, um die Relevanz des Tunings zu belegen.

Für die Evaluation werden daher Speedup-Werte in Abhängigkeit von den drei folgenden Parameterkonfigurationen benötigt:

- **Speedup mit optimaler Parameterkonfiguration.** Gibt den Geschwindigkeitsgewinn an, den die *optimale* Parameterkonfiguration gegenüber der sequentiellen Programmversion erzielt. Es wird also ausgedrückt, um wie viel das parallele optimierte Programm maximal schneller ist als seine sequentielle Version. Zur Ermittlung der optimalen Parameterkonfiguration sei auf die entsprechenden Anmerkungen im vorherigen Abschnitt verwiesen.

- **Speedup mit bester (gefundener) Parameterkonfiguration.** Gibt den Geschwindigkeitsgewinn an, den die *beste* gefundene Parameterkonfiguration des Auto-Tuners gegenüber der sequentiellen Programmversion erzielt. Hier wird demnach die maximale Leistung des Auto-Tuners bewertet. Die Differenz zwischen dem Leistungswert der optimalen und der besten gefundenen Konfiguration ergibt die Fehlerrate des Tuners.

- **Speedup mit schlechtester (gefundener) Parameterkonfiguration.** Gibt den Geschwindigkeitsgewinn an, den die *schlechteste* gefundene Parameterkonfiguration gegenüber der nicht optimierten sequentiellen Programmversion erzielt und ermittelt somit den minimalen Geschwindigkeitsgewinn der parallelen Programmversion.

- **Speedup mit Standardkonfiguration.** Gibt den Geschwindigkeitsgewinn an, der durch eine sinnvolle, aber nicht empirisch ermittelte Standardkonfiguration der Tuning-Parameter erzielt wird. Die konkreten Parameterwerte, die einer Standardkonfiguration im Rahmen dieser Evaluation zu Grunde gelegt wurden, sind Tabelle 7.10 zu entnehmen. Die Wahl der Standardwerte soll ein gewisses Maß an Entwicklerwissen berücksichtigen, jedoch keine Tuning-Heuristiken oder anderweitige Annahmen voraussetzen.

Tuning-Parameter	Standardeinstellung
Tunable Fork/Join	
`NumThreads`	# Rechenkerne
Tunable Producer/Consumer	
`BufferSize`	0 (dynamische Puffergröße)
`BatchSize`	0 (Element sofort lesen)
Tunable Pipeline	
`StageFusion`	vollständig deaktiviert
Tunable Replication	
`NumInstances`	# Rechenkerne (in einem Fließband: 1)
`LoadBalancing`	*static*
`BatchSize`	0 (Element sofort lesen)

Tabelle 7.10: Standardeinstellungen der Tuning-Parameter.

Aus den obigen Maßzahlen für den Geschwindigkeitszuwachs lässt sich die Metrik des *Tuning Performance Gain* ableiten.

- **Tuning Performance Gain (TPG) 1.** Der TPG 1 gibt die Differenz in Prozent zwischen dem Speedup mit der *schlechtesten* und dem mit der *besten* gefundenen Parameterkonfiguration an, die das Suchverfahren ermittelt hat. Mit dieser Metrik wird die Leistungsspanne des parallelen Programms angegeben, die allein durch Konfiguration der Tuning-Parameter durch den Auto-Tuner erreicht werden kann.
- **Tuning Performance Gain (TPG) 2.** Der TPG 2 gibt die Differenz in Prozent zwischen dem Speedup mit der *Standard-Parameterkonfiguration* und dem mit der *besten* gefundenen Parameterkonfiguration an. Hier wird die Leistungsspanne angegeben, die zwischen einer manuell konfigurierten und nicht empirisch bestimmten Konfiguration und der automatische ermittelten besten Parameterkonfiguration liegt.

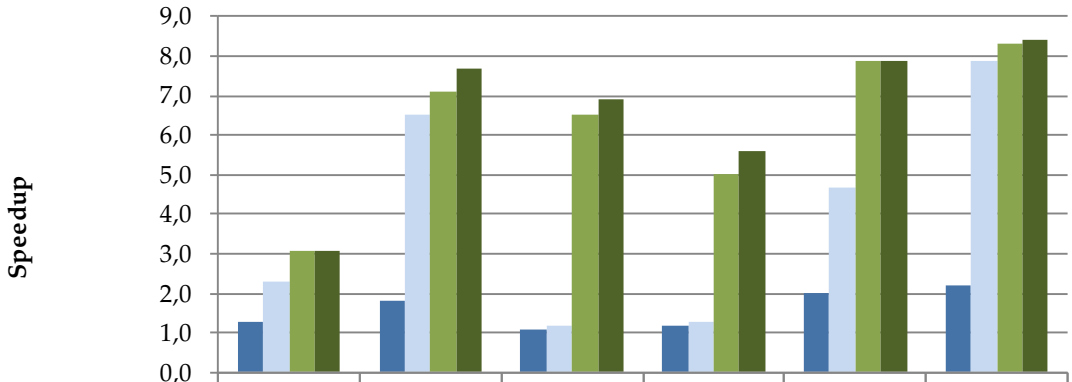
Der TPG ist somit ein Indikator für den Einfluss der Optimierung auf das parallele Programm und gibt die maximal mögliche Leistungsausbeute durch Auto-Tuning an.

Mit der Leistungsmessung wird die Effizienz sowie die Funktionsfähigkeit aller Teilkonzepte dieser Arbeit untersucht, da der erzielte Speedup eines Programms (also die Leistungssteigerung durch Parallelisierung *und* Optimierung) das Ergebnis des Zusammenspiels aller Konzepte darstellt.

Für die Leistungsmessungen wurde pro Fallstudie eines der drei Suchverfahren verwendet. Die Auswahl wurde gemäß den Ergebnissen aus vorherigem Abschnitt getroffen. Für jedes Programm wurde dasjenige Suchverfahren verwendet, das die geringste Fehlerrate aufwies. Demnach wurde für die Optimierung der Desktopsuche, des Videoverarbeitungsprogramm sowie der algorithmischen Fallbeispiele das Hillclimbing-Verfahren mit Check-Back-Schritt verwendet. Bei MetaboliteID und GrGen wurde die Partikel-Schwarm-Optimierung eingesetzt.

7.2.4.2 Ergebnisse im Überblick

Das Diagramm in Abbildung 7.2 stellt die konkreten Ergebnisse dar. Neben den Speedup-Werten für die optimale, die beste und schlechteste gefundene sowie die Standardkonfiguration (*Optimale Konfig.*, *Beste Konfig.*, *Schlechteste Konfig.* bzw. *Standardkonfig.*) sind für jedes der sechs Programme die beiden daraus resultierenden TPG-Werte (*TPG 1* und *TPG 2*) angegeben.



	MID	GrGen	DS	Video	MB	Matrix
Schlechteste Konfig.	1,3	1,8	1,1	1,2	2,0	2,2
Standardkonfig.	2,3	6,5	1,2	1,3	4,7	7,9
Beste Konfig.	3,1	7,1	6,5	5,0	7,9	8,3
Optimale Konfig.	3,1	7,7	6,9	5,6	7,9	8,4
TPG 1 →	138%	294%	491%	317%	295%	277%
TPG 2 →	35%	9%	442%	285%	68%	5%

Abbildung 7.2: Ergebnisse der Leistungsmessungen.

Im Folgenden werden die Ergebnisse der einzelnen Fallstudien aufgeführt und diskutiert.

7.2.4.3 MetaboliteID

Der beste Speedup, der nach der Optimierung der parallelen Version von MetaboliteID erreicht wurde, ist mit $3,1$ auf einem 8-Kern-Rechner nur moderat. Es ist allerdings zu beachten, dass MetaboliteID ausschließlich als sequentielles Programm entworfen und entwickelt wurde. Insbesondere die Ein- und Ausgaberoutinen für das Laden, Aktualisieren und Speichern der Metabolitendatenbank bieten kaum Parallelisierungspotential. Die Ausnutzung von Aufgaben- und Datenparallelität beschleunigt die Ausführung der eigentlichen Algorithmen durchaus beträchtlich; die Beschleunigung des gesamten Programms hingegen ist durch die vielen sequentiellen Teile nach oben beschränkt (siehe hierzu Amdahls Gesetz [Amda67]).

Dennoch erzielte Atune-OPT durch die Optimierung der parallelen Version insgesamt eine Leistungssteigerung von maximal 138% (TPG 1) bzw. 35% (TPG 2).

7.2.4.4 GrGen.NET

Die Datenparallelität in GrGen.NET ließ sich nahezu optimal ausnutzen, so dass nach Parallelisierung und Optimierung ein Speedup von maximal 7,1 erreicht wurde, was auf einem 8-Kern-Rechner nahezu einem linearen Geschwindigkeitszuwachs entspricht. Der Speedup wurde jeweils in den beiden parallelen Programmteilen `PromoterSearch` und `RnaPoly` gemessen und anschließend aggregiert.

Der Optimierungsprozess zeigte sich als äußerst wirksam, da alleine durch das Auto-Tuning bis zu 294% (TPG 1) bzw. 9% (TPG 2) an Geschwindigkeit gewonnen wurde.

7.2.4.5 Desktopsuche

Das Fallbeispiel der Desktopsuche nimmt bei der Leistungsmessung eine besondere Rolle ein. Die Implementierung des Indizierers mittels des KMP-Algorithmus wurde zwar von Atune-OPT gegenüber der naiven Implementierung auf der Basis eines herkömmlichen String-Vergleichers als die schnellere Variante ermittelt, dennoch betrug die Ausführungszeit der sequentiellen Version unter Verwendung des KMP-Algorithmus über 14 Stunden. Die lange Laufzeit ist mit der Funktionsweise des KMP-Algorithmus zu erklären, der ausgehend von einer Liste mit vordefinierten Schlüsselwörtern jede zu indizierende Datei daraufhin überprüft, ob sie eines oder mehrere der Schlüsselwörter enthält. Zusätzlich wird die Anzahl der Treffer pro Datei für jedes Schlüsselwort ermittelt, um bei Suchanfragen eine Gewichtung angeben zu können.

Die separate dritte Variante erstellte den Index wesentlich schneller. Anstatt einer vordefinierten Wortliste bildeten die Wörter in den Dateien selbst die Schlüsselwörter der Indexdatenstruktur. Der Zeichenstrom der geöffneten Datei wurde hierzu zeichenweise gelesen und an Hand von Trennzeichen (wie beispielsweise Leerzeichen oder Tabulatoren) in einzelne Wörter zerlegt. Anschließend wurden die auf diese Weise gefundenen Wörter mit einem entsprechenden Datei-Identifikator in die Indexdatenstruktur eingefügt. Die Laufzeit der sequentiellen Version reduzierte sich auf 239 Sekunden.

Auf Grund der unterschiedlichen Implementierungen und deren Funktionsweise sowie dem extremen Laufzeitunterschied haben wir bei den Leistungsmessungen beide Programmvarianten berücksichtigt, da sich auch bei den jeweiligen Leistungsgewinnen ein unterschiedliches Bild ergab.

Die Variante auf Basis des KMP-Algorithmus ermöglichte erwartungsgemäß eine gute Ausnutzung der Parallelität, da auf Grund des hohen Rechenanteils (Laufzeit des KMP-Algorithmus pro Datei) der limitierende Faktor der E/A-Operationen wenig Auswirkungen zeigte. Somit wurde ein maximaler Speedup von $6,5$ erreicht. Die notwendige Sperre auf der Indexdatenstruktur sowie das Einlesen der Daten von der Festplatte verhinderten einen linearen Speedup. Die tuning-bedingte Leistungssteigerung durch Atune-OPT lag bei maximal 491% (TPG 1) bzw. 442% (TPG 2). Der Auto-Tuner zog den Tunable Producer/Consumer-Konnektor dem Tunable Pipeline-Konnektor wegen geringfügig schnellerer Laufzeit vor.

Dei zweite Variante erreichte einen deutlich geringeren Speedup von maximal $3,1$. Dies begründet sich im Verhältnis zwischen reiner Rechenzeit und Laufzeit der E/A-Operationen. Der wesentlich weniger komplexe Algorithmus zum Auslesen des Zeichenstroms wies bereits in der sequentiellen Programmversion eine derart kurze Laufzeit auf, dass sich in diesem Fall das Einlesen der Daten von der Festplatte als Flaschenhals herausstellte. Der Leistungsgewinn konnte nur durch das Erzeuger/Verbraucher-Konstrukt sowie

durch die Replizierung der Index-Aktualisierung erzielt werden. Das eigentliche Einlesen der Dateien sowie das Zerlegen des Zeichenstroms in Wörter bot kaum Parallelisierungspotential. Entsprechend dem geringeren Speedup lag die tuning-bedingte Leistungssteigerung durch Atune-OPT bei maximal 24% (TPG 1) bzw. 8% (TPG 2).

Das Fallbeispiel zeigt, dass das Parallelisierungspotential einer Anwendung in großem Maße von den verwendeten Implementierungen der Algorithmen abhängt. Des Weiteren werden die Auswirkungen des Gesetzes von Amdahl [Amda67] verdeutlicht, indem zuvor vernachlässigbare E/A-Operationen bei einer zweiten Implementierung mit weniger Rechenaufwand zu einem gravierenden Leistungsengpass führen.

7.2.4.6 Videoverarbeitung

Das Programm zur Videoverarbeitung konnte mit einem Speedup von $5,0$ zufriedenstellend beschleunigt werden.

Auf Grund sehr unterschiedlicher Laufzeiten der replizierbaren Fließbandstufen waren bei der Optimierung insbesondere die Balancierung der Stufen sowie deren geeignete Verkettung entscheidend. Die vom Auto-Tuner ermittelte beste Parameterkonfiguration erzielte einen optimierungsbedingten Leistungsgewinn von maximal 317% (TPG 1) bzw. 285% (TPG 2) und zeigt, welch wichtige Rolle die automatische Performanzoptimierung bei der Parallelisierung einnimmt.

7.2.4.7 Berechnung der Mandelbrotmenge

Die parallelisierte und optimierte Berechnung der Mandelbrotmenge erzielte einen Speedup von $7,9$. Die uneingeschränkte Datenparallelität des algorithmischen Problems führte zu einem fast linearen Geschwindigkeitszuwachs, einzig die Zusammenführung der Einzelergebnisse bremst die Leistung des parallelen Algorithmus leicht ein.

Dennoch konnte Atune-OPT einen Leistungsgewinn von maximal 295% (TPG 1) bzw. 68% (TPG 2) erreichen.

7.2.4.8 Matrix-Matrix-Multiplikation

Die parallele Version der Matrix-Matrix-Multiplikation konnte bei optimaler Konfiguration einen Leistungsgewinn von 8,3 erreichen, was auf einem 8-Kern-Rechner einem optimalen Speedup entspricht. Das algorithmische Problem eignet sich auf Grund der unabhängigen Gebietszerlegung der Daten in bester Weise zur Ausnutzung der Datenparallelität.

Atune-OPT konnte durch Optimierung der Parameter einen Leistungsgewinn von maximal 277% (TPG 1) bzw. 5% (TPG 2) erreichen.

7.2.4.9 Fazit

Atune-OPT konnte durch die Anwendung der Optimierungskonzepte bei allen Applikationen einen beachtlichen Geschwindigkeitszuwachs erzielen. Bei genauerer Betrachtung fällt auf, dass die Werte für den TPG 2 bei Programmen mit reiner Datenparallelität deutlich geringer sind als bei Programmen, die Fließbandparallelität in Kombination mit anderen parallelen Sektionen beinhalten.

Dies lässt sich auf die Tatsache zurückführen, dass in einer datenparallelen Sektion der Tuning-Parameter `NumInstances` (Anzahl an parallelen Instanzen, siehe Tabelle 7.10) zum einen eine hohe Sensitivität besitzt und zum anderen intuitiv mit einer guten Standardeinstellung belegt werden kann. Insbesondere können daher Programme mit nur einer datenparallelen Sektion bereits mit einer Standardkonfiguration akzeptabel beschleunigt werden.

Bei komplexeren Architekturen mit geschachtelter Parallelität führt die Standardkonfiguration durch die vorhandenen Abhängigkeiten und die nicht genau vorhersagbare Beeinflussung der Tuning-Parameter untereinander zu einem weniger guten Ergebnis, so dass eine weiterführende Optimierung dringend erforderlich ist. Gleiches gilt in besonderem Maße für Fließbänder mit replizierbaren Stufen, die ein komplexes paralleles Konstrukt darstellen, das eine große Zahl an Tuning-Parametern bereitstellt. Hier kann nur eine konservative Standardkonfiguration gewählt werden. Die Optimierung führt daher zu einem großen Leistungsgewinn, was durch die hohen Werte des TPG 2 bestätigt wird.

Zusammenfassend kann festgestellt werden, dass die Ergebnisse zum einen zeigen, dass die Parallelisierungsstrategien effizient umgesetzt wurden und zum anderen, dass erst die Kombination aus Parallelisierung und Optimierung einem Programm zu seiner bestmöglichen Leistung verhelfen kann.

7.3 Erfüllung der Thesen

In Kapitel 2 wurden gemäß den Zielsetzungen dieser Arbeit vier Thesen aufgestellt, die durch die oben vorgestellten experimentellen Ergebnisse belegt werden können. Im Folgenden werden die Ergebnisse den einzelnen Thesen zugeordnet.

- **Zu These 1.** Die Leistungsmessungen in den sechs Fallstudien haben gezeigt, dass unter Verwendung aller Konzepte dieser Arbeit ein breites Spektrum an unterschiedlichen Applikationen automatisch optimiert werden kann.
- **Zu These 2.** Die Tuning-Instrumentierungssprache Atune-IL bietet ein einfaches und intuitives Sprachkonzept und ermöglicht neben der Definition herkömmlicher tuning-relevanter Informationen auch die Spezifikation neuartiger Tuning-Instruktionen (wie beispielsweise Tuning-Blöcke). Atune-IL ermöglicht die Suchraumpartitionierung und steigert somit die Effizienz des automatischen Optimierungsprozesses signifikant.

  Des Weiteren wird durch Atune-IL die nötige Trennung zwischen Auto-Tuner und dem zu optimierendem Programm erreicht.
- **Zu These 3.** Mit Atune-TA und der Beschreibungssprache TADL können parallele, optimierbare Architekturen entworfen werden. Die Fallstudien haben gezeigt, dass die automatisierte Umsetzung des Architekturmodells in ein lauffähiges paralleles Programm dank entsprechender Implementierungstechniken funktioniert.

  Hierbei wird von der konkreten Parallelisierung und der für die spätere Optimierung erforderlichen Adaptionsfunktionalität nahezu vollständig abstrahiert, was sowohl den Implementierungsaufwand als auch die Fehleranfälligkeit drastisch senkt.
- **Zu These 4.** Der Einsatz von Tuning-Heuristiken hat gezeigt, dass durch die Ausnutzung von Kontextinformationen über die Programmstruktur sowie über die eingesetzten Parallelisierungsstrategien der Suchraum wesentlich reduziert werden kann, ohne dass die Präzision der Suchverfahren sich wesentlich verschlechtert.

  Die Fallstudien haben veranschaulicht, dass erst durch den Einsatz der kontextbasierten Suchraumreduktion auch große parallele Anwendungen mit suchbasierten Methoden optimiert werden können.

Die durchgeführten Fallstudien konnten die Behauptungen der Thesen lückenlos belegen, womit die Zielsetzung dieser Arbeit erreicht ist.

7.4 Zusammenfassung

In diesem Kapitel wurden die Konzepte und Ansätze dieser Arbeit, aber auch die darauf aufbauenden Implementierungstechniken ausführlich an Hand von sechs Fallstudien evaluiert.

Bei Atune-TA und der Beschreibungssprache TADL war die Einsparung an Implementierungsaufwand sowie die Verringerung der Fehleranfälligkeit bei der Parallelisierung von zentralem Interesse. Der hohe Automatisierungsgrad sowie die Abstraktion von konkreten Parallelisierungstechniken und Optimierungsprozessen verhalfen Atune-TA zu äußerst vielversprechenden Ergebnissen.

Atune-IL wurde insbesondere auf seine Funktionalität sowie auf die Fähigkeit zur Unterstützung der Suchraumpartitionierung untersucht. Die Ergebnisse haben gezeigt, dass Atune-IL zur Effizienz der suchbasierten Optimierung in großem Maße beitragen kann.

Des Weiteren wurde die Effektivität der kontextbasierten Suchraumreduktion untersucht. Die Experimente haben gezeigt, dass Atune-OPT durch die Anwendung von Tuning-Heuristiken bei durchschnittlich kaum größerer Fehlerrate deutlich weniger Tuning-Iterationen benötigt.

Die Leistungsmessungen haben schließlich die Funktionsfähigkeit aller Konzepte belegt: Die parallelisierten und optimierten Programmversionen erreichten im Durchschnitt gute Beschleunigungswerte. Zudem wurde durch die Metrik des *Tuning Performance Gain* der Einfluss des Auto-Tuning auf die Leistung paralleler Programme verdeutlicht sowie die Aussage bekräftigt, dass Optimierung für eine erfolgreiche Parallelisierung unerlässlich ist.

8. Zusammenfassung und Ausblick

In dieser Arbeit wurden mehrere Problemstellungen im Bereich der automatischen Performanzoptimierung paralleler Architekturen bearbeitet und entsprechende Lösungsansätze vorgestellt und erörtert.

Das letzte Kapitel dieser Arbeit fasst die Problemstellungen und zugehörigen Lösungsansätze sowie die Evaluationsergebnisse noch einmal zusammen. Schließlich wird ein Ausblick auf Möglichkeiten zur Weiterentwicklung dieser Arbeit sowie auf noch zu erörternde Forschungsfragen gegeben.

8.1 Zusammenfassung der Arbeit

Zunächst wurden die wichtigsten Grundbegriffe des Auto-Tuning definiert sowie die gängigen Strategien zur Ermittlung des Leistungsoptimums dargestellt. Anschließend wurden die wichtigsten Auto-Tuning-Techniken analysiert und an Hand einer für diesen Zweck entwickelten Taxonomie klassifiziert. Danach wurden die Grundlagen optimierbarer Programme und deren Architektur besprochen sowie der Nutzen paralleler Entwurfsmuster als Architekturbausteine erläutert. Die Definition und Erörterung der fachlichen Grundlagen schloss mit einer Abhandlung über die Herausforderungen, die bei der Erweiterung herkömmlicher Auto-Tuning-Ansätze entstehen können.

Die Diskussion verwandter Arbeiten umfasste die Vorstellung von Ansätzen aus dem Bereich der Tuning-Sprachen und dem Gebiet der konfigurierbaren Architekturen. Ein weiterer Schwerpunkt lag auf Arbeiten, die sich mit automatischer Performanzoptimierung auseinandersetzen. Alle vorgestellten Verfahren wurden entsprechend ihrer fachlichen Zuordnung untereinander verglichen und bewertet.

Die Erörterung der Konzepte dieser Arbeit wurde in vier Bereiche unterteilt. Zu Beginn wurden mit der Auto-Tuner-Separation, dem Auto-Tuning-Zyklus sowie der Suchraumpartitionierung grundlegende Konzepte dargelegt, mit deren Hilfe die automatischen Performanzoptimierung von konkreten Anwendungsbereichen und speziellen Hardware-Plattformen abstrahiert und für Applikationen ausgelegt werden kann, deren Umfang und Komplexität sich in softwaretechnisch relevanten Größenordnungen bewegt.

Anschließend wurden die drei aufeinander aufbauenden Teilkonzepte Atune-IL, Atune-TA und Atune-OPT vorgestellt und erklärt.

Mit Atune-IL wurde eine allgemein einsetzbare Tuning-Instrumentierungssprache präsentiert, deren einfaches und zugleich mächtiges Sprachkonzept die Definition umfangreicher Tuning-Instruktionen ermöglicht. Neben der Deklaration von Tuning-Parametern und Permutationsbereichen werden die Definition von Messpunkten sowie die Spezifikation von Suchraumpartitionen unterstützt, womit sich Atune-IL deutlich von verwandten Arbeiten absetzt.

Anschließend wurde Atune-TA als ein Konzept für den Entwurf und die Entwicklung einer parallelen optimierbaren Programmarchitektur vorgestellt. Die Kernaspekte von Atune-TA sind die Zusammenführung von Parallelisierung und Optimierung, einheitliche Entwurfs- und Implementierungsrichtlinien, die automatisierte Umsetzung des Entwurfs sowie die integrierte Unterstützung für automatische Performanzoptimierung. Mit TADL bietet Atune-TA eine passende Beschreibungssprache, mit der die Interaktion atomarer Programmkomponenten sowie deren parallele Verarbeitung beschrieben werden kann. Nicht zuletzt an Hand von Beispielen wurde gezeigt, dass Atune-TA zusammen mit TADL im Kontext des Architekturentwurfs erstmals Aspekte der Parallelisierung sowie der automatischen Performanzoptimierung kombiniert und somit ein Modell paralleler optimierbarer Architekturen repräsentiert.

Das dritte Teilkonzept beschäftigte sich mit Atune-OPT, einem suchbasierten Offline-Tuner für parallele Architekturen. Atune-OPT zeichnet sich gegenüber existierenden Auto-Tuning-Ansätzen insbesondere durch sein mehrstufiges Optimierungskonzept aus, das es ermöglicht, auch große parallele Applikationen unter Verwendung von Suchraumpartitionierung und kontextbasierter Suchraumreduktion automatisch zu optimieren. Daneben wurden aber auch die verwendeten Suchalgorithmen diskutiert sowie die Möglichkeit zur Parallelisierung des Optimierungsprozesses selbst. Mit Atune-OPT wurden alle Teilkonzepte der Arbeit zusammengeführt, so dass mit dem Gesamtkonzept *Atune* der Prozess vom Entwurf über die Implementierung und Instrumentierung bis hin zur automatischen Optimierung vollständig abgedeckt ist.

Im Anschluss an den konzeptionellen Teil der Arbeit wurde die prototypische Implementierung der drei Teilkonzepte dargelegt. Neben dem Atune-IL-Backend, das den Auto-Tuning-Zyklus implementiert, wurden insbesondere der TADL-Übersetzer und dessen Integration in eine Entwicklungsumgebung erläutert. Hierbei standen das Konzept der Tuning-Wrapper sowie die automatische Generierung von Atune-IL-Instrumentierungen im Mittelpunkt. Schließlich wurde der Prototyp von Atune-OPT mit den entsprechenden Modulen zur Umsetzung des Optimierungsprozesses dargestellt.

Jedes der drei Teilkonzepte wurde an Hand von fünf Fallstudien evaluiert. Es wurde gezeigt, dass Atune-IL in Programmen unterschiedlicher Größe und Anwendungsdomäne eingesetzt werden und zu einer signifikanten Reduktion des Suchraums durch Partitionierung beitragen kann. Bei der Bewertung von Atune-TA wurden die Reduktion des Implementierungsaufwandes und die der Fehleranfälligkeit bei der Parallelisierung bewertet; für beide Metriken wurden vielversprechende Ergebnisse erzielt.

Außerdem wurde die Qualität der Tuning-Heuristiken evaluiert, indem gezeigt wurde, dass durch deren Einsatz die Anzahl an Tuning-Iterationen bei gleicher Präzision der Suchverfahren deutlich verringert werden konnte. Schließlich wurden Leistungsmessungen durchgeführt, die die Funktionsfähigkeit aller Konzepte belegen konnten. Die parallelisierten und optimierten Programmversionen erreichten im Durchschnitt akzeptable Beschleunigungswerte. Zudem wurde durch die Metrik des Tuning Performance Gain der Einfluss des Auto-Tuning auf die Leistung paralleler Programme verdeutlicht sowie die Aussage bekräftigt, dass Optimierung für eine erfolgreiche Parallelisierung unerlässlich ist.

8.2 Ausblick und Forschungspotential

Das Thema der automatischen Performanzoptimierung paralleler Architekturen wurden im Rahmen dieser Arbeit tiefgehend erforscht. Neben den vorgestellten Konzepten und Lösungsansätzen stellt die Bildung einer Grundlage für aufbauende Forschungsansätze einen Beitrag dieser Arbeit dar. Im Folgenden werden daher die wichtigsten Aspekte und Möglichkeiten zur Weiterentwicklung dargelegt.

Als logische Weiterentwicklung der gezeigten Konzepte ist die Kombination von Offline- und Online-Tuning zu nennen. Es sollte untersucht werden, in wie weit Online-Tuning in die bestehenden Ansätze integriert werden kann, um auch langlaufende Programme optimieren zu können. Insbesondere wäre zu erforschen, in welchem Maß die Abhängigkeit des Optimierungsprozesses von den Eingabedaten durch die Integration von Online-Tuning reduziert werden kann.

Im Grundlagen-Kapitel wurden bereits verschiedene Möglichkeiten zur Integration eines Auto-Tuners aufgezeigt. Da in dieser Arbeit das Konzept der eigenständigen Auto-Tuning-Komponente verfolgt wurde, wäre im Besonderen die Integration des Auto-Tuners in Übersetzer oder Betriebssysteme von Relevanz. Durch derartige Verfahren könnten neue Optimierungskonzepte erforscht werden, wie beispielsweise die gleichzeitige Optimierung mehrerer Anwendungen durchgeführt werden kann. Auch die Verbindung von Auto-Tuner und Ablaufplaner des Betriebssystems bietet interessante Problemstellungen, die es zu lösen gilt. An dieser Stelle sei auch auf einen Artikel verwiesen, den wir bereits veröffentlichen konnten [KaSP09].

Im Bereich der optimierbaren Architekturen stellt sich die Frage, ob eine noch schnellere und auf das konkrete Problem abgestimmte Implementierung der Parallelisierungsstrategien möglich ist. Es wäre beispielsweise denkbar, die Parallelisierungslogik nicht in einer Bibliothek bereitzustellen, sondern wie die Tuning-Hüllklassen ebenfalls durch den TADL-Übersetzer generieren zu lassen. Auf diese Weise könnte gerade bei kleinen Programmen mit kurzer Laufzeit der Mehraufwand vermieden werden, der durch das Einbinden der Bibliothek entsteht. Stattdessen würde ausschließlich der für die konkrete Parallelisierung benötigte Quelltext generiert.

Großes Forschungspotential bietet die kontextbasierten Suchraumreduktion. Hier gilt es weitere Heuristiken zu entwickeln, mit denen ein umfangreicheres Spektrum an Parallelisierungsstrategien abgedeckt werden kann. Außerdem stellt sich die Frage, über welche weiteren Wege die erforderlichen Kontextinformationen aus einem Programm extrahiert werden können. Ein möglicher Ansatz wäre beispielsweise die Verwendung von Spracherweiterungen, mit denen parallele Konstrukte ausgedrückt werden können. Der hierfür nötige Übersetzer würde über wesentlich mehr Kontextwissen verfügen, so dass derartige Informationsquellen auf ihre Eignung zur Unterstützung kontextbasierter Optimierung untersucht werden sollten.

Naturgemäß stellen auch die Suchalgorithmen zur Durchführung des Optimierungsprozesses einen zentralen Forschungsbereich dar. In dieser Arbeit konnte gezeigt werden, dass ein einziger Suchalgorithmus in keinem Fall für alle Arten von Programmstrukturen und Parallelisierungsstrategien effizient eingesetzt werden kann. Dies führt zu der Frage, unter welchen Umständen welcher Suchalgorithmus am besten geeignet ist. Die Idee zur automatisierten Auswahl eines passenden Suchalgorithmus wurde im Rahmen der Optimierungstechniken von Atune-OPT vorgestellt. Der Ansatz repräsentiert jedoch nur einen ersten Schritt auf dem Weg zu einem dynamischen Einsatz der Suchalgorithmen, so dass eine Ausarbeitung dieser Idee äußerst lohnenswert erscheint.

A. Sprachgrammatiken

Im Folgenden sind die vollständigen Sprachgrammatiken von *Atune-IL* und der *Tunable Architecture Description Language (TADL)* aufgeführt. Beide Grammatiken wurden für die Verarbeitung durch den Parsergenerator *ANTLR* [antl09] entworfen. Die Syntax nutzt daher zum Teil ANTLR-spezifische Elemente, entspricht aber im Wesentlichen der *Backus-Naur-Form (BNF)*.

Die Produktionen der Grammatiken definieren Nichtterminalsymbole in der Form

```
symbol : <ausdruck>;
```

wobei `symbol` ein Nichtterminalsymbol darstellt und `<ausdruck>` aus einem oder mehreren Symbolen besteht. Symbole, die nie auf der linken Seite einer Produktion erscheinen, sind Terminalsymbole (erkenntlich durch GROSSBUCHSTABEN).

A.1 Atune-IL-Grammatik

Die Sprachgrammatik von Atune-IL definiert die Zusammensetzung der Deklarationen von Tuning-Parametern, Messpunkten und Tuning-Blöcken.

```
start    :  commands*
         ;

commands
         :  PREFIX (setvar|block|gauge) TERMINATOR
         ;


setvar   :  SETVAR IDENTIFIER type DECLARE?
             scale? context? depends? weight?
         ;

type     :  TYPE
             (INTTYPE intvalues (intstep)?
             |BOOLTYPE
             |FLOATTYPE floatvalues floatstep
```

```
            |STRINGTYPE stringvalues
            |GENERICTYPE stringvalues
            )
        ;

scale   :   SCALE (NOMINAL | ORDINAL)
        ;

context :   CONTEXT
            (NUMTHREADS
            |GENERAL
            |NUMPIPELINESTAGES
            )
        ;

depends :   DEPENDS IDENTIFIER (intvalues | floatvalues)
        ;

weight  :   WEIGHT DECIMAL_LITERAL
        ;

block   :   STARTBLOCK (id=IDENTIFIER)?
            (INSIDE (inid=IDENTIFIER))?
            TERMINATOR
            commands*
            PREFIX ENDBLOCK
        ;

gauge   :   GAUGE IDENTIFIER
        ;

intvalues
        :   VALUES intspan (SEMICOLON intspan)*
        ;

floatvalues
        :   VALUES floatspan (SEMICOLON floatspan)*
        ;

intstep
        :   STEP DECIMAL_LITERAL
        ;

floatstep
        :   STEP FLOATING_POINT_LITERAL
        ;

intspan
        :   DECIMAL_LITERAL
            |DECIMAL_LITERAL MINUS DECIMAL_LITERAL;

floatspan
        :   FLOATING_POINT_LITERAL
            |FLOATING_POINT_LITERAL MINUS FLOATING_POINT_LITERAL;

stringvalues
        :   VALUES (STRING) (SEMICOLON (STRING))*;
```

Listing A.1: Vollständige Grammatik der Tuning-Instrumentierungssprache Atune-IL.

A.2 TADL-Grammatik

Die Grammatik der TADL definiert die Kompositionsmöglichkeiten von Konnektoren und atomaren Komponenten.

```
start   :   connector
        ;

connector
        :   (tunablePipeline
            |tunableForkJoin
            |tunableProducerConsumer
            |tunableAlternative
            |sequentialComposition)
        ;

tunablePipeline
        :   TUNABLEPIPELINE (IDENTIFIER)?
            streamDataAttr?
            itemList
        ;

tunableForkJoin
        :   TUNABLEFORKJOIN (IDENTIFIER)?
            batchDataAttr?
            itemList
        ;

tunableProducerConsumer
        :   TUNABLEPRODUCERCONSUMER (IDENTIFIER)?
            streamDataAttr?
            itemList
        ;

sequentialComposition
        :   SEQUENTIALCOMPOSITION (IDENTIFIER)?
            batchDataAttr?
            itemList
        ;

tunableAlternative
        :   TUNABLEALTERNATIVE (IDENTIFIER)?
            itemList
        ;

itemList
        :   '{'!
            (atomicComponent|connector)
            (','! (atomicComponent|connector))*
            '}'!
        ;

atomicComponent
        :   ACIDENTIFIER ('['! REPLICABLE']'!)?
            (('['! PREPARALLEL ']'!)
            |('['! POSTPARALLEL ']'!))?
        ;
```

```
batchDataAttr
        :   '['! inputAttr ';'!  outputAttr ']'!
        ;

streamDataAttr
        :   '['! sourceAttr ';'!  sinkAttr ']'!
        ;

sourceAttr
        :   SOURCE ':'! (ACIDENTIFIER|NULL)
        ;

sinkAttr
        :   SINK ':'! (ACIDENTIFIER|NULL)
        ;

inputAttr
        :   INPUT ':'! acIdentifierList
        ;

outputAttr
        :   OUTPUT ':'! acIdentifierList
        ;

acIdentifierList
        :   (ACIDENTIFIER | NULL)
            (','! (ACIDENTIFIER | NULL))*
        ;
```

Listing A.2: Vollständige Grammatik der Tunable Architecture Description Language (TADL).

B. Quelltextbeispiel für Tuning-Hüllklasse

Im Folgenden wird an Hand eines konkreten Beispiels die Übersetzung eines TADL-Konnektors in eine entsprechende Tuning-Hüllklasse gezeigt. Hierfür soll der Tunable Pipeline-Konnektor aus der Architekturbeschreibung der parallelen Desktopsuche herangezogen werden. Listing B.1 gibt den entsprechenden Ausschnitt aus dem TADL-Skript der Desktopsuche wieder (vgl. hierzu das vollständige TADL-Skript im Evaluations-Kapitel, Abschnitt 7.1.3).

```
...
TunablePipeline
[source:Crawl;sink:CreateIndexFile] {
    TunableAlternative {
        StringSearch1[replicable],
        StringSearch2[replicable]
    },
    UpdateIndex[replicable]
}
...
```

Listing B.1: Auszug aus dem TADL-Skript zur Architekturbeschreibung der parallelen Desktopsuche.

Das Quelltextbeispiel in Listing B.2 zeigt die entsprechende Tuning-Hüllklasse des Tunable Pipeline-Konnektors, wie sie der TADL-Übersetzer generiert hat.

```
1  public class TunablePipeline_1
2  {
3      Engine engine;

5      public void Init(Engine a_engine)
6      {
7          engine = a_engine;
8      }

10 #pragma atune startblock PIPELINE_TunablePipeline_1 pattern PIPELINE
11     public void Run()
12     {
13         String PIPELINE_ALTERNATIVE_Alternative_1 = string.Empty;

15 #pragma atune setvar PIPELINE_ALTERNATIVE_Alternative_1 type string
```

```
16      values "Engine.StringSearch1";"Engine.StringSearch2"
17      startvalue "Engine.StringSearch1" scale nominal

19          Int32 REPL_NUMTHREADS_NumThreads_1_AC_StringSearch1 = 2;
20          Int32 REPL_BATCHSIZE_DlbBatchSize_1_AC_StringSearch1 = 0;
21          LoadBalancing REPL_LB_LoadBalancing_1_AC_StringSearch1 = LoadBalancing.Dynamic;
22          Int32 REPL_NUMTHREADS_NumThreads_2_AC_StringSearch2 = 2;
23          Int32 REPL_BATCHSIZE_DlbBatchSize_2_AC_StringSearch2 = 0;
24          LoadBalancing REPL_LB_LoadBalancing_2_AC_StringSearch2 = LoadBalancing.Dynamic;
25          Int32 REPL_NUMTHREADS_NumThreads_3_AC_UpdateIndex = 2;
26          Int32 REPL_BATCHSIZE_DlbBatchSize_3_AC_UpdateIndex = 0;
27          LoadBalancing REPL_LB_LoadBalancing_3_AC_UpdateIndex = LoadBalancing.Dynamic;

29  #pragma atune setvar REPL_NUMTHREADS_NumThreads_1_AC_StringSearch1 type int
30      values 2-16 startvalue 2 scale ordinal
31      depends PIPELINE_ALTERNATIVE_Alternative_1 values "Engine.StringSearch1"
32  #pragma atune setvar REPL_BATCHSIZE_DlbBatchSize_1_AC_StringSearch1 type int
33      values 0;5;10;15;20 startvalue 0 scale nominal
34      depends (PIPELINE_ALTERNATIVE_Alternative_1="Engine.StringSearch1")&
35          (REPL_LB_LoadBalancing_1_AC_StringSearch1="LoadBalancing.Dynamic")
36  #pragma atune setvar REPL_LB_LoadBalancing_1_AC_StringSearch1 type generic
37      values "LoadBalancing.Static";"LoadBalancing.Dynamic"
38      startvalue LoadBalancing.Dynamic scale nominal
39      depends PIPELINE_ALTERNATIVE_Alternative_1 values "Engine.StringSearch1"

41  #pragma atune setvar REPL_NUMTHREADS_NumThreads_2_AC_StringSearch2 type int
42      values 2-16 startvalue 2 scale ordinal
43      depends PIPELINE_ALTERNATIVE_Alternative_1 values "Engine.StringSearch2"
44  #pragma atune setvar REPL_BATCHSIZE_DlbBatchSize_2_AC_StringSearch2 type int
45      values 0;5;10;15;20 startvalue 0 scale nominal
46      depends (PIPELINE_ALTERNATIVE_Alternative_1="Engine.StringSearch2")&
47          (REPL_LB_LoadBalancing_2_AC_StringSearch2="LoadBalancing.Dynamic")
48  #pragma atune setvar REPL_LB_LoadBalancing_2_AC_StringSearch2 type generic
49      values "LoadBalancing.Static";"LoadBalancing.Dynamic"
50      startvalue LoadBalancing.Dynamic scale nominal
51      depends PIPELINE_ALTERNATIVE_Alternative_1 values "Engine.StringSearch2"

53  #pragma atune setvar REPL_NUMTHREADS_NumThreads_3_AC_UpdateIndex type int
54      values 2-16 startvalue 2 scale ordinal
55  #pragma atune setvar REPL_BATCHSIZE_DlbBatchSize_3_AC_UpdateIndex type int
56      values 0;5;10;15;20 startvalue 0 scale nominal
57      depends REPL_LB_LoadBalancing_3_AC_UpdateIndex values "LoadBalancing.Dynamic"
58  #pragma atune setvar REPL_LB_LoadBalancing_3_AC_UpdateIndex type generic
59      values "LoadBalancing.Static";"LoadBalancing.Dynamic"
60      startvalue LoadBalancing.Dynamic scale nominal

62          ReplicationPolicy ac_StringSearch1_Policy =
63              new ReplicationPolicy(REPL_NUMTHREADS_NumThreads_1_AC_StringSearch1,
64              REPL_BATCHSIZE_DlbBatchSize_1_AC_StringSearch1,
65              REPL_LB_LoadBalancing_1_AC_StringSearch1);
66          ReplicationPolicy ac_StringSearch2_Policy =
67              new ReplicationPolicy(REPL_NUMTHREADS_NumThreads_2_AC_StringSearch2,
68              REPL_BATCHSIZE_DlbBatchSize_2_AC_StringSearch2,
69              REPL_LB_LoadBalancing_2_AC_StringSearch2);
70          ReplicationPolicy ac_UpdateIndex_Policy =
71              new ReplicationPolicy(REPL_NUMTHREADS_NumThreads_3_AC_UpdateIndex,
72              REPL_BATCHSIZE_DlbBatchSize_3_AC_UpdateIndex,
73              REPL_LB_LoadBalancing_3_AC_UpdateIndex);

75          ReplicableItem<String, SearchResult> AC_StringSearch1 =
76              new ReplicableItem<String, SearchResult>(
77              engine.StringSearch1, ac_StringSearch1_Policy)
78              { Condition = ProcessingConditions.ExchangeCandidate };
79          ReplicableItem<String, SearchResult> AC_StringSearch2 =
80              new ReplicableItem<String, ParseResult>(
81              engine.StringSearch2, ac_StringSearch2_Policy)
82              { Condition = ProcessingConditions.Alternative };
83          ReplicableItem<SearchResult> AC_UpdateIndex =
84              new ReplicableItem<SearchResult>(
85              engine.UpdateIndex, ac_UpdateIndex_Policy)
86          ProcessableItem AC_CreateIndexFile =
87              new ProcessableItem(engine.CreateIndexFile);

89          FusionMap map = new FusionMap(2);
```

```
90          bool PIPELINE_STAGEFUSION_1_enable = false;
91  #pragma atune setvar PIPELINE_STAGEFUSION_1_enable type boolean scale nominal
92          bool PIPELINE_STAGEFUSION_2_enable = false;
93  #pragma atune setvar PIPELINE_STAGEFUSION_2_enable type boolean scale nominal
94          map.SetValue(1, PIPELINE_STAGEFUSION_1_enable);
95          map.SetValue(2, PIPELINE_STAGEFUSION_2_enable);

97          PipelinePolicy policy =
98              new PipelinePolicy(map, PIPELINE_ALTERNATIVE_Alternative_1);
99          Pipeline pipeline =
100             new Pipeline(policy, AC_StringSearch1, AC_StringSearch2, AC_UpdateIndex,
101             AC_CreateIndexFile);
102         DataBatchInputHandler idh = new DataBatchInputHandler();

104 #pragma atune gauge PIPELINE_TunablePipeline_1_Gauge_1
105         pipeline.Run(idh);
106         idh.EnqueueBatch(engine.Crawl());
107         idh.ProcessAll();
108         PipelineResultHandler results =
109             (PipelineResultHandler)pipeline.WaitForResults();
110 #pragma atune gauge PIPELINE_TunablePipeline_1_Gauge_1
111     }
112 #pragma atune endblock
113 }
```

Listing B.2: Beispiel der generierten Tuning-Hüllklasse für den Tunable Pipeline-Konnektor der Desktopsuche.

Der Bezeichner einer Tuning-Hüllklasse entspricht dem optionalen TADL-Bezeichner des entsprechenden TADL-Konnektors. Wurde im TADL-Skript kein Bezeichner angegeben, generiert der TADL-Übersetzer einen eindeutigen Klassenbezeichner (im Beispiel `TunablePipeline_1`.

Die Methode `Init()` (Zeile 5–8) nimmt eine Instanz von jedem Typ entgegen, in dem Methoden atomarer Komponenten deklariert sind. Außerdem legt `Init()` eine lokale Referenz auf jede der übergebenen Instanzen an, damit die Methoden später aufgerufen werden können.

Die Methode `Run()` (Zeile 11–111) enthält den Quelltext zur Initialisierung und Ausführung des TALib-Konnektormoduls, welches die Funktionalität des Tunable Pipeline-Konnektors zur Verfügung stellt. Die `Run()`-Methode ist von einem Tuning-Block umgeben (Zeile 10 und 112), der die Methode als parallele Sektion markiert und den Auto-Tuner darüber informiert, dass die Sektion ein Fließband beinhaltet (`pattern PIPELINE`).

In der `Run()`-Methode wird zunächst die Variable für den Tuning-Parameter deklariert, der die Auswahl der beiden String-Matching-Alternativen steuert (Zeile 14). In Zeile 15–17 folgt die entsprechende Instrumentierung mit einer Atune-IL-Anweisung.

Danach werden die Variablen für die Tuning-Parameter der drei Tunable Replication-Konnektoren deklariert (Zeile 19–27). Jeder Konnektor besitzt 3 Tuning-Parameter. Die entsprechenden Atune-IL-Instrumentierungen folgen in den Zeilen 29–60. Die `setvar`-Anweisungen berücksichtigen alle vorhandenen Parameterabhängigkeiten.

In den Zeilen 62–73 werden die einzelnen Parameter jedes Tunable Replication-Konnektors zu einer `ReplicationPolicy` zusammengefasst.

Anschließend werden die Klassen zur Umsetzung der atomaren Komponenten deklariert (Zeile 75–87). Nicht-replizierbare Atomare Komponenten werden durch eine Instanz der generischen Klasse `ProcessableItem` repräsentiert, replizierbare atomare Komponenten durch eine Instanz der generischen Klasse `ReplicableItem`. Die Klassen bilden jeweils eine Hülle um die eigentlichen Methoden, die die atomaren Komponenten implementieren. Das erste Argument des Konstruktors (Zeile 77, 81, 85 und 87) ist jeweils ein

Zeiger auf die auszuführende Methode. Die beiden generischen Klassen erlauben eine typsichere Verarbeitung der atomaren Komponenten.

In den Zeilen 89–95 werden die Tuning-Parameter zur Steuerung der Stufenfusionierung des Tunable Pipeline-Konnektors deklariert und instrumentiert.

Danach werden die Tuning-Parameter des Tunable Pipeline-Konnektors in einer `PipelinePolicy` zusammengefasst (Zeile 97 und 98).

Schließlich wird eine Instanz der Klasse Pipeline erzeugt. Im Konstruktor wird zum einen die Instanz von `PipelinePolicy` übergeben. Zum anderen werden in korrekter Reihenfolge diejenigen atomaren Komponenten übergeben, die als Fließbandstufe fungieren sollten (Zeile 99–101). Des Weiteren wird eine Instanz der Klasse `DataBatchInputHandler` erzeugt, die die Verwaltung der Eingabedaten übernimmt (Zeile 102).

In Zeile 105 wird das Fließband mittels `pipeline.Run()` gestartet. Danach wird der Instanz von `DataBatchInputHandler` die Datenquelle in Form der in TADL spezifizierten Eingabekomponente übergeben (`engine.Crawl`). Der Aufruf `ProcessAll()` in Zeile 107 startet die Verarbeitung der Eingabedaten.

Im nächsten Schritt wird auf die Ergebnisse der gesamten Fließbandoperation gewartet (Zeile 108 und 109). Da die letzte Stufe des Fließbandes (`CreateIndexFile`) keinen Rückgabewert hat, wird die erzeugte Instanz von `PipelineResultHandler` hier nicht verwendet.

Die Ausführung des Fließbandes ist von zwei `gauge`-Anweisungen umgeben, um die Laufzeit zu messen (Zeile 104 und 110).

Literaturverzeichnis

[Agil08] Agilent Technologies. *MassHunter MetaboliteID Software*. Agilent Technologies, 2008. `http://www.chem.agilent.com`.

[AlGa97] Robert Allen und David Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3), 1997, S. 213–249.

[AlIS77] Christopher Alexander, Sara Ishikawa und Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press. 1977.

[Amda67] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the Spring Joint Computer Conference*. ACM, 1967, S. 483–485.

[AMSS+02] J. Anvik, S. MacDonald, D. Szafron, J. Schaeffer, S. Bromling und Kai Tan. Generating Parallel Programs from the Wavefront Design Pattern. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2002, S. 104–111.

[antl09] antlr.org. *ANTLR Parser Generator*. antlr.org, 2009. `http://www.antlr.org`.

[BaGS94] David F. Bacon, Susan L. Graham und Oliver J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4), 1994, S. 345–420.

[BCGH04] Anne Benoit, Murray Cole, Stephen Gilmore und Jane Hillston. Evaluating the Performance of Skeleton-Based High Level Parallel Programs. In *Proceedings of the International Conference on Computational Science*. Springer Verlag, 2004, S. 299–306.

[BeKR09] Steffen Becker, Heiko Koziolek und Ralf Reussner. The Palladio Component Model for Model-driven Performance Prediction. *Journal of Systems and Software*, 82(1), 2009, S. 3–22.

[ChHo04] I-Hsin Chung und J.K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. In *Proceedings of the ACM/IEEE Supercomputing Conference*, 2004.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. MIT Press. 2001.

[CMSL04] E. César, J.G. Mesa, J. Sorribes und E. Luque. Modeling Master/Worker Applications in POETRIES. In *Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, 2004, S. 22–30.

[CoDM91] Alberto Colorni, Marco Dorigo und Vittorio Maniezzo. Distributed Optimization by Ant Colonies. In *European Conference on Artificial Life*, 1991.

[Cole89] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press. 1989.

[CéSL05] Eduardo César, Joan Sorribes und Emilio Luque. Modeling Pipeline Applications in POETRIES. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, 2005, S. 83–92.

[DBRY+06] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, Maria Garzaran, David Padua und Keshav Pingali. A Language for the Compact Representation of Multiple Program Versions. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*, Nr. 4339, 2006, S. 136–151.

[DeTo91] J.E. Dennis und V. Torczon. Direct Search Methods on Parallel Machines. *SIAM Journal of Optimization*, Band 4, 1991, S. 448–474.

[DFHK+93] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp und Q. Wu. Parallel Programming Using Skeleton Functions. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, 1993, S. 146–160.

[Dijk82] Edsger W. Dijkstra. Selected Writings on Computing: A Personal Perspective. 1982. Manuscript.

[EGDP+06] Arkady Epshteyn, Mara Jess Garzaran, Gerald DeJong, David Padua, Gang Ren, Xiaoming Li, Kamen Yotov und Keshav Pingali. Analytic Models and Empirical Search: A Hybrid Approach to Code Optimization. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, Band 4339, 2006, S. 259–273.

[Fahr98] Thomas Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators . *The Journal of Supercomputing*, 12(3), 1998, S. 227–252.

[Flyn72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9), 1972, S. 948 –960.

[FoOW66] L. J. Fogel, A. J. Owens und M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley. 1966.

[FrJo98] M. Frigo und S.G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, Band 3, 1998, S. 1381–1384.

[GaAO94] David Garlan, Robert Allen und John Ockerbloom. Exploiting Style in Architectural Design Environments. *SIGSOFT Software Engineering Notes*, 19(5), 1994, S. 175–188.

[GaSC09] David Garlan, Bradley Schmerl und Shang-Wen Cheng. *Software Architecture-Based Self-Adaptation*, S. 31–55. Nr. ISBN 978-0-387-89827-8. Springer-Verlag. 2009.

[GCC] GCC. *The GNU Compiler Collection*. GCC. `http://gcc.gnu.org/`.

[GCHS+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl und Peter Steenkiste. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, 37(10), 2004, S. 46–54.

[GeBl08] Rubino Geiss und Jakob Blomer. *GrGen.NET*. University of Karlsruhe, IPD Prof. Goos, 2008. `http://www.info.uni-karlsruhe.de/software/grgen/`.

[Gels04] Pat Gelsinger. Speech at Intel Developer Forum. Intel Developer Forum, 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional. 1995.

[GlKL95] Fred Glover, James P. Kelly und Manuel Laguna. Genetic Algorithms and Tabu Search: Hybrids for Optimization. *Computers & Operations Research*, 22(1), 1995, S. 111–134.

[Glov86] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, Band 13, 1986, S. 533–549.

[GoRa91] Michael M. Gorlick und Rami R. Razouk. Using Weaves for Software Construction and Analysis. In *Proceedings of the 13th International Conference on Software Engineering*, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press, S. 23–34.

[HaGR09] Jens Happe, Henning Groenda und Ralf H. Reussner. Performance Evaluation of Scheduling Policies in Symmetric Multiprocessing Environments. In *Proceedings of the 17th IEEE International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2009. To appear.

[HaPo09] Albert Hartono und Sdayappan Ponnuswamy. Annotation-Based Empirical Performance Tuning Using Orio. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium*, 2009.

[Holl92] John H. Holland. *Adaptation in Natural and Artificial Systems*. MIT Press. 1992.

[HuLK04] Chao Huang, Orion Lawlor und L. V. Kalé. Adaptive MPI. In *Languages and Compilers for Parallel Computing*, Band 2958. Springer-Verlag, 2004, S. 306–322.

[KaKKY06] Takahiro Katagiri, Hiroki Honda Kenji Kise und Toshitsugu Yuba. ABCLibScript: A Directive to Support Specification of an Auto-tuning Facility for Numerical Software. *Journal of Parallel Computing*, Band 32, 2006, S. 92–112.

[KaSP09] Thomas Karcher, Christoph Schaefer und Victor Pankratius. Auto-Tuning Support for Manycore Applications: Perspectives for Operating Systems and Compilers. *ACM SIGOPS Operating Systems Review*, 43(2), 2009, S. 96–97.

[KeAl02] Ken Kennedy und John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc. 2002.

[KeEb95] J. Kennedy und R. Eberhart. Particle Swarm Optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, 1995, S. 1942–1948.

[KKHY03] Takahiro Katagiri, Kenji Kise, Hiroaki Honda und Toshitsugu Yuba. FIBER: A Generalized Framework for Auto-tuning Software. In *Proceedings of the International Symposium on High Performance Computing*, Band 2858, 2003, S. 146–159.

[KLMM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier und John Irwin. Aspect-oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Band 1241. Springer-Verlag, 1997, S. 220–242.

[LCMS+00] K.A. Lindlan, J. Cuny, A.D. Malony, S. Shende, B. Mohr, R. Rivenburgh und C. Rasmussen. A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates. In *Proceedings of the ACM/IEEE Supercomputing Conference*, 2000, S. 49.

[LiMa06] Li Li und Allen D. Malony. Model-Based Performance Diagnosis of Master-Worker Parallel Computations. In *Proceedings of the 9th International Euro-Par Conference on Parallel Processing*, Band 4128/2006. Springer-Verlag, 2006, S. 35–46.

[LiMa07] L. Li und A. D. Malony. Knowledge Engineering for Automatic Parallel Performance Diagnosis. *Concurrency and Computation: Practice and Experience*, Band 19, 2007, S. 1497–1515.

[LKAV+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan und Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4), 1995, S. 336–355.

[MaMS99] B. Massingill, T. Mattson und B. Sanders. Patterns for Parallel Application Programs. 1999.

[MaMS00] Berna L. Massingill, Timothy G. Mattson und Beverly A. Sanders. A Pattern Language for Parallel Application Programs. *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*, Band 1900, 2000, S. 678–681.

[MaNi98] Makoto Matsumoto und Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1), 1998, S. 3–30.

[MaSM04] Timothy G. Mattson, Beverly A. Sanders und Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley. 2004.

[MCGS+08] A. Moreno, E. César, A. Guevara, J. Sorribes, T. Margalef und E. Luque. Dynamic Pipeline Mapping (DPM). In *Proceedings of the 12th International Euro-Par Conference on Parallel Processing*, Band 5168, 2008, S. 295–304.

[McKi98] K. I. M. McKinnon. Convergence of the Nelder–Mead Simplex Method to a Nonstationary Point. *SIAM Journal on Optimization*, 9(1), 1998, S. 148–158.

[MCMS+04] Anna Morajko, Eduardo César, Tomàs Margalef, Joan Sorribes und Emilio Luque. MATE: Dynamic Performance Tuning Environment. In *Proceedings of the 7th International Euro-Par Conference on Parallel Processing*, Band 3149. Springer-Verlag, 2004, S. 98–107.

[MCSML07] A. Morajko, P. Caymes-Scutari, T. Margalef und E. Luque. MATE: Monitoring, Analysis and Tuning Environment for Parallel/Distributed Applications. *Concurrency and Computation: Practice and Experience*, Band 19, 2007, S. 1517–1531.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach und Jeff Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*. Springer-Verlag, 1995, S. 137–153.

[MeTa00] Nenad Medvidovic und Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), 2000, S. 70–93.

[Micr] Microsoft Corporation. *Microsoft Developer Network (MSDN)*. Microsoft Corporation. `http://msdn.microsoft.com/`.

[Micr09a] Microsoft Corporation. *Microsoft .NET Framework*. Microsoft Corporation, 2009. `http://msdn.microsoft.com/en-us/netframework/`.

[Micr09b] Microsoft Corporation. *Microsoft Visual Studio 2008*. Microsoft Corporation, 2009. `http://www.microsoft.com/germany/visualstudio/`.

[MiFo00] Zbigniew Michalewicz und David B. Fogel. *How to Solve It: Modern Heuristics*. Springer-Verlag. 2000.

[MoML07] Anna Morajko, Tomàs Margalef und Emilio Luque. Design and Implementation of a Dynamic Tuning Environment. *Parallel and Distributed Computing*, 67(4), 2007, S. 474–490.

[Moor65] Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Journal of Electronics*, Band 38, 1965.

[Moor75] Gordon Moore. Excerpts from A Conversation with Gordon Moore: Moore's Law. Interview, 1975.

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins und Richard N. Taylor. Using Object-Oriented Typing to Support Architectural Design in the C2 Style. *SIGSOFT Software Engineering Notes*, 21(6), 1996, S. 24–32.

[MPI 09] MPI Forum. *The Message Passing Interface (MPI) Standard*. MPI Forum, 2009. `http://www.mcs.anl.gov/research/projects/mpi/`.

[MRSB+07] Holger Muegge, Tobias Rho, Daniel Speicher, Pascal Bihler und Armin Cramers. Programming for Context-based Adaptability – Lessons learned about OOP, SOA, and AOP. In *Proceedings of the Workshop Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme)*. VDE Verlag, 2007, S. 207–218.

[MSSA+02] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling und K. Tan. Generative Design Patterns. In *Proceedings of the 17th International Conference on Automated Software Engineering*, 2002, S. 23–34.

[MSSB00] Steve MacDonald, Duane Szafron, Jonathan Schaeffer und Steven Bromling. Generating Parallel Program Frameworks from Parallel Design Patterns. In *Proceedings of the 6th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, 2000, S. 95–104.

[MuRC07] Holger Muegge, Tobias Rho und Armin B. Cremers. Integrating Aspect-Orientation and Structural Annotations to Support Adaptive Middleware. In *Proceedings of the 1st Workshop on Middleware-Application Interaction*. ACM, 2007, S. 9–14.

[NeMe65] J. A. Nelder und R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4), 1965, S. 308–313.

[OrMT08] Peyman Oreizy, Nenad Medvidovic und Richard N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th International Conference on Software Engineering*. ACM, 2008, S. 899–910.

[PaJT09] Victor Pankratius, Ali Jannesari und Walter F. Tichy. Parallelizing BZip2. A Case Study in Multicore Software Engineering. *IEEE Software*, 26(6), 2009, S. 70–77.

[PeWo92] Dewayne E. Perry und Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4), 1992, S. 40–52.

[PMJP+05] M. Püschel, J.M.F. Moura, J.R. Johnson, D. Padua, M.M. Veloso, B.W. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R.W. Johnson und N. Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE*, 93(2), 2005, S. 232–275.

[PSJT08] Victor Pankratius, Christoph A. Schaefer, Ali Jannesari und Walter F. Tichy. Software Engineering For Multicore Systems: An Experience Report. In *Proceedings of the 1st International Workshop on Multicore Software Engineering*. ACM, 2008, S. 53–60.

[QaKMC06] Apan Qasem, Ken Kennedy und John Mellor-Crummey. Automatic Tuning of Whole Applications using Direct Search and a Performance-based Transformation System. *The Journal of Supercomputing*, 36(2), 2006, S. 183–196.

[RaPo03] Andreas Rasche und Andreas Polze. Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET. In *Proceedings of the 6th International Symposium on Object-Oriented Real-Time Distributed Computing*. IEEE, 2003, S. 164.

[Ricc02] Laura Ricci. Automatic Loop Parallelization: An Abstract Interpretation Approach. *Parallel Computing in Electrical Engineering, International Conference on*, Band 0, 2002, S. 112.

[ScGS09] Jochen Schimmel, Tom Gelhausen und Christoph A. Schaefer. Gene Expression with General Purpose Graph Rewriting Systems. In *Proceedings of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques*, 2009.

[Scha09] Christoph A. Schaefer. Reducing Search Space of Auto-Tuners Using Parallel Patterns. In *Proceedings of the 2nd International Workshop on Multicore Software Engineering*. IEEE, 2009, S. 17–24.

[Schi08] Jochen Schimmel. Parallelisierung von Graphersetzungssystemen. Diplomarbeit, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, 2008.

[ScPT09] Christoph A. Schaefer, Victor Pankratius und Walter F. Tichy. Atune-IL: An Instrumentation Language for Auto-Tuning Parallel Applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Band 5704. Springer-Verlag, 2009, S. 9–20.

[ScPT10] Christoph A. Schaefer, Victor Pankratius und Walter F. Tichy. Engineering Parallel Applications with Tunable Architectures. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, 2010, S. 405–414.

[SDKR+95] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young und Gregory Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), 1995, S. 314–335.

[ShGa96] Mary Shaw und David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall. 1996.

[ShMa06] Sameer S. Shende und Allen D. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2), 2006, S. 287–311.

[StMC79] W. Stevens, G. Myers und L. Constantine. Structured Design. *Classics in Software Engineering*, 1979, S. 205–232.

[Sutt05] Herb Sutter. The Free Lunch Is Over. *Dr. Dobb's Journal*, Band 3, 2005.

[TaCH02] C. Tapus, I-Hsin Chung und J.K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the ACM/IEEE Supercomputing Conference*, 2002.

[TaTH05] V. Tabatabaee, A. Tiwari und J.K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *Proceedings of the ACM/IEEE Supercomputing Conference*, 2005.

[TavdH07] Richard N. Taylor und Andre van der Hoek. Software Design and Architecture The Once and Future Focus of Software Engineering. In *FOSE '07: 2007 Future of Software Engineering*. IEEE, 2007, S. 226–243.

[TiTH09] Ananta Tiwari, Vahid Tabatabaee und Jeffrey K. Hollingsworth. Tuning Parallel Applications in Parallel. *Parallel Computing*, 35(8-9), 2009, S. 475–492.

[TJTK+05] Jie Tao, Jürgen Jeitner, Carsten Trinitis, Wolfgang Karl und Josef Weidendorfer. Comprehensive Cache Inspection with Hardware Monitors. In Victor Malyshkin (Hrsg.), *Parallel Computing Technologies*, Band 3606 der *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005, S. 331–345.

[TSSA+03] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik und Steve MacDonald. Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Band 38. ACM, 2003, S. 203–215.

[Werm97] Michel Wermelinger. A Hierarchic Architecture Model for Dynamic Reconfiguration. In *Proceedings of the 2nd International Workshop on Software Engineering for Parallel and Distributed Systems*. IEEE, 1997.

[WhPD01] R. Clint Whaley, Antoine Petitet und Jack J. Dongarra. Automated Empirical Optimizations of Software and the ATLAS Project. *Journal of Parallel Computing*, Band 27, 2001, S. 3–35.

[WKTi00] Otilia Werner-Kytola und Walter F. Tichy. Self-Tuning Parallelism. In *Proceedings of the 8th International Conference on High-Performance Computing and Networking*. Springer-Verlag, 2000, S. 300–312.

[YSYV+07] Q. Yi, K. Seymour, H. You, R. Vuduc und D. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *Proceedings of International Parallel and Distributed Processing Symposium*, 2007, S. 1–8.