Automatische Verteilung mehrsträngiger Java-Programme

Dissertation

Schriftliche Arbeit zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
an der Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

vorgelegt von

Dinh Khoi Le

Paderborn, April 2006

Datum der mündlichen Prüfung:
08. Juni 2006

Gutachter:
Prof. Dr. Uwe Kastens, Universität Paderborn
Prof. Dr. Odej Kao, Universität Paderborn

Inhaltsverzeichnis

1. Einleitung

Motivation In der Welt des *High-Performance Computing and Communications* findet Java immer mehr Anwendung. Die *High-Performance* Anwendungen aus HP-Fortran, C, C++ mit *message passing*-Technik werden nach und nach in Java übertragen. Neben den typischen Anwendungen aus den Bereichen der geophysikalischen Verfahren und der Finanzmathematik, numerischen Berechnungen etc. finden sich realistische irreguläre Java-Anwendungen im Bereich der Modellierung und Simulation, vor allem bei parallelen und verteilten diskreten ereignisbasierten Simulationen. In diesen vielfältigen Anwendungsgebieten wird Java häufiger als die Sprache für parallele Berechnungen eingesetzt, die sich durch die Verwendung mehrerer leichtgewichtiger Prozesse vom Benutzer ausdrücken lassen. Die verschiedenen Prozesse werden vom Betriebssystem für den Benutzer völlig transparent miteinander verzahnt. Auf speziellen Rechnern mit mehreren Prozessoren kann sogar durch eine echte parallele Ausführung einzelner Prozesse Geschwindigkeitsgewinn erzielt werden. Genau diese Idee, dass sich der Programmierer nur auf die Formulierung seines parallelen Lösungsansatzes konzentriert und von den technischen Details der tatsächlichen parallelen Ausführung seiner Anwendung entlastet wird, soll auf das *JScatter*-System übertragen werden.

Um ein sequentielles *multithreaded* Java-Programm in eine verteilte Version mit *Remote Method Invocation (RMI)* zur Kommunikation umzusetzen, muss ein Programmierer viele aufwändige technische Schritte durchführen. Diese können grob in zwei Phasen aufgeteilt werden: das Umprogrammieren des ursprünglichen Programms in eine verteilte Version mit eincodierten Verteilungshinweisen unter Verwendung von RMI und das verteilte Ausführen auf mehreren Rechnern. Abgesehen davon, dass der Anwender sehr gute Kenntnisse über parallele Programmierung in Java mit der RMI-Technik besitzen muss, sind diese Schritte sehr fehleranfällig und kosten sehr viel Aufwand. Zur verteilten Ausführung auf mehreren Rechnern sind außerdem noch einige nicht triviale Handgriffe erforderlich. Im Fehlerfall ist es auch sehr mühsam, nach Fehlerursachen zu suchen.

Mit dem im *JScatter*-System verwendeten Konzept kann eine mehrsträngig formulierte Java-Anwendung benutzertransparent automatisch verteilt werden. All diese oben genannten notwendigen Schritte werden vollständig automatisiert: Als Ausgangspunkt liegt ein mehrsträngiges Programm vollständig in Form von Java-Bytecode vor. Die Programmanalyse wird eingesetzt, um relevante Eigenschaften des Eingabeprogramms für die automatische Transformation und die automatische Verteilung zu bestimmen. Beruhend auf dieser Information wird anschließend eine verteilte Version des Programms durch den Transformator des Systems automatisch generiert, die dann auf der verteilten Laufzeitumgebung ausgeführt werden kann. Zur Ausführung des Programms hat der Anwender nur noch die Aufgabe, die Rechnerknoten zu bestimmen, auf denen das Programm verteilt laufen soll. Somit wird die Transparenz zwischen der nebenläufigen Formulierung eines Programms und dessen verteilter Ausführung völlig erreicht.

Bei der parallelen Ausführung in einer verteilten Laufzeitumgebung verursachen die Zugriffe auf Programmobjekte über Rechnergrenzen hinweg erheblich größeren Aufwand als lokale Zugriffe. Dadurch kann eine parallele Ausführung im ungünstigen Fall sogar viel langsamer als die verzahnte auf einem einzigen Rechner werden. Daher sollen geschickte Strategien zur Verteilung der Objekte so angewandt werden, dass die Parallelität beim verteilten Rechnen maximal wird, die dabei entstehenden Kommunikationskosten aber so gering wie möglich gehalten werden. Diese Aspekte bilden die Kernpunkte meiner Arbeit.

Beiträge Mit dem Ziel, mehrsträngig formulierte Java-Anwendungen vollständig automatisch zu verteilen und dabei gute Performanz zu erreichen, wird das Konzept *Verteilungsplan* eingesetzt. Über einen Plan können alle strategischen Entscheidungen über Transformation und Verteilung der Objekte getroffen werden. Es wird also entschieden, ob ein zu erzeugendes Objekt ein *remote* oder ein lokales Objekt ist und, im Fall eines *remote* Objekts, auf welcher JVM es erzeugt werden soll. Mit automatischen, tiefgehenden Programmanalysen können Programmeigenschaften bestimmt werden, die relevant für die Verteilung sind. Für bestimmte Verteilungsstrategien werden bestimmte Programmeigenschaften benötigt, die *vor der Laufzeit* und *zur Laufzeit* von entsprechenden *statischen und dynamischen Analyseverfahren* beschafft werden.

Diese Eigenschaften werden innerhalb spezieller Objektverteilungsfunktionen (ODFs) zur Ermittlung der Platzierungsentscheidung verwendet. Be-

stimmte ODFs bilden zusammen eine Verteilungsstrategie und aus mehreren passenden Strategien entsteht ein Verteilungsplan. Die Strategien werden vor der Laufzeit soweit wie möglich berechnet und direkt in das Programm in Form von Anfragen an den Verteilungsplan eincodiert. Zur Laufzeit wird unmittelbar vor einer Objekterzeugung eine Entscheidung getroffen und das Objekt wird nach diesem Hinweis entsprechend platziert. Mit diesem Modell können die Programmeigenschaften, getrennt von ihrer Verwendung in den ODFs, mittels geeigneter Analysetechniken berechnet werden. Die Güte der Analyseergebnisse entscheidet über die Verteilungsgüte des Programms.

Mit diesem Konzept werden grundlegende ODFs formuliert, spezielle Verteilungsstrategien definiert und somit Pläne entwickelt. Die dabei verwendeten statischen und dynamischen Programmeigenschaften werden allgemein aus mehrsträngigen Java-Anwendungen heraus abstrahiert und charakterisieren relevante Verteilungsinformation. Mit dem angegebenen Modell hat man außerdem den Vorteil, neue ODFs formulieren zu können, verschiedene Varianten der Strategien anzuwenden oder neue zu entwickeln und somit unterschiedliche Pläne zur Verteilung eines Programms einzusetzen.

Zur Lokalitätsoptimierung bei der Verteilung habe ich insbesondere das Augenmerk auf die Programmeigenschaft, die die Objektreplikation legitimiert, und die genauere Berechnung der dynamischen Workload-Eigenschaft gerichtet. So wurde zur Replikation eine neuartige Analyse entwickelt, die die so genannte *stagewise-immutable* Eigenschaft der Objekte vor der Laufzeit erkennt. Objekte mit dieser Eigenschaft können direkt nach ihrer Initialisierung (*pure-immutable*) oder ab einem späteren Zeitpunkt (*postponed-immutable*) zur Laufzeit repliziert werden. Nach der Replikation finden alle Lesezugriffe dann lokal statt, so dass viel entfernter Kommunikationsaufwand gespart werden kann.

Zur Verbesserung der vor der Laufzeit abgeschätzten Workload sowie Kommunikationskosten wird der Ansatz zur *hybriden Analyse* erprobt. Die mittels dynamischer Analyse gewonnenen Laufzeitdaten werden mit den statischen Analyseergebnissen kombiniert. Dadurch können einige Laufzeitwerte bestimmt werden, die zur Verteilungsberechnung zukünftig zu erzeugender Objekte eingesetzt werden.

Ergebnisse aus den empirischen Untersuchungen zeigen, dass bei den zusammen mit den verwendeten Bibliothek-Klassen untersuchten Programmen und Benchmarks viele Objektrepräsentanten (etwa 44% aller Erzeugungsstellen) die *stagewise-immutable* Eigenschaft, vorwiegend die *pure-immutable* Eigenschaft, erkannt werden. Durch die Replikation der Objekte, die aus die-

sen als *stagewise-immutable* erkannten Erzeugungsstellen entstehen, können sehr viele Lesezugriffe auf diesen Objekten nach der Replikation (bis zu 15524 oder 6,97% als statisch erkannte Zugriffe auf den *stagewise-immutable* Objektrepräsentanten) lokal ausgeführt werden. Bei der verteilten Ausführung auf 9 miteinander über Standard-Ethernet-100MBit verbundenen Standard-PCs einer diskreten ereignisbasierten Simulation kann ein Speedup von etwa 3.8 erreicht werden. Auf einem speziellen Cluster des Paderborn Center for Parallel Computing (PC^2) konnte sehr viel Performanz beim *mtrt*-Raytracer erzielt werden: Mit den geschickten Verteilungsstrategien, mehr Rechenleistung und schnellere Verbindung konnte dieses Programm bis auf einen Faktor 3 schneller verteilt ausgeführt werden, als die verteilte Ausführung auf einem normalen Pool-Cluster.

Vor- und Nachteile Im Allgemeinen kann die Performanz durch verteilte Ausführung auf mehreren Rechnern nur gut für Anwendungen erzielt werden, die insgesamt viele aufwändige Berechnungen durchführen, aber wenig Kommunikation verursachen. Beispiele dafür sind diskrete ereignisbasierte Simulationen und Programme mit regulären Strukturen, wobei innerhalb eines Threads wesentlich mehr Rechenaufwand als Kommunikation verursacht wird. Dieser Ansatz kann außerdem auch nicht mit denjenigen konkurrieren, bei denen ein Experte im Algorithmen-Bereich eine spezielle algorithmische Lösung für das im Programm verwendete Paradigma programmiert und danach die Objekte verteilt.

Der Vorteil meines Ansatzes ist aber, dass alles vollständig automatisch durchgeführt wird. Durch die verteilte Ausführung in einem Cluster stehen einem auch die verteilten Ressourcen wie Speicher automatisch zur Verfügung. Ein weiterer Vorteil der automatischen Verteilung ist die Skalierbarkeit, die bei Mehrprozessor-Rechnern sehr beschränkt ist. Die automatische Transformation und die verteilte Ausführung eines mehrsträngigen Java-Programms können außerdem einem Programmierer dabei helfen, die echte Parallelität seines Programms durch die verteilte Ausführung auf mehreren Rechnern besser zu verstehen [1], wenn er noch nie solche multithreaded Java-Programme verteilt geschrieben hat.

Da das *JScatter*-System komplett in Java realisiert wurde, kann das System plattformunabhängig verwendet werden, wo eine Standard-JDK vorhanden ist. So wurde *JScatter* zur Evaluation sowohl auf einem Cluster von Standard-

[1] Paralleler Ablauf der Threads (mit Ausgaben) auf verschiedenen Computern

PCs als auch auf einem speziellen Cluster (ARMINIUS-Cluster) des Paderborn Center for Parallel Computing (PC^2) eingesetzt.

Struktur der Arbeit Kapitel 2 beschreibt die wichtigen Aspekte zur verteilten Ausführung eines mehrsträngig formulierten Java-Programms. Diese Aspekte dienen als Grundlage für die Schritte zur automatischen Verteilung. Darauf aufbauend werden die Grundüberlegungen zum gesamten Konzept dargestellt, die im Kapitel 3 beschrieben werden. Vergleiche meines Konzeptes sowie des *JScatter*-Systems mit anderen verwandten Ansätzen und vorhandenen Systemen finden sich im Kapitel 4.

Die beiden anschließenden Kapitel beschreiben die Kernpunkte der Arbeit: Im Kapitel 5 werden das Konzept und die Modellierung der ODFs, der Verteilungsstrategien und somit Verteilungspläne repräsentiert. Die Modellierung der Programmeigenschaften, die zur automatischen Verteilung relevant sind und die Analysetechniken, die diese Eigenschaften berechnen, werden im Kapitel 6 beschrieben. Hierbei werden der Ansatz zur hybriden Analyse und das originäre *stagewise-immutable* Analyseverfahren dargestellt. Eine umfangreiche Evaluation zur Tauglichkeit dieser Programmeigenschaft und empirische Untersuchung der Verteilungsgüte bei einigen Programmen und Benchmarks durch Einsatz einiger vorgestellter Strategien beschreibe ich im Kapitel 8. Zuvor werden im Kapitel 7 einige wichtige Entwurfsaspekte des *JScatter*-Systems dargestellt.

Abschließend werden die Kernpunkte des Konzepts kurz zusammengefasst. Besondere Aspekte über das Konzept zur hybriden Analyse und spezielle Programmeigenschaften, die sehr interessant für die automatische Verteilung sind, jedoch nicht untersucht werden konnten, werden ebenfalls als Erweiterungspunkte beschrieben.

2. Java und sein Modell zur verteilten Ausführung

Inhalt

Java ist eine statisch typisierte, objektorientierte Sprache. Der Programm-Quellcode wird in eine Plattform-unabhängige Repräsentation, auch als Java-Bytecode bezeichnet, übersetzt. Die Ausführung erfolgt dadurch, dass die Bytecode-Instruktionen von einer abstrakten Maschine, der so genannten *Java Virtual Machine (JVM)*, interpretiert werden. Im Vergleich zu anderen ähnlichen objektorientierten Sprachen wie C++ ist Java viel kleiner im Sprachumfang. Stattdessen werden soviel wie möglich die Merkmale vom Sprachkern in die Java Standard-Bibliotheken verlagert. Trotzdem wurden aber fundamentale Datenstrukturen wie Zeichenkette und wichtige Konzepte wie *Multithreading* in die Sprache eingebaut.

In diesem Kapitel gehe ich auf die wichtigsten Aspekte ein, die auf das gesamte Thema Bezug nehmen und als Grundlage für die Schritte zur automatischen Verteilung dienen. Die parallele Programmierung in Java stellt eine notwendige Voraussetzung dar. Mit Hilfe der in die Sprache integrierten Merkmale und Mechanismen aus der Standard-Bibliothek kann die Formulierung einer parallelen Problemlösung erfolgen. Dieses Programm dient als Ausgangspunkt für die automatische Verteilung. Im Abschnitt 2.1 werden das Konzept des *Multithreading* und einige Paradigmen zur parallelen Programmierung in Java vorgestellt.

Die Kommunikation der verteilten Berechnungen wird durch entfernte Methodenaufrufe in Java realisiert. Mittels Java-RMI oder KaRMI, einer schnelleren RMI-Bibliothek der Universität Karlsruhe, kann das grundlegende Kommunikationsmodell festgelegt werden. Die Eigenschaften dieses Modells haben einen starken Einfluss auf das Konzept der automatischen Transformation und Entwurfsentscheidungen für die verteilte Laufzeitumgebung. Auf diesen Aspekt werde ich im Abschnitt 2.2 eingehen.

Eine der wichtigsten Voraussetzungen bei der automatischen Verteilung eines mehrsträngigen Java-Programms ist es zu gewährleisten, dass die Ausführungssemantik des verteilten Programms der Semantik des sequentiellen Programms entspricht. In meinem gesamten Konzept wird ein mehrsträngiges Java-Programm in eine Version transformiert, die dann automatisch verteilt in einer speziellen Laufzeitumgebung ausgeführt werden kann. In dieser Laufzeitumgebung sind mehrere normale JVMs auf verschiedenen Rechnerknoten an einer parallelen Problemlösung beteiligt. Aus Benutzersicht verhält sich diese verteilte Laufzeitumgebung semantikgleich wie eine normale JVM. Dazu muss das transformierte Programm die gleichen semantischen Eigenschaften besitzen wie das Original und alle Funktionalitäten der JVM müssen auf die verteilte Laufzeitumgebung abgebildet werden. Dafür wird tiefgehen-

des Wissen über das Modell zur Ausführung mehrsträngiger Programme im sequentiellen sowie im verteilen Fall benötigt.

2.1. *Multithreading*-Konzept und die parallele Programmierung

Das Konzept *Ausführungsstrang (thread of control)* wird in der Sprache Java direkt unterstützt. Selbst die Eigenschaften eines mehrsträngigen Systems wie *thread scheduling* finden sich in der Java Virtual Machine. Das *Thread*-Konzept selbst ist nicht neu: Früher boten die Betriebssysteme Bibliotheken an, mit deren Hilfe C-Programmierer Threads erzeugen konnten. In Sprachen wie Ada sind Threads Teil der Sprachdefinition. Hier möchte ich eine kleine Einführung in die Java-Threads und das Konzept *Multithreading* darstellen, beginnend mit einem kleinen Auszug aus der Grundlage über Prozesse und ihre Interaktion in der Terminologie eines (Unix-artigen) Betriebssystems. Die fundamentalen Konzepte zur parallelen Programmierung, die Java unterstützt, werden im Anschluss daran vorgestellt.

2.1.1. Prozesse und ihre Interaktion

Prozesse

In modernen Betriebssystemen ist die Abarbeitung von Programmen in *Prozessen* organisiert. Ein Prozess besteht im Wesentlichen aus einem Stück Programmcode, einem oder mehrerer Datenbereiche, auf die er zugreift, und dem aktuellen Zustand der Bearbeitung. Die virtuelle Speicherverwaltung des Betriebssystems trennt die Adressräume der einzelnen Prozesse. So steht jedem Prozess ein eigener Speicherraum zur Verfügung und die Konflikte durch Speicherzugriffe zwischen den Prozessen können vermieden werden. Grundsätzlich sind Adressräume unabhängig von Prozessen zu sehen. Jeder Prozess benötigt zwar zu jedem Zeitpunkt einen Adressraum, es ist aber möglich, dass ein Prozess zwischen mehreren Adressräumen wechselt, genau einen Adressraum besitzt (*heavyweight process*) oder mehrere Prozesse sich einen Adressraum teilen (*lightweight process*). Ein Prozess ist repräsentiert durch einen Prozesskontrollblock (PCB), der wichtige Informationen enthält:

seine Charakteristika (Prozesskennung, Name des Programms), Zustandsinformation (Befehlszähler, Stackzeiger, Registerinhalte) und Verwaltungsdaten (Priorität, Rechte, Statistikdaten).

Prozessumschaltung

In einem Betriebssystem existieren zu einem Zeitpunkt in der Regel viele Prozesse. Ein Prozess, der gerade durch den Prozessor bearbeitet wird, heißt *rechnend*. Es kann in einem Rechner höchstens so viele rechnende Prozesse geben wie Prozessoren vorhanden sind. Um einen anderen Prozess zu bearbeiten, muss umgeschaltet werden: Die Zustandsinformation des rechnenden Prozesses wird gerettet, danach wird die Zustandsinformation des neuen Prozesses in den Prozessor geladen. Das Umschalten kann freiwillig erfolgen oder erzwungen werden, z.B. ausgelöst durch eine Hardware-Unterbrechung (*preemption*). Der Teil eines Betriebssystems, der die Prozessumschaltung übernimmt, heißt *Scheduler*. Der Umschaltungsvorgang geschieht ausschließlich im Betriebssystemkern.

Da das Umschalten eines Prozesses, der einen eigenen Adressraum besitzt, mit einem aufwändigen Adressraumwechsel verbunden ist, nennt man solch einen Unix-artigen Prozess auch *schwergewichtiger Prozess* (*heavyweight process*). Dagegen werden die Prozesse als *leichtgewichtig* (*lightweight*) oder *Threads* bezeichnet, die in einem gemeinsamen Adressraum ablaufen. Diese Möglichkeit wird von modernen Betriebssystemen wie Linux angeboten. Ein derartiger Prozess degeneriert dadurch zu einer Art *Ablaufumgebung* der *Threads*.

Zustände eines Prozesses

Ein Prozess kann drei (aktive) Zustände besitzen: wartend, bereit und rechnend, wie in Abbildung 2.1 verdeutlicht wird. Sie können durch Übergänge erreicht werden, die durch entsprechende Operationen des Betriebssystemkerns realisiert werden. Es gibt dennoch zwei weitere Zustände für einen Prozess: nicht-aktiv und nicht-existent. Bei Java-Threads haben diese beiden Zustände keine zentrale Bedeutung.

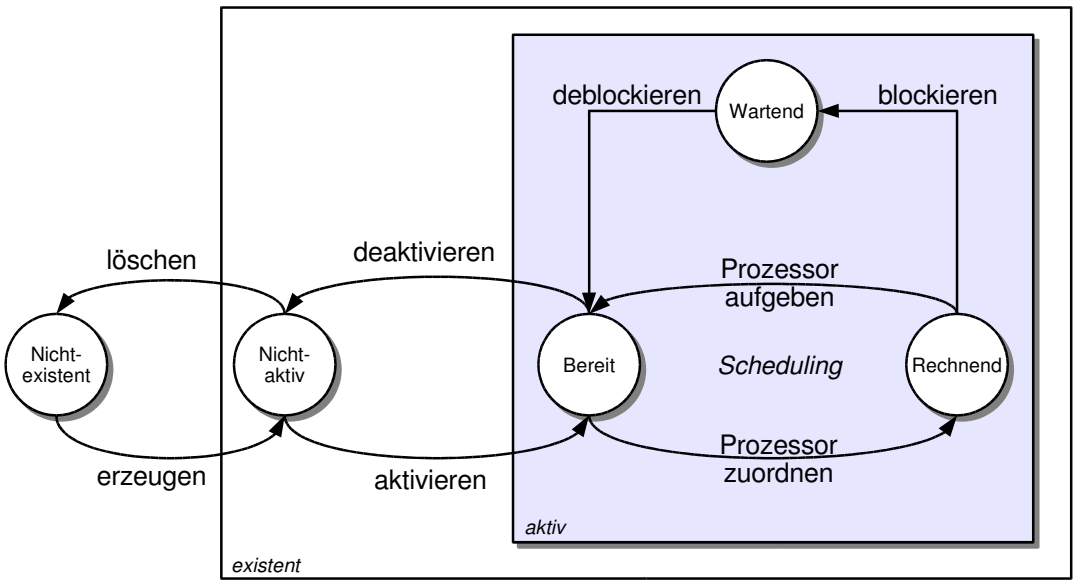

Abbildung 2.1.: Prozesszustandsdiagramm

Prozessinteraktion

In einem komplexen Programmsystem interagieren die Prozesse miteinander unter dem *zeitlichen* und *funktionalen* Aspekt. Beim zeitlichen Aspekt geht es um die Koordination zwischen den beteiligten Prozessen, die durch Synchronisation miteinander abgestimmt werden können. Beim funktionalen Aspekt kooperieren oder kommunizieren die Prozesse miteinander. Wenn mehrere Prozesse auf gemeinsame Daten zugreifen, spricht man von einer *Kooperation*; der explizite Datentransport wird unter dem Begriff *Kommunikation* verstanden. Diese beiden Formen können aufeinander abgebildet werden.

Parallelität und Nebenläufigkeit

In einem Multiprozessor-System können mehrere Prozesse entsprechend der vorhandenen Prozessoren gleichzeitig parallel ausgeführt werden. Man spricht von *parallelen Prozessen*. Im System mit einem einzigen Prozessor werden mehrere Prozesse *verzahnt* ausgeführt. Das heißt, der Prozessor bearbeitet die Prozesse stückweise abwechselnd. Der Umschaltungsvorgang wird vom Prozessverwalter oder von den Prozessen selbst ausgelöst. Was man wahrnimmt, ist eine Quasiparallelität. Da das Umschalten sehr schnell und häufig geschieht, vermittelt es den Eindruck, dass alle Prozesse gleichmäßig fort-

schreiten. Im Folgenden verstehe ich unter *nebenläufigen Prozessen* solche, die parallel oder verzahnt ausgeführt werden.

2.1.2. Java-Threads und das *Multithreading*-Konzept

Java-Threads

In Java hat man die Möglichkeit, seine Programme durch das Konzept Threads nebenläufig zu formulieren. Die Java-Threads oder Ausführungsstränge sind leichtgewichtigte Prozesse, die parallel im gemeinsamen Adressraum (Speicher) ablaufen können und sehr schnell in der Umschaltung sind [1]. Dadurch können die Java-Threads untereinander auf ihre öffentlichen Daten zugreifen. Im Vergleich zu Unix-Prozessen, die durch einen Prozesskontrollblock repräsentiert sind, sind Java-Threads innerhalb eines Programms Objekte der Klasse `java.lang.Thread`. Um Thread-Objekte zu kreieren, müssen die zugehörigen Klassen entweder das Interface `java.lang.Runable` implementieren oder Unterklasse der Klasse `java.lang.Thread` sein. Die Erzeugung eines Thread-Objekts, der Datenaustausch mit anderen über den gemeinsamen Speicher und das Löschen eines Thread-Objekts mittels des Garbage Collectors finden innerhalb der JVM statt.

Weiterhin kann sich jeder Java-Thread in der JVM in einem der folgenden Zustände befinden:

- *Initial:* Dieser Zustand tritt nach der Erzeugung des Thread-Objekts auf, also nach einem Konstruktor-Aufruf, aber bevor die `start()`-Methode aufgerufen wird.
- *Aktiv:* Ein Thread ist in einem *aktiven* Zustand, wenn die `start()`-Methode aufgerufen wurde. Dadurch wird die `run()`-Methode implizit aufgerufen, was veranlasst, dass die JVM diesen Prozess *abarbeiten kann*. Dieser Zustand ist vergleichbar mit dem *Bereit*-Zustand im Diagramm 2.1.
- *Laufend:* Entsprechend dem *Rechnend*-Zustand in Abbildung 2.1 wird einer von den *aktiven* Threads nach einem Auswahlverfahren der JVM ausgewählt und seine `run()`-Methode wird abgearbeitet. Dies ist der Fall

[1] Eine Ausnahme sind Java-Applets, auf die aber in dieser Arbeit nicht näher eingegangen wird.

eines Einprozessor-Systems, wobei nur ein Prozess vom Prozessor abgearbeitet wird. In einem Multiprozessor-System können gleichzeitig so viele Threads bearbeitet werden, wie es verfügbare Prozessoren gibt.

- *Blockiert:* Der Thread wird blockiert, weil er auf ein Ereignis warten muss und nicht laufen kann.
- *Beendet:* Dieser Zustand wird erreicht, nachdem die `run()`-Methode bis zum Ende ausgeführt wurde.

Multithreading-Konzept

Wie Unix-artige Prozesse können mehrere Java-Threads *verzahnt* ablaufen, so dass sich für den Benutzer der Eindruck von Gleichzeitigkeit ergibt. Man kann mehrere nebenläufige Java-Threads innerhalb einer JVM auf die gleiche Weise betrachten, wie die Prozesse innerhalb eines Betriebssystems nebenläufig ausgeführt werden. Abbildung 2.2 verdeutlicht dies. Man nennt diese Umgebung, in der das Programm so abläuft, auch *multithreaded* (mehrsträngig).

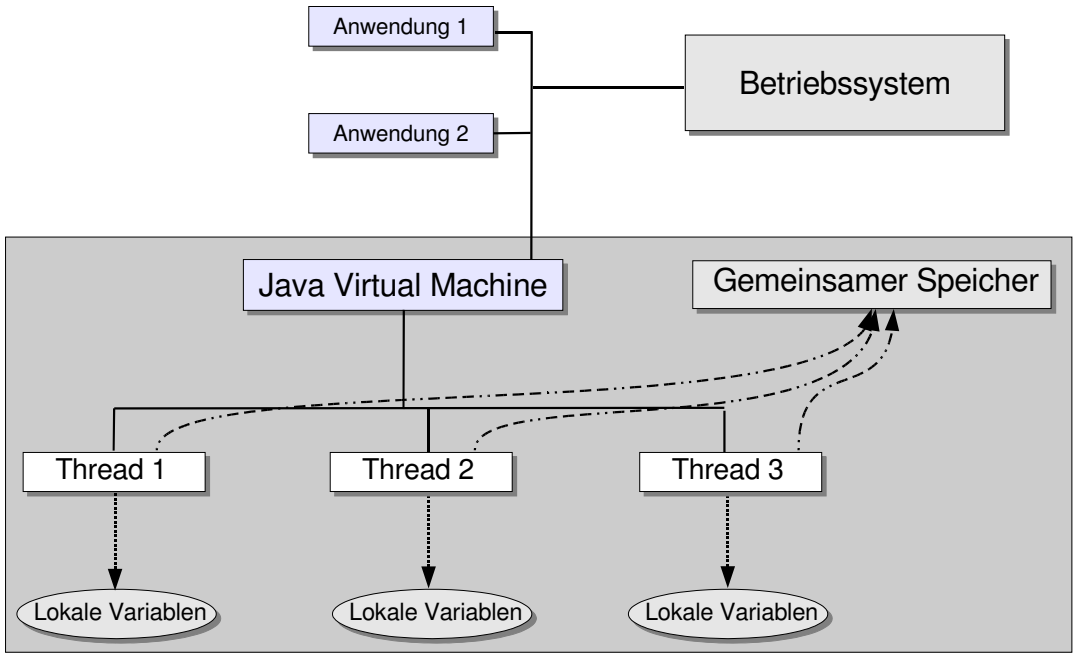

Abbildung 2.2.: Das *Multithreading*-Konzept in Java

Es ist häufig der Fall, dass sich mehrere Threads in einem Java-Programm im *aktiven* Zustand befinden. In diesem Fall wählt die JVM einen Thread aus dem Pool der aktiven Threads aus und bearbeitet ihn, d.h. er *läuft*. Die Auswahlstrategie bestimmt der Scheduler, der ein Teil der JVM ist. Die Umschaltzeit der Java-Threads durch diesen Scheduler ist viel kürzer.

Scheduling von Java-Threads

In der Java Spezifikation ist nicht definiert, welches Scheduling-Modell als Auswahlverfahren zugrunde liegt. Das ist von Plattform zu Plattform unterschiedlich. Das am häufigsten verwendete Verfahren ist ein prioritätsbasiertes Verfahren: Jedem Java-Thread wird eine bestimmte Priorität zugewiesen, also eine positive ganze Zahl im festdefinierten Wertebereich. Wenn ein aktiver Thread eine höhere Priorität als der laufende Thread hat, so schaltet die JVM auf diesen Thread um. Es gibt zwei grundlegende Modelle zum Scheduling von Java-Threads:

- *Das green-thread-Modell:* Der Scheduler der JVM bestimmt selbst die Auswahlstrategie für die Java-Threads. Dies ist das Originalmodell für die JVMs der älteren Generation (bei JDK der Version 1.3 oder älter) und wird heutzutage nicht mehr verwendet. Abhängig von der Implementierung der JVMs folgt es meist dem prioritätsbasierten Scheduling. In der Praxis, wie es in der Spezifikation der JVM [T. 99] beschrieben wurde, wird nicht gewährleistet, wann und wie genau ein Java-Thread vom Scheduler gewählt wird.
- *Das native-thread-Modell:* In diesem Modell wird ein Java-Thread auf einen Betriebssystem-Thread abgebildet, beispielsweise auf einen *pthread* bei Linux. So übernimmt der Scheduler des Betriebssystems, auf dem die JVM läuft, die Aufgabe, die Threads zu schedulen. Das Scheduling ist in diesem Fall bei verschiedenen Betriebssystemen unterschiedlich.

Der Umschaltvorgang beim *green-thread*-Modell dauert viel länger, da er innerhalb der JVM geregelt werden muss. Die verschiedenen Java-Threads bleiben in diesem Fall nur innerhalb der JVM sichtbar. Die Parallelität wird von der virtuellen Maschine simuliert, der Java-Interpreter regelt dann den Ablauf, die Synchronisation und die verzahnte Ausführung. Dieses Modell wird

heutzutage nur in den JVMs implementiert, die auf Betriebssystemen ohne Thread-Unterstützung oder nicht im Multi-Tasking laufen, z.B. bei einigen PDAs oder Handys.

Beim nativen Thread-Modell regelt das Betriebssystem selbst alle Aufgaben für die von Java-Threads abgebildeten *pthreads*. Im Falle eines Multiprozessor-Systems werden die abgebildeten Threads vom Betriebssystem automatisch auf die verschiedenen Prozessoren verteilt. Die Kommunikation kann in diesem Fall als lokal betrachtet werden. In meinem Ansatz wird diese Art der Verteilung nicht in die Betrachtung einbezogen. Trotzdem kann diese Leistungsfähigkeit auch bei der Lastberechnung des automatischen Verteilungsmodells berücksichtigt werden.

Weitere ausführliche Diskussion über verschiedene Implementierungen für obige Thread-Modelle findet sich im Kapitel 6 in [OW99].

2.1.3. Unterstützung der parallelen Programmierung

Für die parallele Programmierung bietet die Java-Quellsprache selbst notwendige Mechanismen für das Schreiben von Thread-sicherem Code an. In diesem *Multithreading*-Modell (Programmiermodell *gemeinsamer Speicher*) findet der Datenaustausch zwischen parallel ablaufenden Ausführungssträngen durch Benutzung gemeinsamer, globaler Variablen oder Objekte statt. Diese befinden sich auf dem Heap und sind daher für alle Threads zugreifbar. Allgemein gibt es zwei Synchronisationsmechanismen, die für das koordinierte Zusammenarbeiten von Prozessen notwendig sind. Das sind *gegenseitiger Ausschluss* (*mutual exclusion*), wobei mehrere Prozesse die kritischen Abschnitte nicht gleichzeitig betreten dürfen, und *Bedingungssynchronisation* (*condition synchronization*), bei der ein Prozess so lange warten muss, bis eine bestimmte Bedingung von einem anderen Prozess erfüllt wird. Bei Java finden sich schon in der Sprachdefinition Mechanismen wie `synchronized` und das Monitor-Konzept[2]. Direkte Unterstützung für die Bedingungssynchronisation hat Java nicht, aber sie kann durch einige Methoden der Klasse `java.lang.Object` wie `wait()`, `notify()` nachgebildet werden.

[2]Monitore sind ein sehr wichtiges Betriebsystemkonzept, das in den frühen Siebzigerjahren von Per Brinch Hansen und Tony Hoare erfunden wurde.

Thread-Interaktion

Zur Synchronisation der Java-Threads verwendet Java *Monitore*. Das sind Programmmodule (Objekte), die Daten (Objektvariablen) und Operationen (Objektmethoden) darauf kapseln. Zu jedem Zeitpunkt darf höchstens ein Java-Thread im Monitor sein. Dadurch kann das Konzept für gegenseitigen Ausschluss und die Bedingungssynchronisation, wobei die Bedingungsvariablen lokal im Monitor benutzt werden, realisiert werden. Die Objekte jeder Klasse in Java können als Monitor verwendet werden. Methoden, die kritische Abschnitte auf Objektvariablen implementieren, werden als *synchronized* gekennzeichnet. Für ein Monitorobjekt werden diese Methoden unter gegenseitigem Ausschluss ausgeführt, wenn mehrere Ausführungsstränge sie gleichzeitig aufrufen.

Die Synchronisation wird durch eine interne Synchronisationsvariable des Objekts (*lock*) implementiert. Die Sprache hat keine direkte Unterstützung, separate Lock- und Unlock-Aktionen auszuführen. Dies muss durch Verwendung der Konstrukte auf höherer Ebene in geeigneter Weise geschehen. Zu bemerken ist, dass die JVM spezielle Bytecode-Instruktionen für Lock- und Unlock-Aktionen unterstützt. Das sind *monitorenter* und *monitorexit*. Wie der Synchronisationsprozess genau abläuft, soll durch folgendes Beispiel verdeutlicht werden: Wenn ein `synchronized` Modifizierer beispielsweise auf eine statische Methode einer Klasse angewandt wird, dann beschafft die Java Virtual Machine das exklusive Lock auf diese Klasse, bevor ein Aufruf der Methode ausgeführt wird. Ein Thread, der in diesen Codeblock einzutreten versucht, muss das Lock holen, bevor der Code im synchronisierten Block tatsächlich ausgeführt wird. Wenn ein anderer Thread gerade zu diesem Zeitpunkt diesen kritischen Abschnitt betritt, wird dieser Thread so lange blockiert, bis der laufende Thread diesen Abschnitt verlässt und das Lock auf diese Klasse freigegeben wird. Für Objektmethoden gilt Entsprechendes.

Die Verwaltung blockierter Threads für jedes Objekt wird durch elementare Methoden, die in der Basisklasse `java.lang.Object` implementiert werden, realisiert. Somit steht dieser Mechanismus automatisch für jede Java-Klasse zur Verfügung. Die elementaren (nativen) Methoden zur Synchronisation `wait()` und `notify()`[3] dürfen nur innerhalb `synchronized` definierter Methoden oder Blöcke verwendet werden, d. h. der Thread muss das Lock auf das Objekt besitzen, deren `wait()` und `notify()` Methoden aufgerufen werden. Sonst wird eine Ausnahme ausgelöst. `notifyAll()` ist ähnlich

[3] Siehe die Java API für mehr Details über diese Methoden.

wie `notify()`, bei der aber alle wartenden Threads aufgeweckt werden. Nur dasjenige, das das Lock des Objekts gewinnt, darf dann laufen. Zu bemerken ist, dass die JVM keine Mechanismen zum Entdecken und Vermeiden von Verklemmungen hat. Das tritt dann auf, wenn ein Thread Ressourcen besitzt, die von einem anderen Thread benötigt werden und so zyklisches Warten auf die Ressourcen entsteht. Es ist dem Programmierer selbst überlassen, diese zu vermeiden.

Thread-Gruppen

In Java besteht die Möglichkeit, die erzeugten Thread-Objekte zu einer Gruppe zusammenzufassen. Jeder erzeugte Thread gehört automatisch zu einer Gruppe, die durch ein `ThreadGroup`-Objekt repräsentiert und von der JVM zur Verfügung gestellt wird. Die Threadgruppen dienen in Java zwei Zwecken: Man kann mehrere Threads innerhalb einer Gruppe durch eine einzige Methode manipulieren; außerdem unterstützt es grundlegende Sicherheitsmechanismen für die Threadinteraktion. Die Beziehung zwischen den Threadgruppen in einem Java-Programm ist eine Baumhierarchie. Die Wurzel des Baumes ist die *System-Thread-Gruppe*, in der einige System-Threads wie Garbage-Collector-Thread, Clock-Thread usw. enthalten sind. Die Klasse `ThreadGroup` stellt dem Programmierer einige nützliche Methoden zur Verwaltung und Manipulation zur Verfügung.

Speichermodell in Java

Nach dem *Multithreading*-Paradigma, wie es in Abbildung 2.2 dargestellt wurde, laufen die Ausführungsstränge in einer gemeinsamen JVM. Jeder Java-Thread hat eigenen Arbeitsspeicher, in dem er Kopien der gemeinsam von allen Threads benutzten Variablen aus dem Heap hält und darauf arbeitet. Nach der Sprachspezifikation für das Speichermodell muss die JVM die Daten konsistent halten. Daher sorgt die JVM dafür, dass die von Threads zwischengespeicherten Daten auf den Heap zurückgeschrieben werden, sobald es erforderlich ist. Der Vorgang läuft grob folgendermaßen ab: Um auf eine gemeinsame Variable zuzugreifen, beschafft ein Thread zunächst ein Lock und leert seinen lokalen Speicher. Anschließend wird der Wert der Variable aus dem Heap geholt und hier gespeichert. Wenn der Thread das Lock wieder zurückgibt, muss er gewährleisten, dass der Wert in seinem lokalen Speicher wieder auf den Heap zurückgeschrieben wurde.

Die vom obigen Modell festgelegten Vorschriften gelten jedoch nicht ganz für Variablen, die als `volatile` gekennzeichnet werden. Werte solcher Variablen werden nicht zwischengespeichert, sondern immer aktuell gehalten. Die parallel ablaufenden Threads sehen immer den aktuellen Variablenwert, da sie ihn vor der Verwendung der Variablen aus dem gemeinsamen Speicher lesen und nach einer Änderung sofort zurückschreiben müssen. Nur Objekt- und Klassenvariablen können als `volatile` deklariert werden. Weitere technische Details können im Abschnitt 17 in [B. 00] nachgelesen werden.

2.2. Kommunikation mittels *Java Remote Method Invocation*

Durch Java RMI ist es möglich, Berechnungen einer mehrsträngigen Anwendung verteilt auf verschiedenen Rechnern durchführen zu können. Hierbei wird der Datenaustausch über Rechnergrenzen hinweg durch die entfernten Methodenaufrufe realisiert. Meinem Gesamtkonzept zur automatischen Verteilung von Java-Anwendungen liegt dieses Kommunikationsmodell zugrunde. Ich werde in diesem Abschnitt das Konzept der Java RMI und die Technik der Objektserialisierung vorstellen, die bei der Datenübertragung über das Netzwerk des RMI-Modells verwendet wird. Aus der semantischen Unterscheidung zwischen lokalem und verteiltem Objektmodell mit RMI ergeben sich einige Aspekte, die bei den Entwurfsentscheidungen für das Transformations- und verteilte Laufzeitkonzept berücksichtigt werden müssen. Darauf gehe ich abschließend ein.

2.2.1. Konzept von Java-RMI

Java-RMI [Mic], [Mic98], [MPA99], das ein Teil der Standard-Bibliothek ist, bietet die Möglichkeit, Objekte über Rechnergrenzen hinweg eindeutig zu identifizieren und deren Methoden entfernt ausführen zu lassen. Objekte einer Klasse, die die Schnittstelle `java.rmi.Remote` implementieren, werden als *remote*-Objekte bezeichnet. Ihre Methoden können von anderen Objekten entfernt aufgerufen werden.

Konzeptionell basiert Java-RMI auf dem Client-Server-Modell. Das *remote*-Objekt befindet sich tatsächlich auf der Server-Seite. Über entfernte Methodenaufrufe kann man das Objekt nicht direkt ansprechen, sondern nur über

das Skeleton. Das ist ein Stellvertreter-Objekt, das alle entfernten Aufrufe entgegennimmt und sie schließlich lokal an die entsprechenden Methoden des Objekts delegiert. Das Resultat eines Methodenaufrufs wird vom Skeleton wieder an den entsprechenden Sender zurückgeschickt.

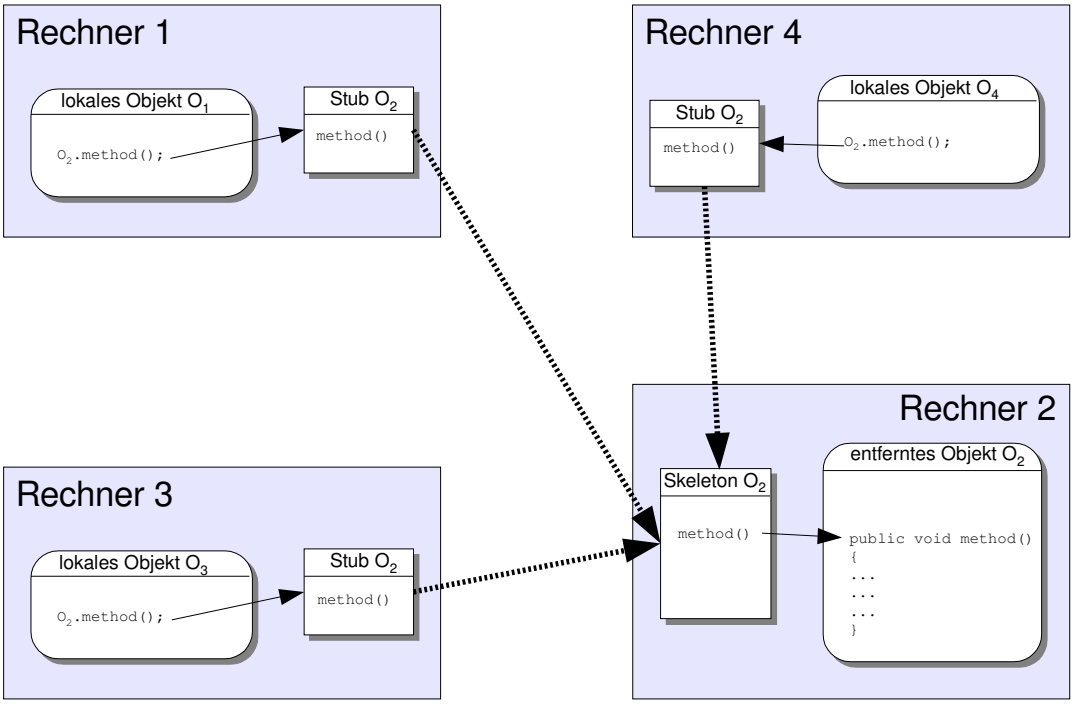

Abbildung 2.3.: Das RMI-Konzept

Zu einem *remote*-Objekt können beliebig viele Stubs existieren. Auf den Client-Seiten dienen diese *Stubs* (oder *Proxies*) als Vertreter des Objekts. Sie sehen bezüglich der entfernt aufrufbaren Methoden genauso aus wie das entsprechende *remote*-Objekt. Über die Stubs können Methoden eines *remote*-Objekts entfernt aufgerufen werden. Der Transfer der Methodenparameter und die Initialisierung des eigentlichen Methodenaufrufs in der virtuellen Maschine auf der Client-Seite werden von den Stubs übernommen. Es ist beim RMI-System möglich, mehrere Methodenaufrufe nebenläufig an einem *remote*-Objekt durchführen zu können. Das Skeleton initialisiert dazu lokale Aufrufe, die dann jeweils in eigenen Threads ausgeführt werden. Stammen mehrere solcher entfernter Methodenaufrufe aus einer gleichen virtuellen Maschine, sorgt das RMI-System dafür, dass diese Aufrufe nacheinander in einem einzigen Thread auf der Server-Seite ausgeführt werden.

Mittels RMI lassen sich *Remote*-Objekte auf dem ganzen verteilten System eindeutig identifizieren. Zuständig dafür ist ein eigenständiger Prozess, ein so genanntes RMI-Registry. Er übernimmt die Aufgabe der Objektverwaltung und des Namensdienstes. Nachdem sich ein *remote*-Objekt auf der Server-Seite beim RMI-Registry erfolgreich registriert hat, kann es Aufrufe der Clients entgegennehmen.

2.2.2. Objekt-Serialisierung

Durch die Technik der Serialisierung in Java können Kopien von Objekten als Parameter oder Rückgabewerte über das Netzwerk gesendet oder empfangen werden. Eine Klasse, deren Objekte serialisierbar sein sollen, muss die Schnittstelle `java.io.Serializable` oder deren Subinterface `java.io.Externalizable` implementieren. Bei der Serialisierung handelt es sich um das Erstellen eines transportierbaren Abbildes eines Objekts, aus dem durch Deserialisierung eine exakte Replik des Originals, also ein Objekt mit identischem Zustand, wieder rekonstruiert werden kann.

Konzeptionell kann der Zustand eines Objekts aus zwei Teilen bestehen: persistenter und nicht-persistenter Zustand. Zum erstgenannten gehören Objektbestandteile, die nach der Deserialisierung des Objekts wieder verfügbar sind. Alle anderen Bestandteile bilden nur vorübergehend einen Teil des Objektzustandes oder können rekonstruiert werden und gehören deshalb zum nicht-persistenten Zustand eines Objekts. Beispiel dafür sind Datenfelder, die nur lokale Ressourcen wie Pfade zu Dateien oder temporäre Werte enthalten. Diese werden auch als `transient` gekennzeichnet. Die Objektserialisierung ist das Speichern des persistenten Zustands eines Objekts.

Bei der Standard Serialisierungstechnik von Sun wird der Zustand eines Objekts in einen Bytestrom gewandelt, zum Ziel durch das Netzwerk transportiert, dann aus dem Bytestrom durch Deserialisierung eine Kopie des Objekts im selben Zustand rekonstruiert. Bei dieser Serialisierung werden alle nicht `transient` und nicht `static` Felder eines Objekts in einen Bytestrom geschrieben. Referenziert ein Feld ein Objekt, so muss es serialisierbar sein. Felder von primitiven Typen sind immer serialisierbar. Wird ein Objekt serialisiert, werden alle seiner Felder auch serialisiert. Auf diese Weise wird der gesamte Objektgraph, der direkt oder indirekt mit einem zu serialisierenden Objekt verbunden ist, mit serialisiert. Sind dabei zyklische Strukturen durch mehrfache Referenzierung auf dasselbe Objekt enthalten, werden diese durch

die Hashing-Technik aufgelöst. Wenn das zu serialisierende Objekt vom Arraytyp ist, wird die Anzahl seiner Elemente und die Elemente analog serialisiert.

Bei der Deserialisierung wird der Objektzustand wieder hergestellt, d.h. den Datenfeldern werden ihre serialisierten Werte zugeordnet und nichtpersistenten Teilen werden typspezifische Initialwerte zugewiesen.

2.2.3. Semantische Unterscheidung zwischen verteiltem und lokalem Objektmodell

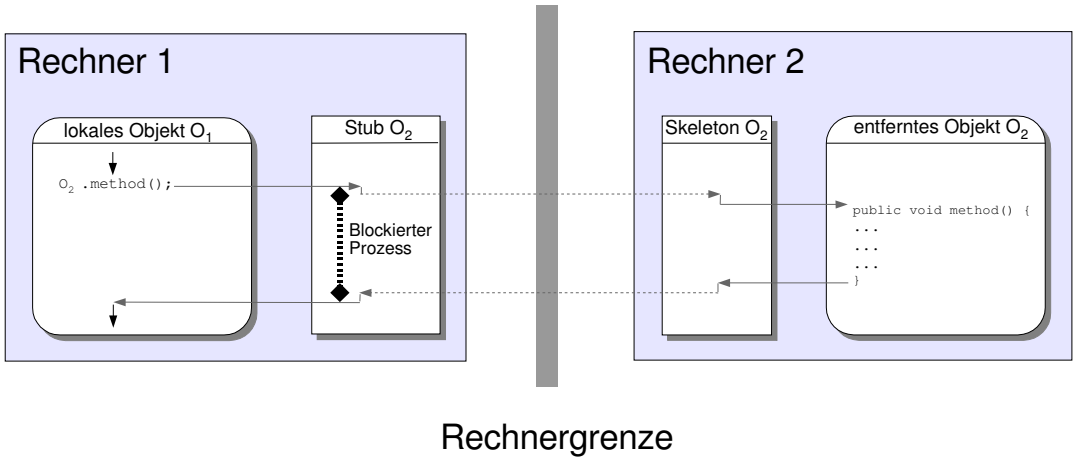

Abbildung 2.4.: Semantik des Methodenaufrufs bei RMI

Die Realisierung von RMI arbeitet mit verschiedenen Prozessen sowohl auf der Client- als auch auf der Server-Seite, wie es in Abbildung 2.4 verdeutlicht wird. Es wird dabei versucht, die Semantik eines gewöhnlichen Methodenaufrufs weitgehend wie im nicht verteilten Fall einzuhalten. Die Ausführung eines entfernten Methodenaufrufs erfolgt synchron, d.h. der Aufruf blockiert so lange, bis die Abarbeitung auf dem Remote-Rechner beendet ist. Die Syntax für einen Methodenaufruf ist, unabhängig davon ob das Objekt lokal oder *remote* ist, identisch, weil das Stub und das *remote*-Objekt bezüglich der entfernt aufrufbaren Methoden den gleichen Typ besitzen. Daher kann der `instanceof`-Operator genauso wie die Typkonversion im normalen Fall verwendet werden. Daneben gibt es aber einige grundlegende Unterscheidungen zwischen dem verteilten Objektmodell mit RMI und lokalem Modell:

- ☞ Das direkte Ansprechen eines *remote*-Objekts ist im verteilten Fall nicht möglich, sondern nur über seine Schnittstelle. Eine Klasse, deren Objekte *remote* sein sollen, muss das Interface `java.rmi.Remote` implementieren. Die Typdeklaration für diese Objekte im Programm muss über dieses Interface erfolgen. Alle entfernten Methodenaufrufe mit dem *remote*-Objekt erfolgen nicht direkt mit dem Objekt, sondern über seine Schnittstelle.

- ☞ Wenn bei einem entfernten Methodenaufruf lokale Objekte, also keine *remote*-Objekte als Argumente übergeben oder als Ergebnis zurückgeliefert werden, dann geschieht dies per Kopie und nicht als Referenzübergabe wie sonst in Java üblich. Eine Referenzübergabe würde in diesem Kontext keinen Sinn ergeben, da eine Referenz auf ein lokales Objekt in einer anderen JVM nicht verwendbar wäre. Außerdem verwendet RMI zur Parameterübergabe und Ergebnisrückgabe die Objektserialisierung, deshalb sind nur serialisierbare Objekte als Argumente für einen entfernten Methodenaufruf zulässig.

- ☞ Die möglicherweise auftretenden Ausnahmen müssen zusätzlich bei einem entfernten Aufruf über RMI abgefangen werden.

- ☞ Entfernte Zugriffe über RMI auf den statischen Anteil kann nicht durchgeführt werden. Dazu gehören statisch initialisierte Blöcke und als statisch deklarierte Variablen und Methoden.

- ☞ Öffentlich zugreifbare Objektvariablen können nicht direkt über RMI angesprochen werden.

- ☞ Ein weiterer Punkt betrifft die Zugriffsrechte der Methoden. Da jede in der *Remote*-Schnittstelle deklarierte Methode implizit den Modifikator `public` bekommt, sind automatisch alle entfernt aufrufbaren Methoden in einem *remote*-Objekt öffentlich zugängig.

- ☞ Die Semantik zur Objektsynchronisation wird mittels RMI nicht mehr gewährleistet. Im lokalen Fall besitzt jedes Java-Objekt ein *lock*, über das gegenseitiger Ausschluss beim Zugriff von mehreren Threads auf dieses Objekt realisiert wird. Nur derjenige, der das Lock besitzt, darf in den kritischen Abschnitt eintreten, die anderen müssen warten. So ist jedes Objekt mit einer *Wartemenge* der Threads assoziiert.

  Da Operationen auf *remote*-Objekten mittels RMI auf indirektem Weg erfolgen, also über Stubs und Skeletons, werden Synchronisationsopera-

tionen auf diesen Stubs[4] angewandt, nicht auf das eigentliche Objekt. Beispiel:

```
synchronized (ro) {
 ...
 notifyAll();
 ...
}
```

Die Anwendung von `synchronized` auf `ro` hat nicht die gewünschte Wirkung wie im lokalen Fall, denn der Zugriff wird lediglich auf den Stub `ro`, nicht auf das Objekt selbst geregelt.

Die hier geschilderten Aspekte, die durch die Verwendung von RMI als Kommunikationsplattform entstehen, müssen berücksichtigt und so gehandhabt werden, dass sie für den Benutzer transparent bleiben, wie im lokalen Objektmodell. Dies gilt insbesondere für das System *JScatter*, das auf dem RMI oder KaRMI aufbaut und mehrsträngige Java-Programme automatisch verteilt, ohne dass der Anwender dafür etwas tun muss. Für das gesamte Konzept und vor allem für die Entwurfsentscheidungen des *JScatter*-Systems spielen diese Erkenntnisse eine wichtige Rolle.

2.3. Multithreaded Java-Programme und ihre Ausführungsmodelle

Durch das *Multithreading*-Konzept eröffnet Java dem Programmierer die Möglichkeit, Nebenläufigkeit der Problemstellung in deren Lösung zu übernehmen oder gezielt nebenläufige Problemlösungen zu entwickeln. Es ist die Aufgabe des Programmierers, die Programme auf der Java-Quellcode-Ebene auf mehrsträngiger Weise zu formulieren. Man kann das Programm direkt übersetzen und auf einer JVM ausführen lassen. In diesem Fall wird das Programm *verzahnt* ausgeführt. Oder man nutzt den RMI-Mechanismus zur Kommunikation und bindet manuell zusätzlichen Code in sein Programm ein. Dieses lässt sich dann auf mehreren JVMs auf verschiedenen Rechnern ausführen. Es wird also *verteilt* ausgeführt. An dieser Stelle möchte ich auf die Aspekte der beiden Ausführungsmodelle eingehen.

[4] zu einem *remote*-Objekt können beliebig viele Stubs existieren.

2.3.1. Verzahntes und verteiltes Ausführungsmodell

Verzahnte Ausführung

Wie bereits im Abschnitt 2.1.2 beschrieben, besteht ein mehrsträngig formuliertes Java-Programm aus mehreren Ausführungssträngen, die sich zur Laufzeit im aktiven Zustand befinden. Innerhalb einer JVM auf einem Einprozessor-System werden diese Threads verzahnt ausgeführt. Hierbei handelt es sich um eine sequentielle Ausführung.

Parallele Ausführung auf Multiprozessor-System

Demgegenüber ist ein Geschwindigkeitsvorteil bei der Programmausführung allerdings nur zu erwarten, wenn die virtuelle Maschine auf einem Multiprozessorrechner zur Ausführung kommt und sie das Ausnutzen mehrerer Prozessoren unterstützt. Das heißt, die Java-Threads werden direkt auf die System-Prozesse des Betriebssystems abgebildet. Die Verteilung der System-Prozesse wird automatisch vom Betriebsystem selbst durchgeführt und man hat hier eine parallele Ausführung. Es können so viele Prozesse gleichzeitig abgearbeitet werden, wie die zur Verfügung stehenden Prozessoren. Sie operieren miteinander über den gemeinsamen Speicher. In den beiden Fällen ist der Kommunikationsaufwand fast null, die Zugriffe auf Objekte können als lokale Operationen gezählt werden.

Verteilte Ausführung

Die (kosten-)günstigere Alternative zur verteilten Ausführung solcher mehrsträngiger Programme ist der Einsatz einer verteilten Laufzeitumgebung. In dieser Umgebung beteiligen sich mehrere virtuelle Maschinen an der parallelen Lösung eines Problems, um eine Geschwindigkeitssteigerung zu erzielen. Die einzelnen virtuellen Maschinen werden auf unterschiedlichen Arbeitsstationen ausgeführt, die untereinander durch ein sehr schnelles Netzwerk gekoppelt sind.

Im Gegensatz zum parallelen Ausführungsmodell auf einem Multiprozessor-System, bei dem ein gemeinsamer Adressraum benutzt wird und alle Ausführungsstränge ihn teilen müssen, besitzt hierbei jede einzelne virtuelle Maschine ihren eigenen Adressraum. Die Objekte (auch Thread-Objekte) auf un-

terschiedlichen Maschinen können durch entfernte Methodenaufrufe miteinander kommunizieren. Die Ausführungsstränge und die Objekte, die einer Maschine zugeteilt sind, werden im gleichen Adressraum ausgeführt.

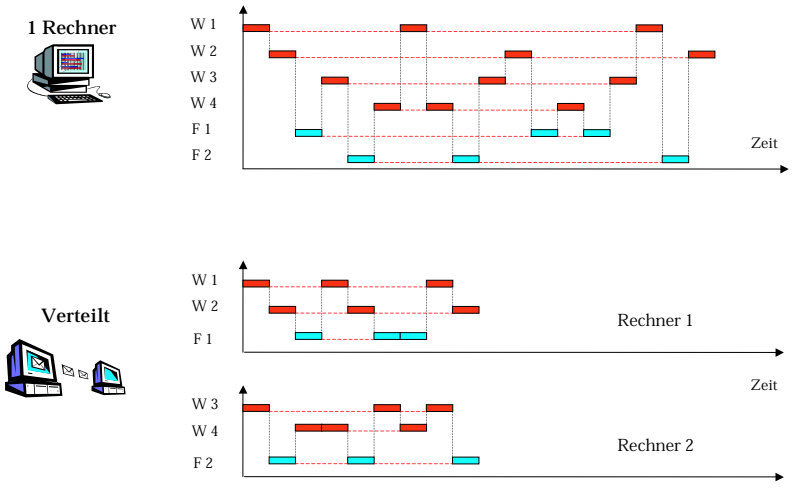

Abbildung 2.5.: Verzahnte und verteilte Ausführung

Abbildung 2.5 verdeutlicht den Unterschied zwischen einer verzahnten Ausführung in einem Adressraum und einem verteilten Ablauf in zwei JVMs auf zwei unterschiedlichen Rechnern. Hierbei handelt es sich um eine ideale Verteilung der Threads ($W_1 \ldots W_4$ stehen für Worker-Threads) und der Objekte (F_1, F_2 sind Fabrik-Objekte). Das heißt, es findet keine entfernte Kommunikationen zwischen den Objekten auf verschiedenen Rechnern statt. Dies ist aber in der realen Welt selten der Fall. Im normalen Fall kommunizieren die Threads über Datenobjekte über Rechnergrenzen hinweg, um Daten miteinander auszutauschen, wie es in Abbildung 2.6 dargestellt wird. Auf dem Bild sind die Thread-Objekte als Rechtecke und die Datenobjekte als Kreise dargestellt. Die Ausführungsstränge in derselben JVM werden *verzahnt* ausgeführt oder sogar parallel im Falle eines Mehrprozessor-Systems. Die Threads auf verschiedenen JVMs werden *parallel* abgearbeitet.

Wie in Abbildung 2.6 visualisiert gibt es zwei Arten von Kommunikationen: lokale und entfernte Zugriffe. Die entfernten Zugriffe über RMI verursachen erheblich größeren Aufwand als die lokalen. Die Konsequenz ist, dass durch die verteilte Ausführung der Threads nicht immer ein Geschwindigkeitsvorteil entsteht. Bei einer ungünstigen Verteilung der Threads und Objekte kann

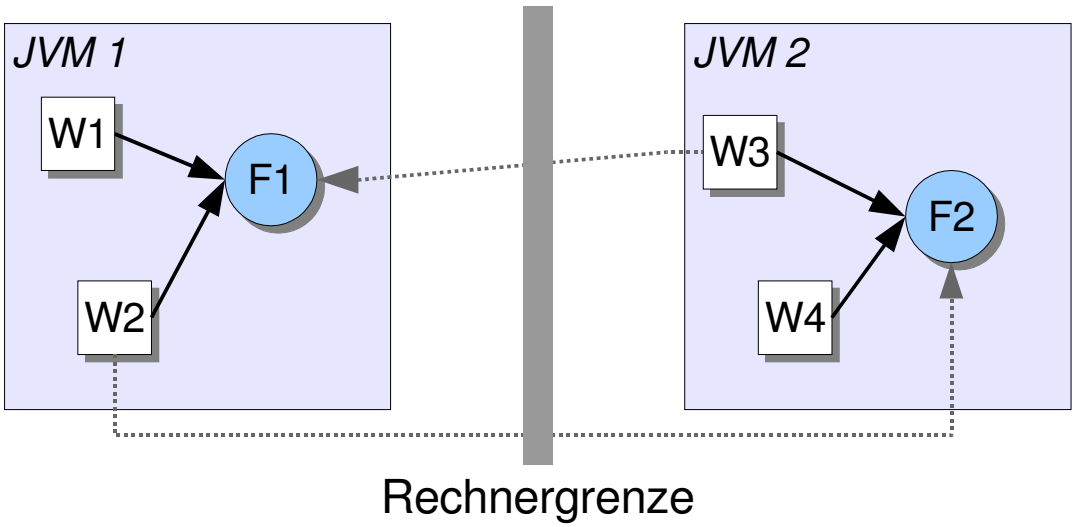

Abbildung 2.6.: Verteilte Ausführung

das dazu führen, dass das Programm sogar langsamer abläuft als die verzahnte Ausführung auf einem einzigen Rechner.

Zur Vereinfachung seien die an einer verteilten Laufzeitumgebung beteiligten Rechner nur Einprozessor-Systeme. Das heißt, es kommt keine echte Parallelität innerhalb einer JVM vor.

2.3.2. Ausführungskontext im verteilten Modell

Um den Ausführungskontext eines Java-Threads im verteilten Fall zu verstehen, betrachten wir nun den eines Ausführungsstrangs im sequentiellen Fall.

Sequentieller Ablauf

Nach der Erzeugung eines Thread-Objekts befindet sich ein Thread noch im initialen Zustand. Durch den Aufruf der `start()`-Methode wechselt er in den aktiven Zustand. Mit der Ausführung der `run()`-Methode, die unmittelbar innerhalb der `start()`-Methode aufgerufen wird, beginnt der eigentliche Ablauf eines Java-Threads. Nachdem die `run()`-Methode *vollständig* abgearbeitet wurde, beendet der Thread seine Ausführung. Während der Ausführung werden die einzelnen Anweisungen innerhalb von `run()` sequentiell abgearbeitet. Wenn innerhalb von `run()` Methodenaufrufe enthalten sind,

werden die aufgerufenen Methoden auf diese Weise genauso behandelt. Alle diese Aufrufe finden synchron statt. Da sich alle Objekte, also auch die Thread-Objekte, im selben Adressraum befinden, umfasst die gesamte Ausführungszeit eines Threads die Zeit zur Abarbeitung aller Anweisungen innerhalb von `run()` und die Abarbeitungszeit aller aufgerufenen Methoden.

Verteilter Ablauf

Im verteilten Fall können ein Thread und Objekte, deren Methoden von diesem Thread aufgerufen werden, in verschiedenen Adressräumen liegen. Die Ausführung eines entfernten Methodenaufrufs über RMI erfolgt auch synchron, wie es im Abschnitt 2.2 beschrieben wurde. Während der Bearbeitung der aufgerufenen Methoden auf anderen Rechnern wird der Thread auch so lange blockiert, bis das endgültige Ergebnis an den ursprünglichen Aufrufsort zurück gegeben wird. Die Semantik eines Aufrufs im verteilten Fall mittels RMI bleibt also erhalten.

Weiterhin wird bei solchen entfernten Aufrufen aber nur auf der Seite Rechenzeit verbraucht, auf der sich die Methoden tatsächlich befinden. Es ist so, als ob ein Thread sich durch einen entfernten Methodenaufruf und weitere Unteraufrufe von seinem ursprünglichen Standort zu anderen Rechnern wandelte, dort die Methode ausführte und schließlich wieder zu seinem Ausgangspunkt zurückkehrte. Zur Bearbeitungszeit eines solchen entfernten Methodenaufrufs müssen neben der Ausführungszeit der aufgerufenen Methoden noch zusätzlich die Kommunikationskosten mitgerechnet werden. Zu den Kommunikationskosten tragen die Abarbeitung der Stubs und Skeleton, die Serialisierung der Parameter und Rückgabewerte und die eigentliche Übertragung der Bytestrom über das Netzkwerk Rechnung. Dies ist der wesentliche Unterschied zur Ausführung eines Java-Threads in einem Adressraum.

2.3.3. Verteilte Ausführungsstränge und einige relevante Eigenschaften

In meinem gesamten Konzept zur automatischen Verteilung von mehrsträngigen Java-Programmen liegt dieses Ausführungsmodell zugrunde. Im Folgenden werden die verteilten Threads und einige relevante Eigenschaften betrachtet.

Aktivitäten - Die verteilten Ausführungsstränge

Das Konzept der verteilten Ausführungsstränge wurde früher von D. Jensen an der Carnegie Mellon University (CMU) [CJR92] als *distributed thread* eingeführt. Es wird im Kern des verteilten Echtzeit-Betriebssystems Alpha verwendet. Ein *distributed thread* ist ein leichgewichtiger Prozess, der sich über mehrere Adressräume erstreckt. In diesem Umfeld ist ein solcher Thread zusätzlich noch mit einer Menge von Parametern und Attributen assoziiert, die das Scheduling- und Echtzeit-Verhalten beeinflussen. Er ist unabhängig von dem Ort, wo er mit der Ausführung beginnt. Ein solcher verteilter Thread wurde im Verteilungskonzept beim JavaParty-System (siehe 4.1.1) *RemoteThread* genannt und gilt dort als eine *Aktivität*. Um ihn vom üblichen Java-Thread zu unterscheiden, möchte ich im Verlauf der Arbeit einen derartigen Thread auch als eine *Aktivität* bezeichnen.

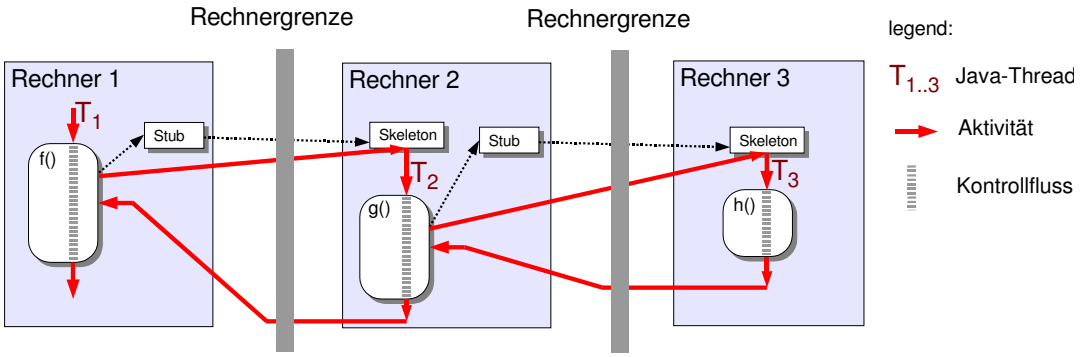

Abbildung 2.7.: Aktivität und verteiltes Ausführungsmodell

Eine Aktivität besteht aus mehreren konventionellen Java-Threads, die sich auf verschiedenen JVMs befinden und die eigentliche lokale Ausführung der einzelnen Methodenaufrufe durchführen. Wie ein konventioneller Java-Thread beginnt eine Aktivität mit der Ausführung der `run()`-Methode seinen Ablauf und endet dann, wenn diese Methode vollständig abgearbeitet wurde. Abbildung 2.7 veranschaulicht den Ablauf einer Aktivität (dicke Pfeile). Sie besteht aus drei konventionellen Ausführungssträngen $T_1 \ldots T_3$, die jeweils die Methoden `f()`, `g()` und `h()` auf drei verschiedenen Rechnern ausführen. Im Beispielcode wird die Methode `g()` von `f()` und `h()` von `g()` aufgerufen. Die Aufrufe gehen aber tatsächlich über die Stubs und Skeletons. Zu beachten ist, dass die vom RMI-System erzeugten Threads für den Empfang der Methodenaufrufe (Skeleton-Threads) von den drei obigen Threads $T_1 \ldots T_3$ zu unterscheiden sind.

Kontrollfluss einer Aktivität

Jeder lokale Thread besitzt eine eigene Ablaufstruktur, dargestellt als lokalen Kontrollflussgraphen (CFG). Eine Aktivität hat auch ihre eigene Ablaufstruktur. Da eine Aktivität aus mehreren verschiedenen lokalen Java-Threads besteht, erstreckt sich ihr Kontrollfluss über die Rechnergrenzen hinaus. Er umfasst also die lokalen Kontrollflüsse der Methoden, die von den zugehörigen Ausführungssträngen lokal ausgeführt werden. Im Vergleich zu einem gewöhnlichen CFG besitzt er Sonderkanten, die zwei verschiedene lokale Kontrollflussgraphen verbinden. Die eine Kante präsentiert den entfernten Aufruf und die andere stellt die Rückgabe dar. Der Datenfluss bei diesen beiden Kanten kann Parameter und Rückgabewert eines Methodenaufrufs enthalten. Betrachten wir das Beispiel in Abbildung 2.7: Der Kontrollfluss der darin dargestellten Aktivität umfasst die lokalen Kontrollflüsse der lokalen Threads $T_1 \dots T_3$. Das sind also die Kontrollflüsse der drei Methoden `f()`, `g()` und `h()`, die sich von den drei lokalen Java-Threads ausführen lassen. Für die Analyse solcher mehrsträngigen Java-Programme spielt dieser Aspekt eine wichtige Rolle, worauf später im Kapitel 6 in Details eingegangen wird.

Identität einer Aktivität

Um die Synchronisation zwischen verschiedenen Aktivitäten und deren Manipulation gewährleisten zu können, muss jede Aktivität eine eindeutige Identität besitzen, wie es beim konventionellen Thread der Fall ist. Wenn eine Identitätsüberprüfung erforderlich ist, beispielsweise bei der Synchronisation zweier verschiedener Aktivitäten über ein Monitor-Objekt, müssen sich alle dazu gehörigen lokalen Threads mit dieser im ganzen System eindeutigen Identität erkennen lassen. Sonst wird die ursprüngliche Ausführungssemantik nicht mehr erhalten bleiben. Es besteht sogar die Gefahr zur Verklemmung und man hat keine Möglichkeit die Aktivitäten zu kontrollieren, was aber bei normalen Java-Threads möglich ist. Dieses Konzept ist nicht nur wichtig bei der Modellierung und Verteilungsanalyse von multithreaded Java-Programmen, sondern muss auch von der verteilten Laufzeitumgebung unterstützt werden. Die entstehende Problematik wurde ausführlich in [WTV02a] und [HMRT03] diskutiert.

Parallelität der Aktivitäten

Ein nebenläufiges Java-Programm besteht konzeptionell aus mehreren Aktivitäten, die die einzelnen Programmobjekte lokal oder entfernt auf eine durch den Programmfluss festgelegte Weise berühren. Wenn zwei verschiedene Aktivitäten die gleiche Methode eines Objekts im selben Adressraum ausführen, wird dies verzahnt geschehen. Diese Verzahnung kommt zustande, weil zwei normale Java-Threads, die jeweils zu einer der beiden Aktivitäten gehören, auf dieser Maschine dieselbe Objektmethode ausführen.

Die Aktivitäten können aber auch parallel auf verschiedenen Rechnern ablaufen. Dies ist der Fall, wenn auf einem Rechner die eine Aktivität einen entfernten Methodenaufruf ausführt und auf das Ergebnis wartet. Während der Wartezeit kann eine andere Aktivität seine Ausführung beginnen bzw. fortsetzen. So betrachtet werden die Aktivitäten im gesamten System parallel ausgeführt.

Der Nachteil dieses Ausführungsmodell ist allerdings, dass der Wechsel einer Aktivität von einer Maschine zu einer anderen und zurück zusätzliche Kommunikationskosten verursacht. Damit eine Geschwindigkeitssteigerung durch die verteilte Ausführung erzielt werden kann, muss der Gewinnanteil an Parallelität größer sein als der Anteil an Kommunikationsaufwand. Es gibt Anwendungsklassen, die sich durch eine gute Verteilung der Objekte auf mehrere Rechner schneller ausführen lassen. Für Anwendungen, bei denen erheblich mehr Kommunikation als Rechnen stattfindet, lohnt es sich nicht zu verteilen.

Zusammenfassung

Die hier behandelten Aspekte stellen eine wichtige Grundlage zum Konzept zur automatischen Verteilung der Objekte eines mehrsträngigen Java-Programms dar. Diese geschilderten Aspekte spielen nicht nur eine wichtige Rolle beim Entwurf des *JScatter*-Systems, sondern auch beim Einsatz sowie bei der Entwicklung der Analyseverfahren zur Berechnung der Programmeigenschaften, die relevant für die automatische Verteilung sind.

3. Grundüberlegungen zur automatischen Verteilung

Inhalt

Nach dem im Abschnitt 2.3 vorgestellten Ausführungsmodell entsteht die Rechenlast dort, wo eine Aktivität die aufgerufene Methode eines Objekts ausführt. Zu einer entfernt aufgerufenen Methode tragen nicht nur die verrichteten Arbeiten zur Ausführung der Methode, sondern auch die teuren Kommunikationskosten Rechnung. Konzeptionell sind in diesem Modell die Aktivitäten die *kleinsten Einheiten* der Parallelität. Daher sollen zugehörige Threads möglichst auf verschiedene Rechner verteilt werden, damit eine echte parallele Ausführung erzielt werden kann. Der entscheidende Faktor ist dennoch, die Objekte, mit denen die Aktivitäten in Berührung kommen, so geschickt mit ihren zugehörigen Aktivitäten zu verteilten, dass so wenig Kommunikationen wie möglich für die entfernten Methodenaufrufe entstehen.

Diesen Aspekt sollte ein Programmierer befolgen, wenn er sein mehrsträngig formuliertes Programm manuell verteilen will. Zunächst sind bestimmte maßgeschneiderte Verteilungsstrategien für dieses Programm aufzustellen, die er später als Verteilungsdirektive in seinen Programmcode einbauen wird. Für diese Strategien hat er davor alle benötigten Informationen über das Programm zu sammeln. Das heißt, er hat das Programm gezielt nach diesem Aspekt *analysiert* und eventuell einige Abläufe des Programms auch *beobachtet*. Nun folgen „nur noch“ die technischen Schritte: Das Programm wird mittels RMI in geeigneter Weise zerlegt, damit es wie gewünscht verteilt auf verschiedenen Rechnern ablaufen kann.

Mein Ansatz ist es, all diese Schritte für jedes mehrsträngige Java-Programm zu automatisieren und dennoch eine gewünschte Verteilung des Programms zu erhalten, durch die eine Geschwindigkeitssteigerung erzielt wird. Als Ausgangspunkt dient ein als multithreaded formuliertes Java-Programm. Um Verteilungshinweise für ein Programm *automatisch* zu berechnen ist der Einsatz der *Programmanalyse* erforderlich. Die Hauptidee ist es, dass zunächst so viel Strategien wie möglich zur Verteilung so aufgestellt werden, damit diese auf jedes Programm angewandt werden können. Zur Umsetzung der Strategien braucht man bestimmte *Programmeigenschaften*, die sich durch Programmanalysen gewinnen lassen. Die für ein Programm ergebenen Verteilungsentscheidungen können in geeigneter Weise im Programm angewandt werden. Mit Hilfe von Werkzeugen kann das ursprüngliche Programm zusammen mit den Verteilungshinweisen in eine verteilte Version transformiert werden. Diese enthält die notwendigen RMI-Kommunikationsmechanismen und kann auf einer verteilten Laufzeitumgebung ausgeführt werden. Die tatsächliche Verteilung der Objekte wird vom System automatisch durchgeführt.

Dieses Kapitel beschreibt die Grundüberlegungen zur automatischen Verteilung solcher mehrsträngig formulierten Java-Programme. Im ersten Abschnitt werden die Schritte vorgestellt, nach denen ein Programmierer normalerweise vorgeht, um sein Programm manuell auf mehrere Rechner verteilt ausführen zu können. Auf diesen aufwändigen Schritten aufbauend wird mein Ansatz zur Automatisierung vorgestellt. Dabei wird beschrieben, welcher Schritt mit welchem Hilfsmittel automatisch realisiert werden kann. Somit wird dann motiviert, dass eine automatische Verteilung sehr hilfreich ist und dem Anwender viele dieser Schritte abnehmen kann. Anschließend werden einige Überlegungen dargestellt, die als Grundlage für das Konzept zur automatischen Verteilung dienen. Einige Voraussetzungen für eine vollständige Automatisierung der Verteilung sowie allgemeine Kriterien werden abschließend diskutiert.

3.1. Manuelle Verteilung eines mehrsträngigen Java-Programms

Beim Konzept entfernter Methodenaufrufe mittels *Remote Method Invocation* ist das Bestreben deutlich erkennbar, die durch entfernte Aufrufe verursachte Netzwerkkommunikation vor dem Programmierer zu verbergen. Mittels RMI wird dem Anwender die Semantik eines entfernten Objekts so dargestellt als ob die eines entsprechenden lokalen Objekts wäre. So kann der Programmierer sich auf die Formulierung seines Programms konzentrieren und wird von der Implementierung der entfernten Zugriffe auf verteilte Objekte nicht mehr belastet. Trotzdem bleibt immerhin noch die Aufgabe des Programmierers die Entscheidung zu treffen, welche Aktivitäten bzw. welche Objekte auf welchen Rechner zu platzieren bzw. zu erzeugen sind. Darüber hinaus muss die Synchronisation zwischen den Aktivitäten und deren Kommunikation gewährleistet werden. Diese Aufgaben sind nicht nur „auf Konzeptpapier“, also durch Analyse des Eingabeprogramms, Treffen der Platzierungsentscheidungen usw., sondern auch in der technischen Realisierung zu erledigen.

3.1.1. Hand-Verteilung

Zunächst hat der Benutzer alle Platzierungsentscheidungen problemspezifisch auf der Entwurfsebene zu treffen. Der erste Schritt ist zu entscheiden,

welche Objekte zu verteilen sind und welche lokal zu lassen sind. Die lokal zu erzeugenden Objekte sind beispielsweise diejenigen, die Hardware-nahe Eigenschaften besitzen und deshalb mit einer lokalen Maschine festgebunden sind. Eine Platzierung solcher Objekte auf andere Maschinen macht keinen Sinn. Kandidaten für lokale Objekte sind auch diejenigen, die als Hilfsobjekte innerhalb anderer Objekte verwendet werden und deshalb nur lokal für umgebende Objekte unmittelbar zugreifbar sind. Für solche Objekte sind keine entfernten Zugriffe erforderlich.

Es sind im Allgemeinen zwei verschiedene Arten von Objekten zu unterscheiden: *Thread-Objekte* und Nicht-Thread-Objekte, im Folgenden nur als *Objekte* bezeichnet. Methoden der Objekte, auf denen mehrere Aktivitäten auf verschiedenen Rechnern Operationen durchführen, müssen über Rechnergrenzen hinweg zugreifbar sein. Operationen können Berechnungen oder Synchronisation sein.

Der zweite Schritt, auch der schwierigere Teil, ist festzulegen, welche Objekte mit welchen zusammen zu gruppieren sind und wie sie auf welchen Zielrechner zu platzieren sind. Im Prinzip sollen die Thread-Objekte möglichst auf verschiedene Maschinen verteilt werden. Nachdem diese Verteilung festgelegt wurde, erfolgt dann die Verteilung der Nicht-Thread-Objekte. Diese Objekte sollen dorthin platziert werden, wo die meisten Aufrufe ihrer Methoden durchgeführt werden und dadurch viel Berechnungsaufwand entsteht. Somit können viele zusätzliche entfernte Kommunikationen vermieden werden.

Nach diesem Prinzip kann der Programmierer verschiedene Strategien aufstellen, die er zur Verteilung der Objekte seines Programms verwendet. Einige Strategien ergänzen sich gut, so dass eine gute Verteilung resultiert. Es gibt aber auch Strategien, die sich nicht gut kombinieren lassen. Beispielsweise besagt die eine Strategie, dass Objekte mit einer Aktivität zusammen verteilt werden, weil diese Aktivität viel Arbeit auf ihnen verrichtet. Eine andere gibt vor, dass diese Objekte mit einer anderen Aktivität zu platzieren sind, weil sie viel mit ihnen kommuniziert. Wenn nicht zueinander passende Strategien kombiniert werden, können unnötige Kommunikationskosten entstehen und dies führt zu einer Geschwindigkeitsreduktion. Der Programmierer hat dann zu entscheiden, welche Strategien mit welchen zu kombinieren sind, damit eine Geschwindigkeitssteigerung durch die verteilte Ausführung erzielt werden kann.

Je nach Verteilungsstrategien werden bestimmte Programmeigenschaften benötigt. Das sind beispielsweise die Kommunikationsstruktur zwischen den

Aktivitäten und Objekten, die Beziehung zwischen den Objekten und/oder die (abgeschätzten) Ausführungskosten, die die Aktivitäten auf ihnen verursachen. Um all diese Informationen zu beschaffen muss der Programmierer sein Programm durch „scharfes Hingucken“ genau *analysieren*. So kann er die Laufzeitstruktur einzelner Methoden, die Hierarchie der Klassen, die Benutzungsbeziehung durch Referenzierung oder Methodenaufrufe zwischen den Objekten usw. bestimmen. Nach diesen Entwurfsschritten muss er das Ganze noch technisch umsetzen.

3.1.2. Technische Umsetzung

Um das Programm mit den darauf anzuwendenden Verteilungsstrategien ohne Hilfe eines Systems wie *JScatter* nur unter Verwendung von RMI verteilt auf vielen Rechnern ausführen zu können, sind viele aufwändige technische Schritte notwendig. Grob wird die Umsetzung in zwei Phasen aufgeteilt: das Umprogrammieren des ursprünglichen Programms in eine verteilte Version mit eincodierten Verteilungshinweisen unter Verwendung von RMI und das verteilte Ausführen von diesem auf mehreren Rechnern.

Umschreiben des Programms

Beim Umprogrammieren des ursprünglichen Programms in eine verteilte Version muss als Erstes festgelegt werden, *wie* die Objekte platziert werden. Es gibt zwei Möglichkeiten dazu: Das Objekt wird sofort am gewünschten Zielort erzeugt oder es wird zunächst lokal erzeugt, dann zu einem anderen Zeitpunkt zum Zielort umgezogen. Der Umzug eines Objekts nach seiner Erzeugung durch die Serialisierungstechnik, die Java-RMI anbietet, ist immer teurer und aufwändiger als die erste Möglichkeit. Deshalb ist die Erzeugung eines Objekts am Zielort vorzuziehen. Der Benutzer muss bei der Formulierung der Problemlösung dann die RMI-Konstrukte nach seiner getroffenen Verteilungsentscheidung in sein Programm an passenden Stellen einbauen bzw. das Programm entsprechend so umgestalten, dass es in verschiedene Teile aufgeteilt wird und somit diese Teile semantikerhaltend auf verschiedenen Rechnern ablaufen können. Hierbei sind die semantischen Aspekte, die in 2.2.3 aufgelistet sind, zu berücksichtigen. Hinzu kommen noch Kommunikations- und Synchronisationsaufgaben und Behandlungen in Fehlerfällen. Außerdem muss der Programmierer selbst für die Serialisierung /

Deserialisierung der Parameter und Ergebnisrückgabe bei entfernt aufgerufenen Methoden sorgen. Bei diesem Entwicklungsprozess ist vorausgesetzt, dass der Programmierer neben der parallelen Programmierung in Java die RMI-Technik schon perfekt beherrscht. Dieser Vorgang ist sehr fehleranfällig und aufwändig.

Ausführen des verteilten Programms

Nach erfolgreicher Übersetzung des umgeschriebenen Programms und der Generierung von Stubs und Skeletons mit Hilfe des im JDK enthaltenen Programms `rmic` sind noch einige Schritte zur eigentlichen Ausführung des als verteilbar vorbereiteten Programms von Hand durchzuführen. Das Programm, die generierten Stubs und Skeletons und die benutzten Bibliotheken müssen auf jedem beteiligten Rechner physikalisch vorhanden sein. Auf jedem beteiligten Rechnerknoten muss der Benutzer die RMI-Registrierung starten, welche die Aufgabe der Verwaltung, des Bindens und der Identifizierung der *remote*-Objekte auf diesem Rechner übernimmt. Nun müssen die Teile, die aus dem ursprünglichen Programm durch das Umprogrammieren entstanden, auf einzelnen Rechnern so gestartet werden, dass es dem ursprünglichen Ablauf entspricht. Wenn Fehler während der Ausführung auftreten, die der Benutzer aber schon beim Programmieren mit Abfangmaßnahmen berücksichtigt, werden sie durch RMI dann weitergeleitet und ausgegeben. Es ist trotzdem sehr mühsam, die Fehlerursache zu finden, z.B. bei Laufzeit- oder Kommunikationsfehlern, denn die Fehlermeldungen können auf verschiedenen Rechnern verstreut sein, obwohl sie logisch zusammengehören.

Es ist also wünschenswert, all diese aufwändigen Schritte so weit wie möglich zu automatisieren. Der Programmierer soll sich nur auf seine parallele Problemlösung konzentrieren und sie anschließend in mehrsträngiger Form in Java wie gewöhnlich formulieren. Zur Ausführung des Programms hat der Benutzer nur noch die Aufgabe, die Rechnerknoten zu bestimmen, auf denen das Programm verteilt laufen soll. Der Rest soll dann von selbst gehen. Wenn dies gelingt, erreicht man die völlige Transparenz zwischen der nebenläufigen Formulierung eines Programms und dessen verteilter Ausführung.

3.2. Konzept zur automatischen Verteilung

Basierend auf dem obigen Vorgehen sollen die durchzuführenden Schritte so weit wie möglich automatisiert werden. Das Treffen der Entscheidung, ob die aus dem Programm entstehenden Objekte zu verteilen sind und wie sie gruppiert werden, bildet den Kernpunkt dieser Arbeit. Dafür sind eine eingehende automatische Analyse solcher nebenläufigen Programmcodes und daraus abgeleitete Verteilungsaussagen notwendige Voraussetzungen. Nachdem solche Verteilungsinformationen zur Verfügung stehen, kann das Programm automatisch in eine verteilte Form transformiert werden, so dass Kommunikationen über Rechnergrenzen hinweg mit RMI oder ähnlicher Technologie erfolgt. Schließlich braucht man zur Ausführung des transformierten Programms eine verteilte Laufzeitumgebung. Diese besondere Laufzeitumgebung enthält mehrere JVMs auf verschiedenen Rechnerknoten mit benötigten Kommunikationsmechanismen. Sie ist in der Lage, das verteilte Programm nebenläufig auszuführen und soll sich aus Benutzersicht transparent wie eine normale JVM verhalten.

3.2.1. Das Szenario

Abbildung 3.1 stellt das gesamte Szenario von der Programmanalyse bis zur verteilten Ausführung des transformierten Programms dar. Ausgangspunkt ist ein mehrsträngig formuliertes Java-Programm, das in Form von *Java-Bytecode* vorliegt. Die Programmanalyse wird eingesetzt, um relevante Eigenschaften des Eingabeprogramms für die automatische Transformation und die automatische Verteilung zu bestimmen. Vor der Laufzeit kommen statische Analyseverfahren zur Berechnung benötigter statischer Programmeigenschaften zum Einsatz. Einige von ihnen müssen der Transformationsphase vollständig zur Verfügung stehen: Wenn beispielsweise alle Objekte einer Klasse in der gesamten Anwendung nur lokal benutzt werden, dann braucht diese Klasse nicht um einen Mechanismus für entfernte Zugriffe, also RMI-Anbindung, erweitert zu werden. Beruhend auf diesen Informationen wird anschließend eine verteilte Version des Programms durch eine automatische Transformation generiert, die dann auf der verteilten Laufzeitumgebung ausgeführt werden kann.

Zur automatischen Verteilung des Programms wird das Konzept *Verteilungsplan* verwendet. Ein Verteilungsplan enthält u.a. allgemein formulier-

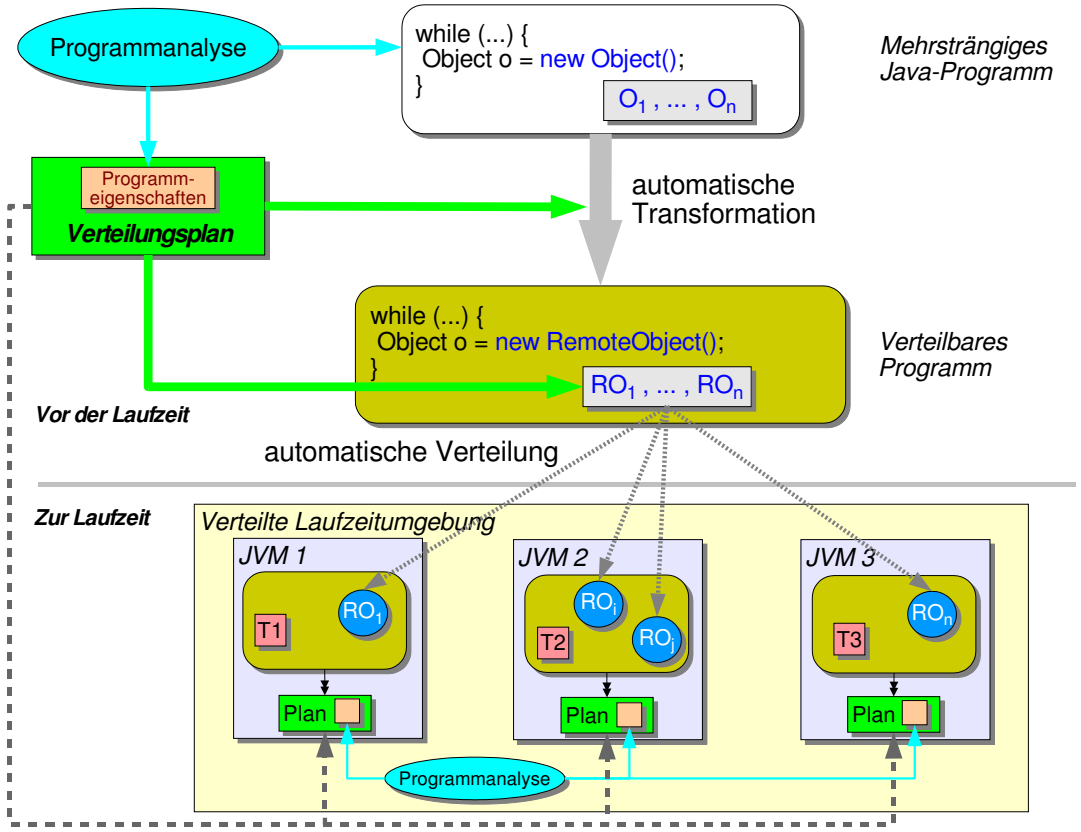

Abbildung 3.1.: Programmanalysen und verteilte Ausführung

te Verteilungsstrategien. Sie liefern die Entscheidung darüber, ob ein zu erzeugendes Objekt ein Remote-Objekt oder ein lokales ist und, im Fall eines Remote-Objektes, auf welcher JVM es erzeugt werden soll. Für bestimmte Verteilungsstrategien werden bestimmte Programmeigenschaften benötigt, die *vor der Laufzeit und zur Laufzeit* von entsprechenden *statischen und dynamischen Analyseverfahren* beschafft werden. Die Strategien werden vor der Laufzeit soweit wie möglich berechnet und direkt in das Programm in Form von Anfragen an den Verteilungsplan eincodiert. Zur Laufzeit wird unmittelbar vor einer Objekterzeugung eine Verteilungsentscheidung getroffen und das Objekt wird entsprechend platziert.

Zur verteilten Ausführung des transformierten Programms wird eine *verteilte Laufzeitumgebung* benutzt. Sie umfasst mehrere JVMs, die sich jeweils auf einem Rechner befinden. Während der Programmausführung werden die Objekte des transformierten Programms auf die beteiligten JVMs platziert. Die

Kommunikation zwischen den Objekten auf verschiedenen JVMs erfolgt über RMI.

Während des Programmlaufs werden ebenfalls bestimmte Laufzeitdaten gesammelt. Sie werden zum einen für einige dynamische Programmeigenschaften benötigt und füllen zum anderen einige Informationslücken, die vor der Laufzeit durch statische Programmanalyse nicht vollständig bestimmt werden konnten und deshalb noch offen gelassen werden. Basierend auf den zusammengesetzten Informationen kann der Verteilungsplan verbesserte Entscheidungen für zukünftig zu platzierende Objekte treffen.

3.2.2. Verteilungsentscheidungen

Aktivitäten greifen auf Objekte auf unterschiedlichen Maschinen zu. Bei entfernten Methodenaufrufen werden hohe Kommunikationskosten verursacht: die Bearbeitungszeit der Stubs und Skeletons für die Weiterleitung der Aufrufe, die Serialisierung der Parameter und Rückgabewerte und die eigentliche Übertragung der Bytestrom-Daten über das Netzwerk. Je mehr die Objekte entfernt miteinander kommunizieren, desto höher ist die Last auf dem Netzwerk. Andererseits wird der Code der aufgerufenen Methode auf der Maschine ausgeführt, auf der sich das diese Methode enthaltende Objekt befindet. Deshalb verursachen die Aktivitäten Ausführungskosten auf allen Maschinen, die die zugegriffenen Programmobjekte enthalten. Also hängen die Parallelität und die zusätzlichen Kommunikationskosten einer verteilten Anwendung stark von der Platzierung der Objekte innerhalb der verteilten Laufzeitumgebung ab.

Wie oben begründet sollte die Platzierung der Objekte während der Objekterzeugung durchgeführt werden. Daher muss die Verteilungsentscheidung für jedes Objekt unmittelbar vor seiner Instanziierung getroffen werden, also zur Ausführungszeit des verteilten Programms. Für verschiedene an einer Erzeugungsstelle zur Laufzeit entstehende Objekte können unterschiedliche Verteilungsentscheidungen getroffen werden. Das hängt von der dort auftretenden Programmeigenschaften ab, die wiederum vom Laufzeitverhalten abhängig sind. Die zuerst getroffene Verteilungsentscheidung für Objekte an einer Erzeugungsstelle heißt *initiale Platzierungsentscheidung*. Wann und in welcher Reihenfolge die Entscheidungen für eine Objekterzeugungsstelle getroffen werden, legen die Verteilungsstrategien fest. Verschiedene Strategien werden in einem *Verteilungsplan* [Klo02] zusammengefasst.

Konzeptionell werden die Verteilungsentscheidungen mittels Funktionen berechnet, sogenannte Objektverteilungsfunktionen (*Object Distribution Functions - ODF*). Zur Transformationszeit liefern die Funktionen für jede Objekterzeugungsstelle das Ergebnis, ob die hier entstehenden Objekte Remote-Objekte sind und daher Transformation dafür notwendig wird. Zur Laufzeit beschreibt das Ergebnis eines Funktionsaufrufs, auf welcher JVM das Objekt allokiert wird. Die ODFs hängen von statischen und dynamischen Eigenschaften des Programms ab, die ohne Benutzereingabe automatisch durch *Programmanalyse* berechnet werden. Diese Eigenschaften werden in Form von Variablen innerhalb der ODFs ausgedrückt, im Folgenden als Programmeigenschaftsvariable (*Program Property Variable - PPV*) bezeichnet. Sie können unterschiedliche Werte annehmen, die sich aus statischen und dynamischen Analyseergebnissen ergeben. Durch dieses Konzept können Ergebnisse statischer und dynamischer Analysen innerhalb der ODFs kombiniert verwendet werden. Die Bindung statisch berechneter Werte zu den PPVs ermöglicht eine partielle Evaluation der ODFs, nachdem eine statische Programmanalyse durchgeführt wurde. Diese Ergebnisse werden unmittelbar für die automatische Transformation verwendet. Abbildung 3.2 verdeutlicht den Ansatz, dass eine Verteilungsstrategie eine Zusammensetzung verschiedener ODFs darstellt und welche Rolle die Programmanalyse hierbei spielt.

Die ODFs werden zur Laufzeit evaluiert, denn einerseits sind Werte einiger Variablen, z.B. die Anzahl verfügbarer JVMs, erst nach dem Start der verteilten Laufzeitumgebung bekannt. Andererseits werden bestimmte Laufzeitinformationen über das Programm, die keineswegs über statische Programmanalyse zu bestimmen sind, benötigt, um Werte für dynamische Programmeigenschaften der ODFs zu berechnen. Zum Beispiel können nur grobe Abschätzungen vor der Laufzeit darüber gemacht werden, wie oft ein bestimmter Programmteil des Eingabeprogramms während der Laufzeit zur Ausführung kommt. Bessere Ergebnisse lassen sich nur erzielen, indem man die statische Abschätzungen durch präzise Daten ersetzt, die während der Laufzeit des Programms gewonnen werden können. Somit liefern die ODFs Entscheidungen für die initiale Verteilung und gegebenenfalls auch während des Programmablaufes verbesserte Entscheidungen zur Platzierung von zukünftig zu erzeugenden Objekten bzw. Hinweise dazu, ob vorhandene Objekte migriert oder repliziert werden sollen.

Die Güte einer Verteilungsentscheidung hängt von der Exaktheit der Analyseergebnisse ab. Da die statischen Analysen vor der Laufzeit durchgeführt werden, haben sie im Gegensatz zur dynamischen Analyse keinen nachteiligen

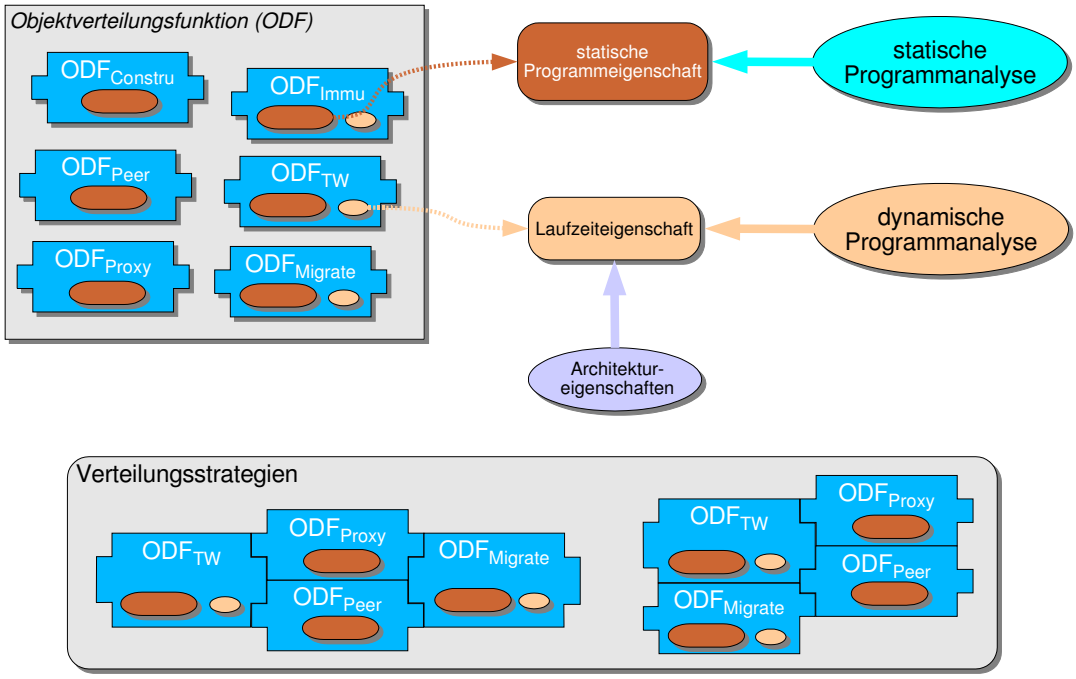

Abbildung 3.2.: Verwendung der Analyseergebnisse in Verteilungsstrategien

Einfluss auf die Ausführungsgeschwindigkeit des Programms. Deshalb können komplexere statische Analyseverfahren eingesetzt werden. Sie müssen die Programmeigenschaften so genau wie möglich bestimmen, und ihre statischen Ergebnisse werden soweit vorbereitet, dass zur Laufzeit nur minimaler Aufwand an dynamischer Analyse entsteht. Denn nicht nur das Beschaffen der Laufzeitinformation, sondern auch der Prozess zur Verbesserung der Platzierungsentscheidungen, also die Kombination statischer und dynamischer Analyseergebnisse und Neuberechnung der Verteilungsentscheidung, verursachen zusätzliche Laufzeitkosten bei der Programmausführung.

In diesem Konzept wird die eigentliche Programmanalyse zur Berechnung der Programmeigenschaften von der Formulierung der ODFs und somit von den Verteilungsstrategien im Verteilungsplan voneinander getrennt. Der Grund ist, dass die Objektverteilungsfunktionen im Prinzip nur die Werte der Programmeigenschaften benutzen, aber nicht davon abhängen, wie sie berechnet werden. Außerdem ermöglicht das verwendete Bindungskonzept den Einsatz verschiedener Werte zu verschiedenen Zeitpunkten. Dadurch können statische und dynamische Analysen kombiniert zum Einsatz kommen. Im Kapitel 5 werde ich ausführlich auf das hier vorgestellte Konzept des

Verteilungsplans sowie die einzelnen Verteilungsstrategien mit ihren ODFs eingehen.

3.2.3. Automatische Transformation

Die Aufgabe, welche während der Transformationsphase erledigt wird, besteht darin, den Programmcode so zu modifizieren, dass zur Laufzeit die Objekte nach den Verteilungsdirektiven aus dem Verteilungsplan auf den JVMs der Laufzeitumgebung instanziiert werden. Außerdem muss gewährleistet werden, dass entfernte Zugriffe aus anderen JVMs auch erfolgen können und die Ausführungssemantik des ursprünglichen Programms erhalten bleibt.

Während der Transformationsphase wird jede Erzeugungsstelle wie folgt betrachtet: Für diese Stelle wird eine Planevaluierung durchgeführt. Wenn der Plan das Ergebnis liefert, dass die Objekte lokal zu erzeugen sind und auf sie keine entfernten Zugriffe in der gesamten Anwendung benötigt werden, dann bleibt der Programmcode an dieser Stelle unverändert. Anderenfalls wird die Stelle entsprechend so transformiert, dass die dort zur Laufzeit entstehenden Objekte auf einer bestimmten JVM erzeugt werden können und eine Anfrage an den Verteilungsplan für die Platzierung angebunden wird. Wenn der Plan zur Laufzeit angefragt wird, werden bestimmte Strategien angewandt. Entsprechend der getroffenen Entscheidung wird das Objekt auf einer der JVMs erzeugt. Außerdem werden alle Zugriffsstellen auf ein Remote-Objekt auch so entsprechend transformiert, dass die Semantik im verteilten Fall erhalten bleibt.

Bei diesem Transformationskonzept wird die Objektsemantik berücksichtigt: Eine urspüngliche Klasse wird in zwei Teile aufgeteilt. Der eine stellt den statischen Anteil der Klasse dar, der andere den Instanzanteil. Aus dem Instanzanteil können mehrere Remote-Objekte entstehen. Jedes von ihnen hat eine eindeutige Objekt-Identität im gesamten verteilten System und kann über eine Referenz entfernt zugegriffen werden. Dafür wird das *Handle*-Konzept verwendet. Es gibt aber nur einen einzigen statischen Anteil einer Klasse während der verteilten Ausführung. Es wird so transformiert, dass Zugriffe auf den statischen Anteil über Rechnergrenzen hinweg erfolgen können.

Zur Migration und Replikation von Objekten während der Programmausführung müssen einige technische Vorbereitungen während der Transformationsphase erledigt werden. Die Klassen, deren Objekte migriert bzw. repliziert werden können, müssen entsprechend transformiert werden. Die Ent-

scheidung dafür, welche Klassen dazu zu transformieren sind, wird vom Verteilungsplan vor der Laufzeit getroffen. Außerdem basieren einige Entscheidungen zur Migration bzw. Replikation auf Laufzeitdaten, deshalb werden an passenden Stellen Instrumentierungscode eintransformiert.

Damit das transformierte Programm hinterher verteilt ausgeführt werden kann, wird nicht nur die Anwendung transformiert, sondern auch die innerhalb dieser Anwendung potentiell benutzten Klassen aus Bibliotheken einschließlich der Java-Standard-Bibliothek. Diese Klassen bilden zusammen mit der Anwendung eine so genannte transitive Hülle. Sie wird auf einen minimalen Umfang so beschränkt, dass nur die erforderlichen Komponenten zur Ausführung des Programms darin enthalten sind. Die gesamte Transformation des Eingabeprogramms inklusive der verwendeten Bibliotheken soll aus der Sicht des Benutzers völlig transparent ablaufen.

Da die Transformation und die verteilte Laufzeitumgebung eng aufeinander abgestimmt sind, müssen Entwurfsentscheidungen für beide gleichzeitig getroffen werden. Details über Entwurfskriterien des Transformators und der verteilten Laufzeitumgebung finden sich im Kapitel 7.

3.2.4. Programmanalyse

Bei nebenläufigen Java-Programmen ergeben sich mehr Anforderungen an die Programmanalyse als bei sequentiellen Programmen. Das Problem besteht darin, dass die Aktivitäten im Programm konzeptionell aus verschiedenen Threads auf unterschiedlichen JVMs bestehen. Der Kontrollflussgraph einer Aktivität erstreckt sich somit über mehrere JVMs und enthält mehrere lokale CF-Graphen der Methoden, die von diesen verschiedenen Threads ausgeführt werden. Eine Datenflussanalyse für solche Aktivitäten muss den gesamten CFG einer Aktivität in Betracht ziehen. Außerdem ist der Kontrollfluss bei den Wechselwirkungen zwischen verschiedenen Aktivitäten nicht mehr so beschränkt wie im sequentiellen Fall. Während im sequentiellen Fall allenfalls durch bedingte Verzweigungen oder die Auflösung dynamischer Methodenbindung eine Auswahl an möglichen Nachfolgern einer bestimmten Instruktion entsteht, kommt im mehrsträngigen Fall jede Instruktion einer nebenläufigen Aktivität als Nachfolger in Frage. Eine einfache Übertragung normaler Analysen würde inkorrekte Informationen liefern. Deshalb muss die Programmanalyse in diesem Kontext in der Lage sein, die möglichen Interaktionen zwischen den Aktivitäten und Objekten einzuschrän-

ken. Es gibt verschiedene Ansätze zur statischen Programmanalyse in diesem Umfeld, z.B. die Objektgraphanalyse, Untersuchung der Wechselwirkungen zwischen den Aktivitäten, Erweiterung der Kontrollflussanalyse oder die Seiteneffekt- und Referenzzielanalyse usw. All diese Verfahren können eingesetzt werden, um gezielt Programmeigenschaften zwecks Verteilung zu berechnen. Auf Einzelheiten werde ich im Kapitel 6 eingehen.

Die Analyseergebnisse werden, wie in Abbildung 3.2 gezeigt wird, in den Verteilungsstrategien verwendet, nämlich in den Objektverteilungsfunktionen. Jede ODF ist charakterisiert durch bestimmte Programmeigenschaften, die sich durch Einsatz passender Analyseverfahren automatisch berechnen lassen. Beispielsweise kann die Proxy-Eigenschaft durch ein statisches Analyseverfahren, die Escape-Analyse für mehrsträngige Java-Programme, bestimmt werden. So kann die Funktion ODF_{Proxy} ihre Entscheidung für Objekte treffen, die nur lokal innerhalb anderer Objekte benutzt werden und deshalb keine entfernten Zugriffe braucht. Sie werden also nicht verteilt sondern nur dort erzeugt, wo die umgebenden Objekte platziert werden.

Zusammenfassung

Zur Berechnung der Programmeigenschaften haben die Programmanalysen relevante Informationen zu liefern. Die Güte der Verteilungsentscheidung hängt sehr stark von den berechneten Daten ab. Mächtige und dafür aufwändige statische Analyseverfahren können hierbei eingesetzt werden, da sie vor der eigentlichen Durchführung des Programms angewendet werden und ihre Ergebnisse unabhängig von einzelnen Programmabläufen sind. Nur statisch berechnete Ergebnisse allein reichen aber nicht für eine gute Verteilung der Objekte aus, weil sich nicht nur der Kontrollfluss eines Programms und dessen Eingabedaten, sondern auch andere Faktoren wie Rechnerlast oder Kommunikationskosten zur Laufzeit ändern können. Deshalb kann eine dynamische Analyse fehlende Information, bespielsweise über bestimmten Häufigkeiten, dazu ergänzen. Reales Laufzeitverhalten und wertvolle Laufzeitdaten lassen sich bei der statischen Analyse nicht herausfinden oder höchstens nur approximieren. Daten, die mit statischen Ergebnissen zu kombinieren sind, bilden die wesentliche Grundlage für die zu treffende Verteilungsentscheidung für zukünftig erzeugte Objekte und für die Replikation und Migration von Objekten.

Auf dieser Grundlage entsteht das Gesamtkonzept zur automatischen Verteilung nebenläufiger Java-Programme. Die eingesetzen Analysetechniken zu diesem Ansatz werden genau im Kapitel 6 beschrieben.

4. Verwandte Arbeiten

Inhalt

In diesem Kapitel werden Systeme und Bibliotheken beschrieben, die derzeit existieren und zur Verteilung von Java-Programmen eingesetzt werden. Die Systeme und Bibliotheken werden nach folgenden Aspekten kategorisiert: Änderung an Sprache und/oder Programmsemantik; getroffene Entscheidungen zur Verteilung basiert auf statischen oder dynamischen Ansätzen; High-Performance Java mit regulären Ansätzen. Die Systeme werden mit *JScatter* auf konzeptioneller Ebene verglichen. Im Verlauf der Arbeit, vor allem in den Kapiteln 5, 7 und 8 wird dann auf die hier beschriebenen Charakteristiken zurückgegriffen, um detaillierte Vergleiche anzustellen. Anschließend werden die Kommunikationsmodelle und -techniken, die meist als untere Schicht der Systeme und Bibliotheken verwendet werden, kurz angegeben. Abschließend gibt eine Tabelle einen Überblick über alle wesentlichen Merkmale der hier beschriebenen Systeme und Bibliotheken.

Verwandte Arbeiten zu Analysetechniken, die zur Berechnung der Programmeigenschaften zur automatischen Verteilung eingesetzt werden, werden im Kapitel 6 extra beschrieben.

4.1. Existierende Systeme, Frameworks und Bibliotheken

Die grundlegenden Arbeiten zu diesem Thema entstanden ab 1997, als Philippsen das JavaParty-System (Abschnitt 4.1.1) der Welt bekannt machte. Seitdem verbreitet sich dieses Thema immer weiter und es existieren derzeit viele Systeme bzw. Bibliotheken aus der ganzen Welt, die sich mit dem Ansatz „automatische Verteilung von Java-Programmen“ beschäftigen. Man bemüht sich ständig, Systeme bzw. Werkzeuge zu schaffen, die dem Programmierer die Verteilung mehrsträngig formulierter Java-Programme abnehmen, so dass sie keine speziellen Verteilungskenntnisse brauchen. Bei allen bekannten Systemen oder Bibliotheken muss der Programmierer bisher Verteilungsdirektiven selbst angeben. Der Programmcode wird dann entsprechend transformiert und mit Hilfe von Werkzeugen oder eigenen Laufzeitumgebungen verteilt ausgeführt. Es wird dabei berücksichtigt, dass das transformierte Endprodukt semantikerhaltend bleibt.

Zunächst wird erklärt, was man unter Änderung der Sprache und der Semantik genau bei der Verteilung von Java-Programmen versteht und was statisch oder dynamisch basierte Verteilungsentscheidungen bei den Systemen bedeuten.

Änderung an der Sprache Es werden neue Sprachkonstrukte in die Java Sprache eingefügt, damit der Programmierer Verteilungsdirektiven explizit angeben kann. Dafür wird ein eigener Compiler benötigt, der diese Erweiterung erkennt und das Programm dementsprechend so transformiert, dass es nach diesen Verteilungshinweisen korrekt verteilt ausgeführt wird. Vertreter hierfür sind das JavaParty-System und ein System namens *Remote Objects in Java* (siehe Abschnitt 4.1.1).

Änderung an der Semantik Verteilungsdirektiven werden meist vom Programmierer explizit in Form von Bibliothek-Methodenaufrufen (ProActive im Abschnitt 4.1.2), spezielle Tags/Kommentare (Doorastha im Abschnitt 4.1.2), Graphpartition auf grafischer Darstellung (Pangaea im Abschnitt 4.1.2) oder direkt in Quellcode durch Spracherweiterung (`remote` bei JavaParty) angegeben. Das ursprüngliche Programm wird entsprechend so transformiert, dass Fernzugriffe auf gekennzeichnete Objekte möglich sind, z.B. Einbindung von RMI-Code, Generierung von Stubs, Skeletons usw. Das Originalprogramm bleibt nach der Transformation bzw. Übersetzung nicht mehr dasselbe, sondern es entstehen daraus eventuell mehrere Programme. Diese können dann in einer geeigneten Laufzeitumgebung ausgeführt werden. Wenn beobachtbare Effekte geändert werden, so dass sie nicht mehr den Beschreibungen der ursprünglichen Sprachdefinition genügen, dann wird das Ursprungprogramm semantisch geändert. Das folgende Beispiel soll dies verdeutlichen: Im Originalcode sollen zwei Objektreferenzen miteinander verglichen werden (entspricht dem Operator `==`). Durch eine Verteilungsstrategie gelangen die beiden referenzierten Objekte auf unterschiedliche Rechner. Nach der Transformation entspricht das Vergleichergebnis nicht dem ursprünglichen Fall. Dann entspricht dies nicht mehr der Semantik des Programms.

Statisch basierte Verteilungsentscheidung Vor der Laufzeit werden statische Analysen eingesetzt, um Programmeigenschaften zu berechnen, die relevant für die Transformation und Verteilung sind. Aus den Analyseergebnissen werden Verteilungsentscheidungen getroffen, die aber dann fest in das Programm integriert werden und zur Ausführungszeit unverändert bleiben. Die meisten Systeme benutzen diesen Ansatz und wenden Analysen während oder nach der Übersetzung des Programms, aber vor der ersten Ausführung an.

Dynamisch basierte Verteilungsentscheidung Im Gegensatz zum statisch basierten Ansatz wird das Programm vor der Laufzeit nicht analysiert, sondern es wird an bestimmten Programmstellen Instrumentierungscode eingefügt. Zur Laufzeit werden an solchen Stellen Laufzeitdaten nach bestimmten Kriterien gesammelt, beispielsweise die Anzahl der Zugriffe zwischen Objekten. Bei vielen Ansätzen wird eine spezielle JVM verwendet, die in der Lage ist, die Last durch Zugriffe auf Objekte und die aktuelle Last der verwendeten Rechner zu berechnen. Diese Daten dienen dann als Grundlage für die Platzierungsentscheidung der Objekte (Juggle-System in 4.1.4, cJVM [AFT99], [AFT+00]).

Nachfolgenden möchte ich basierend auf den obigen geschilderten Kategorien die einzelnen Systeme und Bibliotheken beschreiben.

4.1.1. Systeme mit Sprachänderung

JavaParty

Dies ist eins der ersten Systeme zur automatischen Verteilung von Java-Programmen. Hier wurde Java um einen neuen, die Klassendeklaration ergänzenden, Klassenmodifikator `remote` erweitert. Der Programmierer kennzeichnet damit potentiell entfernt zu realisierende Objekte für den JavaParty-Übersetzer. Der Übersetzer und das verteilte Laufzeitsystem übernehmen die Transformation und Verteilung des Quellprogramms und liefern Kommunikations- und Synchronisationsprimitive für den Zugriff auf verteilte Objekte. Eine erweiterte Typanalyse wurde hierbei zur Übersetzungszeit verwendet, um eine möglichst genaue statische Approximation an den interprozeduralen Kontrollfluss des Programms zu erzielen. Durch Untersuchung des Aufrufgraphen der vorkommenden Aktivitäten im Programm ergibt sich eine Zuordnung von Objekten zu den Aktivitäten. Diese Zuordnung beruht auf Schätzungen von Aufrufhäufigkeiten, Komplexitäten von Methoden und Strukturgrößen. Ein wesentliches Merkmal des verwendeten Konzepts ist die Aufteilung von polymorpher Verwendung der Klassen und Methoden. Somit werden auch die Daten, die in den Klassen gekapselt sind und in Form von gespeicherten Objekten von unterschiedlichen Typen in Instanzvariablen vorliegen, und Methodenparameter in monomorphe Teile aufgeteilt. Diese werden als Konturen oder Schablonen bezeichnet. Dadurch werden die verwendeten Typen eindeutig und das erleichtert die Verteilungsanalyse zur Optimierung der Platzierung von Objekten ([PZ97], [HRP], [BCP96], [PH00]).

Konzeptionell werden aus einer *remote* Klasse zehn Klassendateien generiert, die den statischen Instanz-Anteil und eine lokale Version mit zugehörigen Schnittstellen darstellen. Dadurch existiert eine neue, generierte Klassenhierarchie neben der ursprünglichen. Das Konzept *Objekt-Handles* [Hau98] wurde bei JavaParty benutzt, um u.a. Zugriffprobleme in der neuen Klassenhierarchie zu lösen.

Basierend auf diesen Kenntnissen gibt es im *JScatter*-System auch eine Aufteilung einer Klasse, deren Objekte entfernt zugreifbar sind, in sechs Teile: Instanzenanteil (Cuckoo-Klasse), Instanzenverwalter, statischer Anteil und das Handle plus zwei zugehörige Schnittstellen. Mehr dazu wird im Abschnitt 7 beschrieben. Dadurch haben wir wesentlich weniger Teile zu generieren.

Als Kommunikationstechnik wurde Sun-RMI oder die hauseigene RMI (KaRMI [HP99], im Abschnitt 4.2.2) im JavaParty-Laufzeitsystem verwendet. Diese optimierte RMI-Version ist um einen Faktor 12 schneller gegenüber Sun-RMI. Trotz der neuen entstehenden Klassenhierarchie und der neu generierten Teile bleibt aber das Verhalten der verteilten Ausführung des Programms unverändert. Der Übersetzer beim JavaParty-System sorgt während der Transformation dafür, dass die Semantik des Programms erhalten bleibt.

Remote Objects in Java

Nataraj Nagaratnam und Arvind Srinivasan haben ein verteiltes Java-System implementiert, genannte *Remote Objects in Java* [NS96]. Sie erweiterten die Sprache Java um ein weiteres Schlüsselwort `remotenew`. Dies ist anstelle von `new` zu benutzen, um ein Objekt auf einer entfernten virtuellen Maschine zu erzeugen. Entfernte Methoden können aber mit der üblichen Java-Syntax aufgerufen werden. Sie werden wie beim Java-RMI auf der entfernten Maschine ausgeführt[1]. Objekt-Migration wird nicht unterstützt, d.h. ein *remote object* ist fest an die Maschine gebunden, auf der es instantiiert wurde. Es besitzt zwei Identitäten, eine auf der lokalen Maschine und eine weitere auf der Heimatmaschine, aus der der Erzeugungscode stammt. Eine Abbildungstabelle der lokalen Identität auf die entfernte Identität befindet sich auf der lokalen Maschine. Über diese kann entschieden werden, ob ein Objekt entfernt oder lokal ist. Es gibt zu einer Klasse sowohl lokale als auch entfernte Objekte.

Unser Ansatz ist zu *Remote Objects in Java* in diesem Punkt ähnlich, d.h. sowohl entfernte als auch lokale Objekte können aus einer Klasse stammen. Das

[1] Sun-RMI wird nicht bei *Remote Objects in Java* eingesetzt.

stellt in diesem System aber ein großes Problem dar, das erst zur Laufzeit bekannt ist: Abhängig davon, ob ein Objekt *remote* oder lokal ist, kann dann erst entschieden werden, ob ein Methodenaufruf mit Parametern vom Referenztyp bezüglich eines Objekts überhaupt möglich ist. Dieses Problem haben wir elegant durch den Delegationsmechanismus mittels Objekt-Handles (siehe Kapitel 7) gelöst. Im Falle eines existierenden lokalen Objekts werden lediglich lokale Aufrufe ausgeführt und nicht über den umständlichen RMI-Mechanismus.

Charakteristisch für *Remote Objects in Java* ist auch das Prinzip, dass der Bytecode sämtlicher Klassen anfangs lediglich auf der lokalen virtuellen Maschine bekannt ist. Eine entfernte virtuelle Maschine wartet nach der Initialisierung lediglich auf Anfragen anderer Maschinen. Zur entfernten Ausführung eines Konstruktors bzw. einer Methode muss dann stets der betreffende Bytecode mitgeschickt werden. Dies bringt aber erhebliche Probleme mit sich, da eine gesendete Methode Parameter zu lokalen Variablen auf der einen Maschine enthalten kann und diese Variablen auf der anderen Maschine nicht existieren oder andere Werte besitzen. Außerdem entsteht durch das Hin- und Zurückschicken sehr viel Kommunikationsaufwand.

4.1.2. Systeme und Bibliotheken mit Semantik-Änderung

Pangaea

Pangaea ist ein System zur (halb)automatischen Verteilung von Java-Programmen. Zunächst wird der Quelltext analysiert und ein sogenannter Objektgraph wird dann aufgebaut. Ein Objektgraph ist eine Annäherung der Laufzeitstruktur des Programms und beschreibt, welche Objekte es zur Laufzeit geben wird und wie sie miteinander kommunizieren [Spi99b]. Mit Vorgaben und Rahmenbedingungen vom Programmierer und in Abhängigkeit von der zu verwendeten Verteilungsplattform entscheidet das System über die Verteilung des Programms durch Objektgraphanalyse. Die Festlegung durch den Programmierer geschieht in visueller Form, indem er in einer grafischen Darstellung des Objektgraphen die Objekte fest bestimmten Rechnern zuordnet. Abhängig von der unterstützten Verteilungsplattformen, hier CORBA, JavaParty und Sun-RMI wird vom Pangaea entsprechender Quellcode dafür generiert. Wenn diese Plattformen Objektmigration oder -replikation unterstützen, erzeugt Pangaea dann auch Code dafür [Spi99c], [BLS97], [Spi99a]. Das System ist vor allem geeignet für Client-Server-artige Anwendungen.

Bei Pangaea werden beobachtbare Effekte des auszuführenden Programms verändert. Es gibt auch Einschränkungen des Systems, dass die Objektidentitäten nicht im Typgraphen gespeichert werden. Diese Information ist vor allem wichtig für Vergleiche von Objektreferenzen oder für Aufruf dynamisch gebundener Methoden. Im Allgemeinen bietet die Objektgraphanalyse Möglichkeiten, viel statische Eigenschaften des Programms vor der Laufzeit berechnen zu können. Daher benutzen wir sie auch, um Objektbeziehungen im Programm für Kommunikationsabschätzungen zu bestimmen.

Doorastha

Hierbei handelt es sich um ein System zur Verteilung von Java-Programmen. Es erlaubt entfernte Zugriffe, die dem Programmierer transparent bleiben. Das System bietet zwei verschiedene Möglichkeiten für Fernzugriffe auf Objekte und eine Mischform aus beiden. Durch das Hinzufügen von Tags, die gesonderte Kommentare für Fernzugriffe oder Verteilungsdirektive kennzeichnen, wird ein normales Java-Programm in ein verteiltes transformiert. Die Vererbungshierarchie bleibt dabei erhalten. Wie JavaParty baut es auf RMI auf und benutzt nur den Serialisierungsmechanismus. Zur Notation von Verteilungsanweisungen gibt es spezielle Tags (`remotenew` und Rechnername), die direkt an der Erzeugungsstelle nach `new` als gesonderte Kommentare einzufügen sind [Dah00], [Dahb].

Besonders zu erwähnen ist im Doorastha-System das Konzept *Proxy* und *Wrapper*. Die *Wrappers* nehmen die entfernten Zugriffe entgegen und leiten sie weiter an das lokale Objekt. Die *Proxies* sind Bevollmächtigte auf der entfernten Seite. Durch diesen Mechanismus werden entfernte Zugriffe über Referenzen einfach weiter delegiert. Über das Netzwerk werden dann nur solche Wrappers verschickt, nicht die tatsächlichen Objekte. Während der Transformationsphase wurde auch berücksichtigt, dass trotz Delegation die Semantik der Referenzen und des Programms auch erhalten bleibt. Diesen Vorteil haben wir ausgenutzt und vereinigten *Proxy* und *Wrapper* in einem einzigen Objekt-Handle. Im *JScatter*-System leiten die Handles auf der Client-Seite die Aufrufe an Instanzmanager weiter. Diese verwalten alle Objekte einer Klasse auf einem Rechner und delegieren die Aufrufe auch an sie (siehe Kapitel 7). Außerdem unterstützt das Doorastha-System Objektmigration und -replikation,

welche bei *JScatter* nach dem Konzept in der Diplomarbeit von Weißenborn [Wei03] realisiert wurden.

ProActive

ProActive ist eine Java-Bibliothek, mit der ein Java-Programm in ein verteiltes umgewandelt werden kann. Ein wesentliches Merkmal von ProActive sind *aktive Objekte*, die als Verteilungseinheiten fungieren. Die Objekte des Programms werden dann in sogenannte Subsysteme strukturiert. Ein solches System besteht aus einem aktiven Objekt, das die Kommunikation mit anderen Subsystemen übernimmt, und einer beliebigen Anzahl von normalen (passiven) Objekten. Methoden solcher passiver Objekte werden nur innerhalb ihres Subsystems ausgeführt. Die Subsysteme sind voneinander disjunkt. Um die zu erzeugenden Objekte als aktive Objekte zu kennzeichnen oder passive Objekte in aktive umzuschalten, muss der Programmierer auf Quellcode-Ebene Methodenaufrufe der ProActive-Bibliothek an den zugehörigen Erzeugungsstellen einfügen. Das sind auch die Verteilungshinweise für solche Objekte.

Erwähnenswert bei ProActive ist die Realisierung von asynchronen Methodenaufrufen zwischen den Objekten durch sogenannte *Future-Objekte*. Der Aufrufer kann seine Aufgaben weiter ausführen, während die „Antwort“ abgearbeitet wird. Als weitere Eigenschaften bietet das System Objektmigration, die Wiederverwendung durch erhaltene Polymorphie zwischen aktiven und passiven Objekten und die Kompatibilität mit grafischen Komponenten wie Swing und AWT. Wie die meisten anderen Systeme basiert ProActive auf der Standard Kommunikationstechnik RMI von Sun ([Ant], [CKV98], [ACG00]).

Durch die Umwandlung in aktive Objekte sind Java-Befehle nicht mehr erlaubt, wie zum Beispiel Referenzvergleiche von aktiven Objekten.

Do!

Do! ist ein Framework von Institute Irisa. Es bietet ein paralleles Programmiermodell für mehrsträngige Java-Programme. Ähnlich wie ProActive hat der Benutzer Methodenaufrufe aus der Bibliothek an Erzeugungsstellen einzusetzen, um die zu erzeugenden Objekte als *remote* zu kennzeichnen. Methoden solcher Objekte sind dann entfernt aufrufbar. Zur parallelen Programmierung von Programmen mit strukturierten Threads und Verteilung der Objekte

bietet Do! so genannte „verteilte Collections“. Threads innerhalb einer solchen *Collection* können parallel ausgeführt werden. Dafür ist ein Sprachkonstrukt PAR anzuwenden. Der Preprozessor transformiert das Programm dann in eine verteilbare Version mit eingebundenem Code aus dem Framework. Zu verteilende (aktive) Objekte müssen in solche *Collections* hinzugefügt werden. Für die Verteilung der Elemente in einer „Collection“ ist ein *LAYOUT_MANAGER* im Do!-System zuständig. Damit kann der Programmierer seine Verteilungsstrategie beschreiben oder die vorhandene Blockzyklus- und Zyklus-Verteilung benutzen. Ähnlich wie bei ProActive wird die Semantik der Referenzen über aktive Objekte verändert ([LP98], [LP97]).

4.1.3. Systeme mit statisch basierter Verteilungsentscheidung

In den vorgestellen Systemen und Bibliotheken hat der Programmierer Transformations- oder Verteilungshinweise explizit anzugeben. Zum größten Teil werden hierbei Ergebnisse statischer Programmanalysen benutzt.

In JavaParty wurde die Standard-Typinferenz [PS91] erweitert [PH00], um die polymorphe Verwendung der Klassen und Methoden sowie zugehörige Klassendaten und Parameter in monomorphe Teile zu überführen. Dies ist notwendig, um eine vom Programmierer als *remote* gekennzeichnete Klasse zu transformieren. Basiert auf der Typinformation wird die Verteilungsanalyse anschließend verwendet, um Schätzungen von Aufrufhäufigkeiten, Komplexitäten von Methoden und Strukturgrößen zu machen. Dadurch ergibt sich auch die Zuordnung der Objekte zu Aktivitäten, die für die Verteilung benutzt wird. Diese Analyseverfahren können im allgemeinen die Unschärfe in Kontrollflüssen nicht auflösen und scheitern vor allem bei Problemen mit dynamischer Zuordnung der Aktivitäten, wie z.B. beim Arbeitsvorrat-Paradigma.

Beim Pangaea-System wird das Programm in die Objektgraph-Darstellung umgeformt. Diese Struktur hat den Vorteil, dass die Laufzeitstruktur des Programms und die Beziehungen zwischen potentiellen Laufzeitobjekten hierdurch repräsentiert werden können. Der Objektgraph kann aber die Information über Objektidentität nicht repräsentieren, was nachteilig für eine Verteilung ist.

Mittels der Ergebnisse der Objektgraph-Analyse erhält der Anwender einen Vorschlag über eine Graphpartitionierung des Objektgraphen. Die Verteilungsentscheidung muss aber schließlich vom Programmierer selber getrof-

fen werden. Die Analyseergebnisse werden auch zur Transformation des Programms benutzt.

Die Technik der Programmanalyse wird in Doorastha benutzt, um die von Benutzern angegebenen Anweisungen in Form von speziellen Tags in Java-Kommentaren zu bewerten und das Programm so zu transformieren, dass Objekte über Rechnergrenzen hinweg angesprochen werden können. So ähnlich ist es auch bei ProActive und Do!-Framework, wobei Verteilungsanweisungen explizit vom Programmierer anzugeben sind. Bei ProActive haben die Entwickler versucht, aktive Objekte mittels statischer Analyse (Erweiterung der interprozeduralen Shape-Analyse von Sagiv et al. [SRW98]) bei Programmen mit baumartig strukturierten Prozessen automatisch zu erkennen. Sie haben Kriterien für die Erkennung aufgestellt und dann mittels Analyse überprüft. Die Entwickler haben sich hierbei aber auf eine kleine Teilmenge der Java-Sprache eingeschränkt: Keine Threads, keine statischen Anteile der Klassen und keine dynamische Methodenbindung dürfen vorkommen.

Bei meinem Konzept beruhen Verteilungsentscheidung auf statischen und dynamischen Programmeigenschaften, die innerhalb spezieller Objektverteilungsfunktionen (ODFs) zur Transformations- und Platzierungsberechnung verwendet werden und mittels Programmanalysen automatisch bestimmt werden. Statische Eigenschaften werden vollständig vor der Laufzeit durch verschiedene Analysetechniken berechnet, dynamische Faktoren werden ebenfalls vor der Laufzeit abgeschätzt. Das Modell zur Abschätzung lehnt sich stark an das von JavaParty an. Anfragen an Verteilungsplan werden automatisch in das Programm eintransformiert, so dass die Verteilungshinweise zur Laufzeit vor jeder Erzeugung abgefragt werden können und das Objekt entsprechend danach platziert wird.

4.1.4. Systeme mit dynamisch basiertem Ansatz

Bei diesem Ansatz beruhen die Verteilungsentscheidungen vollständig auf den sich zur Laufzeit ergebenen Daten. Diese Daten werden von einer systemeigenen, verteilten JVM gesammelt. Diese JVM protokolliert die Zugriffe der Objekte zur Laufzeit und verteilt die Objekte nach einem eigenen Lastmodell. Zur Platzierung der Objekte kommt ebenfalls die Technik zur Lastbalancierung zum Einsatz. Der Vorteil an diesem Ansatz ist, dass echte Daten über Objekte, Threads, Kommunikationen usw. zur Verfügung stehen und man diese nicht abzuschätzen braucht. Der Nachteil ist, dass das Sammeln

der Daten und das Treffen der Entscheidungen zur tatsächlichen Laufzeit des Programms dazugezählt werden müssen. Ein Vertreter für diese Kategorie ist das Juggle-System.

Juggle

Das Juggle-System implementiert eine verteilte virtuelle Maschine, die transparent Objekte und Threads nach einer automatischen Lastverteilung auf die beteiligten Rechner verteilt. Die virtuelle Maschine bei Juggle wurde so instrumentiert, dass während des Programmablaufs ständig günstige Positionen für Objekte und Threads errechnet werden. Die neu errechneten Konfigurationen werden über Migrationen und Replikationen umgesetzt [SH98].

Juggle besitzt einige interessante Eigenschaften, die zu erwähnen sind. Jedes Objekt im System ist für jeden beteiligten Rechner ansprechbar. Nur explizit zugegriffene Daten, z.B. nach Zugriff auf eine Instanzvariable eines Objekts, werden zum Zugreifenden migriert. Im Juggle werden nur einmal modifizierte Objekte überall auf allen Knoten repliziert und die Replikate werden nach weiteren Zugriffen entsprechend aktualisiert. Methodenaufrufe finden nicht entfernt statt, sondern der Programmcode wird auf allen Knoten repliziert und dann dort lokal ausgeführt. Dies ist unabhängig davon, ob die Objektdaten lokal liegen oder nicht. Dadurch ist die Position der Threads unabhängig von Code und Daten. Juggle verteilt die Objekte dynamisch und automatisch nach dem Lastmodell, bei dem die Anzahl der Zugriffe aller Threads auf ein Objekt in einer bestimmter Zeitspanne gezählt wird. Die Auslastung der Knoten wird periodisch überprüft und ggf. unter Berücksichtigung der entfernten Objektzugriffe korrigiert. Mit diesem vollständigen dynamischen Ansatz konnte Juggle ein Speedup von 2,14 bis 4 bei der Anwendung „Raytracer - Ball“ erzielen.

Die Objektreplikation wird im *JScatter*-System nach zwei unterschiedlichen Konzepten realisiert: Objekte mit *stagewise-immutable* Eigenschaft werden repliziert. Nach der Replikation dürfen dann nur Lesezugriffe stattfinden. Nach dem in der Diplomarbeit von Weißenborn [Wei03] beschriebenen Konzept werden auch Objekte repliziert, die hinterher eventuell modifiziert werden. Durch nachträgliche Schreibzugriffe werden die Replikate nach der dort entworfenen Strategie aktualisiert, was dem in Juggle realisierten Konzept entspricht.

4.1.5. Andere Systeme

Hyperion

Etwas entfernt mit *JScatter* verwandt ist das System Hyperion [A+01], [ABH+a]. Es enthält einen Transformator, der Java-Bytecode in C transformiert, und eine Laufzeit-Bibliothek zur verteilten Ausführung von Java Threads. Ein mehrsträngig formuliertes Java-Programm in Bytecode wird hierbei zunächst vom Transformator in C-Code übersetzt. Das transformierte Programm enthält Bibliothek-Funktionen und wird ggf. gegen externe Bibliotheken gelinkt. Das Programm kann dann auf Cluster von PCs oder Workstations ausgeführt werden. Das ganze System wurde auf eine verteilte, mehrsträngige Umgebung namens PM2 [ABH+b] aufgebaut. Mittels dieser Umgebung können Prozesse lokal und entfernt erzeugt werden und sie kommunizieren miteinander über *RPC*. Die oberste Schicht von PM2 ist die Plattform DSM-PM2 [AB], [AB01], die mehrsträngige Protokolle in DSM (*Distributed Shared Memory*) unterstützt. Auf diese Schicht werden das Java-Speichermodell sowie die meisten von Hyperion unterstützten Primitive wie Threads, Kommunikation *shared memory* Subsysteme auf Funktionen von PM2 direkt abgebildet. Entfernte Zugriffe auf Objekte werden mittels zweier Protokolle entdeckt: Das eine prüft auf explizite Lokalität der Objekte und das andere basiert auf einer Technik namens *page faults*, die die Zugriffe auf entfernte Objekte überprüft [AH01].

Ähnliche Ansätze mit *verteilten JVMs* finden sich in vielen verschiedenen Projekten wie Java/DSM [YC97], cJVM [AFT99], [AFT+00], Jackal [VHBB01] und JESSICA [MWL00]. Die verteilte JVM wird bei diesen Systemen als ein einziges Systemabbild (SSI - *Single System Image*) modelliert. Hierbei wird das Konzept GOS (*Global Object Space*) benutzt, um nur ein einziges Java Objekt-Heap zu modellieren, das allen beteiligten verteilten Knoten des Clusters zur Verfügung steht. Dadurch bleiben die Objektzugriffe über Rechnergrenzen hinweg in einer solchen verteilten Umgebung transparent. Durch das neue Objektmodell werden *remote* und lokale Objekte voneinander unterschieden, was auch als Grundlage zur Verteilung verwendet wird. Außerdem wird zur Laufzeit die Technik der Lastbalancierung eingesetzt, nach der die Objekte und Threads verteilt werden.

Zur Verteilungsentscheidung während der Programmausführung wird bei meinem Konzept die hybride Analyse benutzt. Hierbei hat man gegenüber nur auf dynamischen Daten basierter Entscheidung den Vorteil, dass stati-

sche Eigenschaften des Programms, insbesondere interprozedurale Informationen, schon zur Verfügung stehen. Die Beschaffung solcher Informationen hat keinen Einfluss auf die Laufzeit des Programms. Dynamische Analyse wird dann nur gezielt eingesetzt, um die noch fehlenden Daten zu ermitteln. Dieser Ansatz und der Einsatz der Lastbalancierungstechnik zur Verteilung sind prinzipiell zwei verschiedene Konzepte. Sie beschaffen die Information zur Platzierungsentscheidung aber auf gleiche Weise. Außerdem besteht die verteilte Laufzeitumgebung des *JScatter*-Systems aus den Standard-JVMs und es wird die Kommunikationstechnik KaRMI verwendet.

J-Orchestra

J-Orchestra [TS01], [StJOG01], [TS02] ist ein System, das monolithische Java-Programme automatisch in eine verteilbare Version partitioniert. Die Transformation findet auf der Bytecode-Ebene statt und die lokalen Objektreferenzen im Programm werden dabei durch entfernte Referenzen mittels der RMI-Technik ersetzt. Das transformierte Programm kann dann verteilt auf mehreren Standard-JVMs ausgeführt werden. Dieses Konzept ist vergleichbar mit dem des *JScatter*-Systems, wobei hier ein anderes Transformationskonzept verwendet wird.

4.1.6. Systeme für High-Performance Java - Datenparallel

Obwohl das Thema High-Performance Java nur entfernt verwandt mit meinem Thema ist, möchte ich hier einen kleinen Ausflug in die Systeme für HP-Java unternehmen. HP-Java hat HP-Fortran als Vorbild und entwickelt sich mit der Beliebtheit von Java mit der Zeit immer weiter. Es ist vor allem für reguläre Anwendungen zum wissenschaftlichen Rechnen geeignet. Die hierbei verwendeten Techniken und Konzepte können aber auch nützlich für irreguläre Ansätze sein.

JavaParty

Philippsen erweiterte in [Phi98] Java um eine `forall`-Anweisung für die datenparallele Berechnung. Eine Source-to-Source-Transformation wurde eingesetzt, um `forall`-Konstrukte auf mehrsträngigen Java-Code abzubilden. Zu-

sätzlich wurde ein Optimierungsverfahren präsentiert, das die Anzahl der benötigten Barrierensynchronisationen verringert. Der entstehende nebenläufige Programmcode wird nicht explizit auf parallele Prozessoren verteilt, sondern bildet die Eingabe für das JavaParty-System. Die größte bekannte Anwendung ist die Umsetzung des *Large-Scale* parallelen geophysikalischen Algorithmus zur Erdkundung.

Manta

Manta [vNMB+99] ist ein System, das Java in ein hierarchisches Supercomputer-Netz (Myrinet-basierter Cluster) integriert. Kommunikationsplattform ist ein eigenes RMI, das aber auf Sun-RMI basiert. Manta erweitert wie JavaParty die Java-Sprachkonstrukte um das Schlüsselwort `remote`.

Orca

Orca [vNMB+99], [BBH+96], [BK93], [BBH+98] ist ein objektbasiertes *distributed shared memory* System. Orca selbst ist eine Sprache, die nicht klassenbasiert sondern objektbasiert ist. Das System enthält einen eigenen Compiler, ein Laufzeitsystem und eine virtuelle Maschine (Panda). Der Compiler generiert reguläre Ausdrücke, die die Verwendung von gemeinsamen Objekten beschreiben. Das Laufzeit-System verwendet diese Informationen zusammen mit Laufzeitdaten zur Verteilung und zur Steuerung der Replizierung.

HPJava

Dies ist eine Erweiterung von Java, um die Programmierung mit verteilten Arrays zu erleichtern (SPMD-Modell). Die Erweiterung enthält drei neue Konstrukte (*on*, *at* und *over*), die nützlich für datenparallele Schleifen sind. Die Kommunikationstechnik ist MPI [CZF+98].

Titanium

Titanium [YSP+98] ist eine Java-basierte Sprache und gleichzeitig auch ein System für High-Performance und paralleles wissenschaftliches Rechnen. Die

wesentlichen Erweiterungen sind *immutable* Klassen, multidimensionale Arrays für explizite SPMD-Berechnungen mit globalem Adressraum und so genanntes bereich-basiertes Speicher-Management.

4.2. Kommunikationsmodelle und -techniken

Als Grundlage für die Kommunikation auf unterster Ebene wird in den meisten Systemen und Bibliotheken das Modell *Remote Method Invocation* verwendet. Mit Hilfe dieser Techniken können Objekte ihre Nachrichten und Daten über das Netzwerk austauschen. Im Folgenden möchte ich einige wichtige Modelle mit ihren relevanten Aspekten beschreiben, die wir direkt in unserem System verwendet haben. Die anderen, die weniger relevant sind, sind in der Kategorie „andere Techniken“ im Abschnitt 4.2.5 aufgeführt.

4.2.1. Sun-RMI

RMI ist eine Client-Server-orientierte Verteilungstechnologie von Sun [Mic], [Mic98], [MPA99]. Instanzen von Klassen werden unter Verwendung des Serialisierungsmechanismus *by-copy* übertragen. Die Klassen, deren Objekte mit RMI zu verteilen sind, müssen das Interface `Remote` implementieren oder von der Klasse `UniCastRemoteObject` erben. Des Weiteren müssen für die zu verteilenden Klassen mit dem Java-Übersetzer `rmic` Stub-(Clientseite) und Skeleton-Objekte (Serverseite) erzeugt werden. Die Stubs übernehmen dann die Rolle der eigentlichen Objekte sowie den Transfer der Argumente und die Initialisierung des eigentlichen Methodenaufrufs. Diese Stellvertreterobjekte sehen bezüglich der entfernt aufrufbaren Methoden genauso aus wie die entsprechenden *remote* Objekte. Die Skeleton-Objekte empfangen die Parameter und rufen ausschließlich lokal die korrespondierenden Methoden auf.

Zum Übermitteln der Argumente und des Resultats bei einem entfernten Methodenaufruf verwendet das RMI-System die Technik der Objekt-Serialisierung [Dow98], [Mic]. Ein als serialisierbar gekennzeichnetes Objekt wird in einen Strom umgewandelt, geschrieben und daraus wieder gelesen. Es gibt in RMI einige Einschränkungen, die aber wichtige Voraussetzung für ein verteiltes Objektmodell darstellen. Sie wurden ausführlich in [HP99] ausgeschildert und in unserem System auch berücksichtigt.

4.2.2. KaRMI

Mit der Begründung, dass das Sun-RMI zu langsam für Hochleistungsanwendungen ist, haben die Karlsruher ein schnelleres RMI mit einer effizienten Serialisierung entwickelt. Durch die Aufschlüsselung der Kommunikationskosten mittels Sun-RMI wurde dort die Serialisierungsphase durch die schlanke Typcodierung, eine neue Variante der `reset()`-Methoden für die Kopiersemantik der Argumente und eine verbesserte Technik zur Pufferung verbessert. Dazu wurde die Struktur von KaRMI gegenüber Sun-RMI komplett neu entworfen, so dass das System neben Geschwindigkeitsverbesserungen auch die Nutzung mehrerer verschiedener Netzwerktechnologien anbietet [HP99].

Wir benutzen neben dem Standard Sun-RMI auch diese Kommunikationstechnik, um eine Geschwindigkeitssteigerung zu erzielen.

4.2.3. Distributed Threads

Eddy Tryen, Danny Weyns und Pierre Verbaeten haben ein Konzept zu *distributed Threads* [TRV+00], [WTV02a], [WTV02b] bei verteilter Ausführung mehrsträngiger Java-Programme entwickelt. Mit diesem Konzept können sie bei verteilten Anwendungen jedem *Thread* eine eindeutige Identität zuordnen und können somit die Semantik des verteilten Kontrollflusses gewährleisten. So kann das Problem unerwarteter Deadlocks gelöst werden. Außerdem kann das Monitor-Konzept wie im sequentiellen Fall einfach benutzt werden, ohne dessen Semantik zu verlieren. Durch die Technik der Bytecode-Transformation können mehrsträngige Anwendungen mit diesem Ansatz realisiert werden. Die Transformation wurde in einen eigenen Klassenlader integriert und so werden Programme in Bytecode erzeugt, die auf einer normalen JVM ablaufen können. Wir haben im System einen ähnlichen Mechanismus zur eindeutigen Erkennung der Aktivitäten, damit die Semantik des verteilten Programms erhalten bleibt. Außerdem verwenden wir auch das verteilte Monitor-Konzept in unserem System.

4.2.4. JavaSpaces

JavaSpaces [FHA99] ist ein spezieller Jini-Service von Sun Microsystems, Inc. Ein *Space* ist ein persistentes Objekt-Lager, das von verschiedenen Prozessen

gemeinsam benutzt wird und über das Netzwerk zugänglich ist. Die Kooperation der Prozesse geschieht dadurch, indem die Objekte einen oder mehrere Spaces betreten und wieder verlassen. Damit kann schwierige Synchronisation der Aktivitäten mit Hilfe des Spaces stattfinden. Die Prozesse können drei einfache Operationen ausführen: Mit `write` schreiben sie Objekte in ein Space, mit `take` werden Objekte aus dem Space herausgenommen und mit `read` lesen die Prozesse Objekte im Space. Solange sich ein Objekt im Space befindet, ist es passiv. Um ein Objekt zu verändern oder eine Methode auf ihm aufzurufen, muss ein Prozess das Objekt explizit aus dem Space entfernen, es eventuell verändern und danach wieder zurücklegen. Die Eigenschaften der Spaces leiten sich aus den Eigenschaften der Jini-Technologie her.

Das Programmiermodell bei JavaSpaces unterscheidet sich von den üblichen Techniken wie *Message Passing* und RMI. Über die Spaces koordinieren die *remote* Java-Prozesse ihre Aktivitäten und tauschen ihre Daten aus.

4.2.5. Andere Techniken

javab

Javab [BG98], [BG97] ist ein Werkzeugsystem zur Parallelisierung von Bytecode unter Verwendung von Bytecode-to-Bytecode Optimierung und Transformation. Durch Kontroll- und Datenfluss-Analyse wird die Parallelität in Schleifen automatisch erkannt.

mpiJava

Dies ist eine API zur allgemeinen Benutzung des *Message Passing Interface* (MPI). Die Funktionalität von MPI wird über JNI-Schnittstellen [BCFK] auf Java-API abgebildet. Diese Schnittstelle wurde unter dem Projekt HPJava [CZF+98] entwickelt, setzt aber keine spezielle Spracherweiterung wie HPJava voraus.

Corba

Corba [MZ96] ist das OMG Standard-Verteilungssystem. Die Schnittstelle von CORBA-Objekten wird durch IDL definiert. Die Objekte werden ebenfalls

mit *by-copy* übertragen. Entfernte Objekte kommunizieren miteinander zur Laufzeit über *Stubs*, die vom CORBA-eigenen Übersetzer erzeugt werden.

4.3. Vergleiche - Überblicke

In diesem Abschnitt werden nochmals die wesentlichen Merkmale der oben vorgestellten Systeme und Bibliotheken tabellarisch zusammengefasst. Hierbei werden nur diejenigen in Betracht gezogen, die direkt mit unserem Ansatz verwandt sind. Legende:

✔: vorhanden, –: nicht vorhanden, ?: konnte nicht in Referenzen gefunden werden

	JavaParty	*Remote Objects*	Pangaea	Doorastha	ProActive	Do!	Juggle	Hyperion
Sprach-änderung	✔ (`remote`)	✔ (`newremote`)	–	–	–	–	–	–
Semantik-erhaltend	✔	–	–	✔	–	–	?	–
Statische Entscheidung	✔	✔	✔	✔	✔	✔	–	✔
Dynamische Entscheidung	–	–	–	–	–	–	✔	–
Eigener Compiler	✔	✔	–	✔	–	Pre-prozessor	–	✔
Eigene Laufzeit-Umgebung	✔	✔	–	–	–	✔	✔	✔
Eigene Kommuni-kation	✔ (KaRMI)	✔ (nicht Sun-RMI)	–	–	–	–	–	✔
Migration	✔	–	plattform-abhängig	✔	✔	?	✔	?
Replikation	–	–	plattform-abhängig	✔	–	?	✔	?
Weitere Merkmale	polymorphe Verwendung in monomorphe Teile zerlegt	Bytecode wird geschickt und dort lokal ausgeführt	grafische Objektgraph-Darstellung zur Angabe der Verteilung	spezielle Tags als Kommentar für die Verteilung	Polymorphie zwischen aktiven und passiven Objekten	*Collection* und neue PAR-Konstrukte für aktive Objekte	Instrumen-tierung in eigener JVM	transfomiert Java-Programme in C-Code

Tabelle 4.1.: Alle Systeme und Bibliotheken - Zusammenfassung

Zusammenfassung

Im *JScatter*-System werden die Konzepte und vor allem die interessanten Merkmale anderer Systeme wie JavaParty, ProActive, Doorastha usw. verwendet. Das Programm wird automatisch in eine verteilte Version transformiert und die Objekte werden vollständig automatisch verteilt. Die Verteilungsentscheidung für Programmobjekte basiert auf statischen sowie dynamischen Programmeigenschaften, die mit Hilfe der statischen und dynamischen Programmanalysen berechnet werden. Hier werden auch verschiedene statische Analysetechniken verwendet, die in anderen Systemen eingesetzt wurden.

5. Verteilungsstrategien und Verteilungsplan: Konzept und Modellierung

Inhalt

Abgesehen von der aufwändigen technischen Umsetzung werden die wichtigen strategischen Schritte nicht mehr von einem Programmierer oder einem Entwickler durchgeführt, sondern durch das im Folgenden vorgestellte Verfahren automatisch erledigt. Der erste Schritt ist das Analysieren des Programms zur Sammlung von Informationen, die Hinweise auf die Verteilung der Programmobjekte liefern. Beruhend auf diesen Hinweisen werden verschiedene Verteilungsstrategien aufgestellt, die zur Entscheidung über die Verteilung der Objekte angewandt werden.

Konzeptionell werden die verschiedenen *Verteilungsstrategien* so entwickelt, dass sie für jede mehrsträngige Java-Anwendung angewandt werden können. Jede Strategie ist eine Zusammensetzung aus bestimmten Verteilungsvorschriften, die als *Objektverteilungsfunktionen* (ODFs) bezeichnet und in einer geeigneten Weise miteinander kombiniert werden. Die ODFs innerhalb einer Strategie berechnen ähnliche Eigenschaften des Programms, beispielsweise statische oder dynamische Eigenschaften. Je nach berechneten Analyseergebnissen liefern die ODFs Entscheidungen für die Programmtransformation an jeder Erzeugungsstelle beziehungsweise für die eigentliche Verteilung der zu erzeugenden Objekte zur Laufzeit. Durch strukturierte Zusammensetzung bestimmter Strategien entsteht ein Verteilungsplan. Abbildung 5.1 stellt die konzeptionellen Unterschiede dieser drei Ebenen (ODFs $\in$ Strategien $\in$ Verteilungsplan) dar.

Der Verteilungsplan ist in diesem Konzept die wichtigste Komponente, die alle strategischen Entscheidungen zur Transformation und zur Verteilung zu treffen hat. Er erledigt die strategischen Aufgaben eines Programmierers. Wie im Abschnitt 3.2 grob angedeutet spielt die Programmanalyse in diesem Konzept die entscheidende Rolle. Die Exaktheit der Analyseergebnisse trägt zur Güte der Verteilungsentscheidung bei.

Dieses Kapitel befasst sich mit den Konzepten und der Modellierung der einzelnen genannten Komponenten, die eine fundamentale Rolle spielen. Das gesamte Schichtenmodell, das als Grundlage für eine automatische Verteilung mehrsträngiger Java-Programme dient, wird hier vorgestellt. Einige zu erfüllende Grundvoraussetzungen und Anforderungen, zusammen mit dem Berechnungsmodell in einem Verteilungsplan werden ebenfalls im ersten Abschnitt dargestellt. Die Modellierung der Verteilungsvorschriften als Funktionen, deren Zusammensetzung in einer geeigneten Weise zu Verteilungsstrategien und das Modell eines Verteilungsplans werden anschließend beschrieben. Mit den erforderlichen Formalismen ist man dann in der Lage, grundlegende Verteilungsfunktionen zu formulieren. Aus diesen konkret definierten

Verteilungsvorschriften werden spezielle Strategien entwickelt, die bei der Planung der Verteilung zum Einsatz kommen. Je nach angewandten Strategien bilden sich somit verschiedene Pläne, die dazu dienen, das eingegebene Programm für die verteilte Ausführung zu transformieren und dessen Aktivitäten und Objekte zur Laufzeit automatisch zu verteilen. So befassen sich die letzten drei Abschnitte mit diesen Gesichtspunkten.

5.1. Modell zur automatischen Verteilung mittels Programmanalyse

Die Kernidee des hier verwendeten Verteilungsansatzes besteht darin, alle strategischen Entscheidungen über Transformation und Verteilung der Objekte vom Verteilungsplan treffen zu lassen. Der Grund dafür, warum solche Entscheidungen nicht direkt im verteilten Programm codiert, sondern durch den Verteilungsplan gekapselt werden, ist, dass dadurch die geforderte Flexibilität und Anpassbarkeit gewährleistet wird. Außerdem werden die Verteilungsentscheidungen im Plan nicht konkret berechnet, sondern nur in Form einer Sammlung von Vorschlägen formuliert. Dadurch bietet sich die Möglichkeit, Informationen über das Programm, welche erst nach der statischen und dynamischen Programmanalyse berechnet werden können, und Information über die verteilte Umgebung, beispielsweise die Anzahl der zur Verfügung stehenden virtuellen Maschinen, in die Entscheidung mit einfließen zu lassen.

Geeignete statische und dynamische Analyseverfahren werden verwendet, um die Programmeigenschaften zu bestimmen, die relevant für die automatische Verteilung sind. Diese Eigenschaften werden dann innerhalb spezieller Objektverteilungsfunktionen (ODFs) benutzt, die elementaren Transformations- und Verteilungsentscheidungen berechnen. Durch geeignete Zusammensetzung bestimmter ODFs entstehen Strategien, die Entscheidungen über Programmtransformation und Objektverteilung treffen. Mehrere passende Strategien bilden zusammen einen Verteilungsplan, der über die Transformation der Objekterzeugungsstellen entscheidet und Aussagen zur Objektplatzierung während der Programmausführung liefert. Abbildung 5.1 stellt das konzeptionelle Modell zur automatischen Verteilung visuell dar, wobei es aus fünf verschiedenen, aufeinander aufbauenden Bereichen besteht.

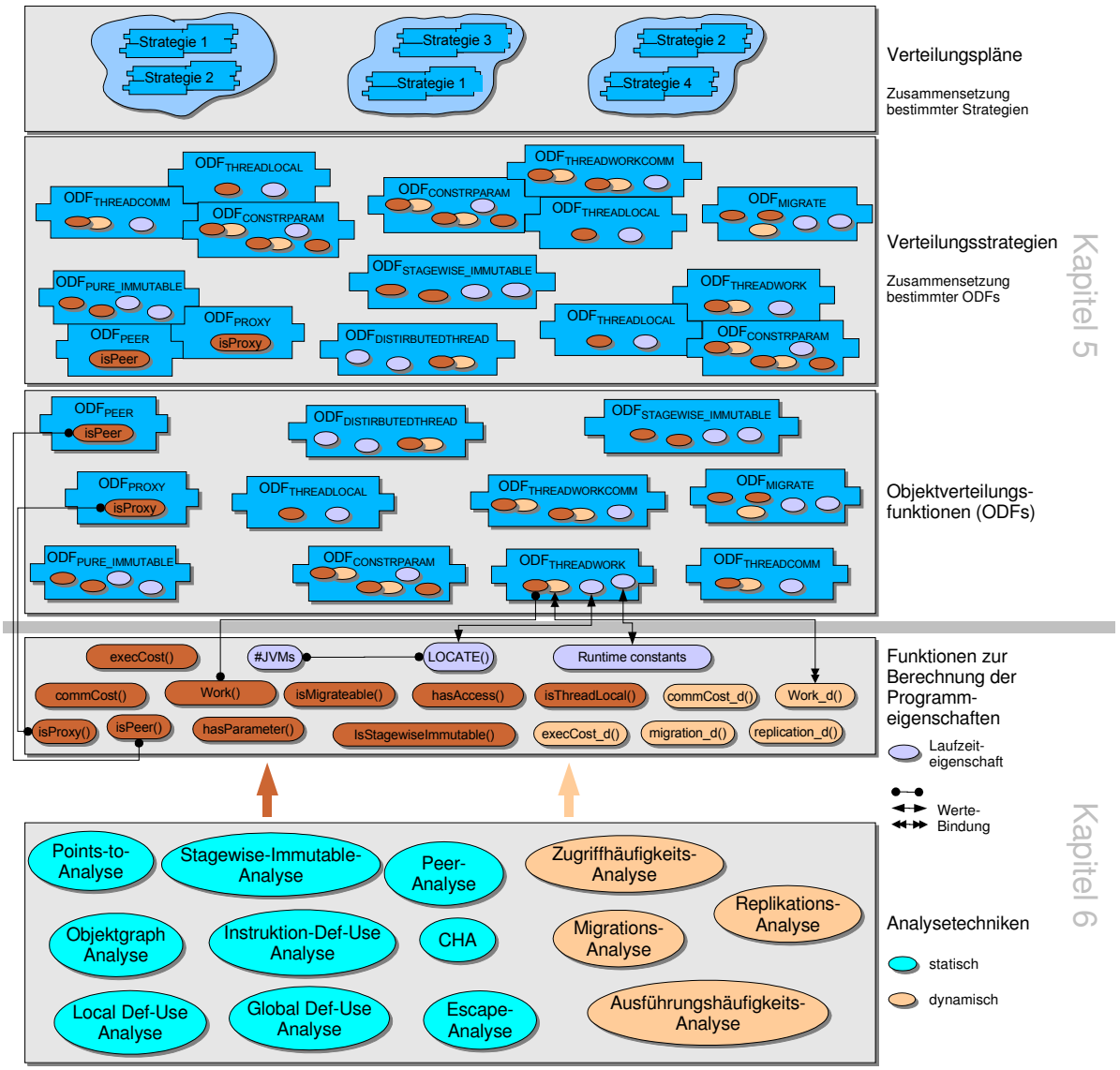

Abbildung 5.1.: Modell zur automatischen Verteilung - Von Programmanalyse zu Verteilungsplänen

Wie in Abbildung 5.1 angedeutet werden die erforderlichen Formalismen zur Formulierung der ODFs, Verteilungsstrategien und somit der Verteilungspläne in diesem Kapitel beschrieben. Hierbei wird auf einzelne spezielle ODFs (angegeben in Abbildung 5.1) im Detail eingegangen. Deren Zusammensetzung als Strategien und darauf aufgebaute Verteilungspläne werden hier ebenfalls dargestellt. Das Konzept, Analyseverfahren zur Bestimmung der Programmeigenschaften für die Verteilung automatisch anzuwenden, insbesondere der Einsatz statischer und dynamischer Analysen, wird noch in die-

sem Abschnitt vorgestellt. Details über einzelne Eigenschaften, angewandte Analysetechniken sowie das formale Modell für die Kombination statischer und dynamischer Eigenschaften finden sich dann im Kapitel 6.

5.1.1. Programmanalyse zur Berechnung der Programmeigenschaften

Die elementaren Eigenschaften eines Programms, die bei der automatischen Verteilung eine wichtige Rolle spielen, werden durch bestimmte Analyseverfahren berechnet. Statische Programmanalyse ist in der Lage, vor der Laufzeit viel Information über das Programm, vor allem verteilungsrelevante Eigenschaften des Programms zu beschaffen. Es steht im Prinzip statischen Analyseverfahren mehr Zeit zur Verfügung, weil sie vor der eigentlichen Ausführung des Programms durchgeführt werden und ihre Ergebnisse unabhängig von einzelnen Programmabläufen sind. Diese Ergebnisse können aber bei jeder konkreten Programmausführung genutzt werden. Somit lohnt sich der dort investierte Aufwand. Allerdings reicht statische Analyse allein nicht aus, da sich nicht nur der Kontrollfluss und die Eingabedaten eines Programms, sondern auch andere Faktoren wie z.B. die Rechnerlast oder die Kommunikationskosten zur Laufzeit ändern können. Diese Faktoren dürfen nicht ignoriert werden, deshalb liegen sie vor der Laufzeit als Verteilungsvariablen vor und werden bei der Formulierung der Verteilungsfunktionen, ebenfalls auch vor der Laufzeit, mit berücksichtigt.

Wie in Abbildung 5.2 verdeutlicht wird, werden *statische* und *dynamische Programmeigenschaften* (als dunkle bzw. helle Ovale visualisiert) mittels Funktionen berechnet. Innerhalb dieser Funktionen liegen die Eigenschaften als *statische* bzw. *dynamische Verteilungsvariablen* vor, die durch entsprechende Verfahren zur Programmanalyse berechnet werden. Außerdem gibt es Laufzeiteigenschaften, deren konkrete Werte erst zur Laufzeit bekannt werden. Sie werden in Abbildung 5.2 im mittleren Bereich als ganz helle Ovale (`#JVM`, `Runtime constants` usw.) dargestellt. Zu verschiedenen Zeitpunkten werden bestimmte *konkrete Werte*, die sich aus statischen oder dynamischen Analyseergebnissen oder durch Bekanntgeben der Anzahl der JVMs ergeben, an diese Verteilungsvariablen *gebunden*. So können elementare Verteilungsergebnisse berechnet werden. Es gibt drei verschiedene Arten von Wertebindungen: Bindung der Werte an statische Variablen, an dynamische Variablen und Bindung an Laufzeitunbekannte.

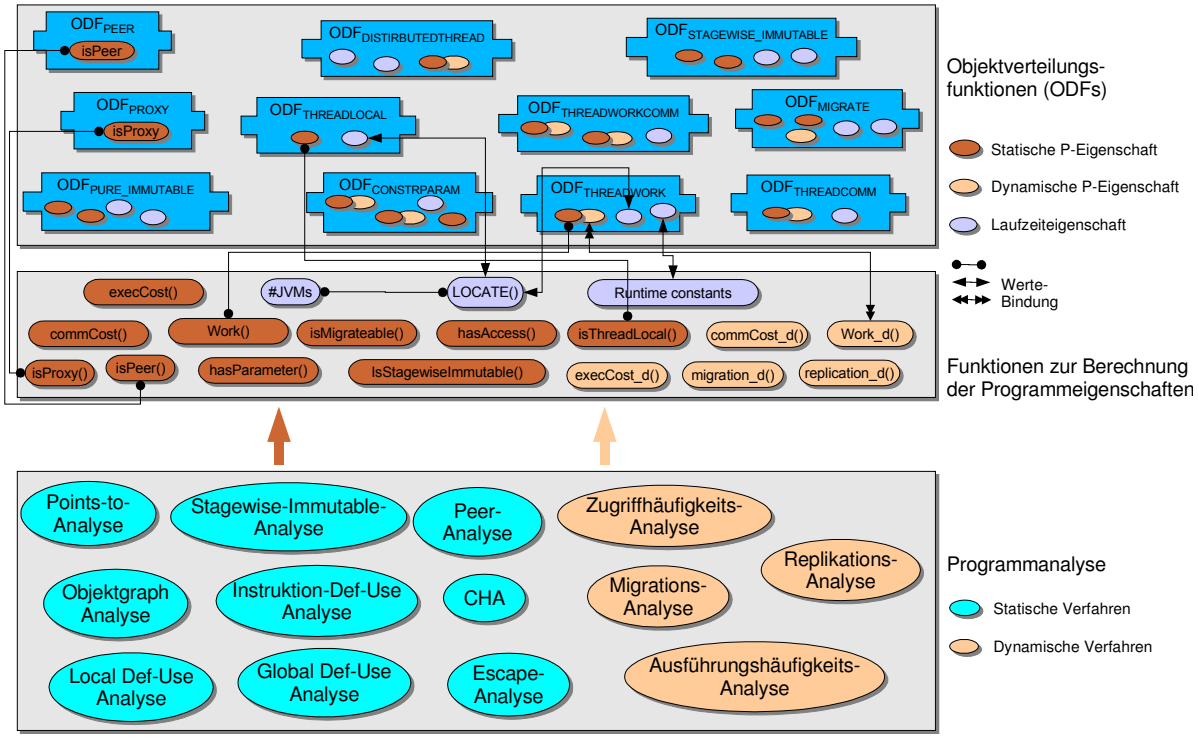

Abbildung 5.2.: Analysetechniken zur Bestimmung der Programmeigenschaften

Jede spezielle Verteilungsfunktion (ODF) verwendet bei ihrer Berechnung der Verteilungsergebnisse bestimmte Eigenschaften des Programms und/oder Laufzeiteigenschaften. Vor der Laufzeit sind die Laufzeiteigenschaften unbekannt, deshalb fließen sie in die Berechnung der ODFs als Variablen ein. Bei einigen ODFs können die verwendeten Programmeigenschaften vollständig vor der Laufzeit ausgerechnet werden. Es gibt aber manche Programmeigenschaften, die von der Programmanalyse vor dem Programmlauf abgeschätzt werden müssen und die abgeschätzten Werte lassen sich zur Laufzeit durch Ergebnisse dynamischer Analyse verbessern. Somit lassen sich die Verteilungsergebnisse der zugehörigen ODFs verbessern. Dies ist auch der Ansatz zur *hybriden Analyse*, die statische und dynamische Analyseergebnisse kombiniert.

Statische Eigenschaften

Statische Analyseverfahren werden hier eingesetzt, um sowohl statische Eigenschaften des Programms zu bestimmen, als auch dynamische Programm-

eigenschaften abzuschätzen, die sich vor der Laufzeit nicht berechnen lassen. Einige Verfahren wurden in Abbildung 5.2 im Analysebereich (unterer Bereich) namentlich genannt. Eine Programmeigenschaft kann aus zwei Teilen bestehen: Der eine beschreibt den statischen berechenbaren Aspekt und der andere repräsentiert den dynamischen. Diese zur Laufzeit ergebenen Eigenschaften modellieren die Ausführungskosten oder Kommunikationskosten. Für den statisch bestimmbaren Anteil einer solchen Eigenschaft kommen ein oder mehrere statische Analysetechniken zum Einsatz.

Einige spezielle ODFs brauchen für ihre Berechnungen vollständige Ergebnisse der verwendeten Programmeigenschaften, die aus solchen statischen und dynamischen Anteilen bestehen. Daher werden ebenfalls statische Verfahren zur Programmanalyse benutzt, um Abschätzungen für dynamische Anteile dieser Programmeigenschaften durchzuführen. Dennoch liegen Laufzeiteigenschaften als Variablen bei den Berechnungen vor. Mit diesem Modell können alle Faktoren, sowohl statische als auch dynamische, bei der Berechnung der Verteilungsergebnisse berücksichtigt werden. Als Beispiel betrachten wir die ODF $ODF_{\text{THREADWORK}}$ im oberen Bereich von Abbildung 5.2: Um Verteilungsergebnisse bestimmen zu können, braucht diese ODF u.a. die Programmeigenschaft `Work`, die prinzipiell aus statischen und dynamischen Anteilen besteht und mittels der beiden Funktionen *Work()* und *Work_d()* (im mittleren Bereich) berechnet wird. Vor der Laufzeit liefert die Funktion *Work_d()* einen durch die statische Analyse abgeschätzten Wert, der zunächst mit zugehörigen Variablen innerhalb der ODF gebunden ist. Die andere Funktion *Work()* kann vollständig ausgerechnet werden. Unter der Verwendung dieser Werte liefert die $ODF_{\text{THREADWORK}}$ ein so genanntes *initiales Verteilungsergebnis*, das eine Zuordnung der an einer Erzeugungsstelle entstehenden Objekte zu abstrakten JVMs beschreibt.

Dynamische Eigenschaften

Dynamische Programmeigenschaften sind solche, die sich bei jeder Programmausführung ändern und sich vor der Laufzeit durch statische Analyse sehr schwer bestimmen lassen. Beispiele dafür sind die Ausführungskosten der Programmschleifen, Kommunikationskosten durch Methodenaufrufe aufgrund der dynamischen Methodenbindung oder Zugriffshäufigkeiten bestimmter Methoden. Statische Analyseverfahren müssen Abschätzungen für solche Eigenschaften machen, damit initiale Verteilungsergebnisse vor der Laufzeit berechnet werden können. Durch das vorgestellte Modell werden

solche Eigenschaften durch Verteilungsvariablen innerhalb der ODFs modelliert, die zu verschiedenen Zeitpunkten unterschiedliche Werte annehmen können. Während der Programmausführung werden einige Häufigkeitsanalysen durchgeführt, die gezielt zu bestimmten Eigenschaften Laufzeitwerte sammeln. Die abgeschätzten Werte werden durch diese ersetzt und somit lassen sich neue Verteilungsergebnisse errechnen. Sie werden mit den initialen Ergebnissen verglichen. Falls sich dabei bessere Platzierungsergebnisse ergeben, werden diese für die demnächst zu erzeugenden Objekte verwendet und die alten werden überschrieben. Dieses Prinzip wird in Abbildung 5.2 durch die Werte-Bindung zwischen der Funktion *Work_d()* und der Verteilungsvariable in der $ODF_{\text{THREADWORK}}$ verdeutlicht.

Ergebnisse solcher Häufigkeitsanalysen sind auch Grundlage für die Objektmigration und -replikation. Nach dem gerechneten Verteilungsergebnis hat man einen Plan, wohin ein Laufzeitobjekt migriert oder repliziert werden kann. Basierend auf den Zugriffshäufigkeiten der Methoden wird dieser Plan zur Laufzeit umgesetzt: Es wird entschieden, zu welcher konkreten JVM das Objekt migriert oder repliziert wird. Ein Beispiel dafür ist die Funktion *migration_d()* in Abbildung 5.2, die zur Laufzeit aktuelle Werte zur Zugriffshäufigkeit für die ODF_{MIGRATE} liefert.

Außer den beiden Programmeigenschaften werden noch Architektur- und Laufzeiteigenschaften in die Berechnung der Verteilung einbezogen. Sie modellieren unter anderem *Laufzeitkonstanten, die vor der Laufzeit unbekannt sind und während der Ausführung konstant bleiben*. Beispiele dafür sind die Anzahl der beteiligten JVMs oder Programmvariablen, die sich bei jeder Programmausführung ändern können, aber während einer Ausführung konstant bleiben.

Unter Berücksichtigung aller relevanten Parameter zur Verteilung können Platzierungsergebnisse mittels Strategien berechnet werden. Zur Laufzeit besteht ein solches Ergebnis aus einer Zuordnung von konkreten Programmobjekten zu Adressräumen (oder JVMs). Die Umsetzung dieser Zuordnung wird vor der Laufzeit durch geeignete Transformationen direkt im Programm codiert. Dies erlaubt bei der Ausführung des transformierten Programms, den Plan zu evaluieren, um dann konkrete Verteilungshinweise zu erhalten und die Objekte entsprechend zu platzieren.

5.1.2. Anforderungen vor der Laufzeit

Zum Zeitpunkt der Transformation hat der Verteilungsplan für jede Objekterzeugungsstelle die Entscheidung zu treffen, ob alle dort zur Laufzeit entstehenden Objekte *remote*-Objekte oder *nicht-remote* Objekte sind. Auf diese Anfrage muss der Plan eine eindeutige Antwort liefern. Wenn festgestellt wird, dass nur auf ein Objekt entfernte Zugriffe stattfinden können, werden auf der Transformator-Seite erforderliche Transformationsmechanismen (siehe Kapitel 7) für die zugehörige Klasse, die Erzeugungsstelle und die Zugriffstellen angewandt. Zur Bindung einer Verteilungsentscheidung an eine Objekterzeugungsstelle wird eine Planevaluierung in Form von Anfragen direkt an dieser Programmstelle eincodiert. Unmittelbar vor der Erzeugung eines Objekts liefert der Plan durch diese Evaluation ein Verteilungsergebnis, also die Zielmaschine, wo das Objekt erzeugt wird.

In diesem Konzept werden Strategien zur automatischen Objektmigration und -replikation eingesetzt, um die entfernten Methodenaufrufe zu minimieren und somit einen sonst entfernt ausgeführten Methodenaufruf lokal auszuführen. Nach diesen Strategien wird entschieden, wann und zu welcher Zielmaschine ein Objekt während des Programmlaufs migriert bzw. repliziert wird. Mit diesem Konzept sowie mit der zugehörigen technischen Umsetzung befasste sich die Diplomarbeit von Marco Weißenborn [Wei03]. Damit die Objekte während der gesamten Ausführung der Anwendung migriert bzw. repliziert werden können, müssen einige technische Vorbereitungen zur Transformationszeit durchgeführt werden. Zuvor hat der Verteilungsplan die Entscheidung zu treffen, ob die entstehenden Objekte einer Klasse migrierbar bzw. replizierbar sind. Zusätzlich ermittelt er Informationen über das Zugriffsverhalten jeder Methode einer Klasse, deren Objekte migriert bzw. repliziert werden können. Für die Transformation ist es notwendig zu erkennen, ob eine Methode den Zustand des Objekts ändert oder nicht, d.h. ob diese Methode nur Lesezugriffe oder auch Schreibzugriffe auf die Instanzvariablen durchführt. Auf der Grundlage der statischen Analyseergebnisse werden solche Eigenschaften möglichst genau berechnet.

Vor der Laufzeit ist Berechnung aller statisch bestimmbaren Eigenschaften mittels Programmanalyse durchzuführen. Analyseergebnisse werden in den ODFs zur Transformation und zur Platzierungsberechnung verwendet. Da der Analyseaufwand die Ausführungszeit des Programms nicht belastet, werden komplexe statische Analyseverfahren eingesetzt, um die Programmeigenschaften so genau wie möglich zu berechnen.

5.1.3. Aufgaben zur Laufzeit

Unmittelbar vor dem Start steht die Anzahl der beteiligten JVMs fest. Diese Architektureigenschaft wird dem Plan bekannt gegeben. Während der gesamten Ausführung des Anwendungsprogramms werden Objekte auf den beteiligten virtuellen Maschinen erzeugt. Vor jeder Objekterzeugung wird eine Planevaluierung durchgeführt, welche durch die bei der Transformation direkt ins Programm eincodierten Anfragen aufgelöst wird. Als Verteilungsergebnis für solch eine Anfrage hat der Plan eine eindeutige Antwort zu liefern, und zwar die Nummer der Zielmaschine, auf der das Objekt erzeugt wird. Zur Berechnung solcher Verteilungsergebnisse greift der Plan intern auf die Ergebnisse der Verteilungsstrategien zurück, welche wiederum Ergebnisse der ODFs verwenden. Diese Berechnungskette entspricht dem Evaluierungsprozess des Planes. Damit die Planevaluation die gesamte Laufzeit des Programms nicht belastet, müssen solche Ergebnisse großenteils vor der Laufzeit bestimmt worden sein. Der Verteilungsplan hat folgende Aufgaben zur Laufzeit zu bewältigen:

- ☞ Aufgrund der vorliegenden Ergebnisse der ODFs plus einiger sich zur Laufzeit ergebender Eigenschaften zur Architektur der Laufzeitumgebung hat der Verteilungsplan nach dem Start des Programms zu jeder Anfrage eine eindeutige Antwort zu liefern. Da die Anwendung verteilt auf vielen JVMs ausgeführt wird, die sich physikalisch auf verschiedenen Rechnern befinden und miteinander über das Netzwerk kommunizieren, muss es gewährleistet sein, dass die berechneten Verteilungsergebnisse des Planes für die gesamte verteilte Anwendung konsistent und korrekt bleiben. Konzeptionell verhält sich der Verteilungsplan nach außen wie eine einzige Komponente, die auf Anfrage Entscheidungen zur Objektverteilung trifft. Dass in der verteilten Laufzeitumgebung auf jeder JVM eine Instanz des Verteilungsplans vorhanden ist und wie diese Instanzen und ihre Daten verwaltet werden, ist ein reiner Implementierungsaspekt.
- ☞ Während der Programmausführung werden einige dynamische Programmeigenschaften und Informationen zur Architektur und zur verteilten Laufzeitumgebung bestimmt, die zur Verteilung Anwendung finden. Diese neu gewonnenen Erkenntnisse werden auch mit in die Verteilungsentscheidung für die nächsten zu erzeugenden Objekte einbezogen. Der Verteilungsplan hat diese neu hinzugekommenen Daten mit

den vorhandenen Eigenschaften zu kombinieren. Bei der nächsten Planevaluierung werden die Ergebnisse der ODFs und somit auch die der Verteilungsstrategien aufgrund der aktualisierten Datenbasis berechnet.

☞ Da Strategien zur Migration und Replikation eingesetzt werden, können vorhandene Programmobjekte zur Laufzeit von einer JVM zu einer anderen migriert oder auf mehreren JVMs repliziert werden. Die Strategien werden auf die Programmobjekte angewandt, die schon erzeugt wurden. Vom Prinzip treffen die Strategien die Entscheidung zur Migration und Replikation aufgrund des Laufzeitverhaltens des Programms zusammen mit einigen aufgestellten Schwellwerten [Wei03]. Diese Strategien sind in der Regel unabhängig von denjenigen im Plan. Aber die Information über migrierte und replizierte Objekte ist auch relevant zur Verbesserung von Verteilungsentscheidungen für die zukünftig zu erzeugenden Objekte. Deshalb werden die Daten über die neuen Aufenthaltsorte der Objekte den Verteilungsstrategien im Plan mitgeteilt und fließen in die Verteilungsergebnisse ein.

Alle zusätzlichen Berechnungen, die zur Laufzeit stattfinden, beeinträchtigen die Ausführungsdauer des Programms. So werden zur Laufzeit Werte zu dynamischen Verteilungsvariablen und Architektureigenschaften gebunden, die bei einer erneuten Berechnung zur verbesserten Verteilung nötig sind. Zuvor werden Häufigkeitsanalysen eingesetzt, um Ausführungshäufigkeiten oder konkrete Aufrufziele dynamisch gebundener Methoden zu bestimmen. Außerdem muss die *Verteilungsdatenbank* nach jeder Replikation oder Objektmigration aktualisiert werden. Ein Verteilungsplan hat über solche Interaktionen zu informieren und seinen Wissensstand über das laufende Programm aktuell zu halten. Dieses Wissen fließt in die nächste Verteilungsberechnung ein.

Nun möchte ich auf die einzelnen Komponenten eines Verteilungsplanes mit passenden Konzepten und Modellierung eingehen.

5.2. Verteilungsplan: die Komponenten, deren Modellierung und verwendete Konzepte

Ein Verteilungsplan besteht aus mehreren auf geeignete Weise zusammengesetzten Verteilungsstrategien, wie es in Abbildung 5.3 dargestellt wird. Die

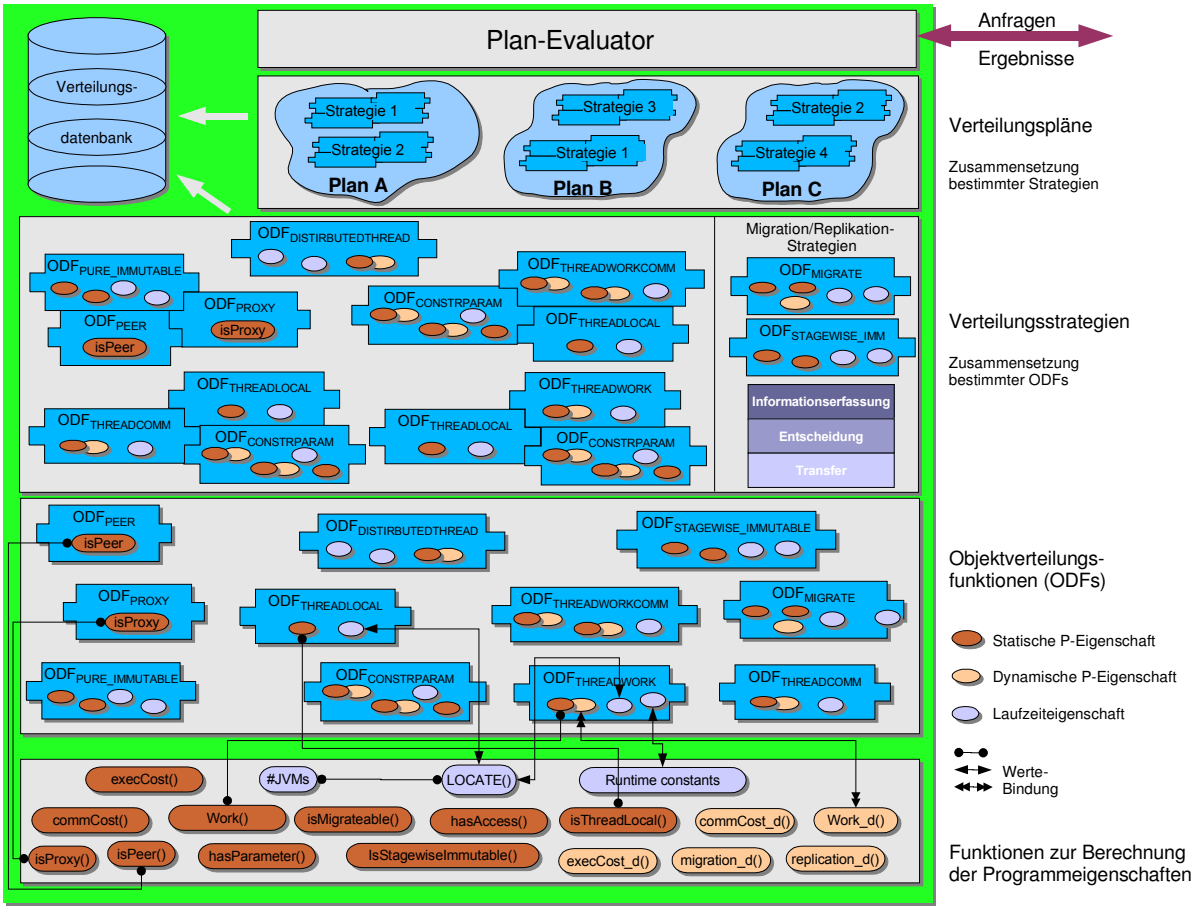

Abbildung 5.3.: Das Zusammenwirken aller Komponenten bei Verteilungsplänen

fundamentalen Einheiten eines Verteilungsplanes sind die Objektverteilungsfunktionen, auch ODFs genannt. Mit deren Hilfe können elementare Verteilungsentscheidungen berechnet werden. Die ODFs benutzen *statische und dynamische Eigenschaften* für ihre Berechnungen. Diese Eigenschaften werden aus einer mehrsträngigen Java-Anwendung heraus abstrahiert und charakterisieren relevante Verteilungsinformation des Programms. Es sind im Wesentlichen statische Programm- und Laufzeiteigenschaften. Außerdem spielen die anwendungsunabhängigen Architektureigenschaften eine wichtige Rolle bei der Berechnung zur Verteilung. Programm- und Laufzeiteigenschaften werden automatisch durch verschiedene Analyseverfahren berechnet, Architektureigenschaften ergeben sich zur Laufzeit unmittelbar vor der Ausführung des verteilten Programms. All diese Eigenschaften werden in Form von Variablen innerhalb der ODFs ausgedrückt. Durch das Bindungskonzept können

sie konkrete Werte zu verschiedenen Zeitpunkten, vor und/oder zur Laufzeit, aufnehmen, die sich aus statischen oder dynamischen Analyseergebnissen ergeben. Sobald alle für die Berechnung einer ODF notwendigen Werte vorliegen, kann diese ODF ein Verteilungsergebnis liefern.

Durch geeignete Zusammensetzung bestimmter ODFs entsteht eine Verteilungsstrategie. Innerhalb einer Strategie lässt sich das Prinzip der funktionalen Abstraktion sehr sinnvoll anwenden: Die Ergebnisse einer ODF können als Berechnungsparameter oder Bedingungen für die nächsten Berechnungsschritte anderer ODFs benutzt werden. Dadurch kann die Abhängigkeit verschiedener ODFs ausgedrückt werden.

Durch geeignete Kombination von bestimmten Strategien erhält man einen Plan für die automatische Verteilung eines mehrsträngigen Java-Programms. Die Verteilungseigenschaften sowie die Strategien, damit auch die Pläne werden so allgemein formuliert, dass sie sich auf jedes mehrsträngige Java-Programm anwenden lassen können. Der Plan-Evaluator bietet die Schnittstelle zur Plananfrage bzw. Planevaluation und liefert Ergebnisse zur Transformation und zur Platzierung der Objekte. Außerdem werden alle getroffenen Entscheidungen zur Verteilung und aktuell nützliche Laufzeitdaten, die durch die dynamische Analyse zur Laufzeit gesammelt werden, inklusive Migrations- und Replikationsentscheidungen in einer Datenbank aufgezeichnet. Sie dienen vor allem als Grundlage zur Berechnung einiger Strategien zur Laufzeit und zur Berechnung der demnächst zu treffenden bzw. zu verbessernden Verteilungsentscheidungen.

Im Folgenden werde ich auf die formale Definition der Komponenten einzeln eingehen und stelle ihre Charakteristika und ihre Zusammenhänge dar.

5.2.1. Modellierung von Verteilungsvorschriften als ODFs

Wie im Abschnitt 3.2.2 eingeführt, berechnet eine ODF eine elementare Transformations- und Verteilungsentscheidung. Ergebnisse einer ODF hängen stark von Werten der darin verwendeten Variablen ab, die die Programm- und Architektureigenschaften repräsentieren und mittels statischer und dynamischer Programmanalyse bestimmt werden. So wird ein Verteilungsergebnis durch eine bestimmte Verteilungsstrategie berechnet. Dieses Ergebnis ist das Resultat aus der Zusammensetzung einer oder mehrerer elementarer Ergebnisse der ODFs, die in der Strategie verwendet werden. Um ODFs for-

mal definieren zu können, sollen zunächst einige der verwendeten Begriffe definiert werden.

Definition 5.0:
Sei OG die Menge aller Objekterzeugungsstellen im gesamten Programm, die beginnend mit 1 fortlaufend durchnummeriert sind. Es sei $\mathbb{O}$ die Menge aller zur Laufzeit entstehenden Objekte. Dann sei $O_{i,j} \in \mathbb{O}$ das j-te Objekt, das an der Erzeugungsstelle $i \in OG$ zur Laufzeit erzeugt wird. O_i sei ein Objektrepräsentant für alle Objekte, die an der Programmstelle $i \in OG$ erzeugt werden. △

Da Objekte erst zur Laufzeit erzeugt werden, kann man vor der Laufzeit verschiedene Objekte nicht explizit unterscheiden, die an derselben Stelle im Programm erzeugt werden. Dies ist zum Beispiel der Fall, wenn eine Erzeugungsstelle in einer Schleife liegt. Diese Objekte werden in äquivalente Klassen eingestuft, die die gleichen statischen Eigenschaften aufweisen, aber unterschiedliches Laufzeitverhalten besitzen können.

Für spezielle Objekte wurde der Begriff *Aktivität* schon im Kapitel 2.3.3 eingeführt, für die nun definiert wird:

Definition 5.1:
Sei $\mathbb{A}$[1]$\subset \mathbb{O}$ die Menge aller Aktivitäten, die zur Laufzeit aktiv sind. $\mathbb{A}_{static}$ sei die Menge der Aktivitätsrepräsentanten. Eine Aktivität $A_i \in \mathbb{A}_{static}$ stellt somit einen Aktivitätsrepräsentanten für alle Aktivitäten dar, die an derselben Programmstelle $i \in OG$ erzeugt werden. Dann sei $A_{i,j} \in \mathbb{A}$ das j-te Thread-Objekt, das an der Erzeugungsstelle $i \in OG$ zur Laufzeit erzeugt wird.

Es seien des Weiteren die Funktionen $ThreadId$ und $ObjectId$ wie folgt definiert, die jeder Aktivität bzw. jedem Objekt eine eindeutige Identität, hier also eine natürliche Zahl, zuordnet:

$$ThreadId : \mathbb{A} \rightarrow \mathbb{N}, ObjectId : \mathbb{O} \rightarrow \mathbb{N}$$

△

Es liegt in der Natur der Sache, dass vor der Laufzeit die Beziehung zwischen Objektrepräsentanten und Aktivitätsrepräsentanten mittels statischer Programmanalyse bestimmt wird. Daher wird zur Berechnung der Verteilung

[1]Das sind Thread-Objekte im ursprünglichen Programm.

vor der Laufzeit hauptsächlich die Menge der Aktivitätsrepräsentanten benutzt. Sie werden dann zur Laufzeit zusammen auf verschiedenen Rechnerknoten platziert. Aus einzelnen Aktivitätsrepräsentanten können zur Laufzeit konkrete Aktivitäten entstehen, die auf die tatsächlichen JVMs platziert werden. Nun kann eine Definition der ODFs erfolgen.

Definition 5.2:
Sei n *die Anzahl der JVMs, die zur Laufzeit in der verteilten Laufzeitumgebung zur Programmausführung zur Verfügung stehen. Dann sei* $AR := \{\mathsf{0}, \ldots, \mathsf{n\text{-}1}\}$ *die Menge der verfügbaren JVMs. Des Weiteren sei* here *ein ausgezeichnetes Element, das die aktuelle JVM zur Laufzeit repräsentiert. Für die Transformationsentscheidung sei die Menge* $R := \{\mathsf{remote}, \mathsf{noremote}\}$ *eingeführt. Diese Elemente sagen aus, ob eine Erzeugungsstelle und ggf. die zugehörige Klasse transformiert werden oder nicht. Außerdem sei ein Element* noans *definiert. Es gibt an, dass die vorliegende ODF zu einer gegebenen Erzeugungsstelle keine Aussage macht. Es sei eine Hilfsfunktion definiert, die den Aufenthaltsort eines Objektes lokalisiert:*

$$LOCATE : \mathbb{O} \rightarrow AR \cup \{\mathsf{noans}\}$$
$$\textit{mit}$$
$$LOCATE(A_i) \neq \mathsf{noans}, \forall A_i \in \mathbb{A}_{\textit{static}}$$

Nun läßt sich eine ODF wie folgt definieren:

$$f_{name} : \mathbb{O} \times \bigotimes_{i \in \mathbb{I}} \mathbb{P}_i \times (\mathbb{O} \rightarrow AR \cup \{\mathsf{noans}\}) \rightarrow (\wp(AR) \times R) \cup \{\mathsf{noans}\}$$

wobei $\mathbb{P}_i$ *die Menge der Funktionen bezeichnet, die die Programmeigenschaft* $i \in \mathbb{I}$ *berechnen.* $\mathbb{I}$ *sei die Indexmenge der Namen aller Programmeigenschaften, die relevant für die automatische Verteilung sind.*

△

Eine Verteilungsfunktion bildet somit die zur Laufzeit erzeugten Objekte, abhängig von darin verwendeten spezifischen Programmeigenschaft(en) und der $LOCATE()$-Funktion, auf eine Menge von Kandidaten der Adressräume oder JVMs, zusammen mit dem Hinweis für eine notwendige Transformation ab. Einer ODF wird ein Tupel von konkreten Funktionen, z.B. $(E_{\texttt{isProxy}}, E_{\texttt{Work}}, \ldots) \in \bigotimes_{i \in \mathbb{I}} \mathbb{P}_i$, übergeben, die alle relevanten Eigenschaften

mittels Programmanalyse berechnen. Innerhalb der ODF werden die entsprechenden Funktionen nach ihren indizierten Eigenschaftnamen aus dieser Eingabe ausgesucht und verwendet. Das berechnete Verteilungsergebnis einer ODF hängt von diesen Eigenschaften ab. Unter der Verwendung der Hilfsfunktion $LOCATE()$, die den Aufenthaltsort eines Objekts oder einer Aktivität modelliert und ebenfalls als Parameter der ODF übergeben wird, können relative Zuordnungen der Objekte innerhalb einer ODF berechnet werden.

Wenn an derselben Erzeugungsstelle zwei ODFs verschiedene Ergebnisse remote und noremote zur Transformation liefern, dann wird noremote bevorzugt. Außerdem wird eine Klasse für entfernte Zugriffe und *remote*-Erzeugung ihre Objekte transformiert, wenn nur an einer Erzeugungsstelle von Objekten dieser Klasse im Programm remote vorliegt. Konzeptionell unterscheidet sich noans vom remote dadurch, wenn keine Aussage über eine Programmeigenschaft gemacht werden kann und auch keine Verteilungsentscheidung getroffen wird.

Definition 5.3:
Es gibt verschiedene Arten von Objektverteilungsfunktionen:

Totale und partielle Funktion: *Eine ODF ist eine partielle Funktion, wenn eine Eingabe aus dem Definitionsbereich existiert, bei der* noans *als Ergebnis geliefert wird. Berechnet eine ODF zu jeder Eingabe ein konkretes Ergebnis ungleich* noans*, so ist sie total.*

Statische und dynamische Funktion: *Eine ODF heißt statisch, wenn alle Werte, die zur Berechnung der Funktion beitragen, vor der Laufzeit vollständig vorliegen. Im Gegensatz dazu können bestimmte Programmeigenschaften, die innerhalb einer ODF verwendet werden, nicht vollständig vor der Laufzeit berechnet, sondern nur abgeschätzt werden. Solche Funktionen sind dynamisch. Außerdem ist eine Funktion auch dynamisch bzw. statisch, wenn ihre Ergebnisse wiederum von Ergebnissen anderer dynamischer bzw. statischer Funktionen abhängen.*

Konstante Funktion: *Eine ODF, die zu jeder Eingabe immer das gleiche Ergebnis liefert, heißt konstant.* △

Zur Verdeutlichung sollen im Folgenden zwei konkrete ODF-Beispiele dargestellt werden.

BEISPIEL 5.0:
Die folgende ODF demonstriert eine Verteilungsvorschrift für die entstehenden Objekte, deren Daten ausschließlich von umgebenden Objekten benutzt

werden, beispielsweise durch Methodenaufrufe. Deshalb bleiben sie *nur* für ihre umgebenden Objekte sichtbar und zugreifbar. Derartige Objekte haben die so genannte *Proxy*-Eigenschaft. Die ODF für solche Objekte sieht folgendermaßen aus:

$$f_{\mathrm{PROXY}}(O_i, E, LOCATE) := \begin{cases} (\{\mathsf{here}\}, \mathsf{noremote}) & \text{if} \quad E_{\mathtt{isProxy}}(O_i), \\ \mathsf{noans} & \text{else} \end{cases} \tag{5.1}$$

Diese Verteilungsvorschrift nimmt Repräsentanten für Objekte als Eingabe, die an der Programmstelle i erzeugt werden. Abhängig von der Programmeigenschaft *Proxy*, welche mittels einer konkreten Funktion $E_{\mathtt{isProxy}}$ berechnet wird, liefert diese ODF das Ergebnis, dass die Transformation an dieser Stelle nicht nötig ist und zur Laufzeit die dort entstehenden Objekte nur lokal auf der JVM erzeugt werden, wo der Programmcode gerade ausgeführt wird. Dies ist eine *partielle Funktion*, denn nicht für alle Objekt-Repräsentanten $O_i \in \mathbb{O}$ ist die Eigenschaft *Proxy* erfüllt und somit kann nicht immer ein Verteilungsergebnis ermittelt werden. In diesem Fall wird noans zurückgeliefert. An der Stelle von E wird dieser ODF ein Tupel von konkreten Funktionen zu allen Programmeigenschaften übergeben. $E_{\mathtt{isProxy}}$ ist ein Tupelelement, welches durch den Index $\mathtt{isProxy} \in \mathbb{I}$ identifiziert wird. □

Aus dem obigen Beispiel für die ODF f_{PROXY} läßt sich erkennen, dass die Programmeigenschaft `Proxy` eine wichtige Rolle spielt. Sie beeinflusst nicht nur die Entscheidung der Verteilungsfunktion, sondern charakterisiert auch, dass diese ODF rein statisch ist. Das bedeutet, dass die verwendeten Eigenschaften vor der Laufzeit durch statische Programmanalyse vollständig berechnet werden können, sodass eine Evaluierung dieser Funktion vor der Laufzeit erfolgen kann. Somit läßt sich eine Teilmenge von Objekterzeugungen bestimmen, deren Platzierung nicht von Laufzeitdaten abhängt. Die Berücksichtigung solcher Verteilungsentscheidungen schon zum Zeitpunkt der Transformation ermöglicht wieder die direkte Umsetzung der Verteilungsentscheidung in Programmcode. Zu bemerken ist, dass eine Programmeigenschaft innerhalb einer ODF über eine Funktion berechnet wird, wie hier die *Proxy*-Eigenschaft über die Funktion $E_{\mathtt{isProxy}}$.

Zur Vereinfachung werden im Folgenden die Namen solcher zur Berechnung der Eigenschaften verwendeten Funktionen direkt als Programmeigenschaften bezeichnet.

Das Ergebnis noremote gilt also definitiv für alle Objekte an einer Erzeugungsstelle, die man nicht zu verteilen braucht. f_{PROXY} ist eine der einfachsten ODFs, die zu den Verteilungsentscheidungen beitragen. Im Folgenden sei eine weitere ODF definiert, die zur Berechnung grundlegender Verteilungsentscheidungen für alle Programmobjekte eingesetzt werden kann.

BEISPIEL 5.1:
Durch folgende ODF kann für jedes zu erzeugende Objekt berechnet werden, mit welcher Aktivität es zusammen platziert wird.

$$f_{\text{THREADWORK}}(O_i, E, LOCATE) :=$$
$$(\{LOCATE(A_k) |\ E_{\texttt{Work}}(A_k, O_i) \geq E_{\texttt{Work}}(A_l, O_i)\ \forall A_l \in \mathbb{A}_{\text{static}}\}, \text{remote}). \quad (5.2)$$

Diese Funktion berechnet, dass jedes O_i zusammen mit der Aktivität platziert wird, die die meisten Arbeiten auf ihm verrichtet. Man erkennt im Wesentlichen zwei besondere Aspekte bei dieser Funktion: Das resultierende Ergebnis ist abhängig von der Berechnung einer anderen Funktion ($LOCATE()$), welche vor der Laufzeit noch keine Nummer einer konkreten JVM liefern kann. Trotzdem wird durch die Definition der Funktion $LOCATE()$ sichergestellt, dass sich als Ergebnis eine konkrete Zielmaschine ergeben wird, sobald die konkrete Anzahl der JVMs zur Verfügung steht. So wird eine relative Zuordnung der Objekte zu den Aktivitäten modelliert. Diese Zuordnung ist eine $n : m$-Relation zwischen n Objekten an der Erzeugungsstelle i und m Aktivitäten an der Stelle k. Diese Funktion berechnet also noch keine Zusammenplatzierung einzelner Laufzeit-Objekte bzw. -Aktivitäten, dennoch reicht das Ergebnis für eine initiale Platzierung aus, deren Berechnung vor der Laufzeit notwendig ist.

Der zweite Aspekt ist, dass die Verwendung und Bewertung der Programmeigenschaft `Work` mittels einer prädikatenlogischen Formel geschieht. Eine solche Formel kann genügend effizient vor der Laufzeit berechnet werden, denn die Menge der Aktivitätsrepräsentanten ist klein und die Menge der Erzeugungsstellen ist endlich. Außerdem kann die Programmeigenschaft `Work` sowohl vor der Laufzeit mittels statischer Programmanalyse abgeschätzt, als auch mit Ergänzung der hinzu kommenden Laufzeitdaten während der Ausführung genauer berechnet werden. Somit lassen sich die Ergebnisse dieser ODF nicht nur zum Zeitpunkt der Transformation, sondern auch zur Laufzeit benutzen. Die abgeschätzte Größe der `Work`-Eigenschaft kann jederzeit zur Verteilungsentscheidung verwendet werden, solange es keine Verbesserung durch dynamische Analyse gibt. □

Durch die beiden obigen Beispiele ist deutlich zu erkennen, dass die Variablen, die die Programmeigenschaften modellieren, eine entscheidende Rolle in einer Objektverteilungsfunktion besitzen. Das berechnete Verteilungsergebnis ist abhängig von den Werten dieser Variablen. Die Modellierung der elementaren Verteilungsvorschriften als Funktion mit Programmeigenschaften als Funktionsvariablen und die Auswertung zusammenwirkender Eigenschaften mittels prädikatenlogischer Formeln hat viele Vorteile und bietet Flexibilität. Die folgenden Eigenschaften sollen dies deutlicher machen:

- ☞ Funktionen können sowohl feste als auch relative Zuordnungen modellieren. Dies erkennt man deutlich an den beiden Beispielen: Die ODF f_{PROXY} bildet bestimmte Mengen der Programmobjekte direkt auf die Menge der Adressräume ab. Derartige Funktionen können schon zur Transformationszeit zur Planevaluierung benutzt werden. Es ist zu bemerken, dass es immer besser ist, die Ergebnisse einer Verteilungsfunktion so früh und so weit wie möglich berechnen zu können, also möglichst vor der Laufzeit. Denn dies hat den Vorteil, dass der hohe Berechnungsaufwand, vor allem von der statischen Programmanalyse, die Programmausführung nicht beeinträchtigen wird.

  Im Gegensatz dazu ordnet die ODF $f_{\text{THREADWORK}}$ jeder Aktivität eine bestimmte Menge der zu erzeugenden Objekte zu. Sie liefert nicht den genauen Adressraum, auf dem die Objekte zur Laufzeit platziert werden. Solche Funktionen können verwendet werden, um Objekte miteinander zu gruppieren. Für eine konkrete Platzierung müssen noch andere Funktionen mit betrachtet werden.

- ☞ Funktionen können Abschätzungen der Programmeigenschaften enthalten. Im Gegensatz zu der Programmeigenschaft `isProxy`, welche vor der Laufzeit mittels passender statischer Analyseverfahren vollständig berechnet werden kann, kann die `Work`-Eigenschaft vor der Programmausführung mit Hilfe der statischen Programmanalyse nur abgeschätzt werden. Es liegt auch in der Natur der Sache, dass solche Ausführungskosten nicht konkret vor der Laufzeit in irgendeiner Weise berechnet werden können. Diese Art der Modellierung, wie es am Beispiel der ODF $f_{\text{THREADWORK}}$ verdeutlicht wurde, ermöglicht es, abgeschätzte Werte für die Programmvariable `Work` aufzunehmen und dadurch die zugehörige Funktion vor der Laufzeit berechnen zu können. Außerdem ist diese Modellierung so flexibel, dass die abgeschätzten Werte später durch ver-

besserte Werte einfach ersetzt werden können und eventuell neue Verteilungsergebnisse dadurch entstehen.

- ☞ Funktionen können verschiedene Verteilungsvorschriften formulieren. Zu einer Objekterzeugungsstelle können mehrere Verteilungsaussagen gemacht werden. Sie können als einzelne elementare Funktionen formuliert und dann bedingt durch die Programmvariablen zueinander in Beziehung gebracht werden. Ein einfaches Beispiel: Zu einer Objekterzeugungsstelle gibt es zwei Vorschriften, modelliert durch zwei verschiedene elementare ODFs. Die eine sagt etwas über die darauf verrichtete Wordload der Aktivitäten aus, wie unsere obige $f_{\text{THREADWORK}}$. Die andere ODF berechnet die entfernten Kommunikationskosten der Aktivitäten, die zur Laufzeit entstehen können. Durch bestimmte Gewichtung können diese beiden ODFs zusammengesetzt werden und sie liefern dann eine passende Entscheidung für die Objekte, die an dieser Stelle erzeugt werden.

- ☞ Partielle Ergebnisse können modelliert werden. Die Funktionen brauchen nicht notwendigerweise konkrete Ergebnisse für jede Eingabe zu liefern. Es kann beispielsweise vorkommen, dass die eingesetzten Analyseverfahren zu einer Objekterzeugung die Programmeigenschaft `isProxy` nicht bestimmen können. Daher kann die zugehörige ODF auch keine Verteilungsentscheidung zu dieser Eigenschaft für diese Stelle treffen. Solche Verteilungsentscheidungen können außerdem von anderen ODFs beziehungsweise bei verwendeten Analyseverfahren bei ihrer Durchführung berücksichtigt werden.

- ☞ Die Auswertung der Funktionen kann sowohl vor als auch zur Ausführungszeit erfolgen. Dieses Merkmal ist am Beispiel $f_{\text{THREADWORK}}$ deutlich zu erkennen. Die statischen Analyseverfahren berechnen Abschätzungen über Laufzeitkosten der Aktivitäten und Objekte vor der Laufzeit und setzen diese als Werte in die Programmvariablen in den vorkommenden prädikatenlogischen Formeln ein. Nach der Auswertung dieser Formeln kann die Funktion zu jeder Erzeugungsstelle vor der eigentlichen Ausführung Verteilungsergebnisse berechnen. Bei der Formulierung dieser ODF werden auch Faktoren mit modelliert, die Laufzeitwerte wie die tatsächlichen Ausführungskosten einer Schleife darstellen. Während der Ausführung ergeben sich realistische Größen zu Laufzeitkosten. Diese Größen werden dann zur eventuellen erneuten Auswertung der Funktion verwendet, um eine verbesserte Entschei-

dung für die nächste Erzeugung zu treffen. Diese Möglichkeit verdeutlicht vor allem das Konzept zur Kombination statischer und dynamischer Programmanalysen.

Das Modell, Verteilungsvorschriften als ODFs zu modellieren, ist somit vollständig und sehr flexibel, so dass alle für die Verteilung als relevant geltenden Programm- und Laufzeiteigenschaften mitmodelliert werden können. Es ist aber unabhängig von der Art und Weise, wie und mit welchen Mitteln sie berechnet werden. Außerdem läßt sich der gesamte Verteilungsplan einfach erweitern oder ergänzen, indem neue Verteilungsvorschriften hinzukommen und daraus neue Verteilungsstrategien gebildet werden.

Dieses Kapitel stellt nur die Verwendung der Programm- und Laufzeiteigenschaften als solche in den hier vorgestellten ODFs und Strategien vor. Wie die Programm- und Laufzeiteigenschaften als Variablen in solchen Formeln modelliert werden und wie sie vor sowie zur Laufzeit ausgewertet werden, werde ich im Kapitel 6 beschreiben.

5.2.2. Modellierung der Verteilungsstrategien

Verteilungsstrategien sind im Prinzip auch Funktionen. Sie bestehen aus einer oder mehreren bestimmten Objektverteilungsfunktionen, die bedingt in einer festgelegten Reihenfolge ausgewertet werden. Eine Strategie trifft eine bestimmte strategische Entscheidung über die zu verteilenden Objekte bzw. Aktivitäten. Abhängig von den darin benutzten ODFs kann eine Verteilungsstrategie das Ergebnis liefern, dessen Güte durch abgeschätzte Werte der Programm- bzw. Laufzeiteigenschaften oder auch exakte Werte bestimmt wird. Die Auswertung der Strategien kann sowohl vor der Laufzeit als auch zur Laufzeit erfolgen. Anders als in den ODFs sind die verwendeten Eigenschaften nicht direkt in den Strategien sichtbar. Außerdem besteht eine Schnittstelle zwischen einer Strategie und der Verteilungsdatenbank. Dadurch können vorher getroffene Entscheidungen, die in der Datenbank gespeichert sind, für die Berechnung innerhalb einer Strategie verwendet werden.

Definition 5.4:
Mit den oben definierten Notationen sei eine Strategie wie folgt definiert:

$$strategy_{name} : \mathbb{O} \times \bigotimes_{i \in \mathbb{I}} \mathbb{P}_i \times (\mathbb{O} \rightarrow AR \cup \{\mathsf{noans}\}) \rightarrow (\wp(AR) \times R) \cup \{\mathsf{noans}\}$$

*Wie eine ODF kann eine Strategie auch **total** oder **partiell** sein. Eingegeben werden Objektrepräsentanten, ein Tupel von Funktionen, die alle zur Verteilung relevanten Eigenschaften bestimmen, und eine Funktion zur Lokalisierung der Objekte. Diese werden innerhalb einer Strategie mittels einiger ODFs berechnet und entsprechend der Strategie wird ein Verteilungsergebnis zurückgeliefert.* △

Nach Definition ist eine Verteilungsstrategie nichts anders als eine Objektverteilungsfunktion. Sie besitzt die gleiche Signatur wie ODF. Konzeptionell fungiert eine Verteilungsstrategie bei der automatischen Verteilung als eine strukturelle Zusammensetzung von einzelnen konkreten ODFs. Man hat dadurch den Vorteil, ODFs mit ähnlicher strategischer Bedeutung für die Verteilung zusammen zu gruppieren und anzuwenden. Dies soll durch das folgende Beispiel verdeutlicht werden.

BEISPIEL 5.2:
Es sei eine grundlegende Strategie definiert, die eine lokale Platzierung für bestimmte Objekte berechnet (informelle Beschreibung):

$strategy_{\text{LOCAL}}$: Zu einer gegebenen Objekterzeugungsstelle wird mittels einer Reihe von ODFs berechnet, ob die dort entstehenden Objekte lokale Eigenschaften besitzen und die zugehörigen Erzeugungsstellen deshalb nicht transformiert zu werden brauchen und dann nur lokal erzeugt werden. In dieser Strategie werden die ODFs zur Überprüfung der `Proxy`-, `ThreadLocal`- und der `pure-immutable`-Eigenschaften angewandt. Die zu replizierenden Objekte werden dort lokal erzeugt, wo auf sie zugegriffen wird.

Alle diese ODFs richten sich nach statischen Programmeigenschaften der zu erzeugenden Objekte. Sie können vor der Laufzeit mittels statischer Programmanalyse vollständig bestimmt oder konservativ abgeschätzt werden, so dass sie während der gesamten Programmausführung unverändert bleiben. Objekte, für die diese Strategie erfolgreich angewandt wurde, brauchen nicht mehr weiter betrachtet zu werden. Ansonsten müssen andere Verteilungsstrategien auf die Objektrepräsentanten angewandt werden. Das Ergebnis dieser Strategie ist entweder noremote oder noans. □

Es gibt verschiedene Arten von Strategien zur Verteilung der Aktivitäten und Nicht-Thread-Objekte, wie es in Abbildung 5.3 schon grob angedeutet wurde.

Definition 5.5:
Insgesamt werden Strategien in drei Kategorien aufgeteilt, abhängig von den darin verwendeten ODFs:

Statische Strategien: *Sie liefern feste Verteilungsentscheidungen für entstehende Objekte vor der Laufzeit. Diese Entscheidungen bleiben dann während der gesamten Ausführung des Programms unverändert. Ergebnis solcher Strategien ist entweder* noremote *oder* noans*.*

Basis-Strategien: *Ihre vor der Laufzeit berechneten Entscheidungen können während der Programmausführung aufgrund erneuter Berechnung verändert werden. Das Endergebnis einer Basis-Strategie darf nicht* noans *sein.*

Dynamische Strategien: *Hierbei werden Platzierungsentscheidungen während der Laufzeit getroffen. Ergebnisse der darin enthaltenen ODFs basieren auf Laufzeitdaten.* △

Bei statischen Strategien hängen die Ergebnisse der darin verwendeten ODFs nicht von veränderlichen Laufzeiteigenschaften ab, sondern höchstens von den Laufzeitkonstanten. Die Funktionen dieser Kategorie sind *partiell*. Exemplarisch für solche Strategien ist $strategy_{\mathrm{LOCAL}}$ im Beispiel 5.2. Im Gegensatz dazu sind die Basis-Strategien *totale* Funktionen, d.h. sie müssen zu jeder Eingabe eine strategische Entscheidung liefern und können im Prinzip auch ohne Kombination mit Strategien anderer Klasse angewandt werden.

Abhängig von der Reihenfolge der angewandten ODFs kann eine Entscheidung zu einer Objekterzeugungsstelle eine früher schon getroffene Entscheidung ungültig machen. Dies betrifft vor allem Entscheidungen, die von Basis-Strategien und dynamischen Strategien geliefert werden. Ausnahmsweise gilt das nicht für Entscheidungen zu Objekten, die nicht *remote* sein sollen: Wenn eine ODF innerhalb einer statischen Strategie noremote liefert, wird die Auswertung anderer ODFs abgebrochen und dies gilt als Endergebnis der Strategie. Das heißt, dass solche Platzierungsentscheidungen erhalten bleiben und nicht überschrieben werden.

Ein Beispiel für solche Überschreibungsphänomene ist die Migrationsstrategie: Eine Basis-Strategie lieferte das Ergebnis, dass Objekte an der Erzeugungsstelle $j \in OG$ zusammen mit der Aktivität A_i platziert werden. Zur Laufzeit werden diese Objekte auf der JVM k erzeugt. Durch die Migration werden diese Objekte auf eine andere Maschine umplatziert. Ab diesem Zeitpunkt gilt für die Objekte O_j nur noch die neue Entscheidung, bis das Programm zu Ende läuft.

Die Auswertung einer Strategie impliziert das Auswerten einzelner Verteilungsfunktionen innerhalb dieser Strategie. Hierbei werden alle verwendeten Programmeigenschaften zur Berechnung der ODFs bestimmt. Deren Werte,

inklusive abgeschätzter Werte, werden für die Auswertung der enthaltenen Formeln eingesetzt und schließlich wird die einzelne ODF berechnet.

5.2.3. Modell eines Verteilungsplanes

Ein Verteilungsplan entsteht durch strukturierte Zusammensetzung bestimmter Verteilungsstrategien in einer geeigneten Reihenfolge, wie es in Abbildung 5.3 dargestellt wurde. Die Aufgabe eines Verteilungsplans ist es, vor der Laufzeit zu jeder Objekterzeugungsstelle die Entscheidung über deren Transformation und über eine initiale Verteilung zu liefern sowie zur Laufzeit eine konkrete Zielmaschine zu jeder Objekterzeugung und zur Migration bzw. Replikation zu ermitteln.

Definition 5.6:
Ein Verteilungsplan sei wie folgt definiert:

$$plan_{name} : \mathbb{O} \times \bigotimes_{i \in \mathbb{I}} \mathbb{P}_i \times (\mathbb{O} \rightarrow AR \cup \{\mathsf{noans}\}) \rightarrow (AR \times R)$$

△

Ein Verteilungsplan muss im Prinzip zwei Berechnungsphasen durchführen: In der ersten Phase, vor der Laufzeit, müssen die Ergebnisse für die Transformation und für eine initiale Verteilung berechnet werden. Die Berechnung einer konkreten Zielmaschine als Platzierungsort zu einer Objekterzeugung gehört zur zweiten Phase zur Laufzeit, wobei die Ergebnisse der ersten Phase als Grundlage für die Berechnung dienen.

In der zweiten Phase werden tatsächliche Objekte an Erzeugungsstellen erzeugt und es liegen die erforderlichen Informationen vor. Das sind beispielsweise die Anzahl der JVMs und die Identitäten der erzeugten Objekte. Mittels der Planabfrage bei einer Objekterzeugung wird der Zielort als Verteilungsergebnis zurückgeliefert.

Planevaluation

Die Anwendung der Funktion $plan_{name}(\ldots)$ entspricht der Planevaluation. Dieser Prozess impliziert die Berechnungen der darin enthaltenen Strategien

und ODFs sowie das Bestimmen der nötigen Eigenschaften zur Platzierungsentscheidung. Zur Programmtransformation und zur Berechnung der initialen Verteilung wird ein Verteilungsplan vor der Laufzeit evaluiert. Das wird als *statische Evaluation* bezeichnet. Während der Programmausführung wird der Plan nach einem konkreten Platzierungsort abgefragt. Dies entspricht der *dynamischen Evaluation* eines Planes.

Bei der statischen Evaluation nimmt ein Verteilungsplan Objektrepräsentanten O_i, die Folge aller Programmeigenschaften und $LOCATE$ als Eingabe. Bei der initialen Verteilung vor der Laufzeit müssen zuerst die Aktivitätsrepräsentanten auf die abstrakten JVMs abgebildet worden sein, bevor irgendwelche Verteilungsberechnungen der normalen Objektrepräsentanten durchgeführt werden. Bei der statischen Evaluation werden ebenfalls an bestimmten Programmstellen passende Codesegmente in das Programm eintransformiert: An der Erzeugungsstelle zu O_i enthält der eintransformierte Code die Planabfragen für die spätere dynamische Evaluation. Außerdem können Instrumentierungscode oder andere Arten von Anfragen in das Programm eingebunden werden, beispielsweise zur Objektmigration oder -replikation.

Zur Laufzeit wird mittels der eingebauten Anfragen die dynamische Evaluation des Verteilungsplans ausgelöst. Zu diesem Zeitpunkt sind die Laufzeit- und Architektureigenschaften schon bekannt, beispielsweise die Anzahl der zur Verfügung stehenden JVMs. Daher kann bei der Evaluation der Verteilungsplan zu jeder einzelnen Objekterzeugung eine konkrete Zielmaschine liefern, auf der das Objekt erzeugt wird.

Ein Beispiel

Im Folgenden wird ein Beispiel vorgestellt, das das Konzept der statischen und dynamischen Planevaluation durch die Anwendung der oben definierten Funktionen verdeutlicht. Gegeben sei ein vereinfachtes mehrsträngiges Java-Programm, das im Folgenden im Programmausdruck 1 zu finden ist.

Zur Vereinfachung der Beschreibung werden Objekt- bzw. Aktivitätsrepräsentanten mit den Namen der Variablen bezeichnet, die ihre Referenzen enthalten. In der `main`-Methode der Klasse `Main` werden zwei Aktivitäten `a1` und `a2` erzeugt. Je nach Bedingung operiert `a1` oder `a2` auf `o`, das entweder an der Stelle 28 oder 32 erzeugt wird. `o` wird ebenfalls als Parameter an die Konstruktor-Methode zur Erzeugung eines Objekts `b` an der Stelle 36 über-

```
public class Activity1 extends Thread
{
    ...
    public void set(Data d) { ... }
}

public class Activity2 extends Thread
{
    ...
    public void setData(Data d) { ... }
}

public class Data { ... }

public class OtherData
{
    public OtherData(Data d) { ... }
}

public class Main
{
    public static void main(String[] args) {
        Activity1 a1 = new Activity1(); // {JVM1}     /* {jvm0} */
        Activity2 a2 = new Activity2(); // {JVM2}     /* {jvm1} */
        ...

        if (...) {
            o = new Data(42); // { LOCATE(a1) }     /* {jvm0} */
            a1.set(o);
        }
        else {
            o = new Data(43); // { LOCATE(a2) }
            a2.setData(o);
        }

        OtherData b = new OtherData(o); // { LOCATE(o_28) , LOCATE(o_32) } /* {jvm0} */

        a1.start(); a2.start();
    }
}
```

Listing 1: Verteilungsentscheidungen mittels statischer und dynamischer Evaluation

geben. Es soll des Weiteren ein Verteilungsplan mit folgenden Strategien zur Verteilung dieser Anwendung verwendet werden (informell beschrieben):

„Die Objekte werden nach ihren lokalen Eigenschaften untersucht und so entsprechend verteilt. Dies entspricht der Strategie $strategy_{\mathrm{LOCAL}}$, die im Beispiel 5.2 beschrieben wurde. Anschließend werden die gefundenen Aktivitäten einfach zyklisch auf die JVMs verteilt. Alle restlichen Objekte werden nach der Workload-Strategie verteilt.“ Die statische Evaluation dieses Verteilungsplans liefert folgende Ergebnisse: Alle erzeugten Objekte sind *remote*, d.h. alle erforderlichen Transformationsschritte werden durchgeführt. Die initialen Verteilungsergebnisse werden als Kommentare an jeder Erzeugungsstelle angegeben: Die beiden Aktivitäten können zyklisch auf die abstrakten JVMs, in diesem Fall $JVM1, JVM2 \in AR$ verteilt werden. Es wurde außerdem festgestellt, dass diese Aktivitäten jeweils auf den Objektrepräsentanten o an der Stelle 28 bzw. 32 die meisten Workloads verrichten (mittels der Methode `a1.set(o)` bzw. `a2.setData(o)`). Deshalb wird o zusammen mit $a1$ bzw. $a2$ durch den Hinweis $LOCATE(a1)$ bzw. $LOCATE(a2)$ platziert. Der Objektrepräsentant b operiert ebenfalls mit o, deshalb wird das dort erzeugte Objekt zusammen mit dem gerade an der Stelle von o (28 oder 32) erzeugten Objekt verteilt werden. Da zu diesem Zeitpunkt noch nicht festgestellt werden kann, welches o genau erzeugt wird, wird für b nur ein Lösungsvorschlag ausgerechnet, also $\{LOCATE(o\_28), LOCATE(o\_32)\}$. Im Allgemeinen werden solche Lösungsvorschläge auch nach dem Schema der Datenflussabhängigkeit, beispielsweise zwischen b und o, berechnet.

Vor der Laufzeit werden außerdem Abhängigkeiten zwischen Verteilungslösungen der verschiedenen Erzeugungsstellen aufgelöst. Das bedeutet, dass für o nicht mehr $\{LOCATE(a1)\}$ bzw. $\{LOCATE(a1)\}$, sondern $\{JVM1\}$ bzw. $\{JVM2\}$ und für b die Lösungsmenge $\{JVM1, JVM2\}$ als Verteilungsergebnis zur Verfügung stehen.

Zur Laufzeit seien zwei JVMs zur verteilten Ausführung verfügbar. So werden die abstrakten JVMs $JVM1$ und $JVM2$ auf die beiden echten Maschinen abgebildet, diese seien $jvm0$ und $jvm1$. Die Aktivitäten $a1$ und $a2$ werden entsprechend auf diesen Maschinen erzeugt. Während der Programmausführung sei die Bedingung der `if`-Abfrage erfüllt, d.h. das Objekt an der Stelle 28 wird auf der JVM $jvm0$ erzeugt und mit einer Objektidentität $ObjectId(o\_28)$ gekennzeichnet. Diese Identität und die Information über den Aufenthaltsort des o_28 sind die dynamischen Programmeigenschaften, die sich zur Laufzeit ergeben und von der Datenbank verwaltet werden. Somit läßt sich ein konkreter Verteilungshinweis für das zu erzeugende Objekt b aus der statischen

Lösungsmenge bestimmen, dass es mit dem gerade erzeugten Objekt o (erkannt durch die Objekt-Id) zusammen platziert wird, also auf $jvm0$. Die Verteilungshinweise, die der Plan für einzelne Objekterzeugungen ermittelt, sind an den Erzeugungsstellen im Programmcode (Listing 1) in den Kommentaren (`/* */`) angegeben.

Die Verwendung von Funktionen höherer Ordnung, um die Verteilungspläne zu modellieren, ermöglicht die Evaluation eines Planes vor der Laufzeit und zur Laufzeit. Initiale Verteilungsergebnisse dienen hauptsächlich als Grundlage zur Berechnung konkreter Platzierungsentscheidungen. Zur Verteilung der Objekte zur Laufzeit werden zusätzlich einige dynamische Programmeigenschaften wie die Identität der Objekte bzw. der Aktivitäten benötigt.

Alternative Verteilungspläne

Mit diesem Modell ist es möglich, verschiedene Verteilungspläne zu formulieren, die auf denselben elementaren ODFs beruhen. Sie unterscheiden sich darin, dass jeweils verschiedene Verteilungsstrategien auf unterschiedliche Weise verwendet werden. Die Ergebnisse der initialen Verteilung, insbesondere der Transformation vor der Laufzeit und die konkrete Verteilung zur Laufzeit sind daher je nach verwendetem Plan auch unterschiedlich. Somit kann man für eine Anwendung verschiedene Pläne mit unterschiedlichen Verteilungsstrategien einsetzen.

5.2.4. Andere Komponenten

Neben den Verteilungsplänen gibt es noch zwei andere Komponenten: den Plan-Evaluator und die Verteilungsdatenbank. Im Folgenden werden ihre Funktionalitäten im Zusammenhang mit einem Verteilungsplan vorgestellt.

Plan-Evaluator

Diese Komponente dient hauptsächlich als Schnittstelle zur Kommunikation mit den Verteilungsplänen, ist aber mit diesen sehr eng gekoppelt. Bei Anfragen, beispielsweise vom Programmtransformator, hat der Plan-Evaluator die Eingaben „richtig umzuformulieren“ und leitet dies an einen entsprechenden Plan weiter. Dies löst den Evaluationsprozess aus. Da ein Verteilungsplan nur die Aufgabe hat, Platzierungsentscheidungen für Objekte zu treffen, muss der

Evaluator zusätzlich zu einem Ergebnis Codesegmente zur Planabfrage für die Transformation konstruieren und zurückliefern. Mittels der eingebauten Anfragen im Programm wird der verwendete Plan auch über den Evaluator dynamisch evaluiert.

Verteilungsdatenbank

Die Aufgabe der Verteilungsdatenbank besteht darin, alle strategischen Entscheidungen zur Verteilung der Objekte und Aktivitäten sowie Vorabberechnungen zur Objektmigration bzw. -replikation, die vor der Laufzeit gefallen sind, aufzubewahren. Vor der Laufzeit können eingetragene Datensätze zur Berechnung weiterer Strategien wiederverwendet werden. Alternative Entscheidungen zur Objektverteilung, die aufgrund fehlender Laufzeiteigenschaften vor der Laufzeit nicht vollständig berechnet werden können, werden ebenfalls in die Datenbank aufgenommen. Zur Laufzeit werden einige Strategien erneut berechnet, die Entscheidungen zu offenen Fragen liefern sollen.

5.3. Grundlegende Objektverteilungsfunktionen

Im vorherigen Abschnitt wurden zwei konkrete ODFs eingeführt, die dort vorwiegend als Beispiele dienen. Die Komplexität einer ODF hängt stark von den verwendeten Formeln ab, die Zusammenhänge der Eigenschaften modellieren und als Bedingung zur Berechnung der Verteilungsergebnisse verwendet werden. In diesem Abschnitt werde ich einige grundlegende Objektverteilungsfunktionen vorstellen, bei denen alle relevanten Programmeigenschaften zur automatischen Verteilung verwendet werden und die ausreichen, um daraus gute Verteilungsstrategien entwickeln zu können. Die hier vorgestellten ODFs gliedern sich in drei verschiedene Gruppen: statische ODFs, Basis-ODFs und dynamische ODFs. Wie im vorherigen Abschnitt schon angemerkt, werde ich die hierbei zum Einsatz kommenden Programm- und Laufzeiteigenschaften nur verwenden, aber nicht auf Details eingehen. Diese finden sich im nächsten Kapitel.

Als Eingabe für die ODFs wird ein Objektrepräsentant O_i verwendet. Damit ist gemeint, dass er für alle zur Laufzeit an der Erzeugungsstelle $i \in OG$ zu erzeugenden Objekte gilt, wenn die ODFs vor der Laufzeit evaluiert werden. Hiermit erreicht man eine initiale Verteilung für alle Objekt- und Aktivitätsrepräsentanten. Zur Laufzeit kann eine ODF verschiedene Verteilungsergeb-

nisse für die Objekte ermitteln, die an derselben Erzeugungsstelle i erzeugt werden. Für die Hilfsfunktion $LOCATE$ gilt das Entsprechende.

5.3.1. Statische Verteilungsfunktionen

Es ist allen statischen ODFs gemein, dass sie schon endgültige Verteilungsergebnisse vor der Laufzeit liefern können. Das bedeutet, dass man die Werte für die verwendeten Verteilungsvariablen mittels statischer Analysetechniken ausrechnen und die Formeln vollständig auswerten kann. Außerdem sind alle statischen ODFs partielle Funktionen. Für Objekte, die keine dieser Eigenschaften erfüllen, können auch keine Verteilungsergebnisse geliefert werden.

Objekte mit der Proxy-Eigenschaft

Wie schon beschrieben sollen Erzeugungsstellen, aus denen Objekte mit der *Proxy*-Eigenschaft entstehen, statisch erkannt werden. Diese Stellen bleiben dann vom Transformationsprozess unberührt und die erzeugten Objekte werden dort lokal erzeugt, wo der Programmcode ausgeführt wird. Zur Erinnerung wird hier nochmals die Funktionsdefinition angegeben:

$$f_{\text{PROXY}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (\{\text{here}\}, \text{noremote}) & \text{if} \quad E_{\texttt{isProxy}}(O_i), \\ \text{noans} & \text{else} \end{cases} \tag{5.3}$$

Ein späterer Umzug dieser Objekte auf eine andere JVM während der Programmausführung ist nur möglich, wenn die umgebenden Objekte durch Migration in einen anderen Adressraum umplatziert oder auf mehrere JVMs repliziert werden. In diesem Fall werden die Proxy-Objekte zusammen mit ihren umgebenden Objekten umgezogen oder repliziert, wobei ihre *Proxy*-Eigenschaft immer erhalten bleibt. Die Programmeigenschaft `isProxy` für ein Objekt ist eine statische Eigenschaft und wird vor der Laufzeit mittels geeigneter Analyseverfahren auf konservative Weise ausgerechnet.

Stagewise-immutable Objekte

Objekte, die im ganzen Programmkontext oder ab einem bestimmten Zeitpunkt während der Ausführung nur lesend zugegriffen werden, werden als

stagewise-immutable gekennzeichnet. Ab dem Zeitpunkt, wann sie nur lesend zugreifbar, können sie dann überall *repliziert* werden, wo Zugriffe darauf stattfinden. Nach der Replikation wird auf diese Objekte nur lokal zugegriffen und eine Synchronisation zwischen ihnen ist auch nicht erforderlich. Die *stagewise-immutable* Eigenschaft gliedert sich in zwei Stufen, die im Folgenden vorgestellt werden. Auf Details über diese Eigenschaft sowie über das statische Analyseverfahren dazu wird ausführlich im Abschnitt 6.4 eingegangen.

Rein-konstante Objekte (*pure-immutable*-Eigenschaft) Objekte mit dieser Eigenschaft werden nach ihrer Initialisierungsphase, also nachdem eine Konstruktor-Methode zur Objekterzeugung vollständig abgearbeitet wurde, nur gelesen. Hiernach können sie repliziert werden. Es werden auch keine entfernten Zugriffe für diese Objekte benötigt. Die Definition dieser ODF sieht folgendermaßen aus:

$$f_{\text{PURE-IMMUTABLE}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (M, \mathsf{noremote}) & \text{if} \quad E_{\texttt{isPureImmutable}}(O_i) \\ \mathsf{noans} & \text{else} \end{cases} \tag{5.4}$$

wobei $M = \{LOCATE(O_j) | E_{\mathsf{hasAccess}}(O_j, O_i) \ \wedge LOCATE(O_j) \neq \mathsf{noans}\}$.

Zu einer Objekterzeugungsstelle kann mittels dieser ODF bestimmt werden, ob die dort entstehenden Objekte die *pure-immutable*-Eigenschaft besitzen. Wenn dies der Fall ist, so werden die Objekte dorthin erzeugt, wo andere Objekte Zugriffe auf sie haben. Die Programmeigenschaften `isPureImmutable` und `hasAccess` sind ebenfalls statische Eigenschaften, die durch passende statische Analysetechniken bestimmt werden. Mit Hilfe dieser Funktion können Erzeugungsstellen zur Transformationszeit erkannt werden, die nicht transformiert zu werden brauchen.

***Postponed-immutable* Objekte** Die Programmeigenschaft `isPureImmutable` zu einem Objekt besagt, dass es während seiner gesamten Lebensdauer nur lesend zugegriffen wird. Mit der *postponed-immutable* Eigenschaft ist ein Objekt *ab einem Zeitpunkt* T genau dann *immutable*, wenn alle Zugriffe nach T nur lesend sind. Somit können vor der Laufzeit mittels statischer Analyseverfahren Erzeugungsstellen von solchen *postponed-immutable* Objekten und zusätzlich Programmstellen erkannt werden, die Replikationspunkte bezeichnen. Dann können Replikationsmechanismen an solchen Stellen eingebaut

werden und zur Laufzeit können die Objekte ab diesen Zeitpunkt dorthin repliziert werden, wo auf sie zugegriffen wird.

Objekte, die diese Eigenschaft besitzen, müssen vor ihrer Replikation auch wie alle anderen verteilt werden. Das bedeutet, dass Verteilungsberechnungen für die zugehörigen Erzeugungsstellen durchgeführt werden müssen. Nachdem Platzierungsergebnisse für alle Erzeugungsstellen im Programm vorlagen, werden diese nach der *postponed-immutable* Eigenschaft überprüft. So können Programmstellen für Objektrepräsentanten mit dieser Eigenschaft erkannt werden, die die Replikationspunkte darstellen. Dort werden erforderliche Transformationsschritte durchgeführt. All diese Schritte finden vor der Laufzeit statt.

Nachdem die Objekte zur Laufzeit repliziert wurden, sind auf sie auch keine entfernten Zugriffe mehr nötig. Prinzipiell können sie aber von ihrer Erzeugung bis zu diesem Replikationspunkt entfernt zugegriffen werden. Daher können solche Objekte auch *remote* sein.

Objekte mit Peer- oder Hardware-nahen Eigenschaften (anwendungsspezifische Muster)

Objekte, die Hardware-Ressourcen benutzen, z.B. Grafiken zeichnen, Maschinencode (*native code*) ausführen oder Interaktion mit Benutzern durchführen, sind eng mit dem Adressraum verbunden, auf dem sie erzeugt werden. Eine Verteilung solcher Objekte würde ihre semantische Bedeutung verletzen. Deshalb sollen sie nach ihrer Erzeugung auch dort bleiben und man braucht auch keine Transformation für solche Erzeugungsstellen durchzuführen. Die Funktionsdefinition dafür ist wie folgt:

$$f_{\mathrm{PEER}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (\{\mathsf{here}\}, \mathsf{remote}) & \text{if} \quad E_{\mathtt{isPeer}}(O_i), \\ \mathsf{noans} & \text{else} \end{cases} \tag{5.5}$$

Aus dem ersten Blick sieht diese ODF sehr einfach aus, aber die Berechnung der Programmeigenschaft `isPeer` ist eine sehr komplexe Aufgabe. Denn jede Art der Benutzung der Hardware-Ressourcen stellt ein anwendungsspezifisches Muster dar und all diese Muster fallen in die Kategorie *Peer*-Eigenschaft. Das Transformationsergebnis remote besagt in diesem Fall, dass die zugehörige Klasse transformiert wird. Der Grund ist, dass die Objekte zur Laufzeit direkt here erzeugt werden, ihre Methoden jedoch auch entfernt zugreifbar

sind. Dennoch ist dies eine statisch bestimmbare Programmeigenschaft und man kann sie mit Hilfe der statischen Programmanalyse vollständig ausrechnen.

Thread-lokale Objekte

Es handelt sich hierbei um Objekte, die von einem Thread erzeugt werden und nur von diesem benutzt werden. Aus der Sicht der Programmanalyse entkommen diese Objekte dem Thread nicht, d.h. sie besitzen den *non-escape*-Status bezüglich des Threads. Der Ansatz besteht darin, solche Objekte mit dem Thread zusammen in einem Adressraum zu platzieren, bezüglich dem sie lokal sind. Die ODF hierfür ist:

$$f_{\text{THREADLOCAL}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (M, \text{noremote}) & \text{if} \quad M \neq \emptyset, \\ \text{noans} & \text{else} \end{cases} \tag{5.6}$$

wobei $M = \{LOCATE(A_j) | E_{\texttt{isThreadLocal}}(A_j, O_i)\}$.

Vor der Laufzeit kann zu einer Objekterzeugungsstelle geprüft werden, ob es einen Thread gibt, bei dem die an dieser Stelle erzeugten Objekte thread-lokal sind. Hierfür kann das statische Analyseverfahren Escape-Analyse eingesetzt werden, um die Programmeigenschaft `isThreadLocal` zu berechnen. Der wesentliche Unterschied zwischen dieser ODF und der f_{PROXY} besteht darin, dass die Proxy-Objekte innerhalb ihrer umgebenden Objekte erzeugt werden und keine entfernte Zugriffsmöglichkeit brauchen. Die `isProxy`-Eigenschaft ist somit schärfer als die `isThreadLocal`-Eigenschaft.

Es wird hierbei gewährleistet, dass das Ergebnis der Funktion $LOCATE$ auf einer Aktivität nicht noans ist. Denn in diesem Ansatz werden zunächst die Aktivitäten auf die verfügbaren JVMs verteilt, dann folgen die Nicht-Thread-Objekte.

5.3.2. Basis-Verteilungsfunktionen

Anders als bei den statischen Funktionen lassen sich die Verteilungsvariablen, die die verwendeten Programm- und Laufzeiteigenschaften in den Verteilungsfunktionen dieser Kategorie modellieren, vor der Laufzeit nicht vollständig ausrechnen. Die Analysetechniken müssen, wenn es nötig ist, die

Werte dieser Variablen abschätzen, damit eine initiale Verteilungslösung gefunden werden kann. Ergebnisse einiger dieser Eigenschaften können sich zur Laufzeit noch verbessern lassen. Die Basis-ODFs können auch partielle Funktionen sein und die Objekte, die nach diesen ODFs verteilt werden, sind alle *remote*-Objekte. Das bedeutet, dass eine Transformation für deren Erzeugungsstellen notwendig ist.

Konstruktorparameter

Man betrachtet hierbei eine Erzeugungsstelle, bei der eine Konstruktor-Methode zur Objekterzeugung aufgerufen wird und eine Objektreferenz als Parameter übergeben bekommt. Es wird dann davon ausgegangen, dass die an dieser Stelle erzeugten Objekte auf jeden Fall mit dem übergebenen Objekt über seine Referenz interagieren. Für eine Verteilung bedeutet dies, dass die zu erzeugenden Objekte mit dem übergebenen Objekt zusammen platziert werden können. Diese Programmeigenschaft kann vor der Laufzeit bestimmt werden. Es ergibt sich somit folgende Definition:

$$f_{\text{CONSTRPARAM}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (M, \text{remote}) & \text{if} \quad M \neq \emptyset \\ \text{noans} & \text{else} \end{cases} \tag{5.7}$$

wobei $M = \{LOCATE(O_j) | E_{\texttt{hasParameter}}(O_i, O_j) \wedge LOCATE(O_j) \neq \text{noans}\}$.

Diese ODF funktioniert aber nur für solche Erzeugungsstellen, bei denen der Konstruktor mit einem einzigen Referenz-Parameter aufgerufen wird. Basierend auf diesem Ansatz kann man die ODF erweitern, so dass sie für beliebig viele Parameter mit Referenztypen funktioniert. Dies läßt sich in folgender Funktion zusammenfassen:

$$f_{\text{EX\_CONSTRPARAM}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (M_1, \text{remote}) & \text{if} \quad M_1 \neq \emptyset \\ (M_2, \text{remote}) & \text{if} \quad M_1 = \emptyset \wedge M_2 \neq \emptyset \\ \text{noans} & \text{else} \end{cases} \tag{5.8}$$

wobei:

$M_1 = \{LOCATE(A_j) | E_{\texttt{hasParameter}}(O_i, A_j) \vee (E_{\texttt{hasParameter}}(O_i, \{A_j, A_k\}) \wedge E_{\texttt{Work}}(A_j, O_i) \geq E_{\texttt{Work}}(A_k, O_i) \; \forall A_k \in \mathbb{A}_{\text{static}})\}$,

$M_2 = \{LOCATE(O_l) | E_{\texttt{hasParameter}}(O_i, \{O_l, O_k\}) \wedge E_{\texttt{commCost}}(O_l, O_i) \geq E_{\texttt{commCost}}(O_k, O_i)) \wedge LOCATE(O_l) \neq \text{noans} \, \forall O_k \in \mathbb{O}\}$.

Im Fall, dass Aktivitäten in der Parameterliste übergeben werden, haben sie Vorrang. Das heißt, wenn nur eine Aktivität übergeben wird, werden die dort entstehenden Objekte zusammen mit dieser Aktivität platziert. Wenn aber mehr als eine Aktivität in der Parameterliste auftaucht, werden die zu erzeugenden Objekte zusammen mit der Aktivität verteilt, die die meiste Workload auf diesen Objekten verrichtet. Für den Fall, dass nur normale Objekte, also keine Aktivitäten, übergeben werden, werden die dort erzeugten Objekte mit dem zusammen platziert, das die meisten Methodenaufrufe auf denen durchführt. Bei dieser ODF erkennt man zwei wichtige Programmeigenschaften: `Work`, welche im Beispiel im vorherigen Abschnitt schon vorkommt, und `commCost`, die die Kommunikationskosten zwischen zwei Objektrepräsentanten modelliert. Die Berechnung dieser beiden Eigenschaften geschieht sowohl vor der Laufzeit, ggf. durch geeignete Abschätzungen, als auch während der Ausführung mit konkreten Laufzeitdaten.

Bei dieser ODF ist deutlich zu erkennen, dass zyklische Abhängigkeiten bei der Verteilungsberechnung auftreten können: Das Ergebnis für O_i ist abhängig von $LOCATE(O_l)$ und dies ist noch unbekannt. Wenn diese ODF für O_l angewandt wird und O_i als Kandidaten hat, dann hängt das Ergebnis für O_l wieder von $LOCATE(O_i)$ ab. Um dieses Problem zu lösen ist ein Constraint-Solver einzusetzen. Somit werden zuerst alle Bedingungen aufgestellt, dann werden die Abhängigkeiten zwischen ihnen überprüft. Die Strategien, also auch die ODFs, werden dann zur Berechnung der Verteilungsergebnisse angewandt.

Threadwork

Nach einer Verteilungsstrategie namens *MaxWork*, die in der Diplomarbeit von Klohs beschrieben wurde [Klo02], [KLK02], wird diese ODF weiterentwickelt. Sie dient als eine der grundlegenden Verteilungsfunktionen, die in jedem Fall angewandt werden. Der Ansatz besteht darin, alle Objekte zusammen mit einer Aktivität zu platzieren, die die meiste Workload auf ihnen verrichtet. Dies läßt sich so formulieren:

$$f_{\text{THREADWORK}}(O_i, E, LOCATE) :=$$

$$(\{LOCATE(A_k) | E_{\texttt{Work}}(A_k, O_i) \geq E_{\texttt{Work}}(A_l, O_i) \, \forall A_l \in \mathbb{A}_{\text{static}}\}, \text{remote}). \quad (5.9)$$

Zu Objekten, die aus einer Erzeugungsstelle entstehen, werden die auf ihnen verrichteten Arbeiten aller statisch erkannten Aktivitäten berechnet, welche durch die Programmeigenschaft `Work` modelliert werden. Sie werden miteinander verglichen und man findet dadurch die Aktivität heraus, die die meiste Workload auf den Objekten verursacht. Die Anwendung dieser ODF setzt voraus, dass Verteilungsergebnisse für alle Aktivitäten schon vorliegen. Wie schon beschrieben müssen einige Werte für die Programmeigenschaft `Work` vor der Laufzeit abgeschätzt werden. Dazu zählen beispielsweise Anzahl der Schleifendurchläufe und Verzweigungen, deren tatsächlichen Werte nur bekannt sind, nachdem die zugehörigen Programmfragmente ausgeführt wurden.

Kommunikation zwischen den Aktivitäten

Objekte können nach minimalem Kommunikationsaufwand verteilt werden. Hier wird der Aufwand, der durch die Kommunikation einer Aktivität mit den Objekten entsteht, als Maß verwendet. Eine weitere grundlegende ODF wird so formuliert:

$$f_{\text{THREADCOMM}}(O_i, E, LOCATE) :=$$

$$(\{LOCATE(A_k) | E_{\texttt{commCost}}(A_k, O_i) \geq E_{\texttt{commCost}}(A_l, O_i)\ \forall A_l \in \mathbb{A}_{\text{static}}\}, \text{remote}). \quad (5.10)$$

Die Programmeigenschaft `commCost`, die die Kommunikationskosten einer Aktivität mit den an einer Stelle erzeugten Objekten darstellt, hängt von vielen der Faktoren ab, deren Werte zum Teil vor der Laufzeit abgeschätzt werden müssen, wie die bei `Work`-Eigenschaft. Solche Faktoren sind beispielsweise die Größe der tatsächlich von einer Aktivität bei einem Objektmethodenaufruf übergebenen Parameter, die Größe der Ergebnisse, die Dauer der Serialisierung bzw. Deserialisierung usw. $\texttt{commCost}(A_k, O_i)$ stellt somit die gesamten Kosten dar, die bei der Kommunikation zwischen einer Aktivität A_k und einem an der Stelle i erzeugten Objekt während des gesamten Programmablaufs entstehen.

Verrichtete Arbeiten und Kommunikation der Aktivitäten

Die ODF $f_{\text{THREADCOMM}}$ und $f_{\text{THREADWORK}}$ stellen somit zwei grundlegende Verteilungsvorschriften dar, die bei jeder Erzeugungsstelle der Nicht-Thread-Objekte angewandt werden können. Es ist aber nicht sinnvoll, beide auf eine

Erzeugungsstelle gleichzeitig anzuwenden. Die gelieferten Ergebnisse können miteinander in Konflikt stehen, so dass keine vernünftige Verteilungsentscheidung getroffen werden kann. Trotzdem man kann diese beiden wichtigen Programmeigenschaften zusammen verwenden, so dass gute Verteilungsergebnisse erzielt werden können. Somit läßt sich eine neue Verteilungsfunktion daraus entwickeln, deren Definition folgt:

$$f_{\text{THREADWORKCOMM}}(O_i, E, LOCATE) :=$$

$$(\{LOCATE(A_k) | \frac{E_{\texttt{Work}}(A_k, O_i)}{E_{\texttt{commCost}}(A_k, O_i)} \geq \frac{E_{\texttt{Work}}(A_l, O_i)}{E_{\texttt{commCost}}(A_l, O_i)} \; \forall A_l \in \mathbb{A}_{\text{static}}\}, \text{remote}). \tag{5.11}$$

Die an der Stelle i entstehenden Objekte werden mit der Aktivität zusammen auf eine Zielmaschine hin platziert, die die meiste Workload auf ihnen verrichtet und die Kommunikationskosten durch diese Zusammenplatzierung minimal werden läßt. Somit werden drei grundsätzliche Verteilungsfunktionen zur Verfügung gestellt, die alternativ in einem Verteilungsplan verwendet werden können.

Zyklische Verteilung der Aktivitäten

Prinzipiell muss zuerst eine Verteilung aller Aktivitäten auf die verfügbaren Maschinen vorgenommen werden, bevor irgendeine Basis-Verteilungsfunktion verwendet wird. Nach dem gesamten Konzept werden die Aktivitäten eines mehrsträngigen Programms einmal verteilt und sie bleiben dann in diesen Adressräumen fest. Eine nachträgliche Umplatzierung oder Umzug der Aktivitäten ist nicht erlaubt. Die Platzierung der Nicht-Thread-Objekte richtet sich nach den schon platzierten Aktivitäten. Ein einfacher Ansatz ist es, alle Aktivitäten, die zur Laufzeit erzeugt werden, auf alle an der Lösung beteiligten JVMs zyklisch zu verteilen. Dies kann mittels folgender Funktion erfolgen:

$$f_{\text{DISTRIBUTEDTHREAD}}(A_{i,j}, E, LOCATE) :=$$

$$(\{i + j \mod |AR|\}, \text{remote}) \tag{5.12}$$

$A_{i,j}$ bezeichnet die j-te Aktivität, die an der Erzeugungsstelle i erzeugt wird. Obwohl die konkrete Anzahl der verfügbaren JVMs vor der Laufzeit noch nicht bekannt ist, kann man trotzdem anhand der abstrakten Menge der JVMs AR eine Zuteilung der Aktivitäten zu den abstrakten JVMs schon vor der

Laufzeit durchführen. Diese ODF zieht auch die einzelnen Aktivitäten in Betracht, die aus derselben Erzeugungsstelle i zur Laufzeit erzeugt werden.

Diesen Ansatz kann weiter verfeinert werden, indem die Ausführungskosten der einzelnen Aktivitäten in Betracht gezogen werden, die an derselben Erzeugungsstelle entstehen. In dem Fall, dass mehr als eine Aktivität an einer Erzeugungsstelle entsteht, so werden sie zusammen auf dieselbe JVM platziert. Dies erfolgt nur, wenn ihre Ausführungskosten kleiner gleich $\frac{1}{f}$ der maximalen Ausführungskosten eines Aktivitätsrepräsentanten sind. Dadurch werden mehrere kleinere Arbeitspakete auf einen Rechner zusammen platziert, große Arbeitspakete auf einzelne Maschinen verteilt. Der Faktor f kann durch den Entwickler festgelegt werden, der den Verteilungsplan zusammenstellt. In diesem Fall soll $f = 5$ sein. Das Ganze kann in einer formalen Definition zusammengefasst werden:

$$f_{\text{EX\_DISTRIBUTEDTHREAD}}(A_{i,j}, E, LOCATE) :=$$

$$\begin{cases} (\{i \mod |AR|\}, \text{remote}) & \text{if } 5 * E_{\texttt{execCost}}(A_{i,j}) \leq max\{E_{\texttt{execCost}}(A_k) | A_k \in \mathbb{A}_{\text{static}}\} \\ (\{i+j \mod |AR|\}, \text{remote}) & \text{else} \end{cases} \tag{5.13}$$

Durch die hier verwendete Bedingung bei dieser zyklischen Verteilung der Aktivitäten ist die Lastverteilung etwas gleichmäßiger auf die beteiligten JVMs. Der Preis dafür ist die Komplexität bei der Auswertung der Programmeigenschaften. Außerdem lassen sich die Verteilungsergebnisse dieser ODF zur Laufzeit durch Einsatz hinzukommender Laufzeitdaten verbessern, die bei der Auswertung der Programmeigenschaft `execCost` eine Rolle spielen. Zur Auswertung dieser Funktion vor der Laufzeit wird j mit dem Wert 0 belegt, d.h. alle Aktivitätsrepräsentanten $A_i \in \mathbb{A}_{\text{static}}$ werden einfach auf die abstrakten JVMs zyklisch verteilt. So ähnlich wie `Work` enthält die Eigenschaft `execCost` viele Faktoren wie die Anzahl der Schleifendurchläufe, die Einfluss auf die gesamten Ausführungskosten hat und vor der Laufzeit nicht genau ausgerechnet werden kann, sondern abgeschätzt werden muss.

5.3.3. Dynamische Verteilungsfunktionen

Verteilungsentscheidungen, die aus den Funktionen dieser Gruppe entstammen, basieren vorwiegend auf Laufzeitdaten. Hierzu gehören Vorschriften zur Objektmigration und -replikation. Weißenborn hat sich in seiner Diplomarbeit [Wei03] mit dem Konzept und verschiedenen Techniken zur Migration und Replikation von Objekten beschäftigt sowie Strategien dazu realisiert,

die vollständig auf Laufzeitdaten die Entscheidung treffen. Die hier vorgestellten Verteilungsfunktionen basieren auf einigen Programmeigenschaften, die zum Teil vor der Laufzeit schon vorab berechnet werden und zur Laufzeit die Ergebnisse noch weiter präzisieren. Jedenfalls entsteht mehr Aufwand zur Berechnung dieser Funktionen zur Laufzeit als bei den ODFs anderer Kategorien.

Migration

Der Grund zur Migration eines schon auf einer JVM erzeugten Objektes ist, dass es insgesamt viel mehr entfernte Zugriffe aus einer anderen JVM gibt als auf der JVM, wo sich das Objekt gerade befindet. Deshalb wird dieses Objekt dorthin umgezogen, wo die meisten Zugriffe darauf stattfinden. Die Verteilungsfunktion zur Migration sieht folgendermaßen aus:

$$f_{\text{MIGRATE}}(O_i, E, LOCATE) :=$$

$$\begin{cases} (M, \mathsf{remote}) & \text{if} \quad E_{\texttt{isMigrateable}}(O_i) \\ \mathsf{noans} & \text{else} \end{cases} \tag{5.14}$$

wobei $M = \{LOCATE(O_j) | E_{\texttt{hasMostAccess}}(O_j, O_i) \ \wedge \ LOCATE(O_j) \neq \mathsf{noans}\}$.

Die notwendige Voraussetzung für eine Migration ist, dass die betrachteten Objekte einer Klasse migrierbar sind. Mit Hilfe statischer Analysetechniken kann man die Klassen identifizieren, deren Objekte migriert werden sollen. In der obigen ODF wird dies als die Programmeigenschaft `isMigrateable` ausgedrückt. So werden erforderliche Transformationsmechanismen auf solche Klassen angewandt. $E_{\texttt{hasMostAccess}}(O_j, O_i)$ modelliert den Aspekt, dass von O_j die meisten Zugriffe auf O_i durchgeführt werden. Unter Zugriffen können Methodenzugriffe zum Lesen bzw. Schreiben oder direkte Zugriffe auf Felder verstanden werden. Bei der Bestimmung dieser Eigenschaft müssen ggf. einige Faktoren, beispielsweise über tatsächlich durchzuführende Methodenzugriffe, abgeschätzt werden. Außerdem werden dabei auch Programmstellen identifiziert und Instrumentierungsmechanismen dort eingeführt, damit Daten über Zugriffe zur Laufzeit gesammelt werden und die tatsächliche Migration durchgeführt werden kann. Die Entscheidung für eine tatsächlich stattfindende Migration wird erst zur Laufzeit getroffen, abhängig von den gesammelten Zugriffshäufigkeiten und von den passenden Migrationsstrategien (siehe [Wei03]).

5.4. Spezielle Verteilungsstrategien

Zur automatischen Verteilung müssen die oben formulierten Verteilungsfunktionen in geeigneter Reihenfolge und so geschickt angewandt werden, dass keine der berechneten Ergebnisse in Konflikt zueinander stehen und gute Verteilungsergebnisse erzielt werden. Durch passende Zusammensetzung einer oder mehrerer ODFs entstehen Verteilungsstrategien. ODFs, die zusammen innerhalb einer Strategie verwendet werden, haben ähnliche Merkmale wie z.B. statische oder dynamische Eigenschaften.

Die Reihenfolge, in der die ODFs innerhalb einer Strategie angewandt werden, ist in dem Fall wichtig, wenn Transformationsentscheidungen getroffen werden müssen. Das heißt, dass man ODFs, die entscheiden, dass Objekte lokal platziert werden, immer vor denjenigen anwendet, deren Ergebnisse zu einer entfernten Platzierung führen. Im Folgenden werde ich auf einige spezielle Verteilungsstrategien, jeweils für eine Kategorie nach Definition 5.5, eingehen. Die Kernausage dieses Abschnitts ist, welche ODFs zu einer Strategie zusammengehören. Außerdem wird hier gezeigt, wie einfach es ist, eine Verteilungsstrategie durch Zusammensetzung der ODFs zu definieren. Zur Vereinfachung der Darstellung wird hierbei auf die lange Parameterliste der verwendeten Funktionen verzichtet.

5.4.1. Statische Verteilungsstrategien

Im Abschnitt 5.2.2 haben wir die Strategie im Beispiel 5.2 kennen gelernt, die die Charakteristik dieser Gruppe darstellt und dort informell beschrieben wurde. Das Ziel dieser Strategie ist es, möglichst viele Erzeugungsstellen mit statischen Programmeigenschaften im Programm vor der Laufzeit zu erkennen. Wenn eine der verwendeten ODFs in dieser Strategie zu einer Erzeugungsstelle ein Verteilungsergebnis berechnen kann, dann braucht diese nicht transformiert zu werden. Außerdem sind damit die erzeugten Objekte lokal, so dass eine erhebliche Geschwindigkeitssteigerung der Programmausführung erzielt werden kann. Die Definition dieser Strategie sieht folgendermaßen aus:

$$strategy_{\text{LOCAL}}(O_i, E, LOCATE) :=$$

$$\begin{cases} f_{\text{PROXY}}(O_i, E, LOCATE) & \text{if} \quad f_{\text{PROXY}}(\ldots) \neq \text{noans} \\ f_{\text{THREADLOCAL}}(O_i, E, LOCATE) & \text{if} \quad f_{\text{THREADLOCAL}}(\ldots) \neq \text{noans} \\ f_{\text{PURE-IMMUTABLE}}(O_i, E, LOCATE) & \text{if} \quad f_{\text{PURE-IMMUTABLE}}(\ldots) \neq \text{noans} \\ \text{noans} & \text{else} \end{cases} \tag{5.15}$$

Die ODFs werden wie angegeben in der Reihenfolge von oben nach unten verwendet. Innerhalb dieser Strategie werden die ODFs angewandt, die Ergebnisse für lokale Platzierungsentscheidung der Objekte ermitteln. Das Endergebnis dieser Strategie ist entweder noremote oder noans. Wenn eine ODF innerhalb dieser Strategie noremote liefert, wird die Auswertung weiterer ODFs abgebrochen und dies gilt als das Endergebnis der Strategie. Die beiden ähnlichen ODFs zu `Proxy` und `ThreadLocal` werden deshalb zusammen in einer Strategie in der angegebenen Reihenfolge angewandt, denn es gibt Objekte, die nur die thread-lokale Eigenschaft erfüllen, aber nicht die `Proxy`. Im Gegensatz dazu sind diejenigen auch thread-lokal, wenn sie die `Proxy`-Eigenschaft besitzen. Die ODF zur Immutabilität wird deshalb hierin als Letzte durchgeführt, weil konstante Objekte auch Proxy- oder Threadlocal-Eigenschaft besitzen können und in solchen Fällen für sie auch keine Transformation benötigt wird. Dies ist ein Optimierungsaspekt für die Transformation. Somit kann an Transformationsaufwand für den einzubauenden Programmcode gespart werden, wodurch zusätzliche Laufzeitkosten vermieden werden. Die Strategie LOCAL ist eine *partielle* Funktion. Deshalb muss nach ihrer Anwendung eine weitere *totale* Strategie folgen.

5.4.2. Basis-Verteilungsstrategien

Die Strategien in dieser Gruppe stellen das Fundament eines Verteilungsplans dar. In einem Plan kann nur eine von den als *global* bezeichneten Strategien verwendet werden. Die Funktionen, die die Strategien dieser Kategorie modellieren, sind *totale Funktionen*. Zu dieser Kategorie gehören eine Strategie zur Verteilung der Aktivitäten, welche als Erste angewandt werden muss, und einige Strategien, die globale Verteilungseigenschaften des Programms darstellen. Da alle globalen Strategien ähnliche Charakteristiken besitzen, wird im Folgenden nur auf eine Vertretende eingegangen.

Strategie zur Verteilung der Aktivitäten

Um Aktivitäten verteilen zu können, werden zwei Verteilungsfunktionen benutzt, die zusammen die Strategie $strategy_{\text{THREADS}}$ darstellen. Diese Strategie kann vor der Laufzeit eingesetzt werden, um vorab Verteilungsentscheidung für alle Aktivitätsrepräsentanten zu berechnen. Dadurch gewinnt man zunächst einen groben Überblick über die Aufteilung der *Arbeitsstränge* auf die abstrakten JVMs. Außerdem ist der Einsatz dieser Strategie eine notwendige Voraussetzung, bevor irgendwelche Verteilungsberechnung für normale Objektrepräsentanten vor der Laufzeit vorgenommen werden kann. Die Funktionsdefinition lautet:

$$strategy_{\text{THREADS}}(A_i, E, LOCATE) :=$$

$$\begin{cases} f_{\text{PEER}}(A_i, E, LOCATE) & \text{if} \quad f_{\text{PEER}}(\ldots) \neq \text{noans} \\ f_{\text{EX\_DISTRIBUTEDTHREAD}}(A_{i,j}, E, LOCATE) & \text{else} \end{cases} \tag{5.16}$$

Die ODF f_{PEER} wird zunächst auf die eingegebene Aktivität angewandt, um zu überprüfen, ob sie die *Peer*-Eigenschaft besitzt. Wenn eine Aktivität (ein Thread-Objekt) diese Eigenschaft erfüllt, dann ist sie semantisch nicht mehr eine Aktivität im verteilten Sinn. Wenn beispielsweise speziell ein Thread für die Zeichnen-Aufgabe in einem Rechner erzeugt wird, dann bleibt dieser Thread lokal auf diesem Rechner. Eine Transformation der Erzeugungsstelle ist deshalb auch nicht notwendig. Es lohnt sich auf jeden Fall, zunächst nach der *Peer*-Eigenschaft zu prüfen, weil dadurch nicht nur Transformationsaufwand gespart wird, sondern man auch an Geschwindigkeit durch die Lokalität dieser Aktivität gewinnt.

Anschließend wird eine zyklische Verteilung der Aktivitäten mittels der ODF $f_{\text{EX\_DISTRIBUTEDTHREAD}}$ vorgenommen. Vor der Laufzeit wird der Parameter j (das j-te erzeugte Thread-Objekt an der Stelle i) auf den Wert 0 gesetzt, damit die Aktivitätsrepräsentanten zyklisch auf die abstrakten JVMs verteilt werden können. Zur Laufzeit wird diese Funktion auf einzelne entstehende Threads angewandt.

Globale Strategien

Zur Verteilung normaler Objekte, also andere als Thread-Objekte, kommen so genannte *globale Verteilungsstrategien* zum Einsatz. Eine globale Strategie ist

eine totale Funktion, d.h. sie liefert zu jeder Eingabe (einen Objektrepräsentanten vor der Laufzeit bzw. eine Objekterzeugung zur Laufzeit) ein Ergebnis zur Objektverteilung. Die Anwendung einer solchen Strategie setzt voraus, dass die Aktivitäten schon auf die abstrakten JVMs bzw. auf die realen Maschinen verteilt wurden. Getroffene Entscheidungen von globalen Strategien lassen sich aber zur Laufzeit gegebenenfalls aufgrund erneuter Berechnung verbessern. Dies liegt daran, dass die meisten Programmeigenschaften der verwendeten Basis-ODFs vor der Laufzeit abgeschätzt werden müssen und sie mit neu hinzukommenden Laufzeitdaten erneut berechnet werden können. Eine beispielhafte globale Verteilungsstrategie sei wie folgt definiert:

$$strategy_{\text{COMMWORK}}(O_i, E, LOCATE) :=$$

$$\begin{cases} f_{\text{EX\_CONSTRPARAM}}(O_i, E, LOCATE) & \text{if} \quad f_{\text{EX\_CONSTRPARAM}}(\ldots) \neq \text{noans} \\ f_{\text{THREADWORKCOMM}}(O_i, E, LOCATE) & \text{else} \end{cases} \tag{5.17}$$

Prinzipiell besteht diese Strategie aus zwei verschiedenen Basis-ODFs in der angegebenen Reihenfolge, wobei die zuletzt anzuwendende eine totale Funktion sein muss. Das liegt daran, weil eine globale Strategie eine totale Funktion ist. Die Idee, warum ausgerechnet diese zwei ODFs zusammen verwendet werden, ist folgende: Die erste ODF hat hauptsächlich die Aufgabe, Objektrepräsentanten mit der Eigenschaft *Konstruktorparameter* „herauszufiltern“, bevor eine Verteilungsentscheidung für sie mittels der ODF $f_{\text{THREADWORKCOMM}}$ berechnet wird. Diese übernimmt die Hauptaufgabe, Verteilungsentscheidungen für Objektrepräsentanten mit Hilfe der *Work*- und *commCost*-Programmeigenschaft zu berechnen.

Anstelle der ODF $f_{\text{THREADWORKCOMM}}$ kann eine andere, totale Basis-Verteilungsfunktion, z.B. $f_{\text{THREADWORK}}$ oder $f_{\text{THREADCOMM}}$ verwendet werden. Daraus entwickelt sich jeweils eine neue Verteilungsstrategie, die eine totale Funktion darstellt und zur initialen Verteilungsberechnung benutzt werden kann. Da jede dieser Funktionen eine eigenständige Strategie zur Verteilung der Objekte modelliert, darf jeweils nur eine für eine Programmausführung angewandt werden. Im Beispiel 5.2.3 auf der Seite 91 wurde diese Strategie zusammen mit der Strategie $strategy_{\text{LOCAL}}$ in einem Plan verwendet.

5.4.3. Strategien mit dynamischen Entscheidungen

Verteilungsfunktionen, die Berechnung zur Migration bzw. Replikation von Objekten durchführen, werden in Strategien dieser Kategorie formuliert. Die

Aufgabe, zur Laufzeit schon platzierte Objekte von einer JVM auf eine andere umzuziehen bzw. auf mehrere JVMs deren Replikaten zu erstellen, wird während der Programmausführung erledigt. Zur Unterstützung dafür können vor der Laufzeit einige Maßnahmen und Vorabentscheidungen getroffen werden, damit so wenig Berechnungen wie möglich zur Laufzeit durchgeführt werden müssen. Dazu gehören u.a. statische Analysen zur Bestimmung der Art der Objektzugriffe, ob Lese- oder Schreibzugriffe stattfinden sowie Abschätzungen der Anzahl der Zugriffe. Dennoch besteht immer noch Rechenaufwand zur Informationserfassung während des Programmablaufs, um zu endgültigen Entscheidungen zu kommen.

Zusammen mit Strategien anderer Kategorien können diese Strategien zur weiteren Geschwindigkeitssteigerung verwendet werden. Vor der Laufzeit treffen sie Entscheidungen zu notwendigen Transformationen und liefern Entscheidungsvorschläge für die Migration bzw. Replikation. Zur Laufzeit werden die Lösungsvorschläge bei genügend vorliegenden Informationen umgesetzt. In dieser Gruppe finden sich zwei spezielle Verteilungsstrategien: Die eine realisiert Replikation von Teil-Immutable-Objekten und die andere liefert Entscheidungsvorschläge, ob und wohin Objekte migriert werden können.

Strategie zur Replikation

Obwohl diese Strategie als dynamische Strategie bezeichnet wird, sind aber die Programmeigenschaften der verwendeten ODF statisch. Das heißt, dass man sie schon vor der Laufzeit vollständig ausrechnen kann. Dynamisch in diesem Sinn ist, dass zur Laufzeit die entstehenden Objekte mittels anderer Strategien schon platziert wurden, und durch diese ihre ursprüngliche Platzierungsentscheidung geändert wird. Nachdem initiale Platzierungshinweise für alle Objekt- und Aktivitätsrepräsentanten schon ausgerechnet wurden, können diese Repräsentanten nach der `postponed-immutable` Eigenschaft überprüft werden. Dadurch werden vor der Laufzeit Replikationspunkte gefunden und Transformationsschritte dort angewandt. Zur Laufzeit werden an solchen Replikationsstellen die Objekte repliziert.

Strategie zur Migration

Die strategische Entscheidung über die Migration der Objekte basiert auf der Grundlage, dass ein schon platziertes Objekt häufig von einem entfernten

Rechner zugegriffen wird. Diese Fernzugriffe können von mehreren Objekten oder Aktivitäten stammen, die sich auf einem anderen Rechner befinden. Aus technischen Gründen muss vor der Laufzeit entschieden werden, ob Objekte einer Klasse zur Laufzeit migriert werden dürfen und daher notwendige Transformationsschritte durchzuführen sind. Zur Bestimmung des Migrationsziels muss die Strategie notwendigerweise vor der Laufzeit alle möglichen Zugriffe auf Objektrepräsentanten und die Art der Zugriffe ausrechnen. Somit kann abgeschätzt werden, welche Aktivitäts- bzw. anderen Objektrepräsentanten die meisten Zugriffe durchführen. So können Zielvorschläge für die Migration als Lösung bestimmt werden. Diese Strategie enthält eine einzige ODF und sei wie folgt definiert:

$$strategy_{\text{MIGRATE}}(O_i, E, LOCATE) :=$$

$$f_{\text{MIGRATE}}(O_i, E, LOCATE) \tag{5.18}$$

Basierend auf den statischen Lösungsvorschlägen aus dieser Strategie werden endgültige Migrationsentscheidungen durch die anderen Komponenten getroffen, die in Abbildung 5.3 als übereinander gestaffelte Rechtecke bei Migrations-/Replikationsstrategien visualisiert und ausführlich in der Diplomarbeit von Weißenborn [Wei03] beschrieben wurden. Die Informationserfassungskomponente sammelt zur Laufzeit zusätzlich zu den statischen Entscheidungsvorschlägen echte Zugriffsdaten für die zu migrierenden Objekte und bestimmt daraus den eigentlichen Zielort für den Objektumzug. Entscheidungen, ob Objekte nur einmal migriert werden dürfen oder die Replikate durch Schreibzugriffe gelöscht oder aktualisiert werden, übernehmen die Komponenten von Weißenborn. Die technischen Schritte zur Migration bzw. Replikation werden auch von der Transfer-Komponente durchgeführt. Wie die Replikationsstrategie ist dies auch eine partielle Funktion.

5.5. Konkrete Verteilungspläne

In diesem Abschnitt wird an einem vollständigen Verteilungsplan gezeigt, wie im Allgemeinen konkrete Verteilungspläne, bestehend jeweils aus einigen ausgewählten speziellen Strategien, formuliert werden können. Alle Berechnungen basieren auf denselben Datenbestandteilen: Das sind die Programm- und Laufzeiteigenschaften, die Strategien und die Datensätze in der Verteilungsdatenbank. Sie unterscheiden sich voneinander nur durch die darin

eingesetzten Verteilungsstrategien und können verschiedene Verteilungsentscheidungen zu einem Programm liefern. Im Prinzip haben alle Verteilungspläne den kompletten Überblick über das Eingabeprogramm. Ein Plan wird statisch und dynamisch evaluiert. Nach der statischen Evaluation liegen alle initialen Verteilungsentscheidungen vor und das Programm wird komplett zur verteilten Ausführung transformiert. Zur Laufzeit wird dynamisch evaluiert, wobei die eigentlichen Platzierungshinweise vom Verteilungsplan geliefert werden und ggf. einige Verteilungsentscheidungen erneut getroffen werden.

Für den vorzustellenden Plan $plan_{\text{WORKCOMM}}$ folgt zunächst eine textuelle Beschreibung:

„Wenn die lokale Strategie keine Antwort liefert, verteile Objekte nach der Strategie „verrichtete Workload und Kommunikation". Strategien zur Migration und Replikation werden ebenfalls angewandt: Wenn noans *geliefert wird, wird das vorher berechnete Verteilungsergebnis beibehalten."*

Die Definition dieses Planes sieht folgendermaßen aus:

$plan_{\text{COMMWORK}}(O_i, E, LOCATE) :=$

$$\begin{cases} strategy_{\text{THREADS}}(O_i, E, LOCATE) & \text{if} \quad O_i \in \mathbb{A}_{\text{static}} \\ strategy_{\text{LOCAL}}(O_i, E, LOCATE) & \text{if} \quad strategy_{\text{LOCAL}}(\ldots) \neq \text{noans} \\ strategy_{\text{COMMWORK}}(O_i, E, LOCATE) & \text{else} \end{cases} \tag{5.19}$$

In dieser Strategie werden vor der Laufzeit alle vorkommenden Aktivitätsrepräsentanten auf die abstrakten JVMs verteilt. Somit hat man einen Überblick über die Platzierungsziele der Threads. Das ist eine notwendige Voraussetzung für alle formulierten Verteilungspläne, bevor irgendwelche Verteilungsberechnung der normalen Objekte stattfindet. Alle Objekte bzw. Objektrepräsentanten, die bei allen vorgegebenen Strategien „durchkommen", werden mittels der $strategy_{\text{COMMWORK}}()$ dorthin zu den Aktivitäten platziert, wo die meiste Workload auf ihnen verrichtet und viel mit ihnen kommuniziert wird.

In der Funktionsdefinition des obigen Plans wurden keine der beiden Strategien zur Migration und Replikation angegeben. Denn diese Programmeigenschaften werden nachträglich berechnet, nachdem Verteilungshinweise aller Objektrepräsentanten vorlagen. Die Strategie zur Migration ist eine partielle Funktion und kann auch noans als Ergebnis liefern. Daher wird die vorherige Entscheidung mittels $strategy_{\text{LOCAL}}$ und $strategy_{\text{COMMWORK}}$ beibehalten, wenn die Strategie zur Migration keine Aussagen macht. Außerdem muss be-

achtet werden, dass Objekte nach ihrer Replikation nicht mehr migriert werden dürfen. Diese Strategien können eingesetzt werden, um eine Geschwindigkeitssteigerung der Programmausführung zu erzielen.

Dies ist ein vollständiger Verteilungsplan, der zur automatischen Verteilung jedes mehrsträngigen Java-Programms eingesetzt werden kann. Im Allgemeinen unterscheiden sich die Verteilungspläne durch die Anwendung verschiedener Strategien und die übergebenen Programmeigenschaften sowie Laufzeiteigenschaften. Das bedeutet, dass ein neuer Plan durch das Hinzufügen, Entfernen und Ersetzen der verwendeten Strategien einfach formuliert werden kann. Beim obigen Plan kann man beispielsweise die Strategien zur Migration und Replikation weglassen und $strategy_{\text{COMMWORK}}()$ durch eine andere Basis-Strategie, z.B. $strategy_{\text{THREADCOMM}}()$, ersetzen. So entsteht ein neuer, einfacherer Verteilungsplan, der evtl. genauso gut für alle *multithreaded* Java-Anwendungen funktioniert.

Durch das vorgestellte Konzept ist man in der Lage, neue Objektverteilungsfunktionen zu formulieren, verschiedene Varianten der Strategien anzuwenden und somit unterschiedliche Pläne zur Verteilung eines Programms einzusetzen. Dadurch kann die Güte der Verteilung evaluiert werden. Die Modellierung der Programmeigenschaften in Formeln, deren Bewertung die Verteilungsergebnisse entscheiden, ermöglichen einem, die Programmanalyse von der Berechnung der Verteilungsfunktion zu trennen, und dennoch statische und dynamische Analysetechniken kombinieren zu können.

6. Analysetechniken zur automatischen Verteilung

Inhalt

Statische Programmeigenschaften, welche zur Entscheidung der Transformation sowie der grundlegenden Verteilungsberechnung beitragen, weisen bestimmte Charakteristiken auf. Sie können durch systematische Anwendung geeigneter statischer Analyseverfahren bestimmt werden. Durch den Einsatz der Analyseverfahren wie zum Beispiel Klassenhierarchie-, Escape- oder Aufrufgraphanalyse sowie Analysen zur Berechnung von Definitions/Verwendungsketten können solche elementaren Charakteristiken ermittelt werden, auf denen viele statische Programmeigenschaften basieren. Damit können vor der Laufzeit Entscheidungen getroffen werden, ob entstehende Objekte zusammen gruppiert und nur lokal erzeugt, oder ob sie wegen ihrer Beziehung zusammen platziert werden sollen. Die zur Migration und Replikation erforderlichen Maßnahmen müssen statisch vorbereitet sein: die Instrumentierungsmechanismen zum Zählen der Zugriffe und bestimmte Migrations- oder/und Replikationshinweise sind beispielsweise in das Programm einzutransformieren. Die elementaren Charakteristiken der statischen Programmeigenschaften werden im ersten Abschnitt beschrieben.

Einige fundamentale Programmeigenschaften, die die Ausführungs- und Kommunikationskosten charakterisieren, sind dynamische Eigenschaften. Diese Kosten sind Grundsteine der Verteilungsberechnung für alle Objekte und Aktivitäten und können vor der Laufzeit nicht genau berechnet werden. Deshalb muss der dynamische Anteil dieser Kosten abgeschätzt werden. Anhand des hier angegebenen Kostenmodells kann *hybride Analyse* angewandt werden, um Ergebnisse der Kostenabschätzung, somit auch der Verteilung, zur Laufzeit zu verbessern. *Hybride Analyse* ist eine Kombination statischer und dynamischer Analyse. Statische Techniken werden dazu benutzt, um dynamische Anteile der Kosten vor der Laufzeit abzuschätzen. Zur Laufzeit werden einige abgeschätzte Werte mittels dynamischer Analyse genau bestimmt. Das führt wiederum zur erneuten Berechnung der Ausführungskosten, was die Verteilung zukünftig zu erzeugender Objekte positiv beeinflusst. Dafür müssen die dynamischen und statischen Aspekte der Kosten genau modelliert werden. Der zweite Abschnitt stellt dieses Vorhaben dar.

Anschließend werde ich einige zur automatischen Verteilung relevante Analysetechniken vorstellen und ihre Anwendung auf die modellierten Programmeigenschaften beschreiben. Das sind eine erweiterte Version der Aufrufgraph-Analyse (*PreciseCallGraph-Analyse*) und die interprozedurale Def-Use-Analyse (*GlobalDefUse-Analyse*). Mit Hilfe der *PreciseCallGraph-Analyse* können dynamisch gebundene Aufrufziele mittels der verwendeten Information zur Typinferenz genauer ermittelt werden. Durch die

GlobalDefUse-Analyse können zu den Erzeugungsstellen ihre Definition-Verwendung-Ketten im gesamten Analysekontext berechnet werden. Diese Information wird fluss- und kontext-insensitiv über Methodengrenzen hinaus verfolgt. Hierbei wird die Technik der *copy-propagation* angewandt. Für formale Parameter und Ergebnisse werden erreichende Definitionen über alle relevanten Aufrufstellen gesammelt.

Der Abschnitt 6.4 beschäftigt sich mit einem besonderen Analyseverfahren, der *stagewise-immutable Analyse*. Diese statische Analysetechnik habe ich entworfen, um Erzeugungsstellen vor der Laufzeit zu erkennen, an denen entstehende Objekte zur Laufzeit repliziert werden können. *Bezogen auf den Programmablauf* wird somit charakterisiert, *in welchen bestimmten Phasen während der Programmausführung* die Lese- und Schreibzugriffe auf ein Objekt stattfinden. Objekte mit der *stagewise-immutable* Eigenschaft werden ab einem bestimmten Zeitpunkt T bei der Programmausführung nur lesend zugegriffen. Nach diesem Zeitpunkt können sie dann repliziert werden. Nach der Replikation werden Lesezugriffe zu lokalen Zugriffen, so dass entfernte Kommunikation dadurch eingespart wird. Abschließend werde ich verwandte Arbeiten im Umfeld der Programmanalysen, insbesondere für den Einsatz zur automatischen Verteilung diskutieren.

Die statische Analyse berechnet die Programmeigenschaften auf der Grundlage von Objekterzeugungsstellen. An einer Erzeugungsstelle gelten die Ergebnisse für alle Objekte, die dort zur Laufzeit entstehen.

6.1. Charakteristiken der statischen Programm- und Laufzeiteigenschaften

Wie im Abschnitt 5.1.1 schematisiert gliedern sich die Eigenschaften, die zur Berechnung der Objektverteilung relevant sind, in drei Kategorien: *statische Programmeigenschaften*, *dynamische Programmeigenschaften* und *Laufzeiteigenschaften*. Im Folgenden werde ich die Gemeinsamkeit sowie die Unterschiede der statischen Programmeigenschaften beschreiben. Durch diese Identifizierung und Klassifizierung können geeignete statische Analyseverfahren dafür eingesetzt werden, die im folgenden Abschnitt ebenfalls beschrieben werden.

Im Gegensatz dazu stellen die *dynamischen Programmeigenschaften* die Laufzeit- und Kommunikationskosten des Programms dar. Teile dieser Kosten müssen vor der Laufzeit abgeschätzt werden. Daher ist eine geeignete Mo-

dellierung dieser Programmeigenschaften in Rahmen eines passenden Kostenmodelles erforderlich. Dieses Modell spiegelt das Konzept *hybrider Analyse* - also eine Kombination statischer Analyseergebnisse mit dynamischen Ergebnissen - wider. Der Übersichtlichkeit wegen wird diese Konzeption einzeln im folgenden Abschnitt beschrieben.

Unabhängig vom Anwendungsprogramm gibt es eine Reihe von Eigenschaften, die die Verteilung der Objekte während der Ausführung beeinflussen. Sie müssen bei der Verteilungsberechnung vor der Laufzeit ebenfalls mit berücksichtigt werden. Die Identifizierung und Charakterisierung solcher Laufzeiteigenschaften schließen diesen Abschnitt ab.

6.1.1. Statische Programmeigenschaften

Allen statischen Programmeigenschaften ist gemein, dass ihre Analyseergebnisse vor der Laufzeit vollständig durch passende Verfahren ausgerechnet bzw. abgeschätzt werden. So kann die Berechnung bestimmter ODFs erfolgen, deren Ergebnisse für die Transformation zur Verfügung stehen müssen. Im Folgenden wird auf einzelne Merkmale der statischen Programmeigenschaften eingegangen.

Eigenschaften zur Lokalität

Die statischen Programmeigenschaften $E_{\texttt{isProxy}}(O_i)$ und $E_{\texttt{isThreadLocal}}(A_j, O_i)$ weisen auf die Lokalität der Objekte hin. Objekte, die diese Eigenschaften besitzen, werden mit anderen Objekten bzw. Threads zusammen auf eine virtuelle Maschine platziert. Das heißt, dass alle Zugriffe von umgebenden Objekten bzw. zugehörigen Threads darauf nur innerhalb eines Adressraumes stattfinden. Alle Zugriffe auf Objekte mit derartigen Eigenschaften werden lokal durchgeführt, so dass sie nicht *remote*-Objekte zu sein brauchen.

Gemeinsamkeit und Unterschiede zwischen `isThreadLocal` und `isProxy` Den beiden statischen Programmeigenschaften `isProxy` und `isThreadLocal` ist gemein, dass Objekte mit diesen Eigenschaften ihrem umgebenden Objekt bzw. ausführenden Thread nicht entkommen. Alle Methodenaufrufe und direkte Feldzugriffe über ihre Referenzen müssen innerhalb ihrer umgebenden Objekte bzw. Threads stattfinden.

Die Referenz zu einem Objekt mit der `isProxy`-Eigenschaft darf nur seinem umgebenden Objekt - also seinem Proxy-Objekt - bekannt sein. Diese Referenz darf also auf keine Weise, z.B. durch Seiteneffekte eines Methodenaufrufs, nach außen sichtbar werden. Alle Zugriffe über diese Referenz müssen vollständig vom Proxy-Objekt gekapselt werden. Diese Eigenschaft gilt konservativerweise nicht für Objekte, die über den *reflection*-Mechanismus erzeugt werden. Denn durch *reflection* wird sogar externer Zugriff auf private Instanzvariablen ermöglicht. Im Prinzip erfüllt ein Objekt die `isProxy`-Eigenschaft nur, wenn sowohl seine Erzeugung (nicht über Reflexion) als auch jeder Zugriff darauf nur innerhalb von Methoden des umgebenden Objekts erfolgt.

Bei der Eigenschaft $E_{\texttt{isThreadLocal}}(A_j, O_i)$ gilt die folgende Bedingung: O_i ist A_j thread-lokal, wenn es keine andere Aktivität $A_l \neq A_j$ gibt, so dass A_l Zugriffe auf O_i durchführt. Im Vergleich zu der `isProxy`-Eigenschaft kann die Referenz eines thread-lokalen Objekts zwar über Parameter oder das Ergebnis eines Methodenaufrufs ihrem umgebeneden Objekt entkommen. Sie darf aber nicht dem Thread entkommen, in dem es erzeugt wird. Aufgrund der Gemeinsamkeit dieser beiden statischen Programmeigenschaften kann das statische Verfahren zur Escape-Analyse eingesetzt werden.

$E_{\texttt{isPeer}}(O_i)$ Objekte, die die `isPeer`-Eigenschaft erfüllen, sind wegen beispielsweise der Benutzung von Hardware-Ressourcen fest im Adressraum verankert, in dem sie erzeugt werden. Peer-Objekte stellen in der Regel anwendungsspezifische Muster dar, die die Nutzung von Hardware-Ressourcen (Netzwerk-Ressourcen), das Zeichnen von Grafiken oder GUI-Komponenten (mittels Java-AWT), die Ausführung von Maschinencode oder Funktionen dynamischer Bibliotheken (*native code*) oder die Benutzerinteraktionen beschreiben. Anhand dieser Muster kann eine statische Analysetechnik die Erzeugungsstellen für Objekte mit dieser Eigenschaft erkennen, indem u.a. an allen Aufrufstellen ihrer Methoden überprüft wird, ob ein potenzielles Aufrufziel eine als *native* deklarierte Methode ist. Die Methoden solcher Objekte sind aber trotzdem *remote* zugreifbar. Daher ist die Transformation der zugehörigen Klasse erforderlich.

Relation zwischen Objekten

Die beiden elementaren, statischen Programmeigenschaften `hasAccess` und `hasParameter` beschreiben die Beziehung zwischen verschiedenen Objekt-

en, die ggf. in einer konservativen Weise abgeschätzt wird. Anders als bei den oben vorgestellten Programmeigenschaften werden die in Beziehung stehenden Objekte nicht notwendigerweise zusammen platziert. Die Eigenschaft stellt ein schwächeres Kriterium zur Objektgruppierung dar.

$E_{\texttt{hasAccess}}(O_j, O_i)$ stellt die Relation zwischen O_j und O_i dar, wobei O_j direkt auf O_i zugreift. Die Zugriffsrelation wird auf eine konservative Weise wie folgt festgelegt:

O_j *„kennt“* O_i: O_j erhält die Referenz auf O_i, indem beispielsweise O_j O_i erzeugt oder O_j die Referenz zu O_i als Parameter oder Ergebnis eines Methodenaufrufs bekommt und

- O_j *calls* O_i: In den verwendeten Methoden von O_j wird mindestens eine Aufrufstelle einer nicht-statischen Methode über die Referenz zu O_i gefunden (fluss-insensitiv), oder
- O_j *greift direkt auf ein Feld von* O_i *zu*.

$E_{\texttt{hasParameter}}(O_i, \{O_j, \ldots, O_k\})$ ist ein spezieller Fall der Programmeigenschaft hasAccess. Hierbei wird an der Erzeugungsstelle von O_i die aufgerufene Konstruktor-Methode darauf hin analysiert, ob Referenzen zu $O_j \ldots O_k$ als Parameter übergeben werden. Hierfür benötigt man die Information über interprozedurale erreichende Definition von $O_j \ldots O_k$, welche mittels der interprozeduralen Def-Use-Analyse (siehe Abschnitt 6.3.2) beschafft wird.

Eigenschaften zur Migration bzw. Replikation

Die notwendige Voraussetzung für eine Migration bzw. Replikation eines Objekts bei diesem Konzept ist, dass es ein passives Objekt sein muss. Das bedeutet, dass es keine Instanz von Thread oder eine Unterklasse davon sein darf. Genau dies beschreibt die statische Programmeigenschaft $E_{\texttt{isMigrateable}}(O_i)$. Mit Hilfe von Klassenhierarchie-Information kann man dies einfach prüfen. Auf Objekte mit dieser Eigenschaft können zur Laufzeit Migrationsstrategien nach [Wei03] angewandt werden.

$E_{\texttt{isStagewiseImmutable}}(O_i)$ Objekte, die diese Programmeigenschaft aufweisen, werden ab einem bestimmten Zeitpunkt T während der gesamten Ausführung des Programms nur lesend zugegriffen. Die *stagewise-immutable* Eigenschaft wird in diesem Konzept in zwei verschiedene Kategorien unterteilt: *pure-immutable* und *postponed-immutable*.

Der Zeitpunkt T bei der *pure-immutable* Eigenschaft ist nach der Objektinitialisierung. Das heißt, dass der Zustand eines *pure-immutable* Objekts bis zum Ende der Ausführung unverändert bleibt, nachdem eine zur Erzeugung aufgerufene Konstruktor-Methode komplett abgearbeitet wurde, also nach seiner Initialisierungsphase. Um diese Eigenschaft für ein Objekt feststellen zu können, muss man vor der Laufzeit die Definitionsstelle dieses Objekts, also seine Erzeugungsstelle, als Ausgangspunkt betrachten: Alle potentiellen Verwendungsstellen des an dieser Stelle entstehenden Objekts, die auch über Methodengrenzen hinaus gehen, werden darauf hin untersucht, ob die Zugriffe auf seine Felder dort, z.B. über Methodenaufrufe oder direkt zugegriffen, lesende oder schreibende Zugriffe darstellen. Ein Objekt besitzt genau dann die *pure-immutable* Eigenschaft, wenn alle Verwendungsstellen seine in den Feldern gespeicherten Daten nur lesen.

Bei der *postponed-immutable* Eigenschaft wird ein Objekt nach seiner Initialisierungsphase noch schreibend zugegriffen, ab einem Zeitpunkt T in seiner Verwendungsphase bis zum Programmende aber dann nur gelesen. Die Immutabilität solcher Objekte wird sozusagen auf einen späteren Zeitpunkt nach seiner Initialisierung verschoben. Auf Details über diese Eigenschaft und die Analysetechnik dazu wird ausführlich im folgenden Abschnitt 6.4 eingegangen.

Objekte, die eine dieser Eigenschaften erfüllen, können ab dem passenden Zeitpunkt auf allen JVMs repliziert werden. Dadurch können viele entfernte Kommunikationen gespart werden.

6.1.2. Architektur- und Laufzeiteigenschaften

Anzahl der verfügbaren JVMs

Die wichtigste Eigenschaft unter dieser Kategorie, die anwendungsunabhängig ist und trotzdem großen Einfluss auf die eigentliche Platzierung der zu erzeugenden Objekte zur Laufzeit hat, ist die Anzahl der verfügbaren virtuellen Maschinen. Die tatsächliche Anzahl der beteiligten JVMs ist erst zur Laufzeit, unmittelbar vor dem Start der Programmausführung, bekannt. Dennoch

wird diese Architektureigenschaft bei der Verteilungsberechnung der Objekte vor der Laufzeit in fast allen ODFs verwendet.

Statisch müssen alle ODFs die Laufzeitkonstante, die die Anzahl der abstrakten JVMs darstellt, für ihre Berechnungen verwenden. Somit kann vor der Laufzeit eine relative Zuordnung der Objekte und Threads zu den abstrakten JVMs berechnet werden. Nachdem die verteilte Laufzeitumgebung gestartet wurde, wird die Zahl der tatsächlich verfügbaren JVMs bekannt. So können die abstrakten JVMs auf die realen abgebildet werden. Diese Laufzeitkonstante bleibt während einer gesamten Programmausführung unverändert. Für die nächste Ausführung kann sie aber einen anderen Wert annehmen, z.B. wenn sich mehr Rechner an der verteilten Ausführung beteiligen sollen.

Rechner- und Netzwerkeigenschaften

Es gibt außerdem noch eine Reihe von Eigenschaften, die direkt die verteilte Ausführung des Programms beeinflussen, aber nicht in der Verteilungsberechnung der Objekte berücksichtigt werden können. Zu nennen sind u.a. die Auslastung der CPUs, auf denen die JVMs gestartet werden. Die Latenz des Netzes spielt bei der entfernten Kommunikation während der Ausführung eine wichtige Rolle, denn durch hohe Latenz kann die gesamte Performanz erheblich beeinträchtigt werden. Die Netzstruktur hat ebenfalls große Auswirkung auf die entfernte Kommunikation während der Programmausführung. Da die Strukturierung des Netzes nicht bei der Platzierungsentscheidung berücksichtigt wird, kostet die Kommunikation während des Programmablaufs bei direkter Vernetzung der Rechner weniger als z.B. bei einer baumartigen.

Es ist technisch machbar, dass einige Faktoren wie die CPU-Last oder die Netzwerktopologie in der verteilten Laufzeitumgebung zur Platzierung der Objekte berücksichtigt werden. Hierbei könnten die verschiedenen Aktivitäten basierend auf der vor der Laufzeit berechneten Workload auf die Rechner je nach CPU-Last verteilt werden. So können zwei Aktivitäten mit geringem Workload zusammen auf einem Rechner platziert, aber eine Aktivität mit größerem Load allein auf einem Rechner erzeugt werden. Bei der empirischen Untersuchung der Verteilung werden die Aktivitäten zusammen mit ihren Datenobjekten einfach zyklisch auf die verfügbaren Rechner verteilt.

Man kann außerdem verschiedene Netztopologien über die Kommunikationsschicht KaRMI, die der verteilten Laufzeitumgebung zugrunde liegt, konfigurieren. So könnte die Hardware-Infrastruktur der vernetzten Rechner wie

z.B. Myrinet/GM damit direkt ausgenutzt werden, so dass der Kommunikationsaufwand reduziert wird. Es könnte einiges an Performanz für die verteilte Ausführung bringen. Solche dynamischen Aspekte werden hierbei aber nicht weiter vertieft untersucht.

6.2. Dynamische Programmeigenschaften und das Kostenmodell zur Objektverteilung

Dynamische Programmeigenschaften modellieren die tatsächlichen Laufzeit- und Kommunikationskosten, die während der Ausführung eines Programms entstehen. Auf der Grundlage dieser Eigenschaften werden fundamentale Verteilungsentscheidungen mittels Basis-Verteilungsfunktionen (siehe Abschnitt 5.3.2) getroffen. Aufgrund der Tatsache, dass diese Programmeigenschaften zum größten Teil das Laufzeitverhalten des Programms widerspiegeln und deshalb vor der Laufzeit nicht komplett ausgerechnet werden können, müssen Teile davon mit geeigneten Strategien abgeschätzt werden. Das gilt für die Programmeigenschaften $E_{\texttt{Work}}(A, O)$, $E_{\texttt{execCost}}(A)$, $E_{\texttt{commCost}}(O_i, O_j)$ und $E_{\texttt{commCost}}(A, O)$, die in einigen vorgestellten ODFs verwendet werden. Im Folgenden werde ich auf die Modellierung der obigen Programmeigenschaften mit einem passenden Kostenmodell eingehen. Die Ausführungs- und Kommunikationskosten werden dabei mit entsprechenden (noch) unbekannten Laufzeitfaktoren modelliert. Auf diesem Modell werden diese Kosten vor der Laufzeit ausgerechnet, indem geeignete Abschätzungsmethoden für die Laufzeitfaktoren angewandt werden. Anschließend wird das Konzept zur hybriden Analyse anhand des zugrundliegenden Kostenmodells zur Objektverteilung nach der `Work`-Eigenschaft dargestellt.

6.2.1. Modellierung dynamischer Programmeigenschaften

Zur Modellierung der obigen Programmeigenschaft $E_{\texttt{Work}}(A, O)$ und $E_{\texttt{commCost}}(A, O)$, auch als verrichteter Workload und verursachte Kommunikation einer Aktivität auf einem Objekt bezeichnet, wird u.a. ein Modell elementarer Kosten benötigt. Sie werden im Folgenden modelliert und die erforderlichen Laufzeitfaktoren werden hierbei definiert.

Ausführungskosten

Der Rumpf einer Methode kann aus einer Sequenz von einfachen Anweisungen, Aufrufstellen, Schleifen und Verzweigungen bestehen. Die letzten drei genannten stellen komplexere Strukturen dar. Deren Kosten sind laufzeitabhängig, daher müssen sie mit bestimmten Laufzeitfaktoren modelliert werden.

Definition 6.0:
Es sei $branch(instr) \in [0,1]$ *die Ausführungswahrscheinlichkeit einer Instruktion innerhalb einer Verzweigungsstruktur und* $loop(l) \geq 0$ *die Ausführungshäufigkeit einer Schleife* l*. Seien* $E_{\texttt{execCost}}(m)$ ***die Ausführungskosten einer Methode*** m*. Dann ist* $E_{\texttt{execCost}}(m)$ *die Summe der Ausführungskosten aller Anweisungen der Methode* m*. Die Ausführungskosten der verschiedenen Anweisungen werden wie folgt festgelegt:*

- *Für eine elementare Anweisung wird ihre Ausführungsdauer* ***einfach*** *(mit Wert 1 gewichtet) ausgerechnet.*
- *Für eine Anweisung, die eine Aufrufstelle für Unteraufrufe lokaler Methoden (über* `this`*-Receiver) oder statischer Methoden darstellt, sind ihre Ausführungskosten die Ausführungsdauer der aufgerufenen Methode.*
- *Für eine Anweisung* $instr$ *innerhalb einer Verzweigungsstruktur wird ihre Ausführungsdauer mit dem Laufzeitfaktor* $branch(instr)$ *multipliziert.*
- *Für eine Anweisung* $instr$ *innerhalb einer Schleife* l *wird ihre Ausführungsdauer mit dem Laufzeitfaktor* $loop(l)$ *multipliziert.*

Die Ausführungskosten der Unteraufrufe von Methoden anderer Objekte werden ignoriert. △

Zwei Laufzeitfaktoren $branch(instr)$ und $loop(l)$ wurden somit definiert. Bei einer Verzweigungsstruktur gibt der Faktor $branch(instr)$ an, mit welcher Wahrscheinlichkeit eine Instruktion innerhalb eines Zweiges ausgeführt wird. Der Laufzeitfaktor $loop(l)$ modelliert die Anzahl der auszuführenden Iterationen der Schleife l zur Laufzeit. Zur Auflösung der Unteraufrufe lokaler Methoden liegt der Aufrufgraph zugrunde, der mittels der PreciseCallGraph-Analyse

(siehe Abschnitt 6.3.1) bestimmt wird. Für den Fall, dass starke Zusammenhangkomponenten oder Rekursion bei der Berechnung der Ausführungskosten für die Unteraufrufe existieren, wird eine Fixpunkt-Berechnung verwendet. In diesem Fall werden die Kosten solange berechnet, bis die Lösung stabil ist. Anderenfalls darf die Anzahl der durchgeführten Berechnungsiterationen einen fest definierten Grenzwert nicht überschreiten.

In diesem Ansatz wird zur Vereinfachung angenommen, dass die Ausführungshäufigkeiten elementarer Anweisungen (Instruktionen) gleich bleiben. Mit dieser Definition kann also die Arbeitslast einer Methode mittels ihrer Ausführungskosten bestimmt werden. Diese Kosten werden tatsächlich verursacht, wenn die Methode in einem Adressraum ausgeführt wird. Das ist auch der Grund, weshalb die Ausführungskosten der Unteraufrufe von Methoden anderer Objekte nicht mit gerechnet werden. Denn diese Kosten entstehen nur im Adressraum, wo sich diese Objekte befinden.

Definition 6.1:
Sei m die zur Laufzeit als Aufrufziel aufgelöste Methode an einer dynamisch gebundenen Aufrufstelle i. Dann sind $E_{\texttt{execCost}}(m)$ die Ausführungskosten der Anweisung i. △

Nun kann die verrichtete Workload einer Aktivität A_i auf einer Methode $m \in O_j$ definiert werden, die von A_i aus *direkt oder indirekt* über andere Objekte und/oder `this`-Receiver aufgerufen wird.

Definition 6.2:
*Seien $E_{\texttt{execCost}_m}(A_i, O_j)$ die Ausführungskosten, die durch die Ausführung der Methode $m \in O_j$ von der Aktivität A_i aus entstehen. $E_{\texttt{execCost}_m}(A_i, O_j)$ ist das Produkt aus der Anzahl der **direkt und indirekt** ausgeführten Aufrufe der Methode m mit $E_{\texttt{execCost}}(m)$. Alle betrachteten Aufrufe finden während der gesamten Abarbeitung der `run()`-Methode von A_i statt.* △

Unter indirekten Aufrufen kann man sich vorstellen, dass A_i eine Methode n eines anderen Objekts O_k aufruft und m wird innerhalb der Methode n aufgerufen. Statisch werden alle Aufrufstellen von A_i aus betrachtet, an denen m direkt oder indirekt aufgerufen werden kann. Die Ausführungskosten an einzelnen Aufrufstellen der Methode m werden ggf. mit entsprechenden Laufzeitfaktoren multipliziert. Somit ergibt sich die gesamte Anzahl der auszuführenden Methodenaufrufe.

Definition 6.3:
$E_{\texttt{Work}}(A, O)$ sei der verrichtete Workload einer Aktivität A auf ein Objekt O. Dann ist $E_{\texttt{Work}}(A, O) := \sum E_{\texttt{execCost}_m}(A, O)$ für alle Methoden $m \in O$, die während der gesamten Abarbeitung der `run()`-Methode von A aufgerufen werden.

Seien $E_{\texttt{execCost}}(A)$ die Ausführungskosten einer Aktivität A. Dann ist dies die Summe der Ausführungskosten der `run()`-Methode von A, $E_{\texttt{execCost}}(run_A)$, und der verrichteten Workload $E_{\texttt{Work}}(A, O)$ auf allen Objekten O, auf die A zugreift. △

Mit diesen Definitionen kann die dynamische Programmeigenschaft zu Workload genau modelliert werden. $E_{\texttt{execCost}}(run_A)$ ist also die Arbeitslast einer Aktivität A. Wie im Abschnitt 6.1.2 beschrieben, könnte die vor der Laufzeit berechneten Kosten verschiedener Aktivitäten zu ihrer Platzierung unter Berücksichtigung der CPU-Load verwendet werden, anstatt sie einfach zyklisch auf die verfügbaren Rechner zu verteilen. Die tatsächlichen Ausführungskosten sind aber erst zur Laufzeit bekannt. Zur Berechnung der Verteilungsentscheidung vor der Laufzeit, die sich auf die Workload bezieht, müssen die Laufzeitfaktoren abgeschätzt werden.

Abschätzungen

Die vor der Laufzeit abzuschätzenden Faktoren sind die Ausführungswahrscheinlichkeit $branch(instr)$ einer Instruktion $instr$ in einer Verzweigungsstruktur, die Ausführungshäufigkeit $loop(l)$ einer Schleife l und Ausführungskosten von Aufrufen dynamisch gebundener Methoden. So wird die Ausführung einer Instruktion $instr$ innerhalb einer n-Verzweigungsstruktur mit einer Wahrscheinlichkeit $branch(instr) = \frac{1}{n}$ abgeschätzt. Die Häufigkeit zur Ausführung einer Schleife l wird mit einem Faktor 10 abgeschätzt. Mittels hybrider Analyse, die im Abschnitt 6.3.3 beschrieben wird, können diese abgeschätzten Faktoren durch reale Werte zur Laufzeit ersetzt werden. Somit können Verteilungsentscheidungen verbessert werden.

An einer Aufrufstelle, an der die statische Auflösung dynamischer Methodenbindung keine eindeutige Lösung liefert, müssen die Ausführungskosten dort ebenfalls abgeschätzt werden. So werden die Ausführungskosten einer solchen Aufrufstelle mit dem Mittelwert abgeschätzt, der aus den Ausführungskosten der statisch aufgelösten Aufrufziele gebildet wird. Die gefundenen Aufrufziele sind das Ergebnis der PreciseCallGraph-Analyse, die im Abschnitt 6.3.1 beschrieben wird.

Die Festlegung, dass die Ausführungsdauer einer elementaren Anweisung nur einfach gerechnet wird (mit Wert „1“), ist ebenfalls eine Abschätzung. Um aber die Präzision bei der Kostenberechnung vor der Laufzeit zu erhöhen, können diese elementaren Instruktionen in verschiedene Gruppen aufgeteilt werden, die bei der realen Ausführung unterschiedliche Ausführungsdauer aufweisen [DPW01]. Nach dieser Gruppierung können Metriken zur Ausführungsdauer der Instruktionen aufgestellt [HD04] und anhand derer die Ausführungskosten genauer modelliert werden.

Mit solchen Abschätzungen können die dynamischen Programmeigenschaften vor der Laufzeit ausgerechnet werden, die die Ausführungskosten des Programms modellieren und in einigen wichtigen Basis-ODFs, $f_{\text{THREADWORK}}()$ und $f_{\text{EX\_DISTRIBUTEDTHREAD}}()$ verwendet werden.

Kommunikationskosten

Unter Kommunikation können alle Arten der Interaktion zwischen den Objekten verstanden werden, bei der Daten über das Netzwerk geschickt werden. Das sind Methodenaufrufe und Feldzugriffe, die im Rahmen der Programmausführung stattfinden. Außerdem entstehen nur Kommunikationskosten, wenn ein Objekt von einem anderen oder von einer Aktivität entfernt platziert wird. Die Modellierung der Kommunikationskosten beruht auf dem Entwurfskonzept des JScatter-Systems: Die Feldzugriffe auf *remote*-Objekte werden zu Methodenaufrufen transformiert. Außerdem wird das Handle-Konzept hier verwendet, damit ein Handle anstatt das tatsächliche *remote* Objekt verwendet wird. Die tatsächlichen Aufrufe werden von Handles an die richtigen Objekte weiter delegiert. Bei entfernten Methodenaufrufen werden die Handles anstelle von *remote* Objekten als Parameter bzw. Ergebnis über das Netzwerk gesendet. Im Gegensatz dazu müssen die *nicht-remote*-Objekte oder lokalen Objekte in entfernten Methodenaufrufen selbst als Werte übergeben werden. Daher spielt die Größe solcher Objekte und der Handles, die allerdings eine einheitliche Größe besitzen, bei der Kommunikation eine wichtige Rolle. Die Größe von Werten eines Grundtyps ist bekannt.

Abschätzung zu Objektgrößen für Kommunikation

Für Handles und normale Objekte gilt, dass deren abgeschätzte Größe die Summe der abgeschätzten Größen seiner nicht-statischen Felder ist. Felder

eines Objekts haben entweder einen Grundtyp oder sind Referenzen auf Objekte. Wenn ein Feld die Referenz auf ein *remote*-Objekt enthält, dann kann die Größe dessen Handles dafür angenommen werden. Im Fall eines lokalen Objekts wird dann wieder dessen geschätzte Größe verwendet. Wenn der Typ für eine Instanzvariable nicht eindeutig ist, müssen die geschätzten Größen aller Objekte berechnet werden, deren Typen vor der Laufzeit in Frage kommen. Dann wird der Mittelwert davon berechnet. Wenn ein Feld rekursive Datenstrukturen enthält, muss ein fester Faktor bei der Berechnung verwendet werden.

Definition 6.4:
Seien $E_{\texttt{commCost}_m}(O_i, O_j)$ die Kommunikationskosten, die durch alle direkten Aufrufe der Methode $m \in O_j$ von einem Objekt O_i aus verursacht werden. Die Kosten für einen Methodenaufruf setzen sich aus den abgeschätzten Größen der Objekte, die als Parameter und Rückgabe übergeben werden, und $E_{\texttt{remoteCallCost}}()$ zusammen.

Die Kommunikationskosten einer Aktivität A mit einem Objekt O über eine Methode m in O, bezeichnet als $E_{\texttt{commCost}_m}(A, O)$, setzen sich ebenfalls aus den abgeschätzten Größen der Objekte, die als Parameter und Rückgabe beim Aufruf von m übergeben werden, und $E_{\texttt{remoteCallCost}}()$ zusammen.

$E_{\texttt{remoteCallCost}}()$ seien die feste Kosten, die bei einem entfernten Methodenaufruf immer verursacht werden. △

Zur Berechnung dieser Programmeigenschaften vor der Laufzeit werden alle Aufrufstellen der Methode m, ggf. mit einer Ausführungshäufigkeit oder -wahrscheinlichkeit multipliziert, mit in die Betrachtung einbezogen. So können statisch die Kommunikationskosten zweier Objekte über eine Methode ermittelt werden. Im Fall einer dynamischen Methodenbindung wird ähnlich wie bei $E_{\texttt{execCost}_m}(A_i, O_j)$ abgeschätzt, also der Mittelwert der Kommunikationskosten aller gefundenen Aufrufziele an einer Aufrufstelle ermittelt.

Da es sich bei Kommunikationskosten nur um entfernte Zugriffe handelt, wird angenommen, dass lokal ausgeführte Methodenaufrufe keine Kommunikation verursachen, obwohl lokale Kommunikation, z.B. zur verteilten Laufzeitumgebung, nicht Null ist. Es gibt viele Faktoren, die direkt die Dauer eines entfernten Methodenaufrufs beeinflussen: die Serialisierung der Parameter bzw. Rückgabewerte, Holen der Stubs, das eigentliche Senden über das Netzwerk, der Kontakt mit dem Skeleton, die Deserialisierung usw. Die meisten von ihnen hängen von den Größen der Parameter bzw. Rückgabewerte ab.

Zur Vereinfachung werden bei der Modellierung der Kommunikationskosten nur die Größen der als Parameter bzw. Rückgabewerte versendeten Strukturen und ein fester Kostenfaktor für jeden entfernten Methodenaufruf, also $E_{\texttt{remoteCallCost}()}$, berücksichtigt. Die restlichen Faktoren werden ignoriert.

Definition 6.5:
$E_{\texttt{commCost}}(O_i, O_j)$ seien die verursachten Kommunikationskosten eines Objekts O_i auf ein Objekt O_j. m, n seien Methoden. Dann ist:

$$
\begin{aligned}
E_{\texttt{commCost}}(O_i, O_j) := & \sum_{m \in O_j}^{O_i \text{ calls } m} E_{\texttt{commCost}_m}(O_i, O_j) + \\
& \sum_{O_l \in M} \sum_{n \in O_l}^{O_j \text{ calls } n} E_{\texttt{commCost}_n}(O_j, O_l), \\
& \textit{wobei } M = \{O_l \in \mathbb{O} | E_{\texttt{commCost}}(O_i, O_l) = 0 \wedge \textit{callsite(n) liegt in } m\}.
\end{aligned}
\tag{6.1}
$$

Analoges gilt für $E_{\texttt{commCost}}(A, O)$. △

Die anfallenden Kommunikationskosten zwischen zwei voneinander entfernt platzierten Objekten O_i und O_j ergeben sich nicht nur aus den Kommunikationskosten entfernter Methodenaufrufe von O_i zu O_j über eine Methode m, sondern auch aus den anfallenden Kosten durch Aufrufe der Methoden anderer zu O_i lokalen Objekte während der Ausführung von m. Der Grund für diese Modellierung der Kommunikationskosten liegt darin, dass durch die Platzierung von O_i und O_j auf zwei verschiedenen JVMs die lokalen Aufrufe von Methoden anderer Objekte während der Kommunikation zwischen O_i und O_j zu entfernten Aufrufen werden. Diese Kosten müssen auch berücksichtigt werden. Wenn eine zyklische Abhängigkeit der Aufrufbeziehung existiert, dann wird die oben beschriebene Fixpunkt-Berechnung für die Kommunikationskosten eingesetzt.

Durch die Abschätzung der Laufzeitfaktoren im letzten Abschnitt und der Objektgrößen ist man in der Lage, dynamische Programmeigenschaften zur Kommunikation vor der Laufzeit auszurechnen. So kann eine Verteilungsentscheidung basierend auf diesen Eigenschaften sowie der *Workload*-Eigenschaften für alle Aktivitäts- und Objektrepräsentanten berechnet werden.

Regeln zur Anwendung dynamischer Programmeigenschaften

Dynamische Programmeigenschaften werden dazu verwendet, um grundlegende Verteilungsentscheidungen mittels Basis-ODFs zu berechnen. Durch Abschätzungen der Ausführungs- und Kommunikationskosten können diese Eigenschaften statisch bestimmt werden. Je nach Basis-ODFs werden die beiden Arten der Kosten einzeln oder auch kombiniert verwendet. Damit es bei der Verwendung nicht zu ungünstigen Ergebnissen kommt, müssen einige Berechnungsregeln beachtet werden.

Die Platzierungsentscheidung der Aktivitäten und Objekte erfolgt immer nach dem Prinzip, dass zunächst alle Aktivitäten auf die abstrakten JVMs verteilt werden. Objekte werden den Aktivitäten nach $E_{\texttt{Work}}(A, O)$ oder $E_{\texttt{commCost}}(A, O)$ zugeordnet, sobald die zugehörigen Programmeigenschaften ausgerechnet werden. Zuletzt wird die Zuordnung der Objekte zu den Objekten bestimmt, die den Aktivitäten schon zugeordnet sind.

Bei der Platzierung eines Objekts O zu einer Aktivität A unter Verwendung beider Arten der Kosten muss sichergestellt werden, dass A die meiste Rechenzeit mit O verbringt. Gleichzeitig müssen die anfallenden Kommunikationskosten minimal gehalten werden. Denn durch diese Zusammenplatzierung müssen andere Aktivitäten und Objekte, die in anderen Adressräumen platziert sind, entfernt auf O zugreifen. Dieses Berechnungsmodell lehnt sich an das Lastmodell in [Hau98] an und wird auch bei den ODFs $f_{\text{THREADWORKCOMM}}$ und $f_{\text{EX\_DISTRIBUTEDTHREAD}}$ eingesetzt.

6.2.2. Konzept zur hybriden Analyse

Die hybride Analyse ist eine Kombination aus statischer und dynamischer Analyse. Ziel der Analyse ist es, möglichst genaue Ausführungskosten zur Platzierungsberechnung zu bestimmen. Die statischen Analysetechniken werden eingesetzt, um die Ausführungskosten einer Methode zu berechnen. So können Anteile der Methode m, die nicht von einem Laufzeitfaktor abhängen, komplett vor der Laufzeit ausgerechnet werden. Im Folgenden werden die Kosten dieser Anteile als $E_{\texttt{fixedCost}}(m)$ bezeichnet. Die von einem Laufzeitfaktor abhängenden Kosten einer Methode, auch als $E_{\texttt{variableCost}}(m)$ bezeichnet, werden dagegen vor der Laufzeit abgeschätzt. Dynamische Analyse kommt zur Laufzeit zum Einsatz, um die tatsächlichen Laufzeitwerte für solche Faktoren zu bestimmen. Sobald ein Laufzeitwert

vorliegt, wird der zugehörige Faktor aktualisiert. Dies löst erneute Berechnung von $E_{\texttt{variableCost}}(m)$ und somit der Ausführungskosten der Methode aus. Infolgedessen werden Verteilungsentscheidungen für die zukünftig zu erzeugenden Objekte basierend auf den genaueren Kosten getroffen.

Da die Kommunikationskosten von vielen Laufzeitfaktoren abhängen, die sich sehr schwierig oder überhaupt nicht modellieren lassen, kann dieses Konzept nicht dafür verwendet werden. Beispiel hierfür ist das eigentliche Versenden der Objektstrukturen über das Netzwerk. Dafür müssen dann Abschätzungen verwendet werden. Als Abhilfe könnten Strategien zur dynamischen Lastbalancierung zur Laufzeit zusätzlich angewandt werden. Somit können viel miteinander kommunizierende Threads und Objekte zur Laufzeit nachträglich zusammen platziert werden, so dass Kommunikationsaufwand reduziert wird. Zur Motivation wird dieses Konzept an einem einfachen Beispiel demonstriert, wobei die Ausführungshäufigkeit der Schleifen genau berechnet wird.

Hybride Analyse für Schleifen

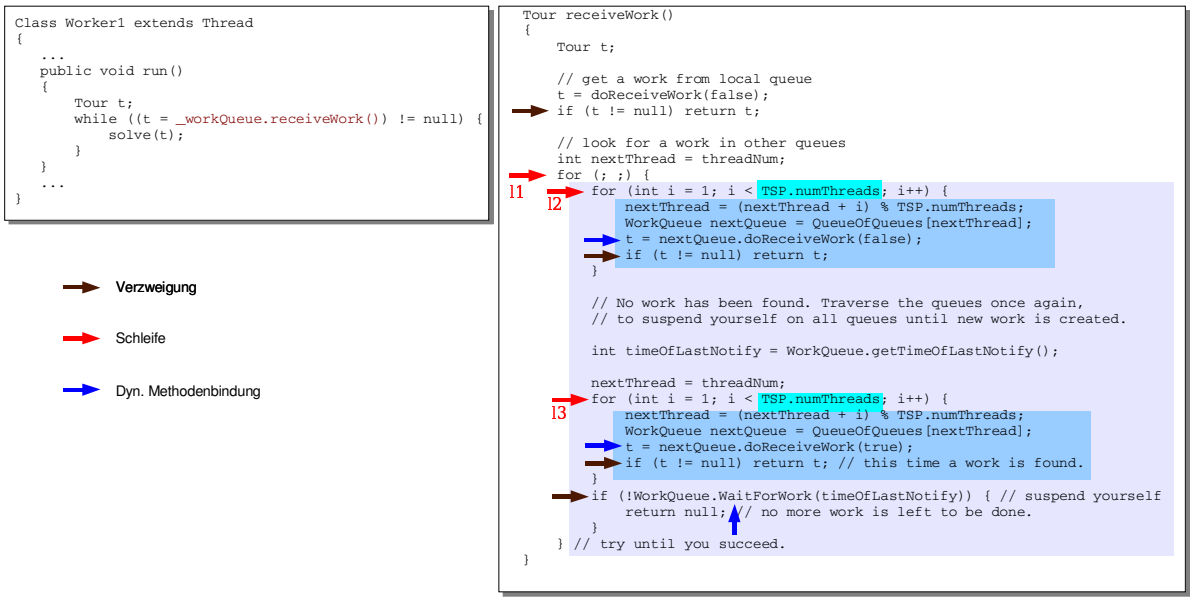

Abbildung 6.1.: TSP-Beispiel zur hybriden Analyse für Schleifen

In Abbildung 6.1 wird ein Ausschnitt eines Programms dargestellt (Traveling Salemans Problem - TSP). In der `run()`-Methode des `Worker1`-Threads

wird die Methode `receiveWork()` aufgerufen, deren Implementierung in der Abbildung ebenfalls dargestellt wird. Dort sind einige besonders gekennzeichnete Stellen (mit einem Pfeil) zu erkennen, an denen die oben definierten Laufzeitfaktoren vorkommen. Betrachten wir nun die drei vorkommenden Schleifen l_1, l_2 und l_3 in der Methode, deren Rumpf mit entsprechenden Rechtecken umfasst wird. Es seien hierbei die Ausführungswahrscheinlichkeit der Verzweigungen und die Aufrufziele an den Aufrufstellen nicht lokaler Methoden fest. Dann sieht die Formel für die Ausführungskosten der Methode `receiveWork()` folgendermaßen aus:

$$\begin{aligned} E_{\texttt{execCost}}(receiveWork()) := & E_{\texttt{fixedCost}}(receiveWork()) + \\ & loop(l_1) * (loop(l_2) * E_{\texttt{execCost}}(l_2) + loop(l_3) * E_{\texttt{execCost}}(l_3) + \\ & E_{\texttt{execCost}}(l_1)) \end{aligned} \tag{6.2}$$

Zur Verteilungsberechnung wird vor der Laufzeit die Anzahl der Durchläufe aller Schleifen mit 10 abgeschätzt, also $loop(l_1) = loop(l_2) = loop(l_3) = 10$. Durch die Abhängigkeitsanalyse für Schleifengrenzen wird statisch festgestellt, dass l_2 und l_3 einen gemeinsamen Grenzwert für ihre Iterationen besitzen. Dieser Grenzwert wird durch `TSP.numThreads` definiert. Für l_1 kann keine genaue Obergrenze berechnet werden, daher bleibt weiterhin $loop(l_1) = 10$. Nun werden Äquivalenzklassen für die Schleifen gebildet, die gemeinsame Schleifengrenzen besitzen. So werden l_2 und l_3 zu einer Äquivalenzklasse $< l_2 >$ zusammengefasst mit $loop(< l_2 >) =$ `TSP.numThreads` $(= loop(l_2) = loop(l_3))$, l_1 wird zu $< l_1 >$ mit $loop(< l_1 >) = 10$. Die Kardinalität von $< l_2 >$ ist 2, die von $< l_1 >$ ist 1. Der obige Term kann nun kompakter dargestellt werden:

$$\begin{aligned} E_{\texttt{execCost}}(receiveWork()) := & E_{\texttt{fixedCost}}(receiveWork()) + \\ & loop(< l_1 >) * (loop(< l_2 >) * (E_{\texttt{execCost}}(l_2) + E_{\texttt{execCost}}(l_3)) + \\ & E_{\texttt{execCost}}(l_1)) \end{aligned} \tag{6.3}$$

Für die nachträgliche Verteilungsberechnung wird nur Äquivalenzklasse mit maximaler Kardinalität in Betracht gezogen, hier also für $< l_2 >$. Nachdem der Wert der Variable `TSP.numThreads` (= 5) zur Laufzeit bekannt war, wird $E_{\texttt{execCost}}(receiveWork())$ mit dem neuen Wert $loop(< l_2 >) = 5$ erneut berechnet. Die Ausführungskosten der Methode `receiveWork()` werden hiermit zur Laufzeit genauer ausgerechnet und die Verteilungsentscheidung

wird erneut getroffen. Dieses Beispiel zeigt eine günstige Situation, denn die Laufzeitkonstante `TSP.numThreads` ist schon bekannt, bevor die Methode `receiveWork()` zum ersten Mal ausgeführt wird.

Im Allgemeinen können die Ausführungskosten einer Methode erst vollständig berechnet werden, nachdem die erste Schleife der betrachteten Äquivalenzklasse $< l >$ vollständig durchlaufen wurde und somit der Wert des zugehörigen Laufzeitfaktors $loop(< l >)$ bekannt ist. So können Platzierungsentscheidungen weiterer zu erzeugender Objekte von solchen genaueren Kosten profitieren.

Ausführungskosten einer Methode

Wenn man die anderen Laufzeitfaktoren betrachtet, können die Ausführungskosten einer Methode im Allgemeinen so modelliert werden:

$$\begin{aligned} E_{\texttt{execCost}}(m) :=& E_{\texttt{fixedCost}}(m) + E_{\texttt{variableCost}}(m) \\ :=& E_{\texttt{fixedCost}}(m) + \\ & \sum_i branch(b_i) * E_{\texttt{execCost}}(b_i) + \sum_j loop(l_j) * E_{\texttt{execCost}}(l_j) + \\ & \sum_k E_{\texttt{execCost}}(target(c_k)) \end{aligned} \tag{6.4}$$

wobei b_i eine Verzweigungsstruktur, l_j eine Schleife und c_k eine Aufrufstelle in der Methode m mit den potentiellen Aufrufzielen $m_{k_1}, \ldots, m_{k_j}$ sei. In *JScatter* habe ich das Konzept zur hybriden Analyse nur für Schleifen angewandt und im Abschnitt 8.3.2 evaluiert. Die beiden anderen Faktoren werden abgeschätzt. Im Folgenden wird ein grobes Verfahren zur hybriden Analyse beschrieben.

Das Verfahren

Um die Ausführungskosten als Terme allgemein zu modellieren, wird zunächst eine einfache statische Analyse [Muc97] zur Erkennung der Schleifen durchgeführt. Wie im obigen Abschnitt über Abschätzungen beschrieben, wird die Anzahl der Schleifendurchläufe zunächst mit einem Faktor 10 abgeschätzt. Mit Hilfe des Verfahrens zur Induktionsvariable können Schleifenbedingungen und im günstigen Fall auch obere Grenzen erkannt werden.

Durch die interprozedurale Def-Use-Analyse können Abhängigkeiten zwischen den Schleifengrenzen herausgefunden werden. Die Schleifen, die gemeinsame Grenzwerte besitzen, werden dann in Äquivalenzklassen zusammengefasst. Die Terme werden danach entsprechend optimiert. Der Faktor *loop* einer Äquivalenzklasse gilt für alle dazu gehörigen Schleifen, wie im obigen Beispiel gezeigt wurde.

Um zur Laufzeit den tatsächlichen Grenzwert für eine Äquivalenzklasse zu finden, muss das Programm an den richtigen Stellen instrumentiert werden. Es sollte Instrumentierung so wenig wie möglich erfolgen, damit keine Zusatzkosten verursacht werden. Dazu werden vor der Laufzeit nur Äquivalenzklassen mit maximaler Kardinalität betrachtet. Durch die interprozedurale Kontrollabhängigkeitsanalyse können die zuerst ausgeführten Schleifen erkannt werden, und dort wird Instrumentierungscode eingeführt.

Zur Laufzeit, nachdem die Ausführungskosten durch Bekanntgabe solcher *loop*-Faktoren erneut ausgerechnet wurden, müssen die von solchen Kosten abhängigen Verteilungsentscheidungen ebenfalls erneut bestimmt werden. Im Abschnitt 6.3.3 werden einige Erweiterungen zur Erhöhung der Genauigkeit bei der hybriden Analyse diskutiert.

6.3. Grundlegende statische Analysen

Alle Programmeigenschaften, sowohl statische als auch dynamische, werden mittels statischer Programmanalysen berechnet und ggf. abgeschätzt. Die Eingabe liegt in Form von Java-Bytecode vor, worauf die Analysetechniken und die Transformation angewandt werden. Zur Beschaffung elementarer Analyseergebnisse wie z.B. des Kontrollflussgraphen, des Dominatorbaumes oder der Schleifenstruktur einer Methodeimplementierung werden viele fundamentale Analysetechniken eingesetzt. Sie sind als verschiedene Module unserer Analyseumgebung (siehe Abschnitt 6.1.2 in [Thi01]) realisiert.

Im Folgenden werde ich auf die Erweiterung der vorhandenen *PreciseCallGraph*-Analysetechnik und die *GlobalDefUse*-Analyse eingehen, die bei der Verteilungsberechnung eine wichtige Rolle spielen. Abschließend wird beschrieben, wie einige ausgewählte Programmeigenschaften mittels vorhandener Analysen berechnet werden.

6.3.1. Aufrufgraph-Analyse zur Auflösung dynamischer Methodenbindung

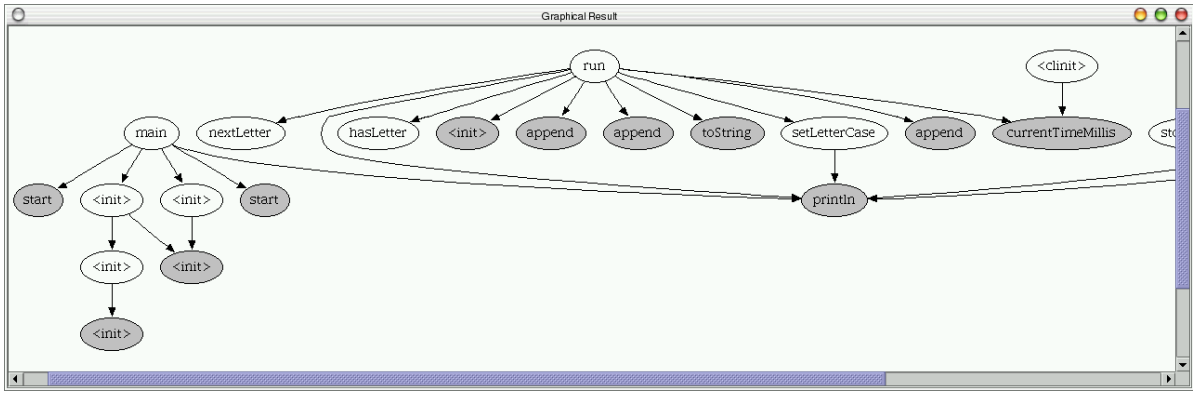

Abbildung 6.2.: Ergebnis der PreciseCallGraph-Analyse

Das statische Analyseverfahren zur Konstruktion eines Aufrufgraphen berechnet für den Analysekontext die Aufrufbeziehung der Methoden auf der Grundlage der Klassenhierarchie und Informationen zur Typinferenz. Ein Analysekontext enthält die Anwendung und alle benutzten Bibliothek-Klassen. Die dort verwendete Typinferenz-Analyse ist eine Variante der Points-to-Analyse [Le00], mit deren Hilfe man den Receivertyp genauer bestimmen kann, um aufgerufene Ziele einzuschränken. In Abbildung 6.2 ist ein Aufrufgraph als Ergebnis der Analyse zu sehen. Die grau markierten Knoten repräsentieren Methoden, die nicht im Analysekontext oder *native* sind und deren Implementierung daher nicht bestimmt werden kann. Die Aufrufrelation zwischen diesen Knoten kann nicht vom ursprünglichen Verfahren beschafft werden. Das sind Aufrufbeziehungen zwischen `start()`- und `run()`-Methoden eines Threads sowie Relation einer Aufrufstelle zur `<clinit>`-Methode.

Erweiterung des Konstruktionsverfahrens

Die genannte, fehlende Information ist wichtig für die Berechnung einiger grundlegender Programmeigenschaften wie `isStagewiseImmutable`, `Work` usw. Die Aufrufrelation zwischen der `start()`- und `run()`-Methode der Thread-Klasse oder Unterklassen davon wurde nicht behandelt, weil `start()` eine *native*-Methode ist. Nach spezieller Betrachtung deren Implementierung läßt sich feststellen, dass die zugehörige `run()`-Methode zur

Laufzeit von dort aus aufgerufen wird. Statisch müsste aber ganz konservativ angenommen werden, dass von jedem Knoten für eine `start()`-Methode Kanten zu jedem `run()`-Knoten existieren sollen. Mit Hilfe der Typinferenz-Analyse kann vor der Laufzeit der Typ des Receivers zum `start()`-Aufruf genauer berechnet werden. Somit können die Aufrufziele eingeschränkt werden und die Anzahl der von `start()`-Knoten ausgehenden Kanten wird dadurch erheblich reduziert.

Wenn eine Klasse einen statischen Anteil, z.B. eine Klassenvariable, besitzt, erzeugt der Compiler bei der Übersetzung automatisch eine so genannte `<clinit>`-Methode, die für die Initialisierung statischer Anteile der Klasse zuständig ist. Nach der JVM-Spezifikation [T. 99] wird diese einmalig aufgerufen und zwar bei der allerersten Benutzung des statischen Anteils der Klasse. Der Aufruf der `<clinit>`-Methode kann durch die Instruktion `PUTSTATIC`, `GETSTATIC` oder die Erzeugung eines Objekts der Klasse durch die Instruktion `NEW` ausgelöst werden. Um zusätzlich Aufrufrelationen zu solchen `<clinit>`-Knoten aufzubauen, werden bei der Konstruktion des Aufrufgraphen nicht nur Aufrufstellen, also `Invoke`-Instruktionen, sondern auch die oben genannten Instruktionen analysiert. Sie können den Aufruf der zugehörigen `<clinit>`-Methode auslösen. Hierbei wird bei jeder solchen Instruktion eine Kante zu der zugehörigen `<clinit>`-Methode konstruiert.

Die zugehörige `<clinit>`-Methode läßt sich einfach finden, denn die Information darüber, welche Klasse bei `GETSTATIC`, `PUTSTATIC` oder `NEW` verwendet wird, ist direkt bei diesen Instruktionen verfügbar. Außerdem muss ggf. die `<clinit>`-Methode der Oberklasse aufgerufen und bearbeitet werden, bevor die erste Anweisung der aktuellen `<clinit>`-Methode ausgeführt wird. Diese automatisch ausgelöste Aufrufkette muss bei der Aufrufgraphen-Konstruktion berücksichtigt werden. So können Kanten zwischen `<clinit>`-Knoten der Unter- und Oberklassen im Aufrufgraphen existieren.

In Abbildung 6.3 wird das Ergebnis der erweiterten Aufrufgraph-Analyse dargestellt. In einem Aufrufgraphen können drei neue Sorten der Kanten vorkommen: von `start()`-Knoten zu `run()`-Knoten, von einem Knoten zu `<clinit>`-Knoten, wobei diese noch voneinander unterschieden werden: eine durch `NEW` und eine durch `PUTSTATIC` oder `GETSTATIC` aufgelöste Beziehung. Der erweiterte Aufrufgraph kann außerdem auch Kanten enthalten, die die `<clinit>`-Knoten mit `<clinit>`-Knoten der Oberklasse verbinden. Das Beispiel in Abbildung besitzt solche Kanten nicht.

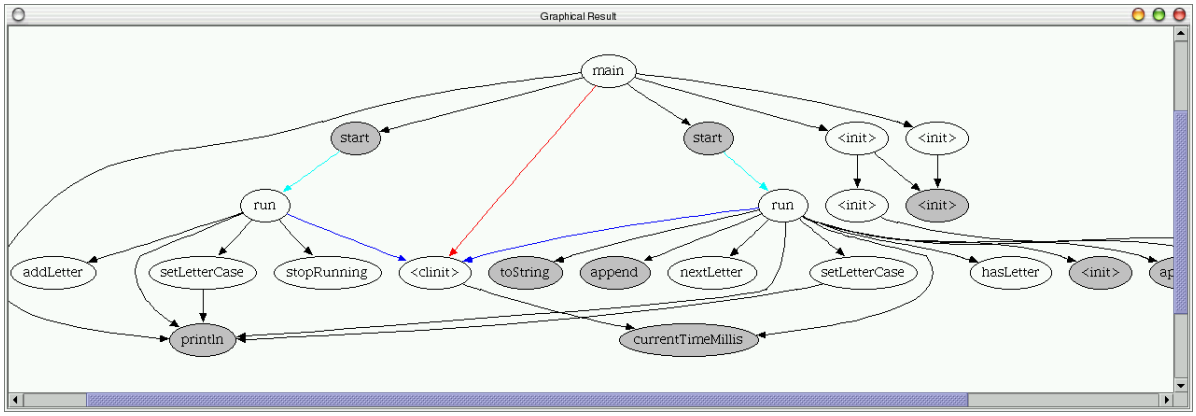

Abbildung 6.3.: Ergebnis der erweiterten PreciseCallGraph-Analyse

Der erweiterte Aufrufgraph enthält außerdem zusätzliche Information darüber, von welcher Instruktion eine Kante ausgeht. Diese Information ist erforderlich für die Berechnung der interprozeduralen Post-Dominator-Relationen (siehe Abschnitt 6.4.2 und [P. 01]). Die üblichen *native*-Methoden oder diejenigen außerhalb des Analysekontexts werden weiterhin als Knoten des Aufrufgraphen modelliert. Nur detailierte Information dazu wie deren Implementierung oder davon ausgehende Kanten, wird nicht berechnet (in Abbildung 6.3 werden sie als graue Knoten dargestellt).

Der berechnete Aufrufgraph dient als Grundlage zur statischen Auflösung der Aufrufziele, insbesondere dynamisch gebundener Methode. Durch die Erweiterung des Konstruktionsverfahrens stehen die Informationen über Aufrufbeziehung innerhalb eines Threads und zu Methoden zur statischen Initialisierung zur Verfügung. Dieser Informationsgehalt ist ausreichend für die hierauf aufbauende Berechnung der Programmeigenschaften.

6.3.2. Interprozedurale Def-Use-Analyse (*GlobalDefUse*-Analyse)

Definitions-/Verwendungsketten für lokale Variablen und *stackslots* innerhalb einer Methode werden mittels der *LocalDefUse*-Analyse aufgebaut, die oben erwähnt und in [Thi01] beschrieben wurde. Dieser Ansatz soll nun so erweitert werden, dass von einer Definition Verbindungsketten zu allen ihrer Verwendungen im gesamten Analysekontext und umgekehrt aufgebaut werden. Solche Verbindungen, die über Methodengrenzen hinausgehen, werden im Folgenden interprozedurale Definitions-/Verwendungsketten genannt, abge-

kürzt als globale Def-Use-Kette. Dieses Verfahren wurde in unserer Analyseumgebung als ein Analysemodul von Dr. Michael Thies [Thi01] realisiert.

Ein Beispiel

Zur Verdeutlichung betrachte man zunächst ein einfaches Beispiel, das in Abbildung 6.4 angegeben ist. Es zeigt uns das Ergebnis der *GlobalDefUse*-Analyse für eine Definition, repräsentiert an der Erzeugungsstelle des Objekts vom Typ `B` innerhalb der Konstruktor-Methode `A()` (die Stelle ist gesondert markiert). Die *GlobalDefUse*-Analyse findet viele (markierte) Programmstellen, an denen sich Verwendungen von Referenzen auf Objekte befinden, die an der Stelle `bo = new B(ai)` erzeugt werden. Der `this`-Receiver zu diesem Objekt (`bo_object`) wird innerhalb der eigenen Konstruktormethode `B(int i)` und in den beiden Methoden `getI()` und `setI()` verwendet.

Außerdem referenziert eine Instanzvariable `bo` in der Klasse `A` das Objekt `bo_object`. Von dieser Variable aus kann das Objekt über die Methode `getBO()` an anderen Stellen verwendet werden: Das Feld `b` in der Klasse `C` speichert diese Referenz zu `bo_object`. Über `b` wird es dann zu Methodenaufrufen in der Methoden `cMethode()` benutzt.

```
class A {
  private int ai;
  private B bo;

  A() {
    ...
    bo = new B(ai);
    ...
  }

  B getBO() { return bo; }
}

class B {
  private int bi;

  B(int i) { this.bi = i; }

  int getI() { return bi; }

  void setI(int i) { bi = i; }
}
```

```
class C {
  private B b;
  private A a;

  C() {
    a = new A();
    b = a.getBO();
    ...
  }

  void cMethod() {
    ...
    if (bed)
      x = b.getI();
    else
      b.setI(x);
  }
}
```

Abbildung 6.4.: Beispiel für interprozedurale Definitions-/Verwendungsketten

Man kann erkennen, dass das Objekt `bo_object` nicht nur innerhalb der Methode verwendet wird, in der es erzeugt wird, sondern über die Methode `getBO()` entkommt. Dadurch kann es auch in Methoden anderer Klasse benutzt werden. Die Analyse hat also die Aufgabe, solche interprozeduralen Verbindungen zwischen den Verwendungsstellen und den Definitionsstellen aufzubauen. Zur Vereinfachung wird im Folgenden das Verfahren anhand des angegebenen Beispiels grob erläutert.

Verfahren

Die Kernidee des Verfahrens basiert auf einer Variante des FTA-Verfahrens [TP00], [Wes04], welches zur Konstruktion des Aufrufgraphen eingesetzt wird und die Verwendungsbeziehungen der Felder berechnet. Das *GlobalDefUse*-Verfahren geht noch einen Schritt weiter, so dass es den Informationsfluss zu den Definitionsstellen von diesen Verwendungen, die im ganzen Programm vorkommen können, weiter progagiert und daraus die Def-Use-Ketten interprozedural bestimmt.

Die Berechnung der globalen Def-Use-Ketten wird prinzipiell in drei Phasen aufgeteilt:

- *1. Phase:* Für jede Methode werden die darin enthaltenen Def-Use-Ketten lokal berechnet. Die Knoten einer lokalen Def-Use-Kette repräsentieren lokale Zugriffe, so genannte *LocalAccesses*, und die Kanten die Def- oder Use-Beziehung. Basiert auf dem Stack-Modell der JVM modelliert ein *LocalAccess* einen einfachen Schreib/Lese-Zugriff auf Operanden bzw. Ergebnis, der mittels einer Bytecode-Instruktion aufgelöst wird. In Abbildung 6.5 wird die lokale Def-Use-Kette innerhalb der Konstruktormethode `A()` dargestellt: So wird das Objekt `bo_object` von vier *LocalAccesses* $l1, l2, l3$ und `bo` verwendet. Ebenfalls wird die lokale Def-Use-Kette der Methode `cMethod()` in der Abbildung angegeben. Solche lokalen Ketten werden durch die *LocalDefUse*-Analyse bestimmt.
- *2. Phase:* Die Abhängigkeit der Parameter, Rückgaben und Felder aller Methoden wird berechnet und in einem Abhängigkeitsgraphen modelliert. Dies ist ein gerichteter Graph. Die Knoten repräsentieren die Felder, Parameter und Rückgaben. Die Kanten stellen die Flussinformation von einem Knoten zu einem anderen dar. Für das obige Beispiel wird dieser Graph in Abbildung 6.5 dargestellt: Eine Kante von einem Parameter $p1$

zu einem Feld f modelliert eine potentielle Zuweisung des Wertes von $p1$ zu f. Die Propagation der Information, also auch der Aufbau des Abhängigkeitsgraphen der Felder, Parameter und Rückgaben, findet auf fluss- und kontext-insensitiver Weise statt. Diese Analyse ist eine einfache Variante der Points-to-Analyse [Ryd03], [Ste96], [Le00].

- *3. Phase:* Die Verbindungen zwischen den *LocalAccesses* der Methoden und den Knoten des Abhängigkeitsgraphen werden in dieser Phase aufgebaut. Lokale Zugriffe auf Parameter, Rückgabewerte oder Felder innerhalb einer Methode werden auf die zugehörigen Knoten des Abhängigkeitsgraphen und umgekehrt abgebildet. Lokale Def-Use-Ketten werden somit zu globalen zusammengesetzt. In Abbildung 6.5 stellen die Kanten zwischen dem *LocalAccess* `bo` bzw. `b` (als Kreise) und dem Knoten bo bzw. b (als Rechtecke) des Graphen solche Verbindungen dar.

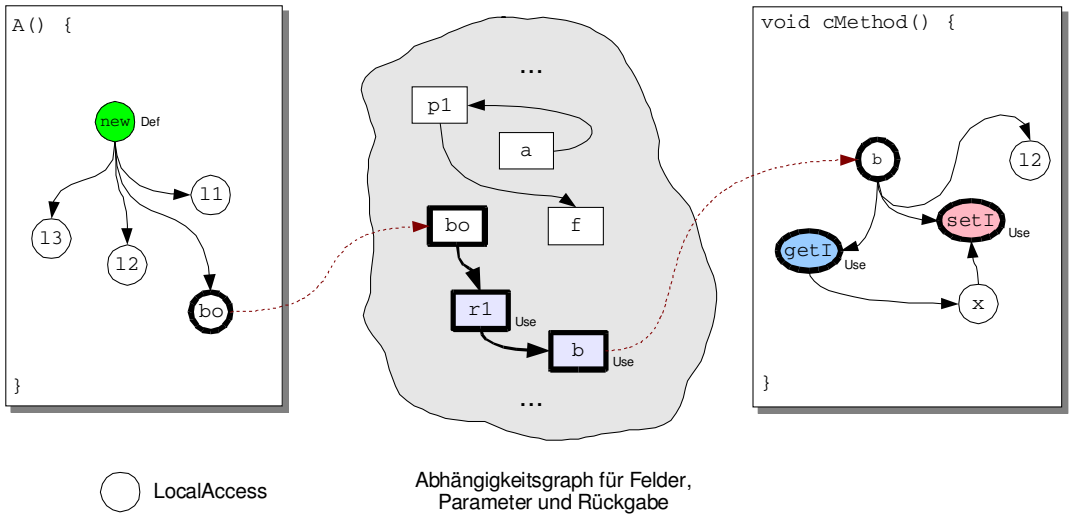

Abbildung 6.5.: Verfahren zum Aufbau der interprozeduralen Definitions-/Verwendungsketten

Als Ergebnis stehen alle interprozeduralen Def-Use-Ketten zur Verfügung. Damit kann man zu einer Erzeugungsstelle, also einer Definition, alle Verwendungsstellen im ganzen Programm bestimmen. Symmetrisch können zu einer Verwendungsstelle ihre Definition(en) gefunden werden.

6.3.3. Analysetechniken zur Berechnung der Programmeigenschaften

Dieser Abschnitt beschreibt die Anwendung der vorgestellten Analysen sowie grundlegender Analysetechniken zur Berechnung und Abschätzung einiger modellierter Programmeigenschaften. Durch das Bestimmen des Escape-Status kann festgestellt werden, ob ein Objekt die Proxy- oder die Thread-lokale Eigenschaft besitzt. Mittels einiger einfacher Techniken können ebenfalls einige Peer-Eigenschaften berechnet werden. Zur Vorbereitung der Objektmigration zur Laufzeit muss statisch bestimmt werden, ob Instanzen einer Klasse migrierbar sind und welche ihrer Methoden Schreib- oder Nur-Lese-Zugriffe darstellen. Diese Information ist vor der Laufzeit erforderlich zur Transformation der Klassen, um bei potentiell schreibenden Methoden einen Mechanismus für Schreibzähler einzubauen. All diese Berechnungen zu statischen Programmeigenschaften können vollständig vor der Laufzeit durchgeführt werden.

Escape-Eigenschaft

Ein typisches Muster für (gute) OO-Programmierung ist die Kapselung eines Objekts in ein anderes, so dass sämtliche Operationen auf das innere Objekt nur durch Aufrufe der Methoden des äußeren Objekts (Proxy-Objekts) ermöglicht werden. Das bedeutet, dass sowohl die Objekterzeugung als auch alle Zugriffe auf diese Objekte innerhalb ihrer Proxy-Objekte stattfinden müssen. Als Zugriffe zählen hierbei Methodenaufrufe und direkte Feldzugriffe über ihre Referenzen. Außerdem dürfen Methoden des inneren Objekts von diesem selbst aufgerufen werden. Das bedeutet, dass die Referenzen der inneren Objekte ihren umgebenden Objekten nicht entkommen.

Im Thread-lokalen Fall wird die Escape-Umgebung der inneren Objekte „etwas vergrößert“. Dort können Objekte ihren umgebenden Objekten über Parameter, Rückgabewerte einer Methode oder in einem für andere zugängig abgespeicherten Feld entkommen, jedoch nicht den sie erzeugenden Threads.

Die Analysetechnik, die diese Bedingungen überprüft, muss dabei auch die Seiteneffekte der Methodenaufrufe auf interprozeduraler Weise berücksichtigen. Außerdem kann zusätzliche Betrachtung von Zugriffen auf als `private` deklarierte Felder viele Treffer für Objekte mit solcher Eigenschaft erzielen. Denn einerseits sind die von privaten Feldern referenzierten Objekte Spitzen-

kandidaten mit dieser Eigenschaft, andererseits muss die Analyse für solche Objekte nur die Methoden der deklarierenden Klasse untersuchen.

Um den Escape-Status statisch für alle Objektrepräsentanten im Programm zu berechnen, wurden u.a. die obige interprozedurale *GlobalDefUse*- und die Klassenhierarchie-Analyse eingesetzt und für jede Objekterzeugungsstelle nach den obigen Kriterien überprüft. Weitere Details über diese Eigenschaft finden sich in der Diplomarbeit von Karsten Klohs [Klo02].

Bestimmung der Peer-Eigenschaft

Zur Peer-Eigenschaft werden zunächst verschiedene Muster spefizifiert, die diese Eigenschaft aufweisen. Mittels unterschiedlicher Analysen, von einfacher Hierarchie-Information bis zu speziellen, komplexeren Techniken werden die Objektrepräsentanten statisch überprüft, ob solche Muster in ihrem Benutzungskontext vorkommen. Beispiele dafür sind Muster zur Erkennung von Benutzerinteraktionen mit dem Programm, z.B. zur Eingabe oder Ausgabe, Muster zur Ausführung maschinenspezifischen Codes (oder Ausführung von als *native* definierter Methoden), Muster zum Netzwerkaufbau, Muster zur grafischen Darstellung, Zeichnung usw. Im Allgemeinen stellen solche Muster verschiedenartige Benutzungen der Ressourcen anderer Programme oder der Hardware dar. Je nach Komplexität und Exaktheit der Berechnung solcher Muster werden sehr einfache bis aufwändige Analysen verwendet. Um diesen Ansatz zu evaluieren werden in dieser Arbeit nur einfache Muster definiert.

Das einfachste Muster ist das zur Ausführung von Maschinencode durch Aufrufe von Methoden, die als *native* definiert sind. Die eingesetzte Analyse hat für jeden Objektrepräsentanten nachzuprüfen, ob bei allen potentiell aufzurufenden Methoden mindestens eine *native*-Methode vorkommt. Durch die obige *PreciseCallGraph*-Analyse und Nachfrage nach dem vorhandenen Status der betrachteten Methode kann dieses Muster einfach bestimmt werden. Es gibt jedoch eine Ausnahme: Thread-Repräsentanten werden nicht untersucht, denn die `start()`-Methode der Thread-Klasse ist eine *native*-Methode. Aber die Aktivitäten sind Hauptbestandteile der Verteilung und werden hierbei behandelt, als ob sie keine Peer-Eigenschaft besitzen würden. Dies ist auch der Grund, weshalb eine nachträgliche Umverteilung eines schon platzierten Threads mit sehr vielen Aufwand verbunden ist und bei diesem Konzept keinen Sinn macht. Ein weiteres Muster, welches sich auch einfach erkennen läßt, stellt die Erkennung der Benutzerinteraktion mit dem Programm

dar. Hierbei ist zu untersuchen, ob in den aufzurufenden Methoden eines Objektrepräsentanten solche Eingabe/Ausgabe-Interaktionen enthalten sind. Sie sind daran zu erkennen, dass bestimmte Klassen oder/und Methoden des `java.io`-Pakets benutzt werden. Neben dem Aufrufgraphen wird auch die `MethodInspector`-Analyse [Thi01] verwendet, um u.a. Details über Instruktionen der Methode zu erfahren.

Um Muster zur Erkennung komplexerer Aufgaben, z.B. zur grafischen Darstellung oder zum Netzwerk-Aufbau zu definieren und zu erkennen, wird mehr Analyseaufwand benötigt und spezielle Kenntnisse über einige Klassen des `java.awt` bzw. `java.net` Pakets vorausgesetzt. Solchen Muster wurden hier nicht evaluiert.

Lese/Schreib-Zugriffsanalyse zur Migration

Ein Kriterium zur Migrationsentscheidung während der Programmausführung ist das Zugriffsverhalten auf Objekte, deshalb werden die tatsächlichen Zugriffe zur Laufzeit gezählt. Es wird außerdem nach Schreib- und Lese-Zugriffen unterschieden. Vor der Laufzeit muss an geeigneten Stellen in entsprechenden Methoden Instrumentierungscode für passende Zähler eintransformiert werden. Dafür wird statisch bestimmt, welche Methoden den Schreib- oder Lese-Status besitzen. Mit Hilfe der Information über die lokalen Def-Use-Ketten kann fluss-insensitiv festgestellt werden, ob eine Methode potentiell eine Instruktion enthält, die einen Schreibzugriff darstellt. Hierbei gelten nicht nur `PUTFIELD` oder `PUTSTATIC` Instruktionen, die offentsichlich Schreibzugriffe auf Felder verursachen. Es muss konservativ abgeschätzt werden, dass die `this`-Referenz auch geschrieben werden kann, die innerhalb eines kritischen Abschnitts (durch die `MONITORENTER` Instruktion) vorkommt oder in einem Array gespeichert wird. Auf diese Weise kann die Transformation korrekt durchgeführt werden.

Berechnung der Programmeigenschaften zu *Workload*

Die Berechnung der Ausführungskosten, die zu Anfang dieses Kapitels modelliert wurden, basiert auf dem Objektflussgraphen [Klo02], [KLK02], der das Ausführungsmodell darstellt. Im Wesentlichen wurde dort so genau wie möglich die dynamische Methodenbindung, welche als Knoten in diesem

Graphen abgebildet wird, zusammen mit der Menge der Erzeugungsstellen als Typbeschränkung anhand Ergebnissen der intraprozeduralen Analyse modelliert. Hierbei dient das Ergebnis der *GlobalDefUse*-Analyse (siehe Abschnitt 6.3.2) als Grundlage. Basierend auf diesem Ausführungsmodell werden die Ausführungs- sowie Kommunikationskosten der Methoden unter Verwendung des Kostemmaßes, das im Abschnitt 6.2.1 definiert wurde, berechnet. Das angewandte Abschätzungsmodell lehnt sich an das im JavaParty-System [Hau98] verwendete Berechnungsmodell an.

Erweiterungen zur hybriden Analyse

Zur Steigerung der Genauigkeit kann man weitere Analysen gezielt einsetzen. Sie können z.B. als Schleifengrenzen verwendete Laufzeitkonstanten identifizieren oder einige häufig verwendete Strukturen wie `java.util.Iteration` oder Arrays speziell behandeln. Für Array beispielsweise kann durch Feststellung der interen Variable `length` die Obergrenze einer Schleife ermittelt oder abgeschätzt werden. Für eine n-Verzweigungsstruktur können Analysetechniken für einige spezifische Strukturen in Java-Programmen eingesetzt werden, um genauere Aussagen über die Ausführungshäufigkeiten der Verzweigungen machen zu können. Beispielsweise kann für den Zweig, der durch ein `throw` zum `Exception` aufgelöst wird, der Faktor $branch($`throw`$) = 0$ angenommen werden. Oder es kann abgeschätzt werden: Die Wahrscheinlichkeit, dass die Bedingung der Form `(i == j)` true ergibt, ist viel geringer als bei einer Bedingung der Form `(a < i && i < b)`. Solche speziellen Kenntnisse können direkt benutzt werden.

Bei der statischen Auflösung dynamischer Methodenbindung wird bisher stets versucht, so genau wie möglich die Aufrufziele an einer dynamisch gebundenen Aufrufstelle durch verschiedene Analysetechniken wie *PreciseCallGraph*-Analyse oder Points-to-Analyse [Ryd03], [Ste96], [WR99], [Le00] zu finden. Wenn kein exaktes Aufrufziel bestimmt werden kann, wird zur Kostenberechnung der Mittelwert der Ausführungs- bzw. Kommunikationskosten der gefundenen Ziele genommen. Ein erweiterter Ansatz zur hybriden Analyse für ein solches Problem ist, vor der Laufzeit mit üblichen Techniken soweit wie möglich die dynamisch gebundenen Aufrufziele einzuschränken. Nur an denjenigen Stellen, wo mehr als 4 Ziele gefunden werden, wird Instrumentierungscode eingefügt, so dass zur Laufzeit die 4 ersten tatsächlichen Aufrufziele gepuffert werden. Aus den vorberechneten Ausführungskosten die-

ser ausgewählten Methoden kann der Mittelwert erneut gebildet werden, der dann zur verbesserten Verteilungsberechnung verwendet wird. Dieser Ansatz ist im Smalltalk-Umfeld bekannt unter dem Begriff *polymorphic in-line cache (PIC)* [DS84], [HCU91], [Höl94] wobei sich aus praktischen Untersuchungen der Grenzwert 4 als sinnvoll ergeben hat.

Die Kommunikationskosten entstehen genau dort, wo entfernte Methodenaufrufe stattfinden. Statisch wird herausgefunden, welche zur Laufzeit entstehenden Objekte *remote*-Objekte werden. Somit wird vor der Laufzeit schon festgelegt, an welchen Aufrufstellen entfernte Kommunikation entsteht. Dafür werden statisch Kommunikationskosten an solchen Stellen ausgerechnet. Diese Kosten werden dann einerseits direkt bei der Verteilungsberechnung in einigen Basis-ODFs verwendet, können andererseits aber zur Laufzeit als Ausgangsgröße zur nachträglichen Kommunikationsberechnung benutzt werden. Denn die vor der Laufzeit berechneten bzw. abgeschätzten Größen der Objektstrukturen bleiben auch für die Berechnungen zur Laufzeit gültig. Nur dynamisch gebundene Aufrufziele lassen sich zur Laufzeit noch verbessern.

6.4. Analyseverfahren zur *stagewise-immutable* Eigenschaft

Ein wichtiges Ziel bei der automatischen Verteilung ist es, Objekte so zu platzieren, dass sie nur lokal miteinander kommunizieren. Messungen in [HP01] zeigten, dass bis zu drei Größenordnungen zwischen einem entfernten und einem konventionellen Methodenaufruf liegen. Neben den vorgestellten Strategien zur Zusammengruppierung der Objekte mittels ihrer `isProxy`- oder `isPeer`-Eigenschaft spielt die Immutabilität der Objekte bei der Lokalitätsoptimierung eine wichtige Rolle. Immutable Objekte können zur Laufzeit repliziert werden. Entfernte Zugriffe darauf finden dann lokal statt.

Unter diesem Aspekt möchte ich in diesem Abschnitt eine neuartige Eigenschaft, die *stagewise-immutable* Eigenschaft, vorstellen. *Bezogen auf den Programmablauf* wird dadurch charakterisiert, *in welchen bestimmten Phasen während der Programmausführung* die Lese- und Schreibzugriffe auf ein Objekt stattfinden. Objekte mit der *stagewise-immutable* Eigenschaft werden ab einem bestimmten Zeitpunkt T bei der Programmausführung nur lesend zugegriffen. Nach diesem Zeitpunkt können sie dann repliziert werden. Ziel ist es,

zu erzeugende Objekte mit dieser Eigenschaft *vor der Laufzeit* zu erkennen und dafür geeignete Programmstellen als Replikationspunkte zu finden. Dazu habe ich ein statisches Analyseverfahren namens *stagewise-immutable Analyse* entwickelt, das die Objektrepräsentanten (Erzeugungsstellen) nach dieser Eigenschaft untersucht und Programmstellen als Replikationspunkte identifiziert. In diesem Abschnitt stelle ich die ablaufbezogene *stagewise-immutable* Eigenschaft dar und beschreibe anschließend das Analyseverfahren *stagewise-immutable* Analyse.

6.4.1. Die *stagewise-immutable* Eigenschaft und ihre Anwendung zur Objektreplikation

Konzeptionell gibt es für ein Laufzeitobjekt verschiedene *Phasen* bei einer Programmausführung: die *Initialisierungsphase* und die *Verwendungsphase*. Bei der Initialisierungsphase werden die Felder des Objekts mit Anfangswerten initialisiert. Nach dieser Phase ist der Zustand des Objekts komplett *initialisiert*. Diese Phase endet, nachdem eine aufgerufene Konstruktor-Methode vollständig abgearbeitet wurde. Anschließend beginnt die *Verwendungsphase*, die bis zum Ende der Programmausführung dauert. Während dieser Phase werden die Daten des Objekts *benutzt*, also *gelesen* oder/und *geschrieben*.

Definition 6.6:
Ein Laufzeitobjekt ist genau dann ***pure-immutable****, wenn seine Daten in der gesamten Verwendungsphase* ***nur gelesen*** *werden.* △

Ein solches Objekt kann zum Beginn der Verwendungsphase repliziert werden, da diese *pure-immutable* Eigenschaft gewährleistet, dass sein Zustand ab diesem Zeitpunkt bis zum Programmende unverändert bleibt. Es beschränkt sich also darauf, dass Objekte nach ihrer Initialisierung nur noch lesend zugegriffen werden dürfen.

Man kann die Immutabilität weiter fassen: Es gibt Objekte, die zwar nach der Initialisierungsphase noch schreibend zugegriffen werden, aber ab einem späteren Zeitpunkt nur noch gelesen werden. Solche Objekte können ab diesem späteren Zeitpunkt repliziert werden. Diese Eigenschaft wird als *postponed-immutable* Eigenschaft bezeichnet. Sie kann wie folgt definiert werden:

Definition 6.7:
Ein Laufzeitobjekt ist genau dann ***postponed-immutable****, wenn seine Daten in der*

Verwendungsphase noch geschrieben, aber ***ab einem bestimmten Zeitpunkt*** T *bis zum Programmende dann* ***nur lesend zugegriffen*** *werden.*

Ein Laufzeitobjekt ist genau dann ***stagewise-immutable****, wenn es die Eigenschaft* ***pure-immutable*** *oder* ***postponed-immutable*** *erfüllt.* △

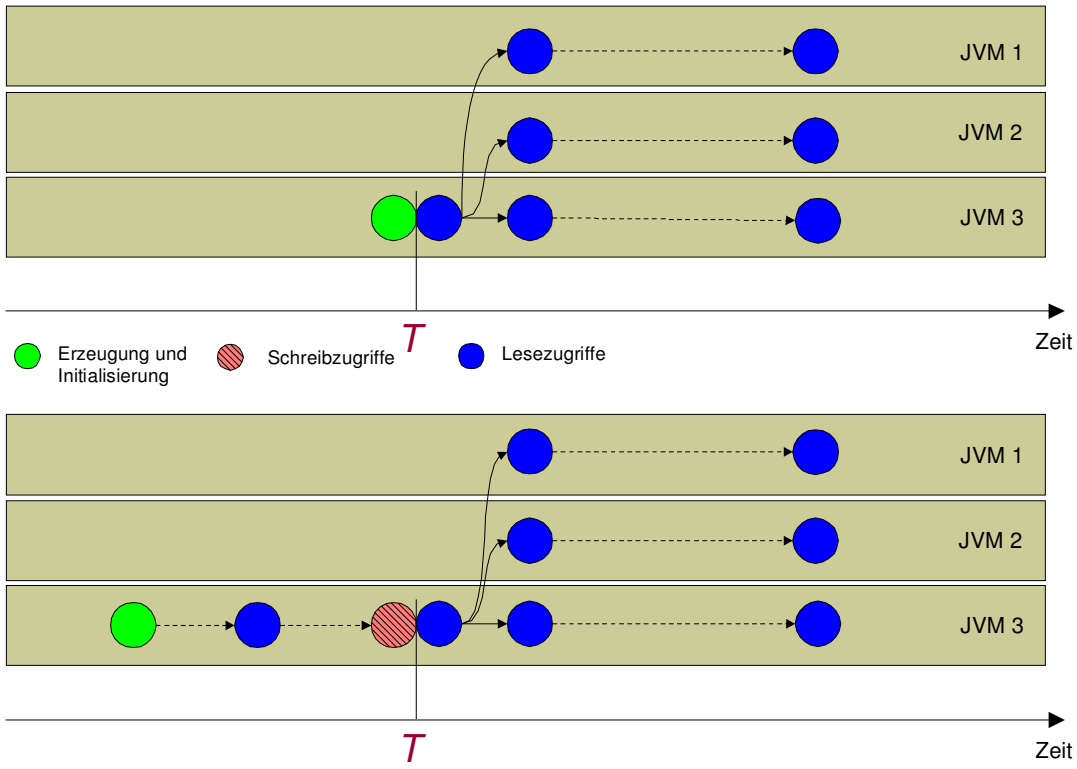

Abbildung 6.6.: Anwendung der *pure-immutable* und *postponed-immutable* Eigenschaft zur Replikation

Stagewise-immutable ist also der Oberbegriff für die beiden speziellen Fälle. Der Zeitpunkt T kennzeichnet im Prinzip das Ende aller Schreibzugriffe auf ein Laufzeitobjekt während des Programmablaufs.

Um die *stagewise-immutable* Eigenschaft eines Objektrepräsentanten zu berechnen, braucht man nur die Zugriffe nach der Initialisierungsphase zu betrachten. Denn Objekte können während ihrer Initialisierung nicht repliziert werden. Ein *stagewise-immutable* Objekt kann nach einem geeigneten Zeitpunkt T, also unmittelbar nach der Initialisierungsphase oder irgendwann in der Verwendungsphase, repliziert werden. Danach stattfindende Lesezu-

griffe werden dann nur lokal durchgeführt. Abbildung 6.6 verdeutlicht diese beiden Fälle.

Die obigen Definitionen beziehen sich auf Objekte, die erst zur Laufzeit existieren. Ziel ist es aber, die *stagewise-immutable* Eigenschaft der Objekte *vor der Laufzeit* durch statische Analyse nachzuweisen und dann Programmstellen als Replikationspunkte ebenfalls vor der Laufzeit zu erkennen. Die statische *stagewise-immutable* Analyse muss auf Objektrepräsentanten, also Erzeugungsstellen, und allen Stellen arbeiten, an denen sie potenziell verwendet werden. Wenn ein Objektrepräsentant diese Eigenschaft erfüllt, dann gilt dies für alle Laufzeitobjekte, die an dieser Stelle während der Programmausführung erzeugt werden. Wegen konservativer statischer Analyse muss ein Objektrepräsentant die *stagewise-immutable* Eigenschaft nicht erfüllen, obwohl die dort zur Laufzeit erzeugten Objekte unter bestimmten Laufzeitbedingungen diese Eigenschaft besitzen können. Die im Folgenden eingeführten Begriffe und Aspekte beziehen sich auf die statische Ablaufstruktur des Programms, auch als *interprozeduralen Ablaufgraphen* bekannt. Dieser Graph enthält alle potenziellen Programmabläufe.

Definition 6.8:
Zu einem Objektrepräsentanten sei der interprozedurale Ablaufgraph in drei disjunkte, aufeinander folgende Phasen eingeteilt: ***Initialisierungsphase****,* ***Schreibphase*** *und* ***NurLesePhase****. Es sei dann* ***Verwendungsphase := (Schreibphase,NurLesePhase)****. Alle Kontrollfluss-Knoten (CF-Knoten) in der Verwendungsphase, die potenzielle Zugriffe auf diesen Objektrepräsentanten darstellen, gehören entweder zur* ***Schreibphase*** *oder* ***NurLesePhase****. Die* ***Schreibphase*** *kann CF-Knoten enthalten, die Lese- oder Schreibzugriffe darstellen. In der* ***NurLesePhase*** *dürfen nur CF-Knoten enthalten sein, die Lesezugriffe repräsentieren. Außerdem muss die* ***NurLesePhase*** *mindestens einen CF-Knoten enthalten, der das Programmende (*`exit`*) darstellt.*

△

Diese Unterteilung innerhalb der Verwendungsphase stellt somit einen Schnitt im Ablaufgraphen dar. Nach diesem Schnitt, also nach der Schreibphase, dürfen keine weiteren Schreibzugriffe mehr erfolgen. Dies gilt für alle Abläufe des Programms. Zu einem Objektrepräsentanten kann seine NurLesePhase nur einen einzigen Knoten enthalten. Abbildung 6.7 stellt die Phasenaufteilung zum Beispielprogramm aus Abbildung 6.8 grafisch dar.

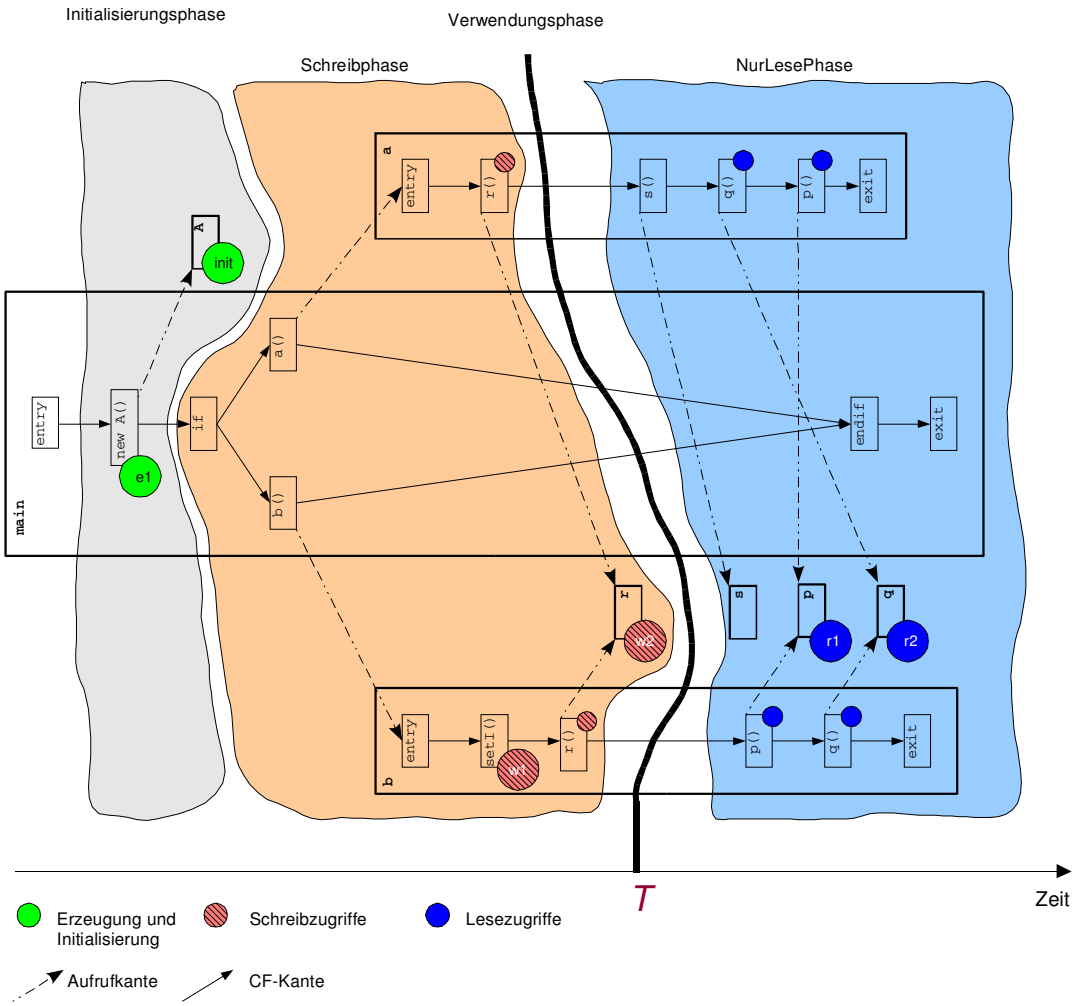

Abbildung 6.7.: Aufteilung des interprozeduralen Ablaufgraphen in drei Phasen

Die Abbildung 6.7 zeigt einen interprozeduralen Ablaufgraphen. Hier finden sich die lokalen Kontrollflussgraphen (in einem Rechteck eingegrenzt) einzelner Methoden sowie ihre Aufrufbeziehungen (gestrichelte Linien). Dies ist auch das Ergebnis der angewandten statischen Analysen. Das bedeutet, dass diese statische Ablaufstruktur alle möglichen Ausführungspfade des Programms enthält, von denen nur einige zur Laufzeit tatsächlich ausgeführt werden. Die *stagewise-immutable* Analyse muss alle diese Pfade vor der Laufzeit betrachten.

Bei diesem Szenario (in Abbildung 6.7) besitzt der Objektrepräsentant `e1` die *postponed-immutable* Eigenschaft. Das heißt, dass alle dort zur Laufzeit entste-

```
public class Test {
  static A ao;
  ... main(String...) {
    ao = new A(20);
    if (...) { a(); }
    else b();
  }
  static void b() {
    ao.setI(100);
    r(); p(); q();
  }
  static void a() {
    r(); s(); q(); p();
  }
```

```
  static void r() {ao.setI(42);}
  static void q() {ao.getI();}
  static void p() {ao.getI();}
  static void s() { do_thing(); }
}
class A {
  int i = 10;
  A(int j) {i = j;}
  int getI() {return i;}
  void setI(int j) {i = j;}
}
```

Abbildung 6.8.: Beispielprogramm zur *stagewise-immutable* Eigenschaft

henden Objekte auch diese Eigenschaft erfüllen. Nach seiner Initialisierung kann `e1` innerhalb der Methode `a()` oder `b()` geschrieben werden, nämlich durch die aufgerufenen Methoden `r()` und/oder `setI()`. Danach werden nur Lesezugriffe mittels der Methoden `q()` und `p()` darauf durchgeführt. Der Schnitt (dicke Linie in Abbildung 6.7) trennt die Verwendungsphase von `e1` in zwei disjunkte Phasen: Die Schreibphase enthält die CF-Knoten, die die Aufrufstellen zu `r()` und `setI()` repräsentieren und die NurLesePhase enthält die Lesezugriffe auf `e1`. Mit diesem Ergebnis können die an `e1` erzeugten Objekte an geeigneten Programmstellen innerhalb der NurLesePhase während der Ausführung repliziert werden. Im besten Fall soll die Replikation nach den letzten Schreibzugriffen erfolgen. Dann erzielt man maximalen Gewinn an lokalen Zugriffen auf `e1`.

Um die Verwendungsphase im Allgemeinen zu trennen, braucht man einen legalen Schnitt des interprozeduralen Ablaufgraphen. Dieser wird wie folgt definiert:

Definition 6.9:
*Ein **Programmschnitt**, der alle gültigen Ausführungspfade in der Verwendungsphase eines Objektrepräsentanten schneidet, heißt ein **legaler Schnitt**, wenn auf solchen Pfaden nach dem Schnittpunkt kein Schreibzugriff stattfindet.* △

Ein trivialer, legaler Schnitt ist die Trennung des `exit`-Knotens von den restlichen des interprozeduralen Ablaufgraphen. Nach diesem Schnitt wird auf jedem Fall der `exit`-Knoten ausgeführt, der weder Schreib- noch Lesezugriffe enthält. Im *pure-immutable* Fall wäre ein legaler Schnitt genau dort, wo die Initialisierungs- und Verwendungsphase getrennt werden. In diesem Fall ist der Replikationspunkt genau die Programmstelle unmittelbar nach der Aufrufstelle der verwendeten Konstruktor-Methode. In Abbildung 6.7 ist ein solcher legaler Schnitt (dicke Linie) angegeben. Dieser ist sogar ein „guter“ Schnitt, denn er liegt unmittelbar nach den „letzten“ Schreibzugriffen.

Intuitiv braucht man für einen Objektrepräsentanten den Schnitt „einfach“ vom `exit`-Knoten des Programms sukzessiv „nach links“ zu schieben, bis ein Schreibzugriff „vom Schnitt getroffen wird“, dann hört man auf. Dabei muss beachtet werden, dass bei der Ausführung der CF-Knoten nach dem Schnitt keine Schreibzugriffe mehr stattfinden. Ein legaler Schnitt unterteilt im Prinzip die Verwendungsphase in die oben genannte Schreib- und Nur-LesePhase.

Auch wenn es einfach erscheinen mag, ist es eine große Herausforderung, diese Eigenschaft für Objektrepräsentanten (Erzeugungsstellen) vor der Laufzeit nachzuprüfen. Außerdem müssen ebenfalls vor der Laufzeit die gültigen Programmstellen zur Anwendung der Replikation gefunden werden, so dass zur Laufzeit nach der Replikation nur Lesezugriffe durchgeführt werden, und zwar für jede Programmausführung.

6.4.2. Analysetechniken zur Berechnung der *stagewise-immutable* Eigenschaft

Die Aufgabe der *stagewise-immutable* Analyse ist es, für jeden Objektrepräsentanten (Erzeugungsstelle) einen legalen, „guten“ Schnitt zu finden, so dass

nach diesem Schnitt Lesezugriffe vorkommen. Dann müssen geeignete Programmstellen nach dem Schnitt als Replikationspunkte identifiziert werden, so dass nach der Replikation der Objekte zur Laufzeit so viel Lesezugriffe wie möglich durchgeführt werden. Replikation wird für Objekte nicht durchgeführt, wenn kein einziger Lesezugriff nach einem gefundenen Schnitt stattfindet.

Kriterien für einen guten Schnitt *Ein legaler Schnitt ist ein **guter Schnitt**, wenn er alle CF-Kanten mit folgenden Eigenschaften schneidet: Diese CF-Kanten werden **zum ersten Mal ausgeführt, nachdem die Schreibzugriffe zum letzten Mal ausgeführt wurden**.*

Ein guter Schnitt liegt also unmittelbar nach den CF-Knoten, die die als Letzte ausgeführten Schreibzugriffe darstellen. Die NurLesePhase enthält somit alle potenziellen Lesezugriffe, die nach dem letzten Schreiben statttfinden können.

Bevor die *stagewise-immutable* Eigenschaft eines Objektrepräsentanten nachgeprüft wird, müssen grundlegende Informationen zum Programm vorliegen. Das sind u.a. die Kontrollfluss-Struktur sowie Aufrufbeziehungen der Methoden des vollständigen Programms. Um diese ablaufbasierte Eigenschaft vor der Laufzeit zu berechnen, braucht man eine statische Programmanalyse, die die Ausführungsreihenfolge der CF-Knoten ermittelt. Diese Reihenfolge wird durch die interprozeduralen Post-Dominator-Relationen zwischen einzelnen CF-Knoten des gesamten Programms modelliert. Mit dieser Information kann statisch bewiesen werden, dass beispielsweise ein CF-Knoten cf_1 zur Laufzeit ausgeführt wird, falls cf_2 ausgeführt worden ist. Für die Berechnung eines guten Schnittes kann mit Hilfe solcher Relation vor der Laufzeit gezeigt werden, dass die erste Ausführung von einem CF-Knoten immer nach der letzten Ausführung von einem anderen CF-Knoten erfolgt.

Außerdem müssen Schreib- und Lese-Zugriffe zu einer Erzeugungsstelle statisch erkannt werden, die in Abbildung 6.7 als Kreise symbolisiert wurden (markierte Kreise für Schreib-Zugriffe). Diese Information wird von der oben vorgestellten *GlobalDefUse*-Analyse beschafft. Die Verfügbarkeit derartiger Information ist eine notwendige Voraussetzung zur *stagewise-immutable* Analyse.

Im Folgenden werde ich kurz auf die interprozedurale Post-Dominator-Relation eingehen, wobei einige für die *stagewise-immutable* Analyse relevante Details ausführlich erläutert werden. Das Verfahren selbst wird sehr aus-

führlich in [P. 01] beschrieben. Anschließend wird die Analysetechnik zur *stagewise-immutable* Eigenschaft sowie das Finden der Replikationspunkte dargestellt.

Interprozedurale Post-Dominator-Relationen

Die Relationen zwischen den Knoten innerhalb eines Kontrollflussgraphen (CFG) einer Methode wie die *dom-*, *idom-* oder *pdom*-Beziehung können mittels klassischer Analyseverfahren [Muc97] bestimmt werden. Seien cf_1 und cf_2 zwei Kontrollflussknoten innerhalb einer Methode. cf_1 post-dominiert cf_2 im klassischen Sinn, wenn cf_1 auf jedem Ausführungspfad von cf_2 zum $exit$-Knoten des zugehörigen CFG liegt. Nach [P. 01] wird hier die Definition für die interprozedurale Post-Dominator-Relation zwischen zwei CF-Knoten angegeben:

Definition 6.10:
*Seien cf_1 und cf_2 zwei CF-Knoten, die in verschiedenen CFGs, also in verschiedenen Methoden liegen können. cf_1 post-dominiert den Knoten cf_2 interprozedural, wenn cf_1 auf **jedem Ausführungspfad** von cf_2 bis zum Programmende $programExit$ liegt. Diese Relation sei mit $cf_1\ IPOSTDOM_{Must}\ cf_2$ bezeichnet und als **Must-IPOSTDOM-Relation** definiert.*

*Ein CF-Knoten cf_1 may-post-dominiert einen anderen Knoten cf_2 interprozedural, wenn cf_1 auf **mindestens einem Ausführungspfad** von cf_2 bis zum Programmende $programExit$ liegt. Diese Relation sei als $cf_1\ IPOSTDOM_{May}\ cf_2$ gekennzeichnet und als **May-IPOSTDOM-Relation** definiert.*

*Die **Must-IPOSTDOM-Relation** und **May-IPOSTDOM-Relation** stellen partielle Ordnungen dar.* △

Das Verfahren in [P. 01] nimmt als Eingabe den gesamten Analysekontext und berechnet solche interprozeduralen (Must-)Post-Dominator-Beziehungen. Durch Änderung des *Meet*-Operators (Θ) in der Phase 2 des Verfahrens kann anstatt das All-Problem das Existenz-Problem gelöst werden. Das heißt, dass die May-IPOSTDOM-Relationen mit $\Theta = \cup$ und die Must-IPOSTDOM-Relationen mit $\Theta = \cap$ berechnet werden können. Die Ergebnisse zur *May-* und *Must-IPOSTDOM-Relationen* werden als DAGs dargestellt. Mit diesen Informationen ist man nun in der Lage, die *stagewise-immutable* Analyse durchzuführen.

Die *stagewise-immutable* Analyse

Basierend auf der interprozeduralen Post-Dominator-Relation kann die Analyse für eine Erzeugungsstelle die Menge der CF-Knoten im Programm bestimmen, die zur NurLesePhase gehören und nach deren Ausführung keine Schreibzugriffe stattfinden. Die Trennung zwischen den CF-Knoten der NurLesePhase und den restlichen Knoten wird durch einen legalen Schnitt definiert. Die notwendige Bedingung für einen solchen CF-Knoten der NurLesePhase lautet, dass nach seiner Ausführung bis zum Programmende kein einziger Schreibzugriff mehr durchgeführt wird. Das heißt, dass *keiner der Schreibzugriffe* diesen CF-Knoten *may-post-dominiert*. Die Kriterien für einen guten legalen Schnitt können mit Hilfe der interprozeduralen Post-Dominator-Relation so umformuliert werden:

*Ein legaler Schnitt ist ein **guter Schnitt**, wenn er alle CF-Kanten schneidet, die zu CF-Knoten mit folgenden Eigenschaften führen: Diese CF-Knoten müssen die vor ihnen auszuführenden Schreibzugriffe **must-post-dominieren, und von keinem der Schreibzugriffe may-post-dominiert werden**.*

Mit dieser Bedingung befinden sich also alle CF-Knoten direkt hinter dem guten Schnitt, die zum ersten Mal ausgeführt werden, nachdem die Schreibzugriffe zum letzten Mal ausgeführt wurden. Durch die Verwendung der Must-IPOSTDOM-Relation wird zum einen die Korrektheit des guten Schnitt auf jedem Fall garantiert. Beim sukzessiven „nach links“ Schieben des Schnittes werden nur die CF-Knoten nach der Must-IPOSTDOM-Relation betrachtet. Dadurch können irrelevante CF-Kanten (z.B. Schleifenkanten) übersprungen werden. Außerdem verursacht die Berechnung für die Must-IPOSTDOM-Beziehung keinen zusätzlichen Aufwand.

Unter Berücksichtigung dieser Kriterien kann man nun einen legalen guten Schnitt finden. Wenn zu einem Objektrepräsentanten ein solcher Schnitt gefunden wird, nach dem auch potentielle Lesezugriffe existieren, dann besitzen alle dort zur Laufzeit entstehenden Objekte die *stagewise-immutable* Eigenschaft. Dafür kann dann nach *geeigneten Programmstellen* als Replikationspunkte gesucht werden.

Beispiel Zur Verdeutlichung der Verwendung der Post-Dominator-Replation zur Ermittlung der *stagewise-immutable* Eigenschaft sowie der Replikationsstellen wird zunächst das Ergebnis der *stagewise-immutable* Analyse zum obigen Programmbeispiel in Abbildung 6.9 gezeigt.

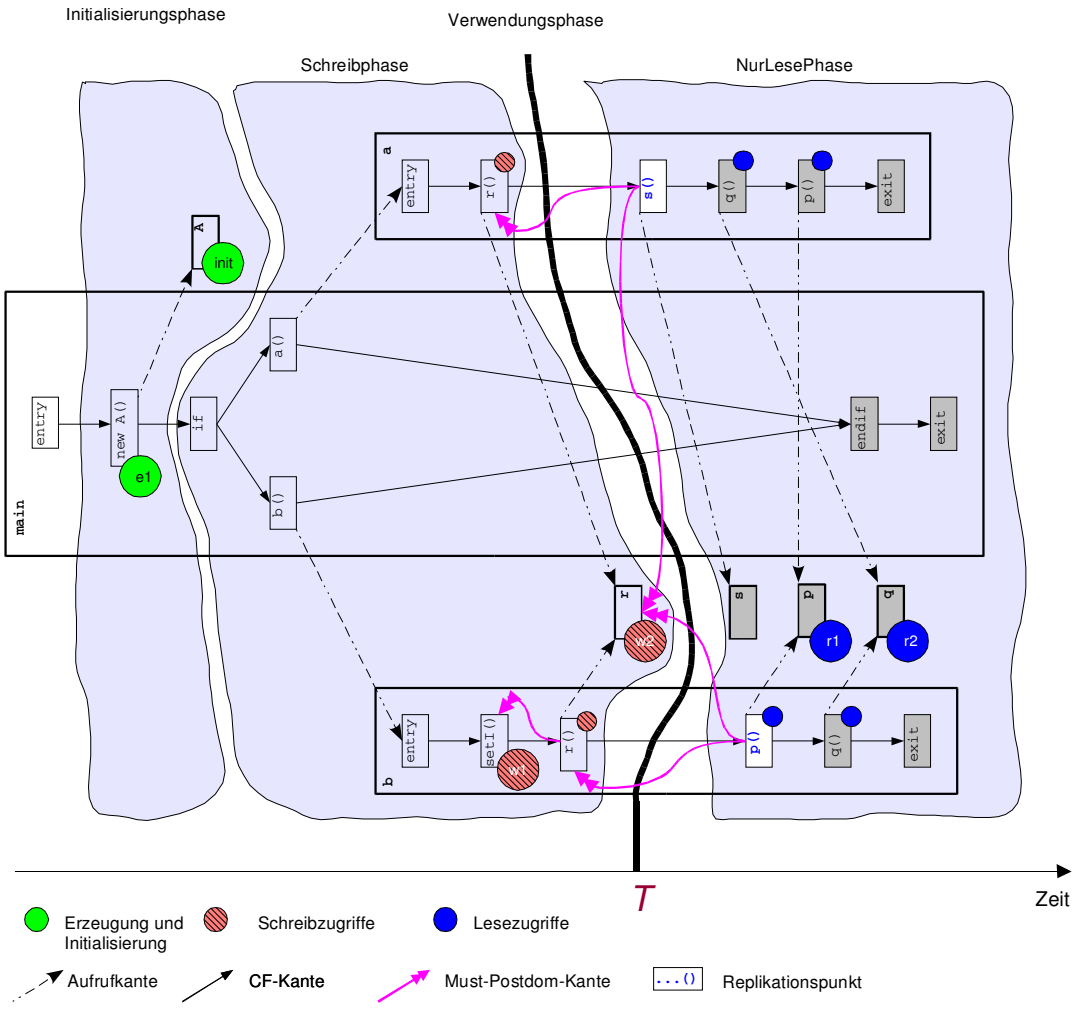

Abbildung 6.9.: Gefundene Replikationspunkte mit Must-Post-Dominator-Bedingung

In Abbildung 6.9 wird der Schnitt unmittelbar nach den als letzten ausgeführten Schreibzugriff-Knoten definiert. Wenn die CF-Knoten hinter dem Schnitt bis zum Programmende ausgeführt werden, findet auch kein einziger Schreibzugriff mehr statt. Außerdem werden am Beispiel auch zwei Stellen als Replikationspunkte (in Weiß hervorgehoben) gefunden, die die Aufrufstelle der Methoden `s()` und `p()` darstellen. Das sind günstige Stellen, weil sie zum einen alle potenziellen Schreibzugriffe must-post-dominieren. Zum anderen befinden sie sich direkt hinter dem Schnitt. Das zweite Kriterium garantiert, dass zur Laufzeit das Objekt nur nach dem letzten Schreibzugriff repliziert wird und somit sein Zustand konsistent bleibt. Mit dem ersten

Kriterium können alle nach dem Schnitt stattfindenden Lesezugriffe auf dem Replikat durchgeführt werden.

Nun kann das Verfahren zum Finden eines guten legalen Schnitts und zum Identifizieren günstiger Replikationsstellen allgemein beschrieben werden.

Finden eines guten, legalen Schnitts Das Finden eines legalen guten Schnitts für einen Objektrepräsentanten läuft folgendermaßen ab: `exit` - also der CF-Knoten für das Programmende - wird in die Ergebnismenge eingefügt. Ausgehend vom `exit` betrachtet die Analyse die Nachfolger-Knoten des DAGs, der die *interprozedurale Must-Post-Dominator-Relation* repräsentiert. Zu beachten ist, dass die Kanten dieses Graphen die Gegenrichtung des interprozeduralen Ablaufgraphen zeigen. Dabei wird überprüft, ob diese CF-Knoten von irgendeinem Schreibzugriff *may-post-dominiert* werden. Wenn das nicht der Fall ist, dann werden sie in die Ergebnismenge aufgenommen. Diese Menge enthält also nur die CF-Knoten, die zur NurLesePhase gehören und hinter dem Schnitt liegen. Sukzessiv für die in die Ergebnismenge aufgenommenen Knoten werden ihre Nachfolger im obigen DAG so weiter analysiert. Der Schnitt wird somit immer weiter nach links verschoben. Wenn ein betrachteter Knoten einen Schreibzugriff darstellt, dann wird dieser Knoten nicht in die Ergebnismenge aufgenommen. Das Verfahren terminiert, wenn der Schnitt unmittelbar vor allen als letzten identifizierten Schreibzugriffen liegt, oder wenn die Initialisierungsphase erreicht wird.

Günstige Replikationspunkte Das beschriebene Verfahren findet dabei auch günstige Replikationspunkte. Das sind Programmstellen, die zur Laufzeit nach den letzten Schreibzugriffen ausgeführt werden und danach möglichst noch viele Lesezugriffe stattfinden. Diese Stellen sollen also direkt hinter dem Schnitt liegen. Außerdem muss die Menge der Replikationspunkte die Menge der Schreibzugriffe *must-post*-dominieren. Diese Mengenrelation bedeutet, dass alle gefundenen Replikationsstellen zusammen alle Schreibzugriffe must-post-dominieren, also nach diesen Schreibzugriffen ausgeführt werden müssen.

Verfahren Zur Vereinfachung wird das beschriebene Verfahren zur Berechnung der *stagewise-immutable* Eigenschaft und zum Finden der Repliaktionspunkte für eine Erzeugungsstelle `e` im Folgenden in einer Pseudo-Form beschrieben. Seien $IMMIPOSTDOM_{\text{Must}}(b)$ die Menge der CF-Knoten, die von b

direkt interprozedural *must-post-dominiert* werden. Das sind also die direkten Nachfolger-Knoten im Graphen (DAG), der die interprozeduralen Must-Post-Dominator-Beziehungen darstellt. $exit$ sei der CF-Knoten, der das Programmende repräsentiert.

***stagewise-immutable* Verfahren**

```
EINGABE: Erzeugungsstelle e

/* Initialisierung */
resultSet = empty;
worklist  = empty;
replicationSet = empty;

/* exit gehört zum Ergebnis, und von exit wird gestartet */
resultSet.insert(exit);
worklist.insert(exit);

WHILE (worklist not empty) {
   b = worklist.removeFirst();
   M = IMMIPOSTDOM_Must(b);
   FORALL m in M DO {
   /* b interprozedural must-post-dominiert m */
      W = WRITESET(e);
      IF (m is not in W) THEN {
      /* m stellt keinen Schreibzugriff dar */
         /* wenn keiner der Schreibzugriffe m may-postdominert,
            füge m in resultSet */
         boolean flag = true;
         FORALL w in W DO {
            IF (w inter may-post-dom m) THEN {
               flag = false;
               break;
            }
         }
         IF (flag) THEN {
            resultSet.insert(m);
            /* Nachfolger von m werden weiter betrachtet,
               um den Schnitt weiter nach links zu schieben */
            worklist.insert(m);
         }
      }
      ELSE {
      /* m stellt einen Schreibzugriff dar */
         replicationSet.insert(b);
      }
   }
}
```

Die günstigen Replikationspunkte werden dabei so gewählt, dass das Objekt nach der letzten Ausführung des letzten Schreibzugriffs repliziert wird. Beim Einfügen des Replikationscodes während der Transformationsphase müssen zusätzlich noch einige Schritte durchgeführt werden. So soll der Replikationscode nicht für ein Objekt mehrfach ausgeführt werden, z.B. innerhalb einer Schleife oder an zwei Programmstellen, die auf jedem Fall hintereinander ausgeführt werden.

Das vorgestellte Verfahren kann einfach für alle Erzeugungsstellen erweitert werden.

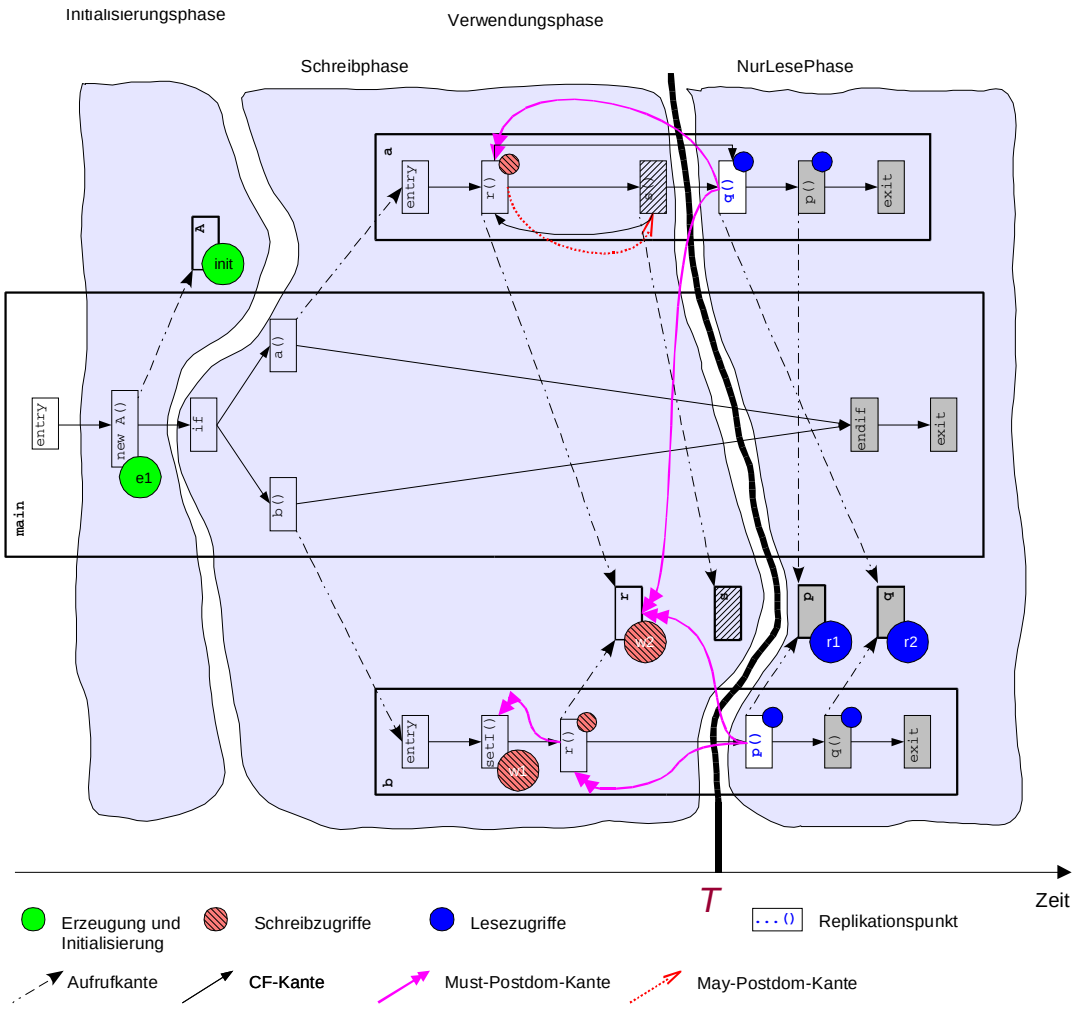

Abbildung 6.10.: MAY- und MUST-POSTDOM

Abbildung 6.10 zeigt ein Beispiel, das die Korrektheit des Verfahrens unter der Verwendung der *may*- und *must-post-dominator*-Relation zeigt. Die Auf-

rufstelle zur Methode `s()` in der Methode `a()` kann hier nicht mehr als Replikationspunkt definiert werden. Denn die Aufrufstellen zur Methode `r()` und `s()` befinden sich in einer Schleife und der `r()`-Knoten, der einen Schreibzugriff darstellt, den `s()`-Knoten may-post-dominiert. Das heißt, dass nach der Ausführung von `s()` die Ausführung von `r()` stattfinden kann. Dagegen erfüllt die Aufrufstelle zu `q()` die Bedingung, dass diese nach der letzten Ausführung von `r()` ausgeführt wird. Dort kann das Objekt `e1` repliziert werden.

6.4.3. Analyse zur *pure-immutable* Eigenschaft

Das oben vorgestellte Analyseverfahren kann im Allgemeinen auch dazu verwendet werden, um Objektrepräsentanten mit der *pure-immutable* Eigenschaft zu erkennen. Der Schnitt würde dann die Trennung der Initialisierungs- und der Verwendungsphase definieren. Aufgrund der konservativen Abschätzung über dynamische Methodenbindung kann die obige *stagewise-immutable* Analyse allerdings nicht alle Erzeugungsstellen erkennen, die die *pure-immutable* Eigenschaft besitzen. Deshalb wurde dieses statische Analyseverfahren speziell entwickelt, um Objektrepräsentanten anhand ihrer Erzeugungsstellen und zugehöriger Zugriffe nach der *pure-immutable*-Eigenschaft zu finden. Das Verfahren startet „vom anderen Ende“, nämlich an der Erzeugungsstelle eines Objekts.

Mittels der interprozeduralen Def-Use-Analyse (siehe Abschnitt 6.3.2) werden zu jeder Erzeugungsstelle (Def), die als Definition betrachtet wird, alle zugehörigen Verwendungen (Uses) im ganzen Analysekontext gefunden. Diese Verwendungen werden einzeln unter die Lupe genommen und danach untersucht, ob es sich hierbei um einen Schreib- oder Lesezugriff handelt. Die Schreibzugriffe werden außerdem genauer analysiert, ob sie zur Initialisierung in Konstruktoren verwendet werden oder nicht. Wenn alle Verwendungen, außer denjenigen zur Initialisierung, nur Lesezugriffe darstellen, dann besitzen die an dieser Erzeugungsstelle entstehenden Objekte die *pure-immutable*-Eigenschaft.

Um festzustellen, ob ein *Use* zur Initialisierung im Konstruktor verwendet wird, muss das Vorwärts-Datenflussproblem innerhalb der Konstruktor-Methode interprozedural gelöst werden. Der Grund dafür ist , dass aus dem Konstruktor auch Methoden mit Zugriffen auf das erzeugte Objekt aufgerufen werden können. Diese statische Analysetechnik ist eine einfachere

Variante der *immutable-fields Analysis* in [Thi01], wobei aufgerufene Super-Konstruktor(en) und Methoden innerhalb des betrachteten Konstruktors mit ihren Seiteneffekten berücksichtigt werden, insbesondere der Escape-Status der `this`-Referenz.

Diese Analyse wird zusätzlich angewandt, um die *pure-immutable* Eigenschaft für Objektrepräsentanten zu berechnen, die Teil der *stagewise-immutable* Eigenschaft ist und mittels der oben beschriebenen *stagewise-immutable* Analyse aufgrund der konservativen Abschätzung nicht entdeckt werden kann. Im Abschnitt 8.2 in Kapitel 8 wird eine empirische Untersuchung nach der *stagewise-immutable* Eigenschaft einiger ausgewählter Anwendungen und Benchmarks beschrieben.

6.5. Verwandte Arbeiten

In diesem Abschnitt werden speziell die verwandten Arbeiten der Analysetechniken diskutiert, die zur Berechnung der Objektverteilung eingesetzt werden, insbesondere zur *stagewise-immutable* Analyse. Hierbei werden die verschiedenen Konzepte zur Objektreplikation in anderen Systemen beschrieben.

Diskussionen und konzeptionelle Vergleiche zwischen den verschiedenen statischen Analyseverfahren wie z.B. Points-to-Analyse, Call-Graph-Konstruktion, Escape-Analyse usw., die in dieser Arbeit als Standardtechniken betrachtet werden, wurden schon in sehr vielen Veröffentlichungen sowie im Kapitel „verwandte Arbeiten bzw. *related works*“ angestellt. Einige sind [Thi01], [Klo02], [KLK02], [P. 01], [Le00]. Deshalb werde ich auf diese nicht nochmal eingehen. Hier werde ich nur die besonderen Aspekte der verwendeten bzw. entworfenen Verfahren zur automatischen Verteilung beschreiben.

6.5.1. Immutable-Analysen

Statische Analysen

Es gibt ein ablaufbasiertes, statisches Analyseverfahren, das die Immutabilität der Objekte, aber nicht die *postponed-immutable* Eigenschaft untersucht. Das ist die die *immutable-fields Analysis* in [Thi01]. Sonst sind mir Verfahren zur *stagewise-immutable* Analyse nicht bekannt. Bei der genannten *immutable-fields*

Analysis werden die Felder einer Klasse bezüglich Immutabilität untersucht. Wenn alle Felder einer Klasse *immutable* sind, dann besitzen alle Objekte dieser Klasse die *immutable* Eigenschaft. Meine Analyse zur *pure-immutable* Eigenschaft basiert auf einer einfachen Variante dieser *immutable-field* Analyse, die zur speziellen Behandlung von Konstruktoren eingesetzt wurde.

Ein zu meiner *stagewise-immutable* Analyse vergleichbarer Ansatz ist die *trust and dependence analysis* [P. 97]. Zur *dependence analysis* muss das Programm zunächst in die dort entworfene Abhängigkeitsalgebra transformiert werden. Mit dieser Algebra können die Datenflüsse in einem Programm als Kanäle mit festgelegten Eigenschaften ausgedrückt werden. Die Daten dürfen über diese Kanäle nur geschickt werden, wenn diese Eigenschaften erfüllt sind. Wenn die aufgestellten Eigenschaften verletzt werden, werden die Programmabschnitte dort als kritisch markiert. Die *trust analysis* wird eingesetzt, um sog. gefährliche Operationen datenflussbasiert aufzusprüren und zu validieren. Gefährliche Operationen, die die Bedingungen verletzen, können diejenigen sein, die beispielsweise den Wert einer Variable ändern, welcher aber nicht geändert werden darf.

Konzeptionell basiert die *trust and dependence analysis* auch auf dem Ablauf und den Datenfluss-Informationen des Programms. Die *stagewise-immutable* sowie *pure-immutable* Analyse entdecken in diesem Sinne auch verschiedene Phasen im Programmablauf, in denen bestimmte Operationen ausgeführt werden, nämlich Schreib- oder Lesezugriffe.

Objektreplikation

Der Ansatz, Objekte in einer verteilten Laufzeitumgebung auf mehreren Rechnern zu replizieren, ist nicht neu. Damit werden sonst entfernte Zugriffe zu lokalen und Kommunikationskosten eingespart.

So können im Manta-System [vNMB+99], [MKB00], [MKB01] Objekt-Cluster (auch als *Wolken* bezeichnet), die nur Verweise auf Objekte desselben Clusters enthält, effizient repliziert werden. Zugriffe auf die Objekte einer Wolke finden mittels eines ausgewählten Objekts, des Root-Objekts, statt. Die replizierten Objekte brauchen nicht konstante Objekte zu sein. Durch statische Programmanalyse werden vor der Laufzeit Methoden identifiziert, die nur Lesezugriffe darstellen. Diese Methoden werden lokal innerhalb der Objekte eines Clusters ausgeführt. Schreibende Methoden werden durch die sog. *function shipping* Technik auf allen Replikaten durchgeführt. Um die Aktualisie-

rung aller Replikate konsistent zu halten, werden die Schreibzugriffe in einer totalen Ordnung mit Hilfe der *totally-ordered group communication* [BBH+98] auf allen Maschinen ausgeführt. Bei diesem Ansatz muss eine Wolke, also ausgewählte Objekte mit einem einzigen Zugang als Root, zur Replikation durch den Programmierer festgelegt werden. In meinem Konzept werden Objektrepräsentanten mit der *stagewise-immutable* Eigenschaft automatisch erkannt und die Objekte zur Laufzeit zum geeigneten Zeitpunkt automatisch repliziert.

Javanaise [HL98] verwendet eine ähnliche Technik, hier muss der Anwender aber selber die zusammenhängenden Objektgruppen identifizieren. Das Mocha-System [TAS98] unterstützt auch Objektreplikation, hier wird aber auf die strikte Konsistenz der Replizierten verzichtet.

Im JavaParty-System wird ebenfalls Replikation unterstützt. Hierbei werden statische Konstanten [PZ97], [Bor02] und konstante Arrays repliziert, welche im Quellcode als `static final` bzw. mit einem eingeführten Modifizierer `const` deklariert werden müssen. Mittels meiner Analysen können nicht nur solche speziellen Objekte (als `static final` deklariert), sondern auch andere nicht explizit als konstant deklarierte Objekte automatisch als *stagewise-immutable* erkannt werden.

Die Diplomarbeit von Weißenborn [Wei03] beschäftigte sich mit Migrations- und Replikationsstrategien. Hierbei wurden auch statische Analysen (siehe Abschnitt 6.3.3) eingesetzt, um den Schreib- und Lese-Status der Methoden zu ermitteln und diese zur Transformation zur Replikation bzw. Migration zu verwenden. Es kann vorkommen, dass einige zu replizierende Objekte nicht konstant sind. Bei nachträglichen Schreibzugriffen werden die Replikate nach der dort entworfenen Strategie aktualisiert. Juggle [SH98] (siehe Abschnitt 4.1.4) setzte Techniken zur Objektreplikation in ähnlicher Weise ein, wenn die Anzahl der Lesezugriffe nach einem Schreibzugriff einen vorkonfigurierbaren Wert überschreitet.

6.5.2. Analysetechniken zur Objektverteilung

Um statische Programmeigenschaften wie `isProxy` bzw. `isThreadLocal` zu bestimmen oder Abschätzung zu Ausführungskosten bzw. Kommunikationskosten durchzuführen, wurde die Escape-Analyse bzw. einige Varianten davon [Thi01], [WR99], [BH99], [CGS+99], [BCCH95], [HBCC99] eingesetzt. Die dort beschriebenen Ansätze wurden für mehr-

strängige Java-Programme entworfen, werden aber meist zu anderen Optimierungszwecken verwendet, z.B. zum Eliminieren überflüssiger Synchronisation in Java-Programmen oder zur Beschleunigung des sequentiellen Ablaufs durch das Verlagern lokaler Objekte vom Heap auf den Stack. Diese Optimierung - also die Erkennung stack-lokaler Objekte - wurde auch für die Lokalitätsoptimierung zur Verteilung benutzt, damit Proxy- oder Thread-lokale Objekte nur lokal erzeugt werden.

In [Rin01a] wurde allgemein über die Points-to-Analyse für *multithreaded* Programme zusammen mit ihren Anwendungsgebieten diskutiert und es wurden Vergleiche zu zahlreichen verwandten Arbeiten gezogen. Wir haben eine einfache Variante der Points-to-Analyse [WR99], [SH97], [Ste96], [Rin01b],[RR99], [Le00] in unserer Analyseumgebung [Thi01] realisiert. Unsere Points-to-Version funktioniert auch für *multithreaded* Java-Programme und ist fluss- und kontext-insensitiv. Sie liefert Informationen zur Typinferenz des gesamten Analysekontextes, welche als Grundlage für weitere Analysetechniken wie die vorgestellte PreciseCallGraph-Analyse (siehe Abschnitt 6.3.1) und die interprozedurale Def-Use-Analyse (im Abschnitt 6.3.2) dienen.

Bei der Konstruktion eines Aufrufgraphen in unserer PreciseCallGraph-Analyse wird das Ergebnis dieser Typinferenz dazu benutzt, um möglichst genau dynamisch gebundene Aufrufziele für den Receiver statisch aufzulösen. Die Problematik bezüglich spezieller Verbindungen im Aufrufgraphen wie Kanten zu `<clinit>`-Methoden oder von `start()`-Methode zu `run()`-Methode wurde in [Wes04] diskutiert. Die Arbeit von Wessels beschäftigte sich ebenfalls mit der Konstruktion von Aufrufgraphen, die auf der Grundlage der Kombination mehrerer, in Schwierigkeitsstufen aufgeteilter Verfahren [TP00] beruht. Der Ansatz zur FTA-Analyse in [TP00] wurde auch in unserer GlobalDefUse-Analyse benutzt, wobei die Analyseinformation über Definitions-/Verwendungsketten fluss- und kontext-insensitiv über alle Methodenaufrufstellen hinaus aufgebaut wird. Hier werden erreichende Definitionen für formale Parameter der Methoden über alle relevanten Aufrufstellen gesammelt. Copy-Propagation wird ebenfalls berücksichtigt und Definitionen werden sowohl zu direkten als auch zu indirekten Benutzungen verkettet.

Die statische Objektgraph-Analyse [Spi99b] stellt eine Approximation der Laufzeitobjekte als Knoten dar und beschreibt die Erzeugungs-, Verwendungs- und Referenzrelationen zwischen diesen durch Kanten. Durch die vom Anwender selbst festgelegten Kriterien kann eine Verteilung daraus abgeleitet werden. Dies ist eine quasi-automatische Verteilung, bei der der

Anwender selbst das Programm partitionieren muss. Außerdem ist die Abschätzung der Objektbeziehungen hierbei sehr konservativ.

Wie im Abschnitt 6.3.3 beschrieben, werden bei der Berechnung und Abschätzung der Workload und Kommunikationskosten verschiedene Analysetechniken verwendet, die das Objektflussgraph-Modell [Klo02] als Berechnungsstruktur benutzen. Anders als bei JavaParty, wo Klassen und Methoden in sog. Konturen aufgesplittet werden und daher die polymorphen Verwendungen von ihnen in monomorphe überführt werden können [Hau98], [PH00], [HP01], werden in meinem Konzept verschiedene Analysetechniken, u.a. die Points-to-, PreciseCallGraph-Analyse verwendet, um die Information zur Typinferenz zu bestimmen.

Das Abschätzungsverfahren zu Workload und zur Kommunikation lehnt sich stark an das Modell der Verteilungsanalyse in [Hau98] an. Das in diesem Konzept verwendete Kostenmodell (siehe Abschnitt 6.2.2) zur hybriden Analyse ist aber flexibel, dass dynamische Aspekte eingebracht werden können und es eine Verbesserung der Verteilungsentscheidung zur Laufzeit ermöglicht.

Die *polymorphic in-line cache (PIC)*-Technik [DS84], [HCU91], ist eine bekannte Technik aus Smalltalk-Implementierungen (und in Self [Höl94]). Hier werden an einer dynamisch gebundenen Aufrufstelle zur Laufzeit die tatsächlichen Aufrufziele gespeichert und beim nächsten Aufruf werden die benutzten Methoden mittels *lookup*-Technik aus dem Cache geholt und wiederverwendet. Diese Idee kann in dem im Abschnitt 6.2.2 beschriebenen Kostenmodell verwendet werden, um die Kosten an den dynamisch gebundenen Aufrufstellen besser zu berechnen. Dieses Konzept wurde aber noch nicht im JScatter-System erprobt.

All diese dynamischen Aspekte, die bei der hybriden Analyse betrachtet und zur Verbesserung der Verteilungsentscheidung zur Laufzeit angewandt werden, haben ein ähnliches Ziel wie die Konzepte zur dynamischen Lastbalancierung, die in Systemen wie Juggle (beschrieben in Abschnitt 4.1.4, [SH98]) oder cJVM [AFT99], [AFT+00] verwendet werden. Die automatische Lastverteilung wird in der System-eigenen JVM realisiert: das Lastmaß bei Juggle richtet sich nach der Anzahl der Zugriffe aller Threads auf ein Objekt, die in einer bestimmten Zeitspanne gezählt werden. Die Auslastung der beteiligten Rechnerknoten wird periodisch überprüft und nach jeder Objektplatzierung korrigiert.

Mit diesem Ansatz werden zur Laufzeit anhand der gesammelten Laufzeitdaten sowie der ständig aktualisierten Rechnerlasten die Verteilungsentschei-

dungen verbessert und die Objekte danach entsprechend (um)platziert. Bei der hybriden Analyse werden vor der Laufzeit alle Programmeigenschaften soweit wie möglich berechnet. Die fehlende Information wird nachträglich zur Laufzeit beschafft und zur Verbesserung der Entscheidung verwendet. Im Vergleich mit der Lastbalancierungstechnik ist die Methodik und die Art und Weise der Informationsverwendung im *JScatter* dennoch sehr verschieden, obwohl die Ziele sich ähneln.

7. Infrastruktur und Realisierung des Systems *JScatter*

Inhalt

Das *JScatter*-System wurde entwickelt, um mehrsträngige Java-Anwendungen auf die virtuellen Maschinen in verschiedenen miteinander vernetzten Rechnern automatisch zu verteilen. Es entstand ursprünglich im Rahmen der Projektgruppe „Verteilung von parallelen Java-Anwendungen“ [FFF+02] an der Universität Paderborn unter dem Namen *ParJava* und wird seitdem weiterentwickelt.

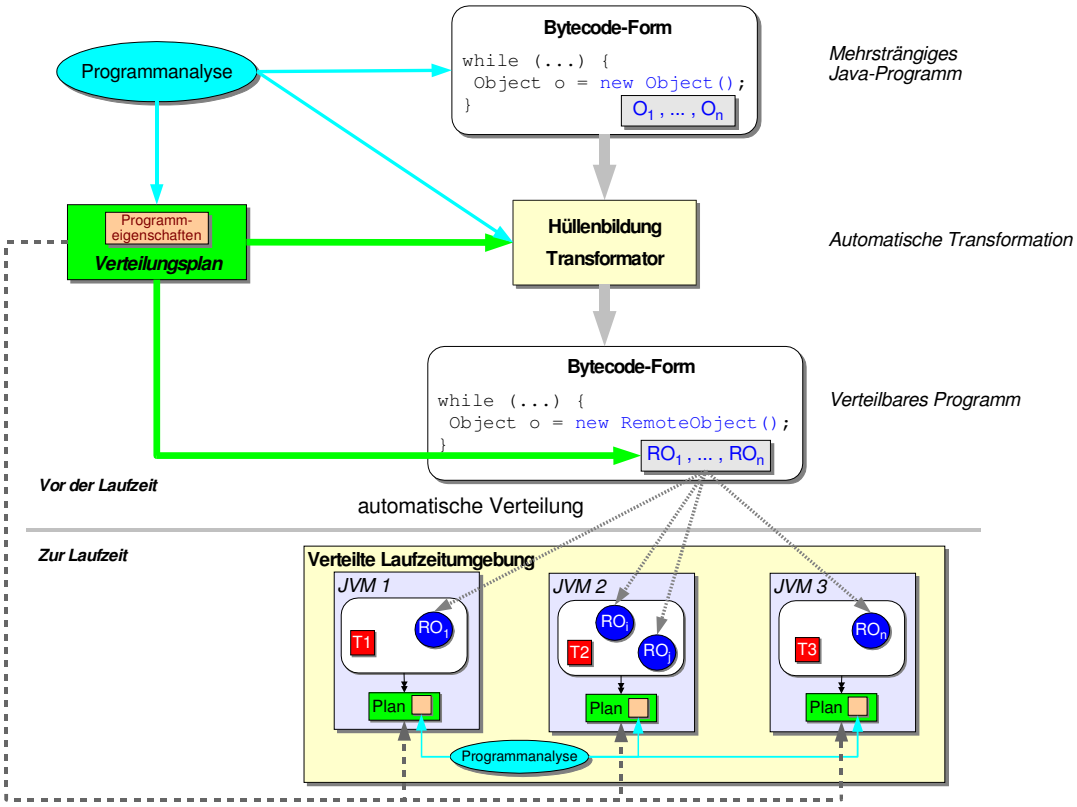

Abbildung 7.1.: Das JScatter-System mit seinen Komponenten

Abbildung 7.1 stellt die gesamte Infrastruktur des Systems mit seinen Komponenten grafisch dar. Als Eingabe liegt ein multithreaded Java-Programm in *Java-Bytecode* vor. Zusammen mit den verwendeten Bibliotheken wird eine transitive Hülle [Wes04] gebildet. Sie enthält alle Methoden und Klassen der Anwendung und diejenigen der verwendeten Bibliotheken, die zur Ausführung des Programms benötigt werden. Darauf wird die statische Programmanalyse angewandt, um die für die Transformation relevanten Eigenschaften des Eingabeprogramms sowie diejenigen, die zur Berechnung der Verteilungsentscheidung benötigt werden, vor der Laufzeit zu bestimmen. Anschließend wird eine verteilte Version des Programms mittels des Transfor-

mators, ebenfalls in Bytecode, generiert, die dann auf der verteilten Laufzeitumgebung des Systems mit den Platzierungshinweisen aus dem Verteilungsplan ausgeführt werden kann. Während der Ausführung werden Laufzeitdaten mittels dynamischer Analyse bestimmt und dann zur Verbesserung der Verteilungsentscheidung eingesetzt.

Dieses Kapitel beschreibt einige ausgewählte Realisierungsaspekte des gesamten Systems. Die Analyseverfahren, die zur Berechnung der statischen Programmeigenschaften im Verteilungsplan zum Einsatz kommen, werden in unserer Analyseumgebung realisiert. Ihre Struktur und Funktionalität werden im ersten Abschnitt vorgestellt. Anschließend wird das Transformationskonzept beschrieben, mit dem das Programm in eine verteilte Version transformiert wird. Zur Platzierung der entstehenden Objekte werden zunächst die für die Verteilung relevanten Programmeigenschaften berechnet. Sie werden unmittelbar in den ODFs der verwendeten Strategien und somit im Verteilungsplan verwendet. Die Realisierung und das Zusammenwirken der Strategien innerhalb eines Verteilungsplanes werden hierbei dargestellt. Das transformierte Programm wird in einer verteilten Laufzeitumgebung, einer Komponente des *JScatter*-Systems, ausgeführt. Die Entwurfsaspekte dazu werden im folgenden Abschnitt beschrieben. Abschließend werden einige Verbesserungsmöglichkeiten für das System diskutiert.

7.1. Die PAULI - Analyseumgebung

Die Analyseverfahren zur Berechnung der Programmeigenschaften wurden als Module unserer Analyseumgebung realisiert. Diese von unserer Fachgruppe „Programmiersprachen und Übersetzer“ entwickelte Analyseumgebung enthält viele grundlegende Analysen. Sie lässt sich einfach um weitere Analysen erweitern. Im Folgenden werde ich die Struktur und die Funktionalität der Analyseumgebung beschreiben. Weshalb sie hier verwendet wird, wird anschließend begründet.

7.1.1. Struktur der Analyseumgebung

Abbildung 7.2 stellt die PAULI-Struktur dar. Das Anwendungsprogramm zusammen mit seinen benutzten Bibliothek-Klassen (transitive Hülle) bildet einen Analysekontext und wird der Analyseumgebung in Bytecode-Form

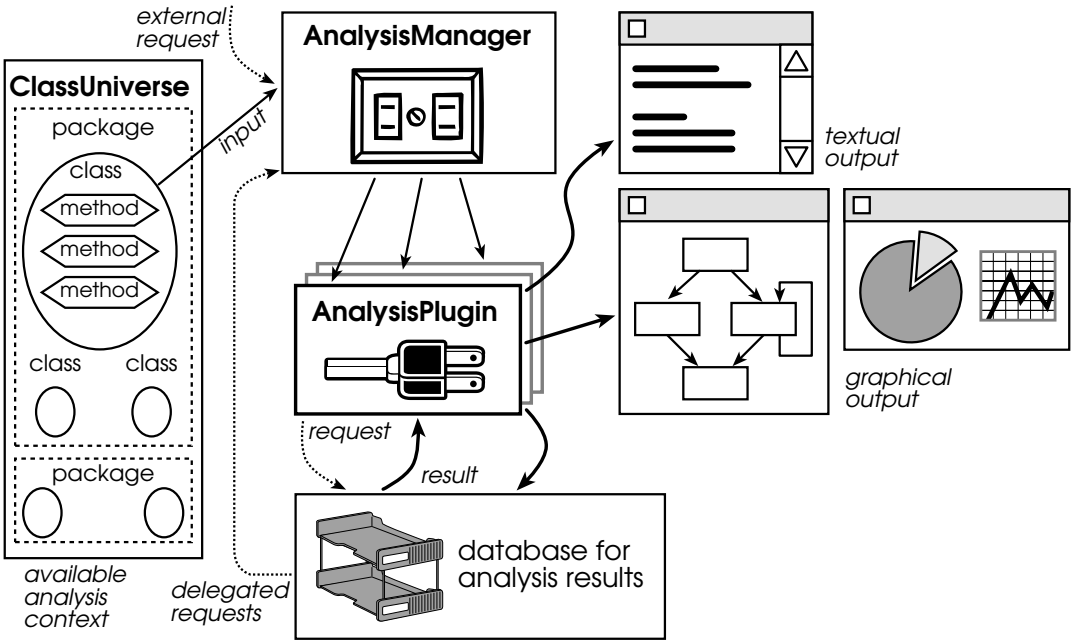

Abbildung 7.2.: Die Infrastruktur der Analyseumgebung [Thi01]

als Eingabe übergeben. Der Bytecode wird zunächst untersucht und mittels BCEL [Daha] - eine an der Freien Universität Berlin entwickelte *Byte Code Engineering Library* - in interne Datenstrukturen überführt, auf denen sämtliche Analysen arbeiten. Abbildung 7.3 stellt das Objektmodell der zu analysierenden Software in Form eines UML-Klassendiagramms dar. Die Assoziationen beschreiben die Zuordnung der erzeugten Repräsentanten für die elementaren Strukturen eines Java-Programms wie Pakete, Klassen, Methoden oder Felder. *JavaClass library* ist hier die Bezeichnung für die BCEL-Bibliothek.

In einem so genannten *ClassUniverse* in Abbildung 7.2 können neben den Grundstrukturen auch mehrere Pakete zusammengefasst werden. Der gesamte Analysekontext wird somit in mehrere Teilmengen eingeteilt und für bestimmte Analysen eingeschränkt, während andere Analysen ggf. den gesamten Kontext betrachten müssen. Die Komponente *AnalysisManager* ist für die Verwaltung und Steuerung der *AnalysisPlugins* zuständig. Ein Analyseverfahren wird durch ein *AnalysisPlugin*, also ein Analyse-Modul, realisiert. Ergebnisse verschiedener Analysen werden in einer Datenbank verwaltet, damit bestimmte Analysen auf Ergebnisse anderer zugreifen und diese wiederverwenden können. Liegt das angefragte Ergebnis noch nicht vor, wird der *AnalysisManager* informiert, so dass dieser das erforderliche Modul aufruft.

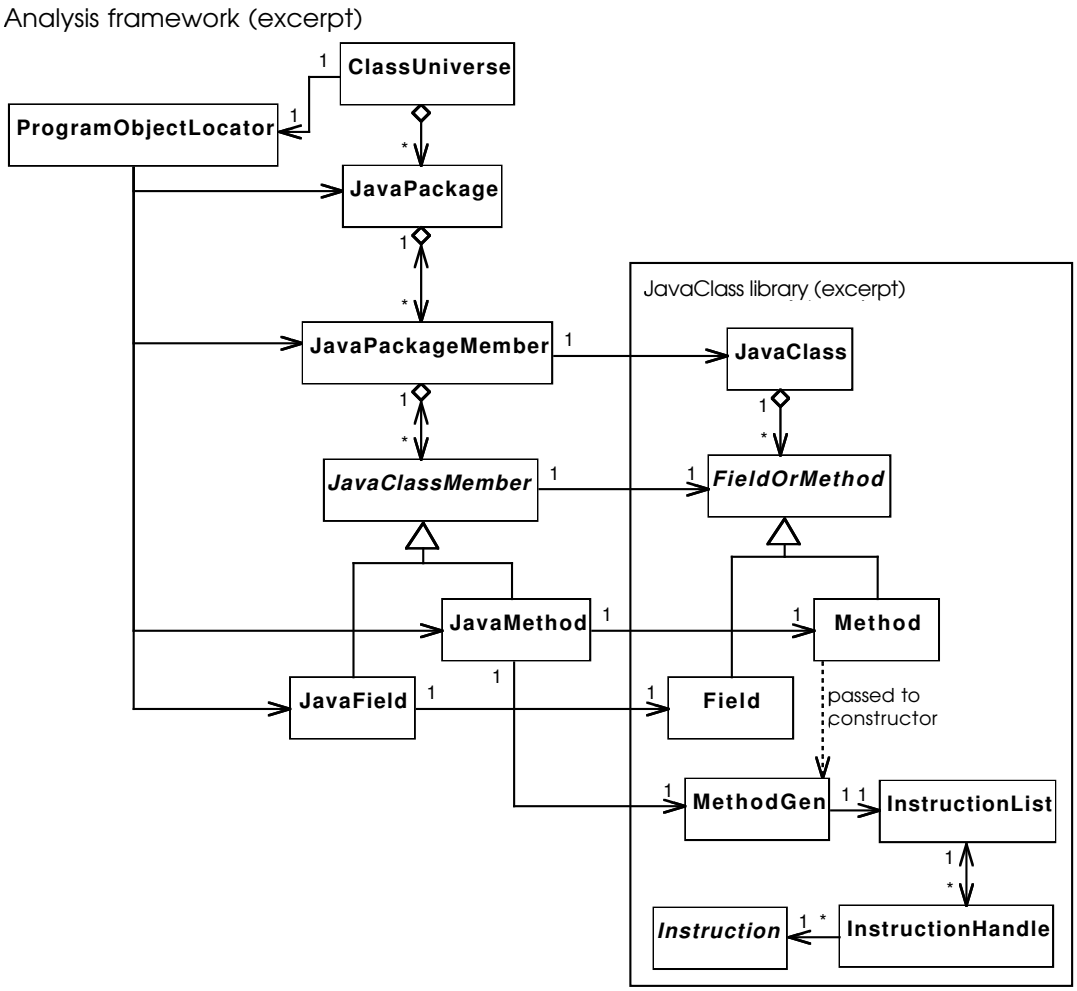

Abbildung 7.3.: Objektmodell für die zu analysierende Software [Thi01]

Mit diesem Mechanismus findet der gesamte Ablauf für den Benutzer völlig transparent statt und es wird dadurch sichergestellt, dass jedes Analyseergebnis nur einmal berechnet wird. Eine detaillierte Beschreibung der Analyseumgebung findet sich in [Thi01] im Kapitel 6.

7.1.2. Vorteile durch den Einsatz der Analyseumgebung

Mit PAULI werden elementare Teile eines Java-Programms in entsprechender Form repräsentiert, wobei sich alle wichtigen Programmeigenschaften finden lassen, auch diejenigen für die Verteilungsberechnung. Zur Berechnung solcher Eigenschaften werden neue statische Analysen auf einfache Weise imple-

mentiert, die noch den Vorteil zur Wiederverwendung vorhandener, grundlegender Analyseergebnisse sowie elementarer Datenstrukturen wie z.B. gerichteter Graphen nutzen.

Durch die internen Strukturen zur Repräsentation von Java-Programmobjekten und passenden Analyseergebnissen für die Verteilungsstrategien, sowie Zugriffsmöglichkeiten auf die interne Verwaltungsstrukturen wird die Transformation des Programms in eine verteilte Version auf Bytecode-Ebene erheblich erleichtert. Die Analyse sowie die Transformation des Programms auf Bytecode-Ebene haben außerdem den Vorteil, dass man unabhängig davon ist, ob der Quellcode eines Programms oder der verwendeten Bibliotheken zur Verfügung steht oder nicht. So können z.B. Erzeugungsstellen und Verwendungsstellen für die dort zu erzeugenden Objekte transformiert werden.

7.2. Programmtransformation

Ziel ist es, das ursprüngliche Programm so zu transformieren, dass die transformierte Anwendung in der verteilten Laufzeitumgebung ausgeführt werden kann. Damit das Programm mit eingebautem RMI-Mechanismus verteilt abläuft, wird jede Klasse des Originalprogramms in mehrere Klassen zerlegt. Hierbei muss beachtet werden, dass bei der verteilten Ausführung die Semantik der neu entstehenden Klassen die Semantik der ursprünglichen Klasse nicht verletzen darf. Wie bei der sequentiellen Ausführung darf es den statischen Anteil, im Folgenden Klassenanteil genannt, auch nur einmal während des gesamten verteilten Ablaufs geben. Im Gegensatz dazu kann der Instanzenanteil jedoch mehrfach auf jedem beteiligten Rechner vorhanden sein, von jedem entfernten Rechner aus zugegriffen werden und muss eine eindeutige Identität besitzen. Aus diesem Grund wurde ein Konzept zur Transformation entwickelt, das aber auch die Infrastruktur der verteilten Laufzeitumgebung beeinflusst.

7.2.1. Transformationskonzept

Angelehnt am verwendeten Konzept in JavaParty-System (siehe Abschnitt 4.1.1), bei dem aus einer Klasse sechs Klassen und sieben Interfaces entstehen, wird eine Java-Klasse im *JScatter*-System in vier Klassen und zwei Interfaces zerlegt. Abbildung 7.4 zeigt die Klassenaufteilung grafisch.

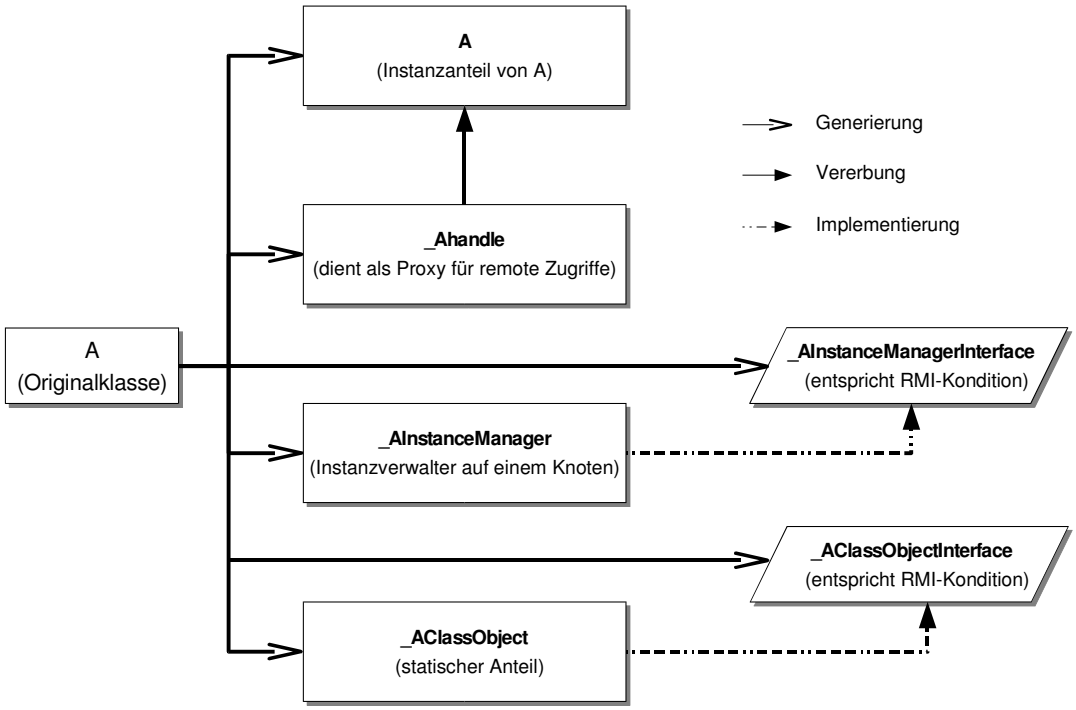

Abbildung 7.4.: Klassenaufteilung einer Klasse

Die neue gleichnamige Klasse repräsentiert den Instanzenanteil (sog. Cuckoo-Klasse) der ursprünglichen Klasse. Statische Felder und Methoden der Originalklasse bilden den statischen Anteil, der im Beispiel als `_AClassObject` zu erkennen ist. In dieser Klasse sind jedoch die ursprünglich als statisch deklarierten Felder und Methoden nicht mehr statisch, sondern als Instanzenfelder und -methoden definiert. Dieser Transformationsschritt ist erforderlich, denn der statische Bestandteil einer Klasse existiert nur einmal in einer JVM. Die verteilte Laufzeitumgebung besteht aber aus mehreren JVMs, die sich auf verschiedenen Rechnern aufhalten. Daher wird zur Laufzeit eine einzige den statischen Anteil einer Klasse darstellende Instanz einmal auf einer bestimmten JVM erzeugt und existiert auch nur einmal für die gesamte Umgebung. Um aber Zugriffe auf diese Instanz aus verschiedenen Adressräumen zu ermöglichen, steht in allen JVMs ein Stellvertreter auf diese eindeutige Instanz zur Verfügung. Für die Kommunikation über RMI zwischen dem Stellvertreter und dem Original wird ein Interface namens `_AClassObjectInterface` generiert, das die entfernte Schnittstelle auf `_AClassObject` definiert.

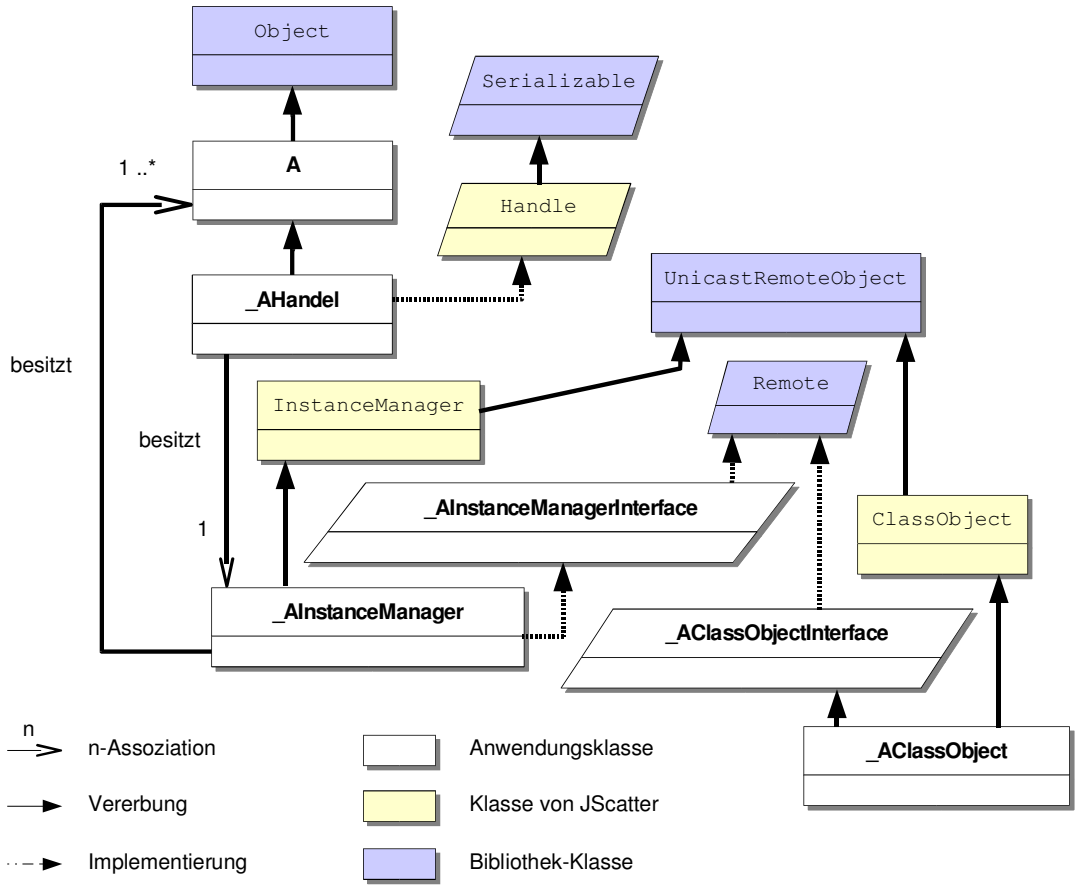

Abbildung 7.5.: Hierarchie der aufgeteilten Klassen

Die generierte Klasse `_AInstanceManager` verwaltet alle Instanzenanteile repräsentierende Objekte der ursprünglichen Klasse innerhalb einer JVM. Damit ist es wie beim sequentiellen Ablauf möglich, mehrere Instanzen einer Klasse in der gesamten verteilten Umgebung zu erzeugen und jede von ihnen lässt sich dadurch eindeutig identifizieren. So existiert zur Laufzeit auf jeder beteiligten JVM genau eine Verwaltungsinstanz pro Originalklasse. Die Zugriffe auf den Instanzenanteil eines *remote*-Objekts finden bei diesem Konzept durch `Handles` statt. Hier wird der Delegationsmechanismus verwendet, um alle Zugriffe auf ein *remote*-Objekt über das zugehörige Handle zu ermöglichen. Methodenaufrufe und Feldzugriffe werden vom Handle an die Verwaltungsinstanz weitergeleitet, die diese dann an das richtige Objekt weitergibt. So wird für eine Klasse `A` die Handle-Klasse `_AHandle` generiert. Ein Handle eines Objekts kennt seine Verwaltungsinstanz und damit auch das auf einem Rechner platzierte Objekt. Die Handles werden ebenfalls als Parameter

und Rückgabeergebnis verwendet. Abbildung 7.5 zeigt die generierten Klassen mit ihrer hierarchischen Beziehung zu denjenigen des JScatter-Systems.

7.2.2. Transformation

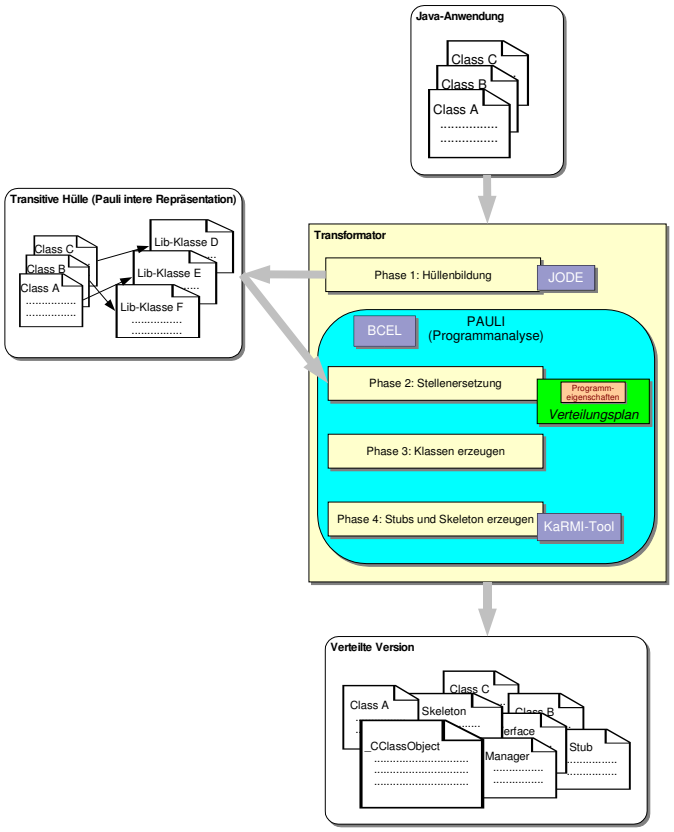

Abbildung 7.6.: Transformationsschritte mit verwendeter Software

Die Transformation besteht im Wesentlichen aus 4 Phasen. Die erste Phase ist die Hüllenbildung. Aus der eingegebenen Anwendung wird zunächst mittels *JODE (Java Optimize and Decompile Environment)* [Hoe] die zugehörige transitive Hülle [Wes04] gebildet. Die Analyseumgebung PAULI nimmt diese als Analysekontext und überführt sie in eine interne Darstellungsstruktur. Darauf werden die eigentlichen Transformationsschritte durchgeführt.

Unter Verwendung der Platzierungsstrategien in einem Verteilungsplan werden Erzeugungsstellen zu Objekten transformiert, die mittels statischer Analyse als *remote* gekennzeichnet werden. So werden Instruktionen zur Abfrage an den Verteilungsplan zur Erzeugungsinstruktion hinzugefügt, damit die

zur Laufzeit getroffene Entscheidung zur entfernten Erzeugung der Objekte genutzt wird. Außerdem werden alle betroffenen Stellen wie z.B. Verwendungsstellen der *remote* Objekte, deren Methoden usw. entsprechend so geändert, dass entfernte Zugriffe darauf zur Laufzeit durchgeführt werden können. Anschließend werden Klassen und Interfaces entsprechend dem obigen Konzept generiert. Mittels Werkzeugen von KaRMI werden die für entfernte Kommunikation benötigten Stubs und Skeletons erzeugt.

Die Phasen 2 bis 4 arbeiten prinzipiell auf den internen Strukturen der Analyseumgebung und somit BCEL. Dadurch wird die Transformation auf Bytecode-Ebene erleichtert. Außerdem müssen während der Transformation bestimmte statische Analysen durchgeführt werden. So müssen Instruktionen zur Definition aktuell übergebener Parameter einer Methodenaufrufstelle sowie ihre in Frage kommenden Typen gefunden werden, damit z.B. entsprechende Handle-Typen erzeugt werden. Bei Verwendungsstellen muss festgestellt werden, ob ein Zugriff hier ein lokaler oder entfernter ist und dementsprechend müssen Instruktionen zu Handle-Aufrufen generiert werden. Durch Verwendung der Analyseumgebung hat man den Vorteil, dass vorhandene Analyseergebnisse wiederverwendet und ggf. auch notwendige Analysen durchgeführt werden. Mittels BCEL lassen sich die Transformationen der Stellen im Programm sowie die Manipulation, Erzeugung neuer Instruktionen und neuer Klassen einfacher handhaben.

7.3. Berechnung der Programmeigenschaften

Verantwortlich für die Verteilungsberechnung sind die ODFs. Durch Anwendung einiger von ihnen in bestimmter Reihenfolge können Ergebnisse der verwendeten Strategien, somit der eingesetzten Verteilungspläne ermittelt werden. Abbildung 7.7 stellt die gesamte Struktur der ODFs, ihre Zusammenhänge bei der Berechnung der Verteilungsentscheidung in den Strategien sowie in den Verteilungsplänen dar. Dort sind auch einige ausgewählte Analyseverfahren angegeben, die zur Berechnung der Programmeigenschaften verwendet und in der oben vorgestellten Analyseumgebung implementiert wurden. Diese Komponenten wurden realisiert, um das Konzept zur automatischen Verteilung unter Einsatz der Programmanalysen zu evaluieren.

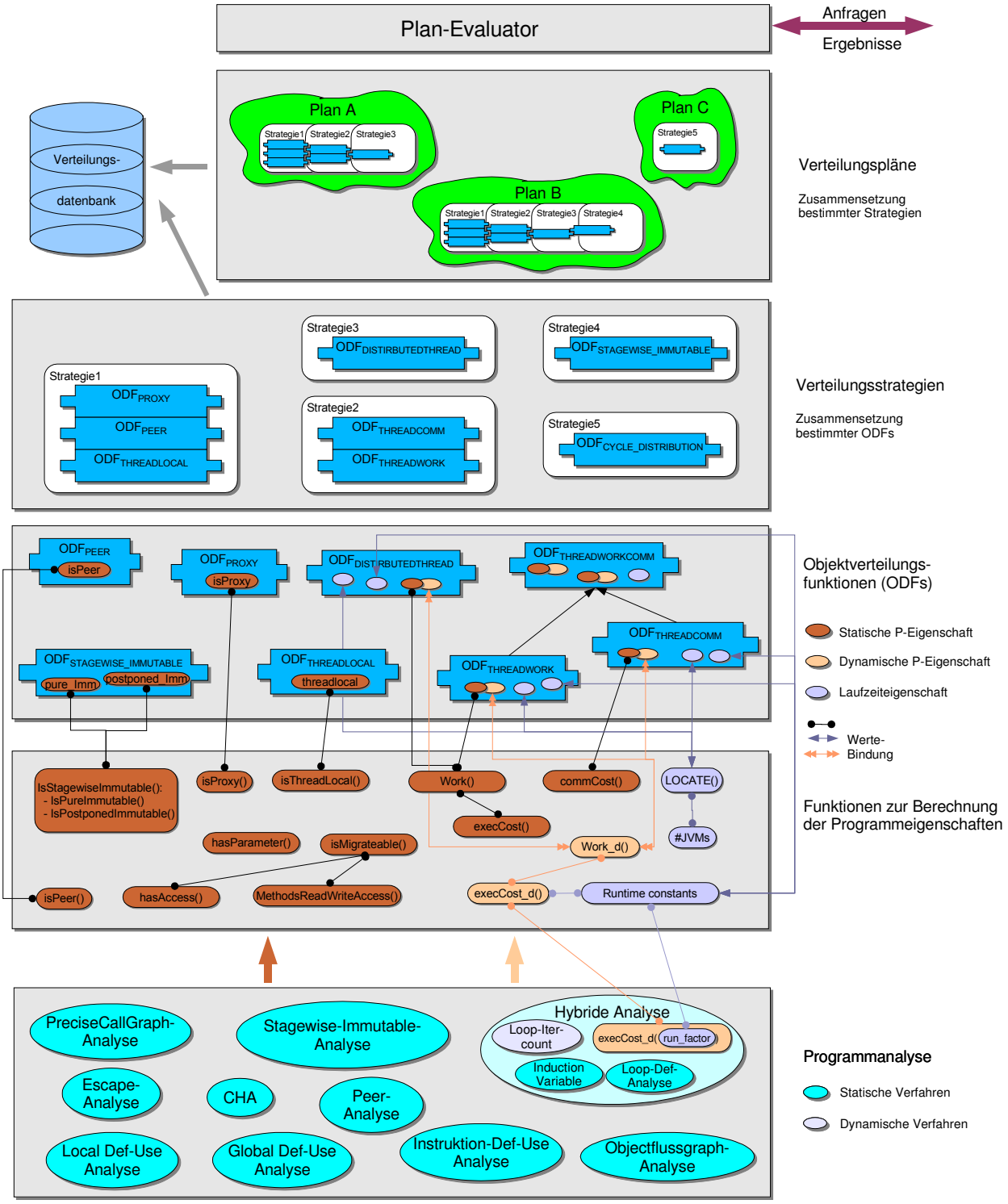

Abbildung 7.7.: Infrastruktur der verwendeten ODFs, Strategien und Verteilungspläne

7.3.1. Verteilungsberechnung der Verteilungspläne

Modellierte ODFs

In Abbildung 7.7 sind die realisierten ODFs angegeben, die innerhalb der zur Evaluation eingesetzten Verteilungspläne verwendet wurden. Das sind einige statische ODFs, z.B. zur Berechnung der *ThreadLocal*- oder *stagewise-immutable* Eigenschaften. Zur grundlegenden Verteilungsentscheidung der Aktivitäten sowie der Datenobjekte benötigte man Basis-Verteilungsfunktionen, die die dynamischen Programmeigenschaften zum Workload und Kommunikationsaufwand vor der Laufzeit berechnen und ggf. abschätzen.

In diesem Konzept wird die Berechnung der Programmeigenschaften mittels Programmanalyse von der Verteilungsberechnung mittels der ODFs deutlich getrennt. Somit hängt also die Güte der Ergebnisse nur davon ab, mit welchen Analysen man solche Programmeigenschaften bestimmt. Sie werden bei der Realisierung mittels Funktionen (siehe zweite Schicht von unten in Abbildung 7.7) berechnet. Ihre Ergebnisse werden direkt in den ODFs verwendet, um dort elementare Verteilungsentscheidung ermitteln zu können.

Neben den statischen Programmeigenschaften gibt es einige Eigenschaften, die vom Laufzeitverhalten des Programms abhängig sind oder Architekturinformation beschreiben. Diese werden als variable Laufzeitfaktoren in den Berechnungen modelliert. Sobald ihre konkreten Werte, z.B. die Anzahl der JVMs oder die Anzahl der schon durchgelaufenen Iterationen einer Schleife, während der Programmausführung bekannt sind, werden sie an ihren Verwendungsstellen gebunden.

Verteilungsstrategien

Die Zusammenfassung bestimmter ODFs definiert dann eine Verteilungsstrategie. Die bei der empirischen Untersuchung im nächsten Kapitel verwendeten Strategien zur Verteilung der Objekte sind in Abbildung 7.7 angegeben. Das sind die Strategien zur Lokalitätsoptimierung (*Strategie1*), zur Verteilung der Aktivitäten (*Strategie3*), zur Gruppierung der Datenobjekte mit ihren Aktivitäten anhand Workload und Kommunikation (*Strategie2*) und zur Replikation (*Strategie4*) nach der besonderen Eigenschaft *stagewise-immutable*. (*Strategie5*) definiert eine einfache zyklische Verteilung aller *remote*-Objekte.

Ein Verteilungsplan wird durch Verwendung bestimmter Strategien definiert. Die vor der Laufzeit getroffenen Platzierungsentscheidungen für Objektre-

präsentanten eines Planes werden in eine Datenbank zur Verwaltung eingetragen. Ebenfalls wird benötigte Information für jede Erzeugung zur Laufzeit hierin aufgenommen, z.B. welches Objekt (also mit welchem ID) aus welcher Erzeugunsstelle auf welchen Rechner platziert wird.

Plan-Evaluator

Bei der Transformation werden Erzeugungsstellen für *remote*-Objekte um Anfragen an den Plan so erweitert, dass unmittelbar vor der tatsächlichen entfernten Erzeugung der Verteilungshinweis abgefragt werden kann. So hat der Plan-Evaluator die Aufgabe, vor und zur Laufzeit Anfragen nach Transformations- und Verteilungshinweisen vom Programm zu beantworten. Vor der Laufzeit erhält er Fragen zur Transformationsentscheidung, ob eine Erzeugungsstelle zu *remote* transformiert wird oder nicht. Nach der Planevaluation liefert er neben Transformationsergebnissen auch Codesegmente zur Planabfrage, die an einer Erzeugungsstelle eingebaut werden müssen, wenn das zu erzeugende Objekt ein *remote*-Objekt wird. Zur Laufzeit nimmt der Plan-Evaluator die gestellte Anfrage entgegen, leitet sie an den Verteilungsplan weiter und gibt die Entscheidung des Planes an den Fragenden zurück. Wie eine entfernte Erzeugung unter Verwendung des Verteilungshinweises abläuft, wird im folgenden Abschnitt 7.4 beschrieben.

7.3.2. Programmanalyse

Alle Analyseverfahren, die zur Berechnung der Eigenschaften zum Einsatz kommen, wurden in der Analyseumgebung PAULI realisiert. In Abbildung 7.7 sind einige von ihnen dargestellt. Jede Analyse wurde als ein Analyse-Modul (*Analysis-Plugin*) in PAULI implementiert. Durch die Infrastruktur der Analyseumgebung können Ergebnisse vorhandener Analysen wie z.B. CHA, Local-Def-Use-Analyse usw. innerhalb der Zwischenschritte der neuen Verfahren wiederverwendet werden. Das Konzept zur hybriden Analyse wurde in dieser Arbeit zur Workload-Berechnung für Schleifenstrukturen erprobt, wie es in Abbildung 7.7 gezeigt wird: Hier findet sich ein unbekannter Laufzeitfaktor in den Berechnungsschritten zum dynamischen Workload (run_factor). Dieser Faktor modelliert die Anzahl der auszuführenden Iterationen einer Schleife. Sein Wert ergibt sich zur Laufzeit und wird in die Berechnung direkt eingesetzt. Somit wird die Funktion `execCost_d()` zum

Workload erneut berechnet. Verteilungsstrategien, die auf diesen Kosten basieren, verbessern dann ihre Platzierungsentscheidungen für zukünftig entfernt zu erzeugende Objekte. Durch das hier benutzte Bindungskonzept kann diese Berechnungskette automatisch ausgelöst werden.

7.4. Verteilte Laufzeitumgebung

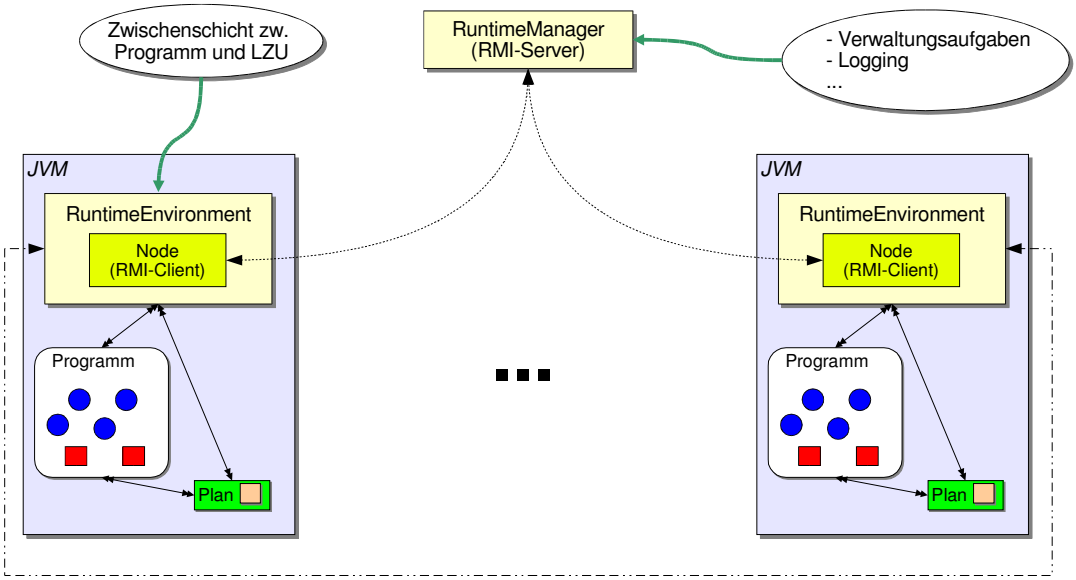

Abbildung 7.8.: Abstraktion der verteilten Laufzeitumgebung

Die für den Benutzer transparente Ausführung der verteilten Version des Programms übernimmt die verteilte Laufzeitumgebung des *JScatter*-Systems. Sie verbindet die auf den beteiligten Rechnern vorhandenen JVMs miteinander. Abbildung 7.8 zeigt die Abstraktion der verteilten Laufzeitumgebung (LZU).

7.4.1. Zusammenarbeit der Komponenten

Zur entfernten Kommunikation wird hierbei KaRMI - eine schnellere RMI-Version der Universität Karlsruhe - verwendet. Ein `Node` dient als RMI-Client und repräsentiert eine JVM. Zur Verwaltung der mit der LZU verbundenen Nodes ist eine Instanz der Klasse `RuntimeManager` zuständig. So werden Fehlermeldungen der JVMs sowie Ausgaben des verteilten Programms an

diese Instanz weitergeleitet und dort ausgegeben. Sie fungiert auch als der RMI-Server. Als Zwischenschicht oder Schnittstelle zwischen der LZU und dem ausgeführten Programm wird die Klasse `RuntimeEnvironment` verwendet. Zur Laufzeit existiert pro JVM genau eine Instanz dieser Klasse, in der die Instanzen- und Klassenanteile repräsentierenden Objekte verwaltet werden. Über diese Schnittstelle kann das verteilte Programm auf diese Anteile zur Laufzeit zurückgreifen und entfernte bzw. lokale Zugriffe durchführen.

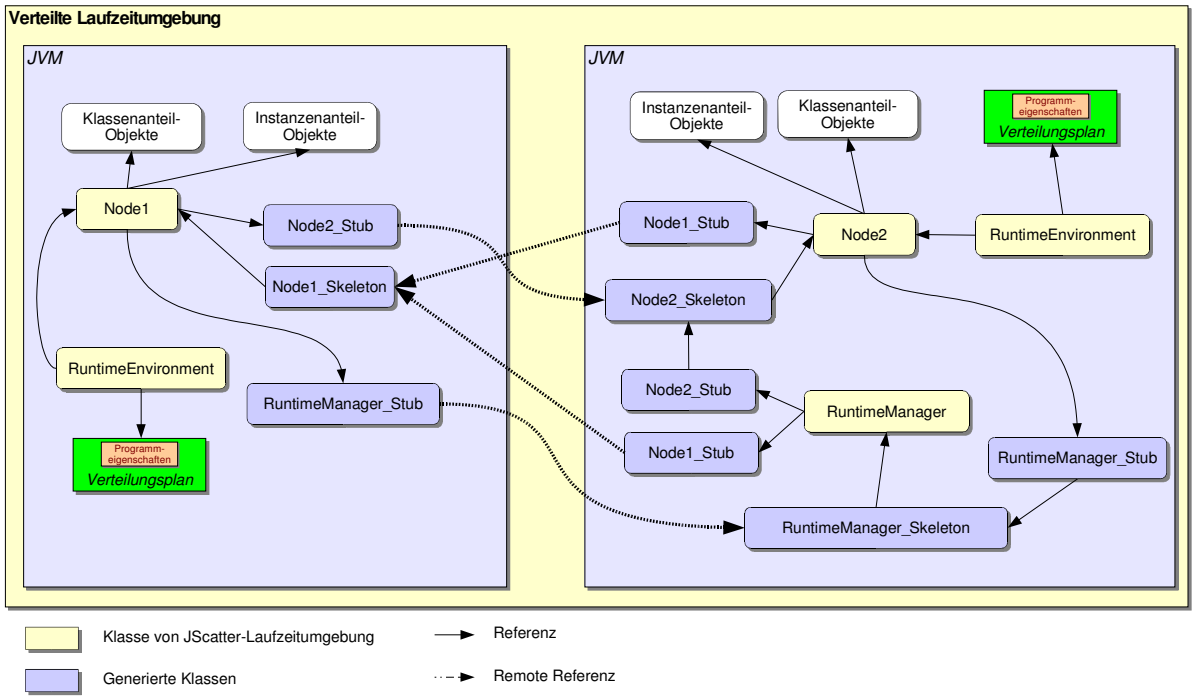

Abbildung 7.9.: Komponenten der verteilten Laufzeitumgebung

Zur Verteilung der erzeugten Objekte existiert auf jeder JVM eine Instanz des verwendeten Verteilungsplans. Somit kann für jede (entfernte) Objekterzeugung im Programm die Planabfrage durchgeführt werden. Abbildung 7.9 stellt die Zusammenarbeit der vorgestellten Komponenten der LZU grafisch dar.

7.4.2. Verwendung des Verteilungsplans

Nach der Transformation des Programms befindet sich an jeder Erzeugungsstelle für *remote*-Objekte eine Planabfrage an die lokale Instanz des Vertei-

lungsplans. Mittels dieses Aufrufs kann vor der entfernten Erzeugung eines Objekts die Zielmaschine bestimmt werden. Danach führt die LZU die Platzierung durch. Abbildung 7.10 stellt dieses Szenario grafisch dar.

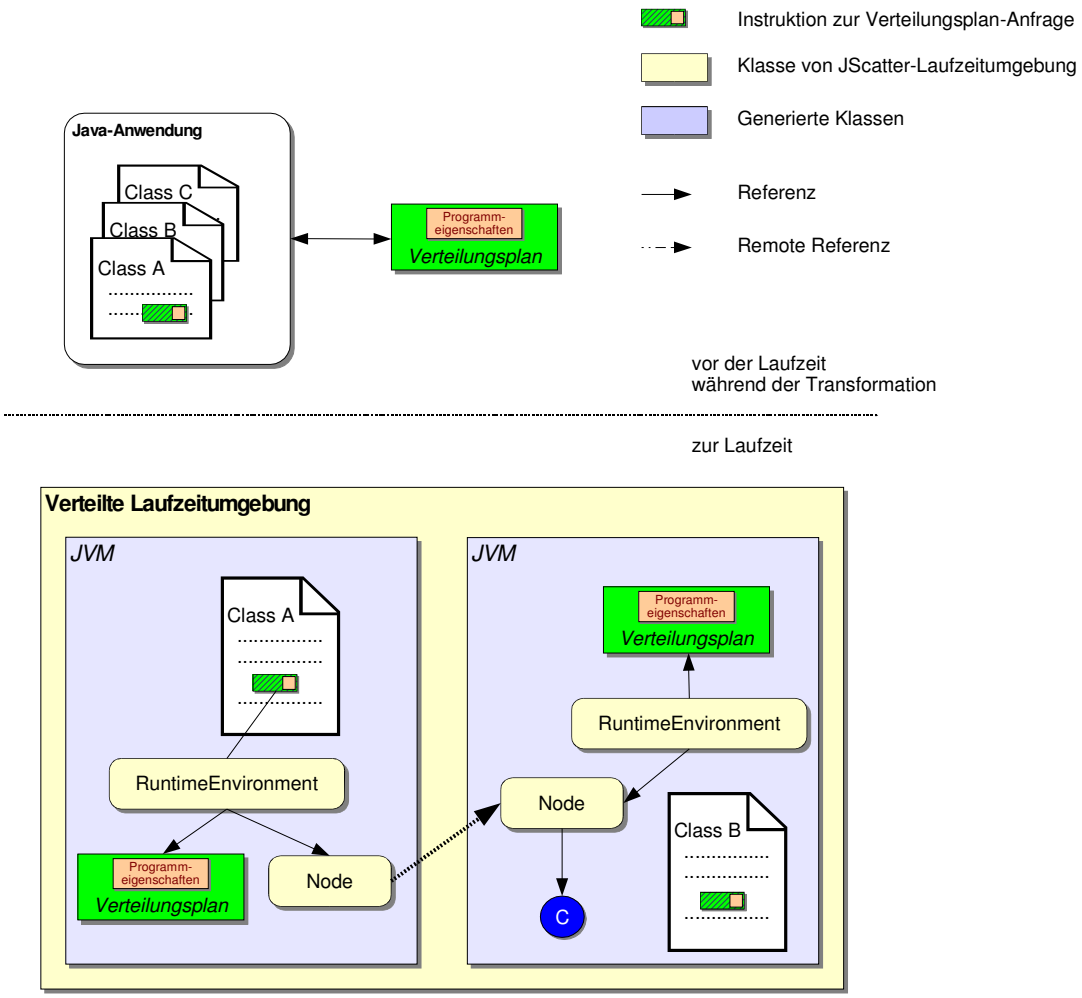

Abbildung 7.10.: remote NEW

Hierbei soll bei der Abarbeitung einer Methode von Objekt `A` ein Objekt der Klasse `C` auf einem anderen Rechner entfernt erzeugt werden. An der zugehörigen Erzeugungsstelle für das Objekt `C` wird die Abfrage an den lokalen Verteilungsplan über die `RuntimeEnvironment`-Instanz gestellt. Dort wird entschieden, dass das Objekt `C` auf dem zweiten Rechner erzeugt werden soll. Die Laufzeitumgebung führt diese Aufgabe über die lokale `RuntimeEnvironment` und über die `Nodes` durch, die die beiden JVMs repräsentieren. Nachdem das Objekt erzeugt wurde, wird die Referenz darauf

über einen zugehörigen Handle zurückgegeben. Über diesen Handle kann das Objekt während der Programmausführung angesprochen werden.

Technische Details sowie Entwurfsentscheidungen des Systems wurden ausführlich in [FFF+02] beschrieben. Die Realisierung der Komponenten zur Migration und Replikation wurde ausführlich in der Diplomarbeit von Weißenborn [Wei03] beschrieben. Ein zu erwähnender Aspekt der Laufzeitumgebung ist der Folgende: Wenn während der Programmausführung ein Knoten ausfällt oder das Programm abbricht, z.B. aufgrund einer `Exception`, muss die gesamte Umgebung mit dem Programm erneut gestartet werden. Dies macht die Realisierung der verteilten Laufzeitumgebung einfacher. Sonst müsste man im Fall des Ausfall eines Knotens u.a. die Laufzeitobjekte mit ihren aktuellen Zuständen retten und diese auf einen anderen Knoten verteilen. Dies ist vergleichbar mit dem Mechanismus zum Retten der Prozesse bei verteilten Betriebssystemen.

7.5. Systemvoraussetzungen und Verbesserungen

7.5.1. Systemanforderungen

Das *JScatter*-System wurde entworfen, um das Konzept zur automatischen Verteilung mehrsträngiger Java-Anwendungen zu erproben. Zusammen mit PAULI ist es komplett in Java implementiert und kann deshalb plattformunabhängig verwendet werden.

Die Grundvoraussetzung für den Einsatz des Systems ist eine vorinstallierte JDK ab Version 1.3.1. Zur Analyse, Hüllenbildung sowie Transformation des multithreaded Java-Programms in eine verteilte Version mit (Ka)RMI werden zusätzlich Java-Bibliotheken zu PAULI, JODE, KaRMI und JScatter selbst benötigt. Um das transformierte Programm verteilt auszuführen, müssen neben der Standard-Java-Bibliothek das JScatter-System, KaRMI und das eigentliche transformierte Programm (in Form von JAR-Dateien) vorliegen. Zur Ausführung müssen die Klassen aus den verwendeten Bibliotheken sowie aus der Anwendung selbst auf jedem Rechner-Knoten geladen werden können. Das heißt, dass die in JAR-Dateien vorliegenden Klassen jedem Rechner-Knoten auf irgendeine Weise zur Verfügung stehen müssen. Diese JAR-Dateien müssen beispielsweise über NFS, AFS usw. zugreifbar sein oder sich lokal auf jedem Knoten befinden.

Zur Kommunikation während der verteilten Ausführung wird (Ka)RMI, also *Remote Method Invocation*, eingesetzt. Die auszutauschenden Stromdaten werden über das Netzwerk mit Hilfe eines von KaRMI unterstützten Netzwerk-Protokolls verschickt. Die Verbindung zwischen den verwendeten Rechnern zur verteilten Berechnung muss also über dieses Protokoll aufgebaut werden. Derzeit unterstützt KaRMI zwei *Netzwerk-Technologien*: Die Standard-Übertragung über TCP/IP (Socket-Kommunikation) und Myrinet/GM. KaRMI bietet aber auch die Möglichkeit, weitere Netzwerk-Technologien für spezielle Protokolle oder Übertragungstechniken zu entwickeln.

Man kann also das *JScatter*-System zur automatischen Verteilung mehrsträngiger Java-Anwendungen überall benutzen, wo die oben genannten, einfachen Voraussetzungen erfüllt werden. So wird im Kapitel 8 gezeigt, dass das *JScatter*-System zur empirischen Untersuchung nicht nur auf einem normalen PC-Cluster, sondern auch auf einem speziellen parallelen Rechner-System (ARMINIUS - Compute and Visualization Cluster) unseres Paderborn Center for Parallel Computing (PC^2)[Pad] eingesetzt wurde.

7.5.2. Verbesserungen

Viele Entwurfsentscheidungen des *JScatter*-Systems basieren darauf, dass die Komponenten einfach realisiert werden sollten. Die Effizienz spielte dabei nur eine Nebenrolle. Deshalb gibt es noch viel Potenzial zur Verbesserung der Effizienz.

Zur Steigerung der Effizienz des Systems können quantitative Untersuchungen durchgeführt werden. Dabei kann die verteilte Laufzeitumgebung so instrumentiert werden, dass wichtige Performanz-Daten bei der verteilten Ausführung der Anwendungen erhoben werden. Dazu zählen z.B., wie oft entfernte Methodenaufrufe stattfinden, wie viele Bytes an Daten dabei verschickt werden, wie viele Handles für entfernte Objekte erzeugt werden etc. Somit können Schwächen und Bottlenecks in der aktuellen Realisierung aufgedeckt und verbessert werden.

Der Transformator ist zurzeit noch nicht in der Lage, alle Java-Anwendungen zu transformieren. So können Arrays im Programm noch nicht behandelt werden. Es existiert ein Entwurf zum Umgang mit Arrays. Dieser wurde aber zur Evaluationszeit nicht erprobt. Da KaRMI zur entfernten Kommunikation verwendet wird, wirken sich die dort enthaltenen Fehler auch auf *JScatter* aus.

8. Evaluation

Inhalt

Dieses Kapitel umfasst die Evaluation meines Konzepts zur automatischen Verteilung mehrsträngiger Programme. So wird die Güte der Verteilung unter Einsatz von Verteilungsplänen mit geschickten Strategien anhand einiger Anwendungsprogramme evaluiert. Zunächst werden die verwendeten Anwendungen bzw. Benchmarks vorgestellt. Diskrete ereignisbasierte Simulationen sind besonders geeignet für verteilte Berechnungen. Hier werde ich einen kleinen Überblick über dieses Anwendungsgebiet geben und seine Eignung zur Verteilung zeigen. Außerdem wird die bei der Untersuchung eingesetzte Hard- und Software vorgestellt.

Im Abschnitt 6.4.1 wurde ein neues Analyseverfahren vorgestellt, das entworfen wurde, um die *stagewise-immutable*-Eigenschaft der zu erzeugenden Objekte zu berechnen. Solche Objekte können zur Laufzeit repliziert werden, damit die Lesezugriffe nach der Replikation nur lokal stattfinden. Im folgenden Abschnitt werde ich die Messergebnisse zu dieser Eigenschaft der verwendeten Anwendungsprogramme vorstellen. Der folgende Abschnitt beschäftigt sich dann mit der empirischen Untersuchung des Konzepts. Hierbei werden verschiedene Aspekte zur Performanz, Kommunikation sowie zum Speicherverbrauch evaluiert. Besonders wird die Verteilungsgüte zur Geschwindigkeitssteigerung unter Einsatz der Verteilungspläne mit verschiedenen Strategien untersucht, u.a. eine Strategie zur *stagewise-immutable* Eigenschaft sowie die Auswirkung auf die Verteilung bei Anwendung hybrider Analyse.

Die Laufzeitmessung wurde sowohl auf einem normalen Pool-Cluster, der aus Standard-PCs (Pentium 4 mit 1.7GHz) mit Ethernet-100MBit-Verbindung besteht, als auch auf einem speziellen Cluster aus unserem parallelen Rechenzentrum PC^2 durchgeführt. Dieser Arminius-Cluster umfasst 200 Rechen-Knoten (Dual-Systeme mit INTEL Xeon 3.2GHz), die über Ethernet-GBit-Verbindung und auch über ein schnelleres Netzwerk - Infiniband HCA - kommunizieren.

Aus dieser empirischen Untersuchung können viele Erfahrungen zur automatischen Verteilung und Erkenntnisse über die Vor- und Nachteile der Verteilungsstrategien für bestimmte Fälle gewonnen werden. Somit können Beurteilungen über die eingesetzten Strategien und allgemein über das Konzept gemacht werden. Das Ganze zusammen mit einigen Erweiterungsansätzen wird abschließend präsentiert.

8.1. Vorstellung der Anwendungen und Benchmarks - Vorbereitung zur Evaluation

Zur Evaluation des gesamten Konzepts habe ich einige Anwendungsprogramme und Benchmarks als Untersuchungsgegenstand ausgesucht. Diskrete ereignisbasierte Simulation scheint für meinen automatischen Verteilungsansatz am besten geeignet zu sein. Sie besitzen passende Berechnungsstrukturen, wobei die Abarbeitung der Ereignisse aus einem Ausschnitt der Simulation viel Rechenaufwand erfordert, aber wenig Kommunikation zwischen den beteiligten Simulationsobjekten benötigt. In diesem Abschnitt werde ich einen kleinen Überblick über das Anwendungsgebiet der diskreten ereignisbasierten Simulation geben. Mit dieser kleinen Einführung können Programm- und Verteilungseigenschaften eines Untersuchungsgegenstands, also einer mittelgroßen diskreten ereignisbasierten Simulation, einfacher verstanden werden. Anschließend werden einige Charakteristiken der untersuchten Programme beschrieben, die direkte Auswirkung auf die Analyse- und infolge dessen auf die Verteilungsergebnisse haben. Außerdem werden Testbedingungen vorgestellt, z.B. mit welcher Software und Hardware sowie unter welchen Netzwerkinfrastrukturen die Anwendungen evaluiert werden.

8.1.1. Ein Anwendungsgebiet: diskrete ereignisbasierte Simulation

Java findet immer mehr Anwendung in der Welt des *High-Performance Computing and Communications*. Die *High-Performance* Anwendungen aus HP-Fortran oder C bzw. C++ unter Verwendung der *message passing*-Technik werden nach und nach in Java übertragen. Typische Anwendungen hierfür stammen z.B. aus dem Bereich der geophysikalischen Verfahren und der Finanzmathematik, numerischen Berechnungen, Raytracer und Renderer. Man kann sie als reguläre Anwendungen klassifizieren. Eine andere Art von Anwendungen, die zum Teil nicht ganz regulär sind und meist als Benchmarks verwendet werden, sind klassische Probleme wie TSP (*Traveling Salesperson Problem*), SOR (*Successive Overrelaxation*), ASP (*All-pairs Shortest Paths*), IDA^* (*Iterative Deepening* A^*) oder einige Suchstrategien eingesetzt im *Data Mining*-Bereich. Ein anderes Gebiet, wo Java zur Implementierung von parallelen Berechnungen eingesetzt wird, ist die wissenschaftliche und technische Simulation und Modellierung. Die Probleme auf diesem Gebiet lassen sich in vier Ty-

pen klassifizieren: Datenparallelität, funktionale Parallelität, Objektparallelität und Metaprobleme.

Klassifizierung

Realistische irreguläre OO-Anwendungen finden sich u.a. im Bereich der Modellierung und Simulation, vor allem bei parallelen und verteilten diskreten ereignisbasierten Simulationen. Abbildung 8.1 zeigt eine Einordnung dieser Art der Simulation. Sie gehört zur Kategorie „funktionale Parallelität“.

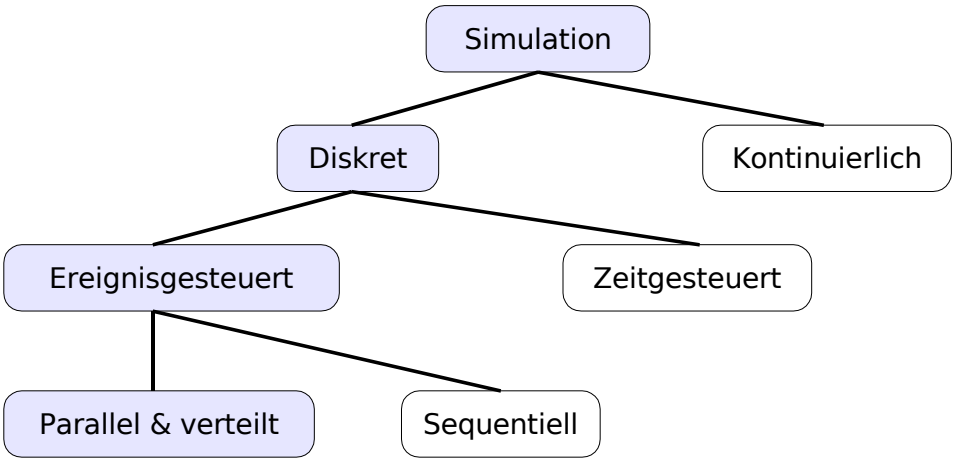

Abbildung 8.1.: Klassifikation verschiedener Arten von Simulationen

Bei der diskreten Simulation beschränkt man sich auf abhängige Größen, die nur diskrete Werte annehmen. Diese Werte verändern sich sprunghaft in einem bestimmten Intervall im Zeitablauf. Es gibt hierbei keine Zwischengrößen. Beispiele für solche diskreten Größen sind die Anzahl von Kassen in einem Geschäft oder der Zustand einer Ampel an einer Kreuzung. Dies ist der Unterschied zur kontinuierlichen Simulation.

Diskrete ereignisbasierte Simulation

Unter Simulation eines zeitdiskreten dynamischen Systems kann man das Berechnen der Ausgabedaten aus den vorgegebenen Eingabedaten durch das fortlaufende Verfolgen der Übergänge verstehen. Eine diskrete ereignisbasierte Simulation ist die Simulation eines zeitdiskreten dynamischen Systems durch Abarbeiten von Ereignissen in der Reihenfolge ihres Eintreffens. Die

Abarbeitung eines Ereignisses bedeutet, dass der Zustand des Systems durch Systemübergänge geändert wird und die Nachfolgeereignisse und ihre Zeiten zum Eintreffen erzeugt werden.

Ein solches System enthält sowohl statische als auch dynamische Strukturen. Das bedeutet, dass eine Anordnung von Objekten - also Simulationsobjekte, nicht Java-Objekte - existiert. Diese Objekte beeinflussen sich gegenseitig und der Zustand des Systems wird durch Objekte, Attribute und Aktionen zu einer gegebenen Zeit bestimmt. Damit sich man diese Abstraktion einfacher vorstellen kann, wird im Folgenden ein kleines Beispiel derartiger Simulationen gegeben.

Ein Beispiel

Um die Wartezeit an einer Supermarktkasse zu untersuchen, betrachtet man den Ausschnitt „Kassenbereich“ des Supermarktes. Hierbei handelt es sich um ein zeitdiskretes dynamisches System. Es ändert sich nur zu den Zeitpunkten, zu denen ein Kunde den Kassenbereich zum Bezahlen betritt und zu denen ein Kunde den Kassenbereich verläßt, nachdem er bezahlt hat.

Die Komponenten des System sind „Kasse“ und „Kunde“, die Komponentendaten sind: geöffnet oder geschlossen, Kassiererin bzw. Kassierer schnell oder langsam, Länge der Schlange für die Kasse; Eintrittszeitpunkt, Waren im Warenkorb, Kassenauswahlstrategie, Austrittszeitpunkt der Kunden. Die Eingaben für die Supermarktkasse sind Kunden, die bezahlen wollen. Ausgaben sind die Kunden, die bezahlt haben. Aus diesen Ein- und Ausgabedaten kann man interessante Zusammenhänge untersuchen und ermitteln, wie beispielsweise die durchschnittliche oder maximale Wartezeit der Kunden oder die durchschnittliche oder maximale Länge der Warteschlangen.

Für dieses Beispiel kann man das Ereignis „ein Kunde stellt sich an Kasse i an“ betrachten, das wie folgt definiert ist:

- Im neuen Systemzustand ist die Warteschlange der Kasse i um Eins verlängert.
- Es gibt kein Folgeereignis.

Ein anderes Ereignis „ein Kunde wird an Kasse i bedient“ kann wie folgt definiert werden:

- Im neuen Systemzustand ist die Warteschlange der Kasse i um Eins verkürzt.
- Das Folgeereignis ist definiert als „ein Kunde verläßt den Kassenbereich“ zu der Zeit „aktuelle Zeit plus Anzahl der Artikel im Warenkorb des Kunden mal fünf Sekunden“.

Zum Beginn der Simulation wird das Ereignis „ein Kunde kommt in den Kassenbereich“ als Eingabe abgearbeitet. Dieses Ereignis wird dann wie folgt definiert:

- Der Systemzustand bleibt gleich.
- Als Folgeereignis wird durch die Kassenauswahlstrategie des Kunden die Kasse i ausgewählt. Dann wird das Folgeereignis „ein Kunde stellt sich an Kasse i an“ zur Zeit „eine Sekunde später“ erzeugt.

Solche Art der Simulation kann eingesetzt werden, um quantitative Aussagen über das Verhaltens eines Ausschnitts der realen Welt zu machen. Es gibt aber unzählig viele solcher Ausschnitte. Viele bekannte und sinnvolle Anwendungen dafür finden sich in der Betriebswirtschaftslehre, insbesondere in der Produktionswirtschaft, Verkehrsteuerung, im Computerbereich wie Prozessnetzwerke, Spiele usw. Dadurch bietet sich eine Vielzahl von irregulären OO-Anwendungen, in denen viel Rechenleistung in möglichst kurzer Zeit benötigt wird. Sie eignen sich daher sehr gut für das verteilte Rechnen.

8.1.2. Untersuchte Simulations- und Benchmark-Programme

Für die empirische Untersuchung zur automatischen Verteilung wurden drei Anwendungsprogramme aus drei verschiedenen Kategorien ausgesucht: Eine mittelgroße diskrete ereignisbasierte Simulation eines Gemüseladens (*grocery*), das multithreaded Raytracer-Programm aus dem SPECjvm98 *mtrt* [Cor] und das klassische TSP-Programm von IBM. Zur Evaluation der *stagewise-immutable* Eigenschaft werden außerdem zusätzlich eine von uns selbst entwickelte diskrete ereignisbasierte Simulation (*wash&wax*) mit ihrem Simulationsframework und einige bekannte Benchmark-Programme betrachtet. Das sind die n-Körper-Simulation von IBM (*tnbody*) und die drei Programme aus *section3* des Multithreaded-JavaGrande-Benchmarks in der Version 1.0 (*mt_moldyn, mt_montecarlo* und *mt_raytracer*) [Cen].

	Klassen	Methoden	Bytecodes
grocery	9	55	1536
mtrt	34	326	9619
tsp	5	30	1070
tnbody	7	30	1412
wash&wax	19	110	1998
mt_moldyn	12	29	2688
mt_montecarlo	20	182	3032
mt_raytracer	21	103	2968

Tabelle 8.1.: Kenndaten der Anwendungsprogramme

Die Grocery-Anwendung simuliert das Einkaufverhalten der Kunden eines Gemüseladens, den Laden selbst während der Einkaufzeit und den Kalorienverbrauch der Kunden. Die Komponenten des Systems sind der Laden, die Produkte, das Produktlager, Regale mit einer bestimmten Anzahl der Produkte bzw. nachzufüllenden Produkten, Einkaufswagen, Kassen sowie die Kunden mit ihren Einkaufslisten. Ereignisse während der Simulation sind u.a. „ein Kunde kommt einen Einkaufswagen holen“, „ein Kunde kauft die Produkte nach seiner Einkaufsliste“, „ein Kunde kommt in den Kassenbereich“, „Nachfüllen bestimmter Produkte für ein Regal“, „ein Kunde verbraucht Kalorien aus seinen eingekauften Produkten“ usw. Das Ereignis „ein Kunde kommt einen Einkaufswagen holen“ wird z.B. wie folgt definiert: Im neuen Systemzustand ist die Menge der verfügbaren Einkaufswagen um Eins kleiner oder bleibt Null, falls kein Wagen mehr dort steht. Als Folgeereignisse werden „ein Kunde verläßt den Wagenbereich“ und „ein Kunde wartet auf einen freien Wagen“ definiert. Die Simulationszeit wird entsprechend mit einigen Details angepasst. Das System wurde so modelliert, dass viel Rechenaufwand bei einigen dieser Ereignisse im Vergleich zum Kommunikationsaufwand gebraucht wird. Hierbei werden der Laden und die Kunden als Threads modelliert. Alle anderen Komponenten sind normale Objekte.

Die *Wash&Wax*-Anwendung simuliert eine Autowaschanlage, wobei verschiedene Autotypen wie Geländewagen, Stadtautos usw. je nach ihren Nutzungseigenschaften in der Anlage gewaschen bzw. gewachst werden. Der Berechnungs- und Kommunikationsaufwand dieser Simulation sowie die Simulation insgesamt stellt ein ähnliches Muster dar, wie die Grocery-Simulation. In Tabelle 8.1 sind einige Kenndaten der vorgestellten Anwendungen und der Benchmark-Programme angegeben.

Beim *mtrt* ist ein Unterpaket dabei, das u.a. die Gültigkeitsprüfung der gerenderten Daten ermöglicht. Durch die Hinzunahme dieses Unterpakets werden die zugehörigen Kenndaten etwas vergrößert, nämlich die Anzahl der Methoden und somit der Bytecode-Instruktionen. Es wurde mit *mtrt* zusammen als eine Einheit analysiert und verteilt. Die Implementierung des TSP-Programms von IBM basiert auf dem Lösungsansatz nach dem Branch-and-Bound-Verfahren. Jeder `Worker`-Thread besitzt eine eigene *WorkQueue*, welche die zu evaluierenden Routen enthält. Wenn die lokale Queue keine Arbeitspakete mehr enthält, sucht der zugehörige Worker in den anderen Queues nach Arbeit und nimmt ggf. diese den anderen Workern ab. Im folgenden Abschnitt über die empirischen Untersuchung werde ich auf die einzelnen Eigenschaften sowie die dort verwendeten Implementierungstechniken detaillierter eingehen, welche auch das jeweilige Laufzeitverhalten der untersuchten Programme beeinflussen.

Die Anwendungen und Benchmark-Programme wurden zusammen mit ihrer transitiven Hülle [Wes04] nach Verteilungseigenschaften analysiert. Das bedeutet, dass alle Verwendungen der im Programm zu erzeugenden Objekte nicht nur innerhalb der Anwendung selbst, sondern auch in den aufgerufenen Methoden der Java-Standard-Bibliothek betrachtet werden. Z.B. wurden auch benutzte Methoden der Klasse `java.util.HashMap` analysiert, die zum Speichern der Objektreferenzen des Anwendungsprogramms verwendet wurden. So wird gewährleistet, dass alle möglichen Seiteneffekte auf die entstehenden Objekte berücksichtigt werden.

8.1.3. Eingesetzte Hardware und Software

Es wurde die Standard JDK von Sun in der Version 1.4.2_07 unter Redhat Enterprise Linux WS Release 3 verwendet, um die untersuchten Programme verteilt ausführen zu lassen. Dem JScatter-System (siehe Kapitel 7) und der Analyseumgebung [Thi01], die zur Programmanalyse nach Verteilungseigenschaften und zur Transformation sowie zur verteilten Ausführung der Anwendungen zum Einsatz kommen, liegt auch diese JDK-Version zugrunde. Für die darunter liegende Kommunikation wurde die schnelle RMI von der Universität Karlsruhe [HMP] in der Version 1.07i mit der JDK-Serialisierung verwendet. Jedoch habe ich zur Bildung der transitiven Hülle des jeweiligen Programms die Standard-Bibliothek der jdk1.3.1 benutzt. Sie ist vom Umfang her viel kleiner als die der jdk1.4.x, stellt aber alle erforderlichen Klassen und Methoden für die zu analysierenden Anwendungsprogramme zur Ver-

fügung. Das bedeutet, dass ein untersuchtes Programm zusammen mit der gebildeten Hülle als „komplett“ darstellt wird. Es kann dann mit seiner Hülle ausgeführt werden. Eine Anwendung zusammen mit ihrer Hülle ist auch der Ausgangspunkt der *stagewise-immutable* Analyse, wobei für die Objekterzeugungsstellen im Anwendungsprogramm alle ihre Verwendungen inklusive derjenigen in der Standard-Bibliothek nach Lese- und Schreibzugriffen untersucht werden. Im Allgemeinen bleibt die Ausführungssemantik des Programms bei Verwendung der jdk1.3.1 erhalten, somit auch die Analyseergebnisse.

Die Verteilung der Anwendungen fand zunächst auf einem PC-Cluster statt, dessen beteiligte Rechner über Standard-Ethernet 100MBit gemeinsam an einem Switch angeschlossen sind und die zusammen zu einem Subnetz gehören. Mit dieser Art der Vernetzung findet die Kommunikation von einem Rechner zum anderen direkt statt und durch die Subnetz-Infrastruktur werden die Datenpakete nur innerhalb dieses Subnetzes verschickt. Außerdem wurde der Cluster so konfiguriert, dass der Datenverkehr während der verteilten Ausführung nicht von außen gestört wird. Jeder PC im Cluster ist mit einem Pentium 4 1.7GHz und 1GB Hauptspeicher ausgerüstet. Das Betriebssystem ist ebenfalls Redhat Enterprise Linux WS Release 3, der Kern hat die Version 2.4.21-37.EL.

Bei den Performance-Messungen wurde außerdem eine spezielle Mehrprozessor-Maschine verwendet, um die Verteilung der Anwendung auf mehreren PCs mit der Ausführung auf solch einem Mehrprozessor-Rechner zu vergleichen und zu evaluieren. Die hier genannte Maschine ist die Sun-Fire der Firma Sun. Sie verfügt in der Hardware-Konfiguration über 8 Sparc-Prozessoren, jeweils mit 900MHz getaktet und besitzt insgesamt 8GB Hauptspeicher. Ein solcher Sparc-Prozessor ist bezüglich der Rechenleistung mit einem doppelt so schnell getakteten Pentium 4 vergleichbar, also etwa 1.7GHz, wie die Cluster-PC. Die Sun-Fire läuft unter SunOS der Version 5.9 und die dort verwendete JVM hat die JDK-Version 1.3.1_06, welche die Abbildung der Java-Threads auf die Betriebssystem-Prozesse unterstützt und somit die Verteilung dieser Prozesse auf die Prozessoren mit dem gemeinsamen Hauptspeicher vom Betriebssystem aus automatisch durchführt.

Das *JScatter*-System wurde außerdem in einem speziellen parallelen Rechnersystem unseres Rechenzentrums PC^2 an der Universität Paderborn [Pad] zur verteilten Ausführung der untersuchten Programme eingesetzt. Dieses System - Arminius Fujitsu Siemens Computers - ist ein spezieller Cluster, der

200 Rechen-Knoten zur Berechnung und 8 Knoten zur Visualisierung umfasst. Die Konfiguration jedes Rechen-Knotens besteht aus Dual-INTEL Xeon 3.2 GHz EM64T mit 4 GByte (DDR-2) Hauptspeicher. Diese Knoten können über ein Hochgeschwindigkeitsnetzwerk *InfiniBand HCA PCI-e* und auch über Standard-Ethernet 1000MBit kommunizieren. Als Betriebssystem wird 64-bit Linux Redhat AS 4 mit dem Kern in der Version 2.6.9 verwendet. Auf den Knoten ist eine JDK in der Version 1.5.0_06-b05 installiert, die speziell für solche Mehrprozessor-Maschinen konstruiert wurde. Die Java-Threads hierbei werden auf die Betriebssystem-Prozesse direkt abgebildet, die dann automatisch vom Betriebssystem auf die zwei Prozessoren verteilt werden.

8.2. Empirische Untersuchung der *stagewise-immutable* Eigenschaften

Die erste Evaluation beschäftigt sich mit der *stagewise-immutable* Eigenschaft der untersuchten Anwendungsprogramme. Als Analysekontext, also die Eingabe für die Analyse, wurde jeweils eine Anwendung zusammen mit ihrer transitiven Hülle übergeben. Dadurch wird gewährleistet, dass alle potenziellen Zugriffe und Seiteneffekte auf die entstehenden Objekte innerhalb der Anwendung nicht nur im Anwendungsprogramm selbst, sondern auch in den benutzten Methoden der verwendeten Bibliotheken berücksichtigt werden. Die Analyse sowie die Transformation des ursprünglichen Programms zu einer verteilten Version finden auf Bytecode-Ebene vor der Laufzeit statt.

Innerhalb des Anwendungsprogramms wurden zunächst alle Erzeugungsstellen identifiziert. Die Analyse untersucht dann alle Verwendungen zur jeweiligen Erzeugungsstelle über den gesamten Analysekontext und ermittelt die Information über die *stagewise-immutable* Eigenschaft zu den Erzeugungsstellen. Konzeptionell besagt die *stagewise-immutable* Eigenschaft, dass ein Objekt ab einem bestimmten Zeitpunkt während des Programmablaufs nur lesend zugegriffen wird. Bei der Evaluation wird die *stagewise-immutable* Eigenschaft in zwei feinere Stufen unterteilt, und zwar in *pure-immutable* und *postponed-immutable*. Wie im Abschnitt 6.4 definiert bedeutet *pure-immutable*, dass ein Objekt nach seiner Initialisierungsphase - also nachdem eine zur Erzeugung aufgerufene Konstruktormethode komplett abgearbeitet wurde - bis zum Programmende nur lesend zugegriffen wird. *Postponed-immutable* Objekte werden nach ihrer Initialisierung noch schreibend zugegriffen, werden aber irgendwann in der Verwendungsphase bis zum Programmende

	CF-Nodes	C2M	M2C	M2M
grocery	15391	141	176	110
mtrt	27000	512	499	1053
tsp	15000	89	101	87
tnbody	15109	127	137	102
wash&wax	19371	50	113	106
mt_moldyn	15375	41	75	88
mt_montecarlo	16297	68	140	214
mt_raytracer	15511	56	113	116

Tabelle 8.2.: Einige Kenndaten über Ergebnisse der *stagewise-immutable* Analyse

nur gelesen. *pure-immutable* Objekte können sofort nach der Aufrufstelle der Konstruktormethode repliziert werden. Technisch ist das Finden solcher Replikationspunkte viel einfacher. Außerdem braucht man dafür nur einmal die Replikationsanweisung nach der Konstruktoraufrufstelle einzufügen. Für *postponed-immutable* Objekte muss die Analyse geeignete Replikationsstellen finden und Replikationscode wird, wenn nötig, an mehreren Stellen eingefügt.

Die Tabelle 8.2 zeigt zusätzlich einige Kenndaten über die Ergebnisse der interprozeduralen Post-Dominator-Analyse. Die azyklischen Graphen, die diese interprozeduralen Post-Dominator-Beziehungen repräsentieren, dienen als Eingabe der *stagewise-immutable* Analyse. Ein solcher Graph enthält die Kontrollfluss-Knoten aller Methoden der Anwendungs- und Bibliotheksklassen, deren Anzahl in der ersten Spalte (CF-Nodes) angegeben ist. Die Kanten sind die normalen Kontrollfluss-Kanten und zusätzlich diejenigen, die interprozedurale Post-Dominator-Aufrufbeziehungen darstellen. Ihre Anzahl im jeweiligen Programm ist in der Tabelle 8.2 unter den Namen C2M (Call-2-Method), M2C (Method-2-Call) und M2M (Method-2-Method) zu finden. Zu bemerken ist, dass eine Aufrufstelle bei der Analyse als ein eigenständiger Kontrollflussknoten (Basisblock) modelliert wurde. Ein urprüngliches Basisblock kann aber mehrere Aufrufstellen enthalten. Dadurch wird die Genauigkeit der interprozeduralen Post-Dominator-Relation erhöht.

So durchlief die *stagewise-immutable* Analyse diesen Graphen, gestartet vom jeweiligen Programm-Exit-Knoten, und berechnete für jede Erzeugungsstelle die *stagewise-immutable* Eigenschaft. Aus den Kenndaten kann man erkennen, dass die Anzahl der interprozeduralen Post-Dominator-Kanten sehr klein ist gegenüber der Anzahl derjenigen, die die CF-Knoten verbinden. Dies be-

deutet, dass die *stagewise-immutable* Analyse die Überprüfung der *stagewise-immutable* Eigenschaft sowie die Suche nach geeigneten Replikationsstellen für jede Erzeugungsstelle sehr schnell durchführen kann. Die Analyse muss im schlimmsten Fall alle dieser interprozeduralen Kanten betrachten, aber nicht alle CF-Kanten der Methoden. Diese CF-Relation innerhalb einer Methode wird ggf. bei Bedarf konstruiert und untersucht.

Im Folgenden werde ich Analyseergebnisse der *stagewise-immutable* Eigenschaft unter zwei verschiedenen Gesichtspunkten repräsentieren. Im ersten Abschnitt werden nur *stagewise-immutable* Ergebnisse innerhalb der Anwendungsprogramme dargestellt. Danach werden Ergebnisse für den gesamten Analysekontext, also die Anwendung und ihre transitive Hülle, präsentiert.

8.2.1. Analyseergebnisse der Anwendungen

Die *stagewise-immutable* Analyse wurde auf die acht oben vorgestellten Anwendungsprogramme angewandt. Als Eingabe für die Analyse wurde nicht nur jeweilige Anwendung, sondern der Vollständigkeit wegen auch ihre gebildete Hülle übergeben. Die hier dargestellten Ergebnisse sind statische Ergebnisse und geben die Anzahl der Erzeugungsstellen mit der jeweiligen Eigenschaft an, die ebenfalls vor der Laufzeit identifiziert wurden. Die Grafik in Abbildung 8.2 stellt die Analyseergebnisse visuell dar. Die X-Achse repräsentiert die einzelne Anwendung mit ihren Ergebnissen über die Erzeugungsstellen (`NEW`-Stelle), davon gefundene *mutable* und *stagewise-immutable* Stellen, die jeweils als Balken von links nach rechts innerhalb einer Anwendung dargestellt werden. Auf der Y-Achse wird die Anzahl der Stellen aufgetragen. Die unterstrichenen Zahlen präsentieren die gewonnenen Lesezugriffe, die lokal durchgeführt würden, wenn entstehende Objekte aus allen als *stagewise-immutable* identifizierten Erzeugungsstellen repliziert würden. Die Anzahl der gewonnenen Lesezugriffe wurde ebenfalls statisch erkannt.

Die identifizierten `NEW`-Stellen umfassen alle Erzeugungsstellen innerhalb des untersuchten Anwendungsprogramms. Die gewonnenen Lesezugriffe können aber nicht nur in der Anwendung, sondern auch in den aufgerufenen Methoden aus verwendeten Bibliotheken vorkommen. Da die Analyse ein statisches Verfahren ist, wird eine in einer Schleife liegende Erzeugungsstelle nur einfach gezählt.

Die ganz links stehenden Balken stellen die gesamte Anzahl der identifizierten `NEW`-Stellen dar, die nur im Anwendungsprogramm vorkommen. Davon

als *mutable* bzw. als *stagewise-immutable* erkannte Erzeugungsstellen werden als mittlere bzw. rechts daneben stehende Balken dargestellt. Diese *stagewise-immutable* Stellen umfassen *pure-* und *postponed-immutable* Stellen. Ihre separaten Anteile werden in Abbildung 8.3 dargestellt.

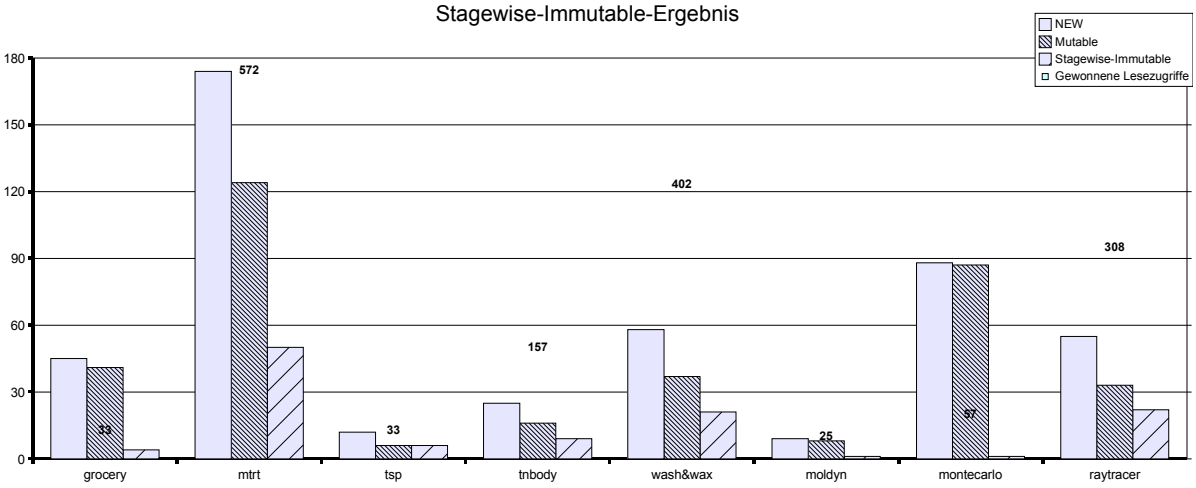

Abbildung 8.2.: Analyseergebnisse zur *stagewise-immutable* Eigenschaft

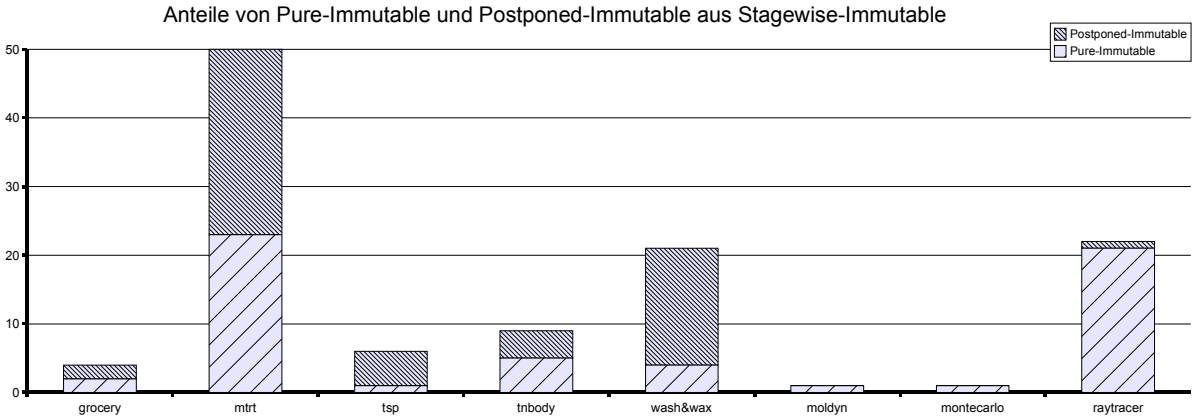

Abbildung 8.3.: Ergebnisse zur Pure- und Postponed-Immutable-Eigenschaft

Bei `mtrt` können sehr viele (572[1]) aus vorhandenen Lesezugriffe lokal durchgeführt werden, dank der 50 gefundenen *stagewise-immutable* `NEW`-Stellen, wovon 23 *pure-immutable* und 27 *postponed-immutable* sind. Wie die Erzeugungsstellen sind die Werte zu den gewonnenen Lesezugriffen sowie die *immutable*

[1]Das entspricht etwa 13,44% aller statisch gezählten Lesezugriffe.

Werte statische Zahlen. Das heißt, dass die Lesezugriffe der potentiellen Verwendungsstellen von der Programmanalyse gefunden werden. Sie können innerhalb einer Schleife auftreten und mehrfach ausgeführt werden oder aber aufgrund bestimmter Laufzeitbedingungen nicht ausgeführt werden.

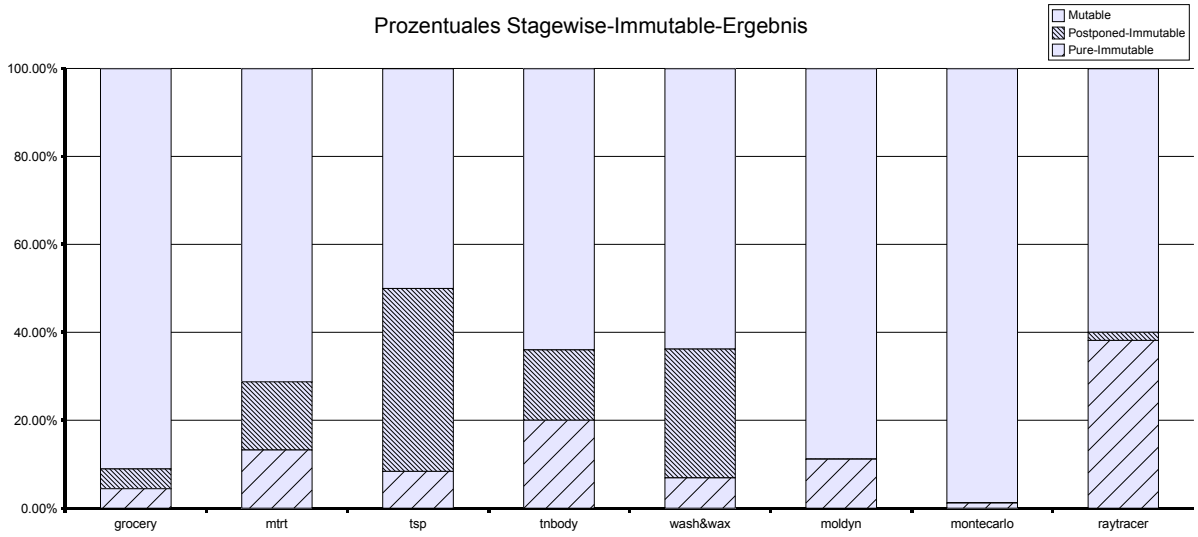

Abbildung 8.4.: Anteile der als *stagewise-* und *pure-immutable* erkannten Stellen zu gesamten `NEW`-Stellen

Zum einfachen Vergleich werden die gefundenen Erzeugungsstellen der Kategorien *pure-immutable*, *postponed-immutable* sowie *mutable* in Abbildung 8.4 prozentual dargestellt. Im `tsp`-Programm konnten besonders viele *postponed-immutable* Stellen erkannt werden. Das sind meistens `StringBuffer`-Objekte (z.B. `sb`), die innerhalb von `System.out.println(sb)` verwendet wurden und die Form `sb=s1+s2;` besitzen, wobei `s1` und `s2` selbst auch String-Objekte sind. Da der Java-Übersetzer einen solchen Ausdruck in `new StringBuffer().append(s1).append(s2).toString()` übersetzt, bedeutet dies für die Analyse, dass das zu erzeugende `StringBuffer`-Objekt nach dieser Aufrufkette, wobei das Objekt modifiziert wird, dann *postponed-immutable* wird.

Da Objekte von Bibliothek-Klassen (noch) nicht verteilt werden, sondern nur diejenigen der Anwendungsklassen, sind zurzeit nicht alle *stagewise-immutable* Erzeugungen zur Verteilung nutzbar. Von solchen Fällen abgesehen können aber durch die Analyseergebnisse viele zu erzeugende Objekte repliziert werden, wie z.B. bei `mtrt` oder `raytracer`, wodurch viele aufwändige entfernte Zugriffe lokal werden. Bei der empirischen Untersuchung zur Ver-

teilung ist zu sehen, dass hierdurch viel Kommunikation eingespart und die gesamte Laufzeit damit auch verkürzt wurde.

8.2.2. Analyseergebnisse der Anwendungen zusammen mit ihren transitiven Hüllen

Im obigen Abschnitt wurden nur Analyseergebnisse für `NEW`-Stellen gezeigt, die innerhalb des betrachteten Anwendungsprogramms vorkommen. Es gibt daneben aber auch sehr viele Objekte, die außerhalb der eigentlichen Anwendung, aber innerhalb der verwendeten Methode in der transitiven Hülle erzeugt werden. Deshalb lohnt es sich, nach *stagewise-immutable* Eigenschaft solcher in den Bibliotheks-Klassen erzeugter Objekte zu suchen. In diesem Abschnitt wird bei jeder Anwendung ihr gesamter Analysekontext analysiert. Die Balkendiagramme in Abbildung 8.5 sowie in Abbildung 8.6 mit gleicher Bedeutung wie im letzten Abschnitt zeigen solche Analyseergebnisse.

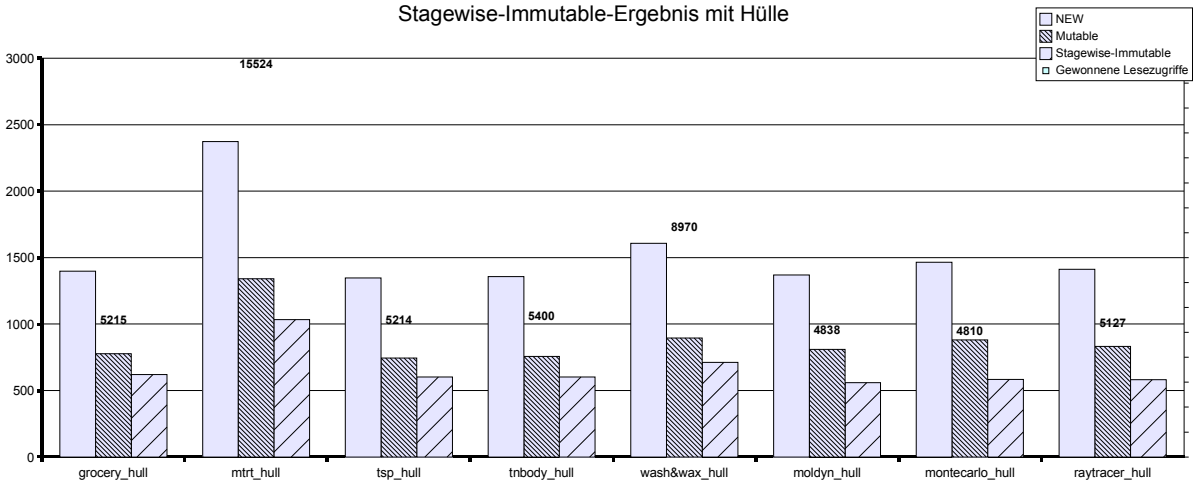

Abbildung 8.5.: Analyseergebnisse zur *stagewise-* und *pure-immutable* Eigenschaft der Anwendungen mit ihrer transitiven Hülle

Durch Hinzunahme der Erzeugungsstellen außerhalb der Anwendungen wurden bei den meisten untersuchten Programmen insgesamt erheblich viel mehr Erzeugungsstellen identifiziert. Der Grund dafür ist, dass die Vielzahl neu hinzukommender Erzeugungsstellen aus Bibliotheken die wenigen innerhalb der Anwendung komplett dominieren. Somit steigt die Anzahl der gesamten `NEW`-Stellen um das 13fache (mtrt) bis 152fache (moldyn) an. Bei

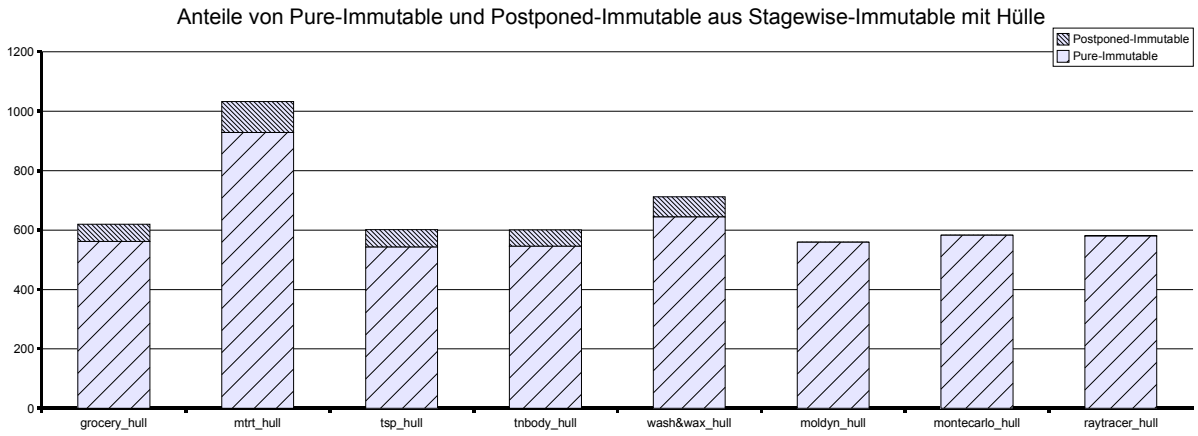

Abbildung 8.6.: Anteile der als *stagewise-* und *pure-immutable* erkannten Stellen im Verhältnis zur Gesamtanzahl der `NEW`-Stellen der Anwendungen und ihrer transitiver Hülle

der Betrachtung der Anwendung allein konnte man erkennen, dass nicht soviel *pure-immutable* Stellen gefunden werden. Mit der Hülle zusammen sind die Balken für diese Eigenschaft (in Abbildung 8.6), und somit auch die Balken für *stagewise-immutable* (in Abbildung 8.5), bei allen Programmen deutlich höher. Auch die statisch gewonnenen Lesezugriffe wachsen bei allen Analysekontexten gleichmäßig an, etwa um einen Faktor zwischen 22 und 193. So kann man daraus schließen, dass viele Erzeugungsstellen in den verwendeten Bibliotheken unabhängig von der Anwendung *pure-immutable* sind. Dies erkennt man am deutlichsten bei den drei Anwendungen aus dem Java-Grande-Benchmark: Hierbei werden nach wie vor keine bzw. bei `raytracer` nur eine `NEW`-Stelle mit der *postponed-immutable* Eigenschaft gefunden, denn nach Definition ist die *postponed-immutable* Eigenschaft stark anwendungsabhängig. Die *pure-immutable* Kandidaten sind nur abhängig davon, welche ihrer Methoden benutzt werden, die potentiell Schreib- oder Lesezugriffe darstellen. Diese Aussage läßt sich durch die prozentuale Darstellung der Analyseergebnisse bestätigen, die in Abbildung 8.7 zu sehen sind.

Bei der prozentualen Darstellung der Ergebnisse ist deutlich erkennbar, dass bei allen Anwendungen etwa 40% der `NEW`-Stellen über den gesamten Analysekontext die *pure-immutable* Eigenschaft besitzen. Die gefundenen *postponed-immutable* Stellen sind im Vergleich dazu nicht viel, machen aber auch etwa 4% Anteil aus. Insgesamt können durch etwa 44% der `NEW`-Stellen bis zu 15524 Lesezugriffe (etwa 6,97% aller statisch gezählten Lesezugriffe) nach

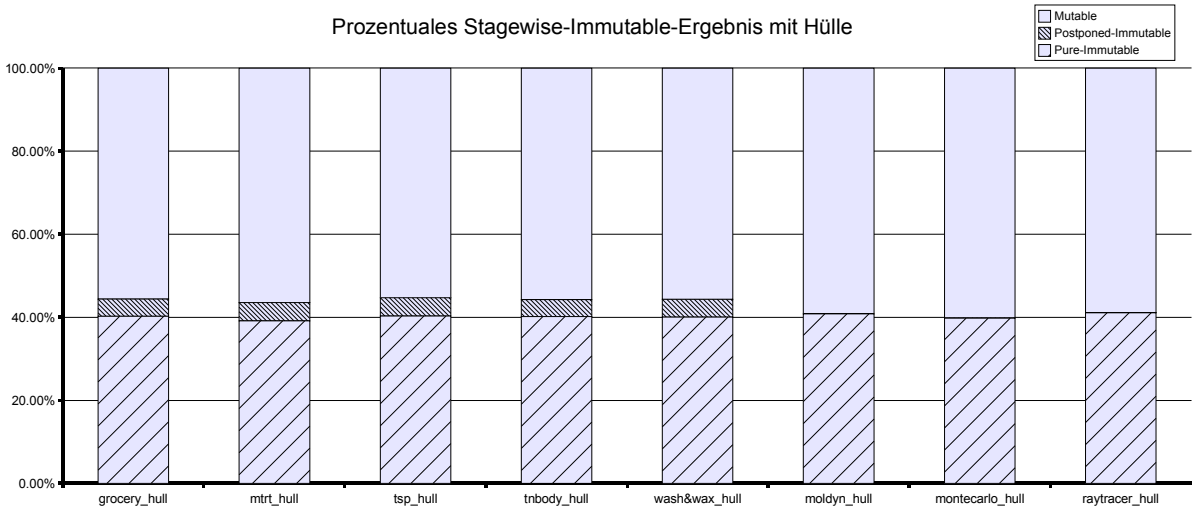

Abbildung 8.7.: Anteile der *stagewise-* und *pure-immutable* Stellen über den Analysekontext

der Replikation lokal werden, so dass entfernter Kommunikationsaufwand erheblich reduziert werden kann.

Diese schönen Erkenntnisse für den gesamten Analysekontext können leider noch nicht in vollem Umfang zur automatischen Verteilung genutzt werden. Denn das JScatter-System ist technisch noch nicht so ausgereift, dass die Anwendung und ihre gesamte transitive Hülle automatisch komplett fehlerfrei transformiert werden. Für die empirische Untersuchung zur Performanz der verteilten Ausführung wurden nur die Anwendungen allein transformiert und deren Objekte automatisch verteilt. Objekte, die in der Standard-Bibliothek erzeugt werden, wurden nicht verteilt. Sie werden in ihrer vollständigen Größe über das Netzwerk verschickt. Trotz dieses Nachteils kann die Immutabilität, selbst wenn sie nur in der Anwendung ausgenutzt wird, schon sehr viel entfernte Kommunikationskosten einsparen, wie wir im folgenden Abschnitt erfahren.

8.3. Empirische Untersuchungen zur Performanz der verteilten Anwendungen

Zur empirischen Untersuchung meines Konzepts zur automatischen Verteilung mehrsträngiger Java-Anwendungen habe ich drei verschiedene Anwen-

dungsprogramme verwendet, die zu drei unterschiedlichen Kategorien gehören. Das sind die *Grocery*-Simulation, der *multithreaded raytracer (mtrt)* aus dem SPECjvm98-Benchmark und das klassische Programm TSP von IBM. Die erste Anwendung repräsentiert eine diskrete ereignisbasierte Simulation, wobei die Threads viel zu berechnen haben, jedoch nicht so viel miteinander kommunizieren. Das mtrt-Programm besitzt eine reguläre Berechnungsstruktur, wobei die Worker-Threads nach der Initialisierung der Daten ihre eigenen Berechnungen durchführen und das Ergebnis jedesmal aktualisieren. Im Gegensatz dazu stellt TSP ein Muster dar, bei dem wenige Berechnungen aber viel Kommunikationen erforderlich sind. Außerdem erschwert die verwendete Load-Balancierungstechnik der Problemlösung die statische Analyse, so dass prinzipiell keine eindeutige Objektzuordnung zu den Worker-Threads gefunden werden konnte.

Für jede Art der Messungen, z.B. Messung der verteilten Ausführung auf 2 Rechnern, wurden 10 Testläufe hintereinander durchgeführt. Die Messergebnisse der ersten beiden Testläufe wurden ignoriert, da hierbei Unregelmäßigkeiten wie beispielsweise das allererste Laden der Klassen der verteilten Laufzeitumgebung über NFS oder Netzwerkverkehr auftreten können. Von den restlichen Ergebnissen wurden dann die Mittelwerte gebildet und ausgewertet.

Im Allgemeinen wurden die Anwendungsprogramme mit Blick auf die Geschwindigkeitssteigerung unter Einsatz der Verteilungspläne mit verschiedenen Platzierungsstrategien evaluiert. So wurde die Laufzeit der verteilten Ausführung auf einer bestimmten Anzahl der beteiligten Rechner gegenüber dem sequentiellen Ablauf auf einem Rechner und der Ausführung auf der Mehrprozessor-Maschine (fireball) verglichen. Die Verteilungsstrategien in einem Verteilungsplan wurden wie folgt angewandt: „Die entstehenden Threads werden einfach zyklisch den verfügbaren JVMs zugeordnet. Nicht-Thread-Objekte werden zuerst nach ihren Lokalitätseigenschaften verteilt, d.h. falls die `isPeer`, `isProxy`- oder `isThreadLocal`-Eigenschaft erfüllt ist, können sie einfach im lokalen Adressraum erzeugt werden. Sonst werden sie zu der Aktivität zugeordnet, die die meiste Workload auf ihnen verrichtet und bei der wenig Kommunikationskosten entstehen ($f_{\text{THREADWORKCOMM}}$)." Zusätzlich wurde der Ansatz zur hybriden Analyse, die die dynamische Programmeigenschaft zur Workload-Abschätzung innerhalb der ODF$f_{\text{THREADWORKCOMM}}$ berechnet, für die Anwendung *mtrt* erprobt. Um die Auswirkung der Immutabilitätseigenschaft auf die Verteilungsgüte zu untersuchen, habe ich bei der Grocery-Simulation separat Mes-

sungen mit und ohne Replikation durchgeführt, bei den beiden anderen aber mit dieser evaluiert. Bei allen drei Anwendungen wurde der statische Anteil einer Klasse auf dem Rechner erzeugt, auf dem der allererste Zugriff darauf stattfand.

Wie im Abschnitt 8.1.3 erläutert, entspricht die Rechenleistung eines mit 900MHz getakteten Sparc-Prozessors (Fireball-Maschine) theoretisch der Leistung eines Pentiums 4 mit doppelter Taktfrequenz. Praktisch zeigt sich die erreichte Performanz eines solchen Sparc-Prozessors aus den Messungen etwa 2,66 fach langsamer als die eines Pentiums 4 mit 1.7GHz. Der Vorteil einer solchen Mehrprozessor-Maschine liegt nur darin, dass der Kommunikationsaufwand zwischen den einzelnen Prozessoren fast Null ist, während die verteilte Ausführung auf einem PC-Cluster auch Zeit zur Kommunikation benötigt. Wenn nur der Rechenaufwand ohne Kommunikation der verteilten Ausführung auf einem Cluster mit dem Rechenaufwand auf der Fireball-Maschine verglichen wird, dann wird mehr Performanz (auch etwa 2,4 fach) beim Cluster erreicht. Mit Kommunikation kann eine Anwendung auf der Mehrprozessor-Maschine schneller ausgeführt werden.

Um zu zeigen, dass das *JScatter*-System auch einfach auf unterschiedlichen Cluster-Architekturen eingesetzt werden kann, wurden zusätzliche Messungen für die *Grocery*-Simulation und das Benchmarkprogramm *mtrt* auf dem speziellen Arminius-Cluster des PC^2 durchgeführt. Als Verteilungsstrategien wurden dafür diejenigen genommen, die bei den vorherigen Messungen auf dem normalen PC-Cluster die besten Verteilungsergebnisse lieferten. Der Arnimius verfügt über zwei unterschiedliche Netzwerk-Architekturen: Das sind die Standard-Ethernet-Verbindung mit 1000MBit und die Hochgeschwindigkeitsverbindung über das InfiniBand [Ass]. Bei der empirischen Untersuchung fand die Kommunikation über die Ethernet-Gbit-Verbindung statt.

Im Folgenden wird die Evaluation nach den drei vorgestellten Anwendungsprogrammen jeweils auch mit zusätzlichen Evaluationsaspekten beschrieben.

8.3.1. Grocery-Simulation

Performanz-Messungen auf einem normalen PC-Cluster

Diese Simulation erzeugt zur Laufzeit insgesamt 16 Threads, davon sind 15 die Kunden und einer der Laden selbst. Um aber zunächst den Unterschied des Laufzeit- und Kommunikationsaufwands zwischen der origina-

len, sequentiellen Version und der transformierten, verteilten Version des Programms zu untersuchen, wurden die beiden Versionen auf demselben Rechner ausgeführt. An den Messergebnissen sollte erkannt werden, welche zusätzliche Belastungsfaktoren auf die tatsächliche Ausführungszeit des Programms entstehen. Das sind der Delegationsmechanismus an `Handle-` und an `InstanceManager`, die Zugriffe auf Klassenanteile eines Objekts, welche auch zu *remote* geworden sind, und Delegationen an erzeugten Stubs und Skeleton.

Die Messungen ergeben, dass die verteilte Version auf einem Rechner um etwa 9% langsamer als die Originalversion ist. Dies entspricht einem „speeddown“ von etwa 0,917. Der Kommunikationsaufwand ist etwa um das 11fache erhöht. Diese Verlangsamung ist auch zu erwarten, denn ein normaler Methodenaufruf im sequentiellen Fall wird in der verteilten Version zu einem *remote* Aufruf. Bis zum eigentlichen Aufruf der Methode muss zunächst die ganze Delegationskette des JScatter-Systems verarbeitet werden: als der Aufruf des Handles, des InstanceManagers, Abfragen an die verteilte Laufzeitumgebung usw. Außerdem kommen noch Aufrufe der von *karmi* generierten Stubs und Skeleton hinzu. Für Parameter und Rückgabewerte müssen die Werte noch serialisiert bzw. deserialisiert werden. Bei der tatsächlichen verteilten Ausführung auf mehreren Rechnern wird der Kommunikationsaufwand nochmals durch das eigentliche Versenden der Daten über das Netz und die Netzlatenz gesteigert.

Zur verteilten Ausführung wurden drei Verteilungspläne zur automatischen Verteilung eingesetzt: Der erste umfasst eine einfache zyklische Verteilung aller Objekte, der zweite enthält die oben beschriebenen Strategien, aber ohne Replikation der *pure-immutable* und *postponed-immutable* Objekte und der dritte mit Replikation. Bei der zyklischen Verteilung wurden die erzeugten Objekte im Ablauf des Programms einfach zyklisch auf die verfügbaren Rechner verteilt, ohne dabei Threads und Nicht-Thread-Objekte zu unterscheiden und auch ohne komplexe Strategien anzuwenden. Alle Objekte wurden dabei als *remote* Objekte behandelt. Für die Simulation wurden Messungen der Verteilung auf 2 bis 9 Rechnern durchgeführt. Abbildung 8.8 zeigt die Ergebnisse zur Ausführungsdauer der Simulation, die mit verschiedenen Verteilungsstrategien ausgeführt wurde.

Auf der X-Achse ist die Anzahl der beteiligten Rechner aufgetragen, wobei auf einem Rechner (sequentiell) und auf der Mehrprozssor-Maschine (fireball) die originale, untransformierte Version des Programms ausgeführt wurde. Die Y-Achse präsentiert die gemessenen Laufzeitwerte in Sekunden. Wie

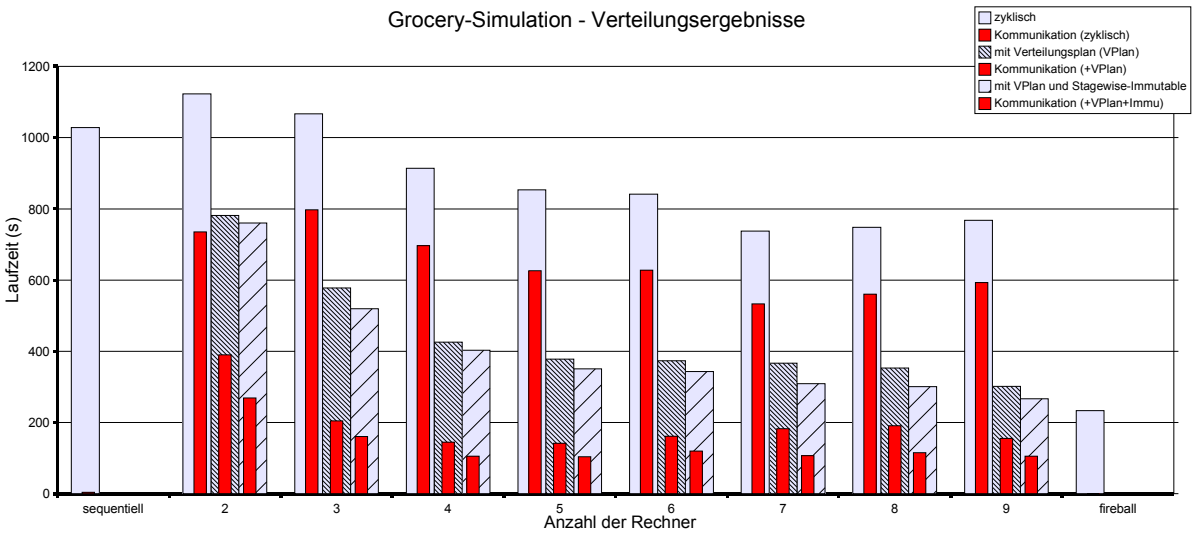

Abbildung 8.8.: Verteilungsergebnisse der Grocery-Simulation mit verschiedenen Verteilungsstrategien

bereits erläutert, wurden hier drei verschiedene Verteilungspläne angewandt, deren Messergebnisse in drei nebeneinander stehenden Balken mit unterschiedlichen Farben dargestellt werden. Von links nach rechts innerhalb einer Gruppe präsentieren die ganz links stehenden Balken Messergebnisse unter Verwendung des Verteilungsplans mit einfacher zyklischer Verteilungsstrategie, die mittleren Balken repräsentieren die oben beschriebenen Strategien und die rechten Balken den Plan mit Replikation von *stagewise-immutable* Objekten. Die inneren schmalen Balken, die jeweils innerhalb der Laufzeitbalken vorkommen, repräsentieren die anteiligen Kommunikationskosten.

Durch die zyklische Verteilung werden die Objekte einfach der Reihe des Programmablaufs nach auf die beteiligten Rechner verteilt. Das verursacht viel entfernte Kommunikation, so dass die Simulation in bestimmten Fällen, z.B. bei Verteilung auf 2 oder 3 Rechner, noch langsamer als die sequentielle ablief. Mit mehr als 7 Rechnern verbesserte sich auch die Laufzeit nicht mehr, das Programm lief sogar noch etwas langsamer. Dieses Phänomen läßt sich auch deutlich am verursachten Kommunikationsaufwand erkennen.

Beim Verteilungsplan mit intelligenten Strategien wird die Performanz spürbar verbessert. Durch Analyseergebnisse konnte die Lokalität vieler Objekte erkannt werden, wie z.B. die Proxy-Eigenschaft und Thread-Lokalität. Außerdem entstand dadurch eine bessere Zuordnung der Objekte zu den Aktivitäten anhand der maximalen Workload und minimaler Kommunikation.

Somit wurden Objekte zusammen mit ihrer Aktivität zusammen platziert, so dass die durch die Transformation „entfernt“ werdenden Methodenaufrufe nur noch innerhalb eines Adressraums stattfinden. Bei manchen Objekten mit `isProxy`- und `isThreadLocal`-Eigenschaft wurden ihre Methodenaufrufe sogar direkt durchgeführt. Durch all diese Verbesserungen können viel entfernte Kommunikationen eingespart werden, was zusätzlich zu einer Geschwindigkeitssteigerung führt.

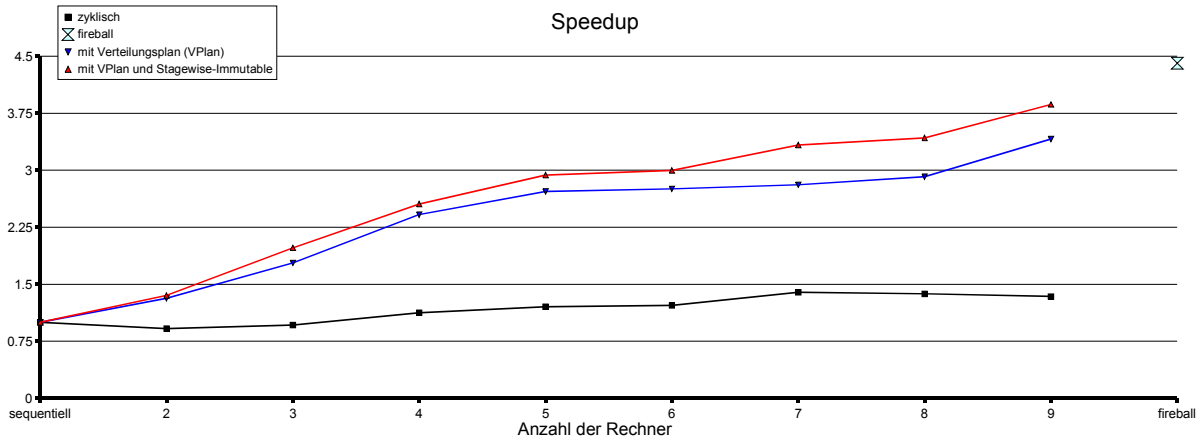

Abbildung 8.9.: Speedup für Grocery-Simulation

Die Performanzsteigerung erkennt man ganz deutlich an den sinkenden inneren Balken für Kommunikationskosten im Vergleich zu denjenigen bei der zyklischen Verteilung. Dieser eingesparte Anteil an entfernter Kommunikation ist auch etwa gleich dem Gewinn an gesamter Ausführungsdauer des Programms, die an den mitteren Balken zu erkennen ist: Schon bei 2 Rechnern lief das Programm schneller als sequentiell und ab 4 Rechnern fällt die gesamte Laufzeit weiter bis unter 40% des sequentiellen Ablaufs. Durch Replikation der als *stagewise-immutable* erkannten Objekte, in der Simulation 4 von insgesamt 45 `NEW`-Stellen, konnte zusätzlich noch bis zu etwa 9.9% an Geschwindigkeitsgewinn (die rechts stehenden Balken) erzielt werden. Durch die lokalisierten Lesezugriffe auf die replizierten Objekte konnte einiges an entfernter Kommunikation eingespart werden, was auch zu den sinkenden (inneren) Balken führte. Somit ist der ersparte Anteil an entfernten Kommunikationen auch der Zusatzgewinnanteil an der gesamten Laufzeit des Programms. Wie man auch an der Grafik erkennt, bleiben die Kommunikationskosten unter Einsatz des Verteilungsplans mit Replikation ab 4 Rechnern fast unverändert, nur die gesamte Laufzeit wird mit steigender Anzahl der beteiligten Rechner verbessert. Das heißt, dass die Parallelität zur Berechnung

der Kunden-Threads bei gleichbleibenden Kommunikationskosten besser genutzt wird. Abbildung 8.9 zeigt uns die Speedup-Werte der Simulation.

Trotz Einsatz der 9 Rechner erreichte die Performanz leider noch nicht ganz die der 8-Prozessor-Maschine. Das läßt sich dadurch erklären, dass der Kommunikationsaufwand auf dem PC-Cluster höher ist. Denn wenn nur der Rechenanteil, also ohne Kommunikation, der gesamten Laufzeit betrachtet würde, würde die Performanz beim Einsatz von 6 Rechnern bereits die Performanz der Fireball-Maschine erreichen. Diese steigt dann mit zunehmender Anzahl der beteiligten Rechner an. Der Vorteil des Einsatzes eines PC-Clusters ist außerdem natürlich die Skalierbarkeit und die Anschaffungskosten gegenüber einem solchen speziellen Rechner. Außerdem wird bei einem solchen Cluster nicht nur die Anzahl der Prozessoren, sondern auch die verfügbare Speicherkapazität mitskaliert. Im Gegensatz dazu sind diese Faktoren bei Mehrprozessor-Maschinen beschränkt.

Neben der Evaluation nach Performanz wurde auch der Aspekt *Startup* des verteilten Programms gegenüber der sequentiellen Version sowie auf der fireball-Maschine untersucht. Die *Startup*-Differenz ist die benötigte Zeit vom Programmstart bis zum Zeitpunkt, an dem der erste arbeitende Thread beginnt zu starten. Während dieser Zeit bei der Simulation werden alle Initialisierungsschritte durchgeführt, beispielsweise Erzeugung der für die Simulation relevanten Objekte, Initialisierung der statischen Werte, Konstanten usw. Im sequentiellen Fall werden in dieser Phase u.a. Klassen geladen, `<clinit>`-Methoden werden zur Initialisierung der statischen Anteile der Klassen durchgeführt oder Thread- bzw. Datenobjekte werden erzeugt.

Abbildung 8.10 zeigt diesen Aspekt für die Simulation auf verschiedenen Rechnern unter Einsatz der drei besprochenen Verteilungspläne. Wie man daran erkennt, wurde zum Startup im sequentiellen Fall sowie auf der fireball-Maschine fast keine Zeit benötigt. Obwohl diese Startup-Differenz im verteilten Fall gemessen wurde, nachdem die gesamte verteilte Laufzeitumgebung betriebsbereit gewesen war, wurde viel Overhead in dieser Phase verursacht. Dabei müssen u.a. alle beteiligten JVMs ihre System- und Anwendungsklassen laden, zu verteilende Objekte ggf. auch entfernt instanziieren, und beim ersten Zugriff auf einen statischen Anteil einer Klasse muss auch die eindeutige repräsentative Instanz dafür erzeugt werden, inklusive aller erforderlichen Schritte für entfernte Zugriffe usw.

Dieser Unterschied läßt sich deutlich an der Grafik erkennen. Mit den beiden anderen Verteilungsplänen wurden die Messergebnisse zu der Startup-Differenz wie erwartet geliefert, wobei viele Objekte lokal erzeugt wurden.

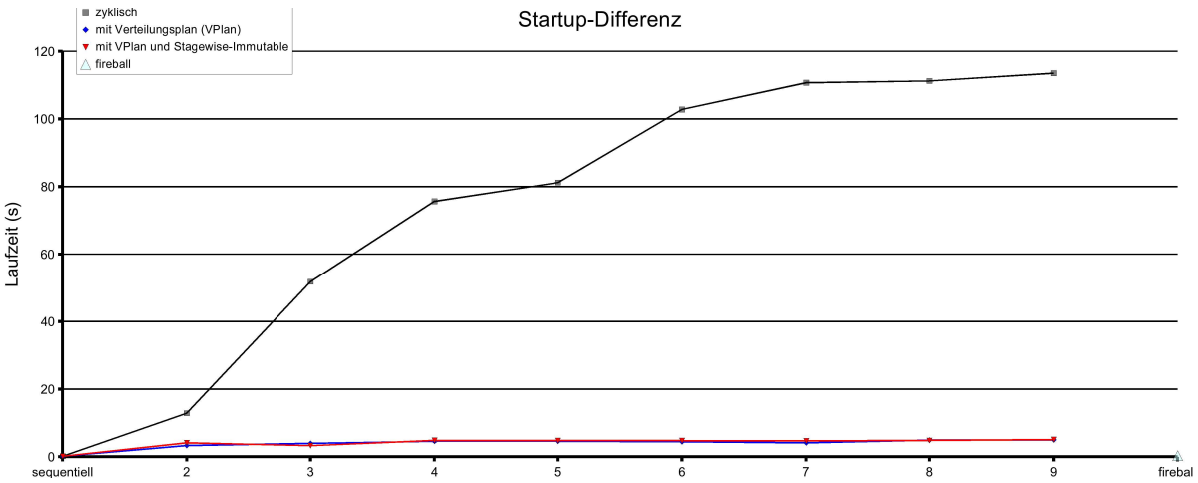

Abbildung 8.10.: Startup-Zeit bei der verteilten Grocery-Simulation

Der Overhead kommt höchstens durch entfernte Erzeugung von Klassenanteilen, Thread-Objekten und einigen Datenobjekten zustande. Durch den Einsatz intelligenter Verteilungsstrategien konnte die Startup-Zeit hierbei recht gering gehalten werden. Der Vergleich mit der zyklischen Verteilung zeigt diesen Unterschied ganz deutlich. Bei der zyklischen Verteilung wächst die Startup-Differenz auch mit zunehmender Anzahl der Rechner, in großem Maße im Bereich zwischen 2 bis 6 Rechnern, dann mit immer kleiner werdenden Zeitdifferenzen. Der Grund dafür ist, dass neben der ständig entfernten Instanziierung der zu erzeugenden Objekte auch statische Anteile initialisiert werden, was auch entfernter Erzeugung von solchen für Klassenanteile eindeutig präsentierenden Instanzen entspricht.

Um aber die Güte der Verteilung und damit das Konzept zur automatischen Verteilung fair zu evaluieren, wurden bei allen drei untersuchten Programmen die Messergebnisse zum Performanzvergleich erst nach Ausführung des Startups aufgenommen und in den Diagrammen dargestellt.

Performanz-Messungen auf dem Arminius-Cluster (PC^2)

Zusätzlich wurden Performanz-Messungen der *Grocery*-Simulation auf dem Arminius-Cluster unseres PC^2 durchgeführt. Die Untersuchung wurde auf die gleiche Weise durchgeführt: Das originale, untransformierte Programm wurde auf einem Knoten ausgeführt. Daraus ergaben sich die Messwerte für die sequentielle Version. Die Simulation wurde dann auf verschiedenen

Rechner-Knoten, also zwischen 2 bis 9 Knoten, verteilt. Die Kommunikation fand über eine Standard-Ethernet-Gbit-Verbindung statt. Die Messungen wurden mehrmals (mehr als 10) durchgeführt. Die Messergebnisse bilden sich aus den jeweiligen Mittelwerten. Als Verteilungsstrategien wurden die Strategien genommen, die die besten Verteilungsergebnisse bei der Messung auf dem normalen Pool-Cluster lieferten. Im Folgenden werden die Verteilungsergebnisse miteinander verglichen, die sich aus der Verteilung auf dem normalen Pool-Cluster und auf dem Arminius-Cluster über Ethernet-Gbit-Verbindung ergaben.

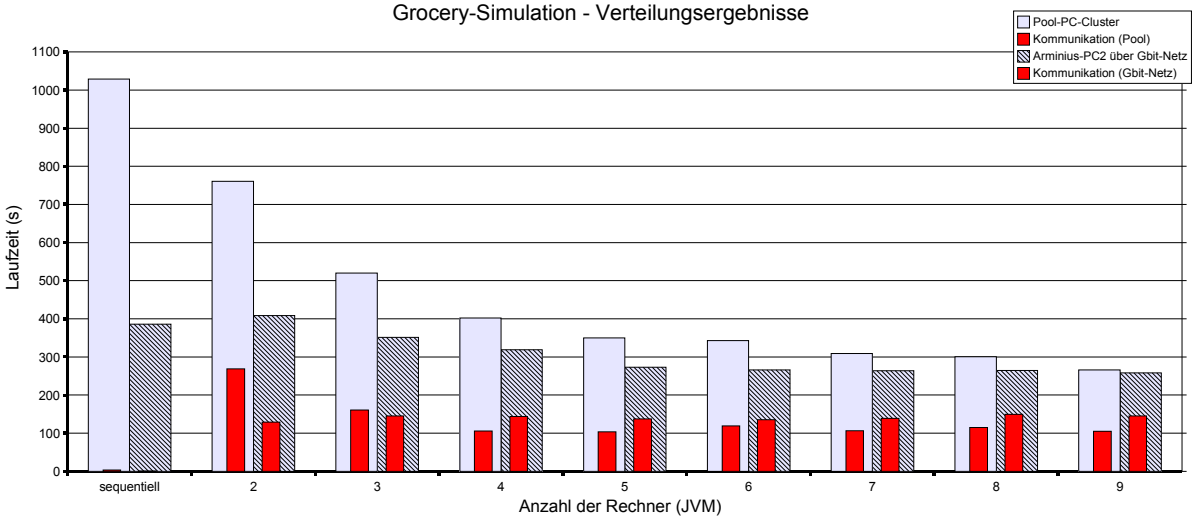

Abbildung 8.11.: Verteilungsergebnisse der Simulation auf normalem Pool-Cluster und auf Arminius-Cluster

Abbildung 8.11 repräsentiert die genannten Ergebnisse: Die linken Balken stellen die Laufzeitwerte auf dem normalen Pool-Cluster, die rechten Balken die Werte auf dem Arminius-Cluster dar. Die darin enthaltenen schmaleren Balken zeigen die jeweiligen anteiligen Kommunikationszeiten. Zu beachten ist, dass auf der X-Achse die Anzahl der Rechner (Knoten) aufgetragen ist. Ein Knoten auf dem Arminius-Cluster besitzt aber zwei CPUs, die die Prozesse parallel ausführen können. Wie oben schon erläutert unterstützt die JVM auf jedem Rechner-Knoten des Arminius-Clusters die Abbildung der Java-Threads auf die Unix-Prozesse. Die Prozesse werden dann vom Betriebssystem automatisch auf die CPUs verteilt und parallel ausgeführt. Das heißt, dass die echte Parallelität bereits auf einem solchen Knoten stattfindet. Dies ist auch der Grund, warum die Laufzeit der sequentiellen Version auf einem Arminius-Knoten viel kürzer ist als die auf einem normalen PC, und zwar um

einen Faktor von 2.7. Außerdem besitzt eine CPU eines solchen Dual-System-Rechners eine andere Architektur als die eines normalen PCs: Ein Xeon-CPU ist mit mehr 2nd-Level-Cache (1MB) ausgerüstet und viel höher (3,2GHz) getaktet. Ein Rechen-Knoten des PC^2 besitzt auch mehr und schnelleren Speicher (4Gbyte DDR-2).

Insgesamt kann man mit so einer speziellen Architektur mehr Performanz erreichen als mit herkömmlichen PC-Clustern. Der beste erreichte Faktor bei einer verteilten Ausführung im Vergleich zu der sequentiellen auf demselben Arminius-Cluster beträgt etwa 1.86. Aufgrund des hohen Kommunikationsaufwands ist dieser Gewinn aber nicht so groß. Dies kann man deutlich an den anteiligen Kommunikationskosten der gesamten Laufzeiten erkennen. Wie in Abbildung 8.11 zu sehen lief das verteilte Programm auf zwei Arminius-Knoten noch langsamer als die sequentielle Version, ab 3 Knoten ist es dann schneller. Abgesehen vom Kommunikationsaufwand kann die Parallelität durch die Verteilung auf mehreren Rechnern, der jeweils 2 CPUs besitzt, erzielt werden. Dadurch bekommt man insgesamt bessere Performanz als beim normalen PC-Cluster. Abbildung 8.12 zeigt die Speedup-Werte. Bei verteilter Ausführung auf dem Arminius-Cluster konnte ab 6 Knoten nicht mehr viel Performanz erzielt werden, dafür nahm aber der Kommunikationsaufwand zu.

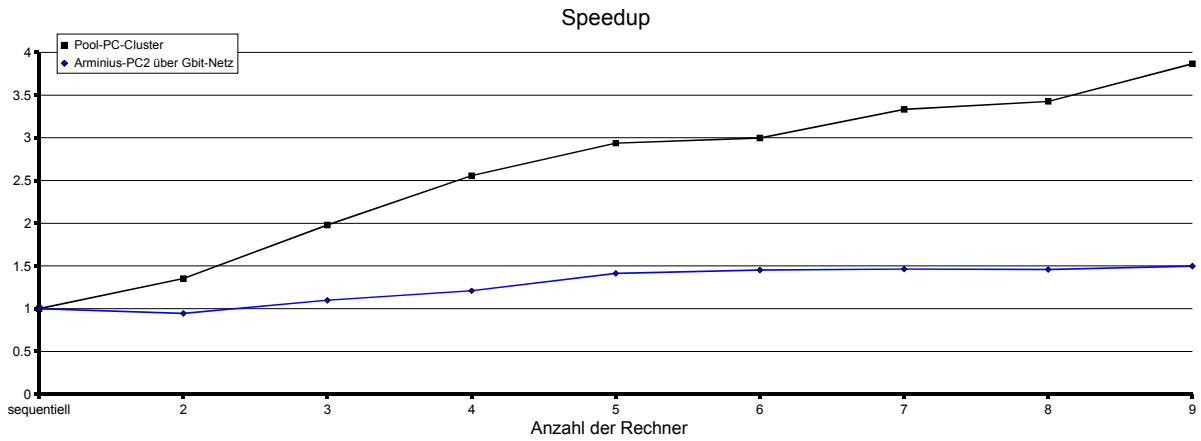

Abbildung 8.12.: Speedup-Werte - Verteilung auf normalem Pool-Cluster und auf Arminius-Cluster

Das Phänomen, ab einer bestimmten Anzahl beteiligter Rechner keinen Performanzgewinn mehr zu erreichen, ist auch zu erwarten. Denn die Grocery-Simulation ist eine irreguläre Anwendung, die im Allgemeinen willkürliche Kommunikationsstrukturen enthalten kann. Durch die wachsende An-

zahl der Threads im Programm entsteht auch mehr Kommunikationsaufwand. Man erreicht die Geschwindigkeitssteigerung nur bis zu einer bestimmten Anzahl der Knoten. Danach kann keine Performanz mehr gewonnen werden, das Programm läuft sogar noch langsamer durch zunehmende Kommunikationskosten. Auf dem Arminius-Cluster nahmen die Kommunikationskosten ab 6 Knoten (entspricht 12 CPUs) ständig zu: Im Vergleich zu Kommunikationskosten auf 6 Rechner wachsen sie bei 7 Knoten um 3.03%, bei 8 Knoten um 10.39% und bei 9 Knoten um 7.49%. Trotz zunehmender Kommunikationskosten bleiben die gesamten Laufzeiten ab 7 Knoten aber fast unverändert. Das liegt daran, dass mehr Rechenleistung zur Verfügung steht und dies den wachsenden Kommunikationsaufwand ausgleicht.

In Abbildung 8.11 ist auch deutlich zu erkennen, dass der Kommunikationsaufwand beim Arminius-Cluster ab 4 Knoten etwas höher ist als beim PC-Cluster, obwohl die Knoten dort über eine Ethernet-Gbit-Verbindung kommunizieren. Der Grund ist, dass die Latenz zu den Zeitpunkten der Messungen zu hoch war. Die Rechner des Pool-Clusters sind an einen 100Mbit-Switch angeschlossen und die Verbindung zwischen ihnen findet direkt statt. Während der Messungen auf dem PC-Pool waren auch keine anderen Störfaktoren vorhanden. Daher war die Netzlatenz nicht hoch. Beim PC^2 sind aber alle 200 Knoten an einen einzigen Gbit-Switch angeschlossen. Und dies ist der Flaschenhals: Intern besitzt der Switch auch mehrere *Backplanes*, die für das Verbinden von 200 Knoten zuständig sind. Im günstigen Fall sind die für die Evaluation reservierten Knoten durch eine *Backplane* physikalisch direkt verbunden und können direkt miteinander kommunizieren. Hier ist die Latenz nicht so hoch. Wenn die reservierten Knoten physikalisch an verschiedenen *Backplanes* angeschlossen sind, muss der Switch intern den Datenverkehr der Knoten auch noch über diese *Platinen* leiten. Zusätzlich ist die Latenz noch davon abhängig, wie stark die Nutzung des gesamten Netzes ist, und somit der Engpass des Switches. Alle dieser Aspekte zusammen führten zu hoher Latenz während der Messungen, daher musste die Performanz insgesamt darunter leiden.

Die Evaluation auf dem Arminius-Cluster des PC^2 zeigte, dass durch mehr Rechnerleistung mehr Parallelität und somit auch mehr Performanz erzielt werden kann. Ein wichtiger Schlüssel zur Geschwindigkeitssteigerung durch die geschickte verteilte Ausführung ist die schnelle Kommunikation, wobei die Latenz eine sehr wichtige Rolle spielt. Im Allgemeinen besitzen irreguläre Anwendungen wie die Grocery-Simulation Charakteristiken, die nicht typisch für High-Performance-Computing sind wie numerische Berechnun-

gen, Renderer usw. Daher konnte die parallele Ausführung mit geschickten Verteilungsstrategien keine sehr guten Ergebnisse auf dem Arminius-Cluster erreichen, der aber für solche HP-Anwendungen viel besser geeignet ist. Im folgenden Abschnitt werden wir erfahren, dass die verteilte Ausführung des Raytracer-Programms *mtrt* mit regulären Strukturen sehr gute Performanz auf diesem speziellen Cluster erreichte.

8.3.2. Multithreaded Raytracer (mtrt) aus dem SPECjvm98

Der nächste zu untersuchende Kandidat ist das bekannte Raytracer-Programm aus dem SPECjvm98-Benchmark [Cor] in der mehrsträngigen Version. Man kann die Größe des Canvas des zu rendernden Bildes und die Anzahl der Threads anfangs definieren. Wenn mehr als ein Thread angegeben wird, z.B. $n > 1$ Threads, besitzt jeder die Daten der gesamten Szene - jeder erzeugt also ein eigenes Szene-Objekt - aber rendert nur eine Kachel der Größe $\frac{1}{n}$. Zum Start lesen die Threads aus einer Eingabedatei die Spezifikationsdaten des zu rendernden Bildes. Hierin sind die Objekte mit ihren Koordinaten, Formen, Farb- und Lichtinformationen usw. definiert. Aus diesen Daten werden jeweils zugehörige Objekte aus entsprechenden Klassen erzeugt, die die Datenstrukturen repräsentieren, z.B. `mtrt.Vector`, `mtrt.Point`, `mtrt.Light` ... Nachdem die erforderlichen initialen Strukturen komplett aufgebaut wurden, fängt jeder Worker-Thread an, seine Kachel zu rendern. Hier steckt die eigentliche Arbeit bzw. der Workload. So durchläuft jeder Worker den gesamten Canvas mittels zweier geschachtelter Schleifen in Breite und Höhe und rendert jedes seiner Kachel zugehörige Pixel. Dies wurde mittels der Methode `Shade()` in der Klasse `mtrt.Scene` realisiert. In der originalen Version des Programms wird jedes fertig gerenderte Pixel sofort im Canvas mittels der `mtrt.Canvas.Write(...)`-Methode aktualisiert. Seien *height* und *width* die definierte Höhe und Breite des Bildes, dann werden diese beiden Methoden bei jedem Worker-Thread $height \times \frac{1}{n} \times width$-mal ausgeführt.

Zur Evaluation wurden fünf Threads eingesetzt, um ein Canvas der Größe 805×805 zu rendern. Der Grund für diese ungewöhnliche Größe ist, dass unabhängig von der Anzahl der Threads die Überprüfung der Pixels mittels der `mtrt.IntersectPt.FindNearestIsect()`-Methode an den Kachelgrenzen bei bestimmten Kachelgrößen zu endloser Rekursion führt. Dies ist ein reproduzierbarer Fehler im originalen Programm. Außerdem habe ich bei der Untersuchung der Verteilung einige Optimierungen und Erweiterungen der ursprünglichen Version des Programms eingebaut: Im originalen Pro-

gramm wird bei jeder Schleifeniteration jeweils ein neues `mtrt.Color`- und `mtrt.Point`-Objekt erzeugt, dies führt zu unnötigem Speicherverbrauch. Deshalb wurde es so optimiert, dass die beiden Objekte außerhalb der Schleife erzeugt und bei jeder Iteration erneut initialisiert und wiederverwendet werden. Neben der Aktualisierung des Canvas nach jeder Berechnung eines Pixels in der originalen Version, die im Folgenden mit *opt0* bezeichnet wird, wurde das gesamte Rendering um zwei weitere Versionen erweitert: Bei der einen wird die Aktualisierung des Canvas nach fertigem Rendern einer Spalte der Kachel (*opt1*) und bei der anderen nach dem kompletten Rendern der Kachel (*opt2*) durchgeführt. Hierdurch werden die Kommunikationsschritte der Threads mit dem Canvas reduziert, dafür werden pro Kommunikationsschritt aber mehr Daten verschickt. Diese Änderungen der ursprünglichen Version dienen nur dem Zweck, die Güte der angewandten Verteilungsstrategien zu evaluieren. Man kann vor allem dadurch zeigen, für welche Berechnungs- sowie Kommunikationsstrukturen sich der Einsatz bestimmter Verteilungsstrategien eignet.

Performanz-Messungen auf einem normalen PC-Cluster

Eine Laufzeitmessung bei der Evaluation erfolgt ab dem Start des ersten Threads und endet, nachdem der Canvas komplett gerendert wurde. Wie bei der Grocery-Simulation werden die Laufzeit und der Kommunikationsaufwand der sequentiellen, originalen Version gegenüber der transformierten Version *opt0*, ausgeführt auf einem Rechner verglichen. Hierbei kann man den durch die Transformations- und verteilten Laufzeitumgebung verursachten Overhead erkennen. So lief die transformierte Version auf einem Rechner etwa 77% langsamer als die sequentielle, dies entspricht etwa einem *speed-down* von 0.566. Diese verursachte etwa 157fach mehr Kommunikationsaufwand als die Originale. Dieser höhere Kommunikationsaufwand entsteht durch das ständig Aktualisieren des Canvas bei jeder Iteration plus zusätzlicher Kosten für entfernte Kommunikation, die bei der verteilten Version trotz der Ausführung auf einem Rechner entstehen.

Bei der Evaluation des mtrt-Programms wurde die zyklische Verteilung nicht eingesetzt. Die hier verwendeten Verteilungsstrategien sind die gleichen wie bei der Grocery-Simulation. Außerdem wurde das Konzept zur hybriden Analyse hierbei angewandt: Zur Abschätzung des Workloads und Kommunikationsaufwands mittels der ODF $f_{\text{THREADWORKCOMM}}$ wurde zunächst bei allen vorkommenden Schleifen die Anzahl der Durchläufe jeweils der

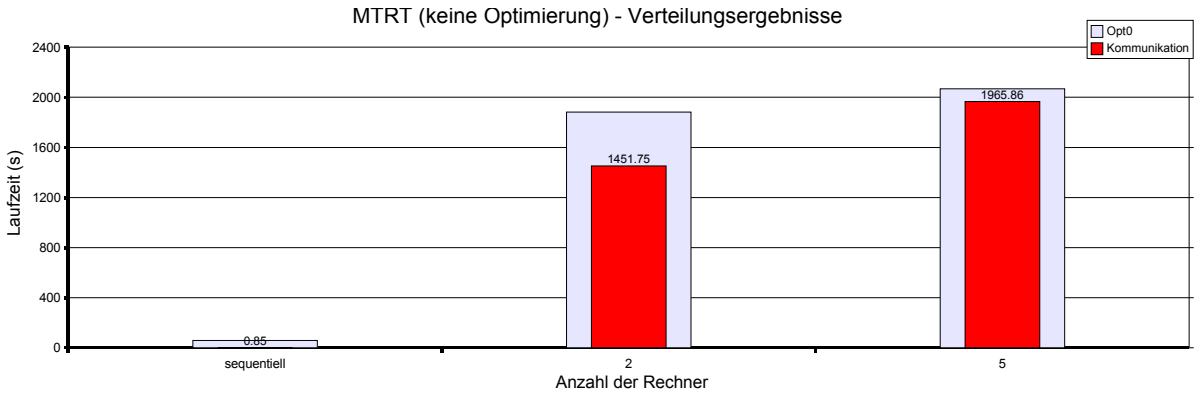

Abbildung 8.13.: Verteilungsergebnisse der originalen Version des mtrt-Programms

Wert 10 angenommen. Da die Laufzeitkonstanten zur Höhe und Breite (jeweils 805) des Canvas zur Laufzeit bekannt waren, bevor die Worker anfangen zu starten, konnten diese Werte direkt in die Verteilungsberechnungen eingesetzt werden. Dies führte zur erneuten Berechnung der Platzierungsentscheidung der Objekte zu den Workern, bevor die Objekte tatsächlich erzeugt wurden (alle zum Rendern benötigten Datenobjekte wurden innerhalb der Szene erzeugt, also nachdem ein Worker gestartet wurde). Durch die hybride Analyse zeigten die neu berechneten Verteilungshinweise, dass alle innerhalb der geschachtelten Schleife zum Rendern zu erzeugenden Objekte zusammen mit dem zugehörigen Worker platziert werden sollen. Durch die Bekanntgabe der Laufzeitkonstanten verrichtet ein Worker die meiste Workload und Methodenaufrufe auf diesen Objekten. Zusammen mit anderen statischen Strategien wie `isPeer`, `isProxy`, `isThreadLocal` und `isStagewiseImmutable` ergaben sich die Platzierungsentscheidungen: Außer dem Canvas-Objekt, auf das alle Worker zugreifen, wurde die komplette Szene (`Scene`-Objekt) mit allen dazu gehörigen Datenobjekten zusammen mit ihrem zugehörigen Worker-Thread in einen Adressraum platziert. Die hier als *stagewise-immutable* erkannten Erzeugungsstellen besitzen auch die `isThreadLocal`-Eigenschaft, so dass die entstehenden Objekte ebenfalls lokal erzeugt werden, anstatt diese zu replizieren. Der einzige entfernte Methodenaufruf ist das Schreiben des Canvas.

Die ersten Messergebnisse des mtrt-Programms werden in Abbildung 8.13 dargestellt. Es handelt sich hierbei um das originale Programm (*opt0*). Wie man erkennt, wurde die meiste Zeit des Ablaufs für Kommunikation benötigt, also etwa 77% bzw. 95% der gesamten Laufzeit auf 2 bzw. auf 5 Rech-

nern. Die anteiligen Kommunikationskosten werden durch die jeweiligen inneren Balken visualisiert. Das wurde durch Aktualisierung des Canvas bei jeder Iteration verursacht, indem die `Write(...)`-Methode bei jedem Thread $805 \times \frac{1}{5} \times 805 = 129605$mal entfernt aufgerufen wurde. Trotz der Lokalitätsoptimierung der Datenobjekte mittels geschickter Verteilungsstrategien wurde das Programm noch erheblich verlangsamt, auch im Fall mit mehr beteiligten Rechnern. Daraus kann man schließen, dass die Performanz bei der verteilten Ausführung insgesamt hauptsächlich durch entfernte Methodenaufrufe beeinträchtigt wird. Um dies genauer zu untersuchen, wurde deshalb das mtrt-Programm mit Kommunikationsoptimierungen *opt1* und *opt2* verteilt ausgeführt, also mit einmaliger Aktualisierung einer Spalte bzw. der ganzen Kachel. Abbildung 8.14 zeigt die Verteilungsergebnisse.

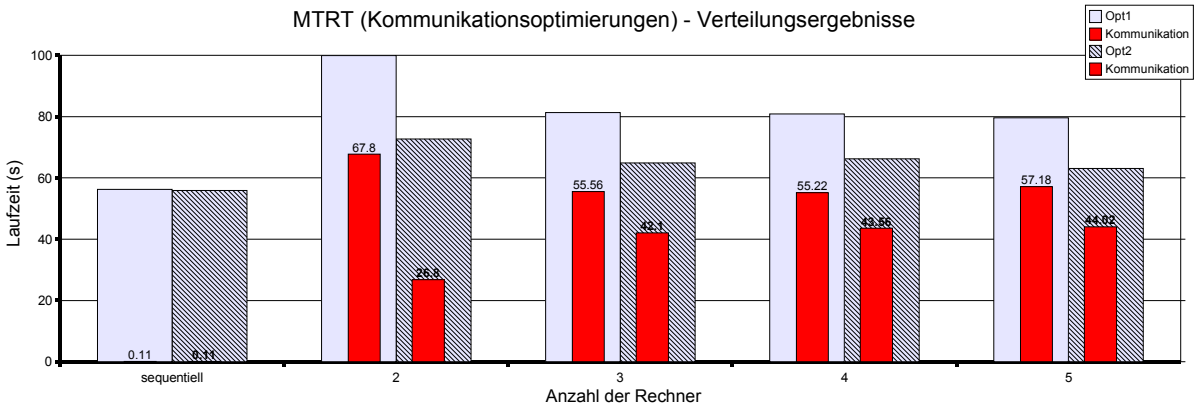

Abbildung 8.14.: Verteilungsergebnisse mit Kommunikationsoptimierung *opt1* und *opt2* für mtrt

Gegenüber der vorherigen verteilten Ausführung kann hier sehr viel Performanz aufgrund der Kommunikationsoptimierung herausgeholt werden. Beim Einsatz von 5 Rechnern konnte das Programm mit *opt1* um etwa 25fach, und mit *opt2* 32fach schneller gegenüber der unoptimierten Version laufen. Die sequentielle Ausführung wurde dadurch auch etwas schneller, da hierdurch anscheinend sehr viele Methodenaufrufe eingespart werden. Diese Optimierungsschritte basieren auf den Erkenntnissen, dass ein IP-Paket mindestens 15 Bytes fester Verwaltungsdaten (2 Bytes für MAC-Adresse, 2 für IP ...) und die Nutzdaten enthält. Somit wird ein Paket bis einer Gesamtgröße von maximal 1,5 KBytes übertragen. Daher besteht nur sehr wenig Unterschied bei der Übertragungsdauer zwischen den Daten eines Pixels und den Daten

der gesamten Spalte (zweidimensionales Array für die ganze Spalte mit Farbinformation) bzw. denen des gesamten Canvas. (dreidimensionales Array).

Der entstehende Aufwand zum Verschicken von mehr Daten kann in Kauf genommen werden. Dies wird ganz deutlich durch die inneren Balken für Kommunikationszeit gezeigt. Bei der *opt1*-Version kann bis zu etwa 97% an Kommunikationskosten bei der verteilten Ausführung auf 5 Rechnern eingespart werden. Obwohl bei der Optimierung *opt2* mehr Daten aber nur einmalig nach dem fertigen Rendern verschickt wurden, wurden trotzdem Kommunikationskosten eingespart und somit die gesamte Laufzeit verkürzt. Bei der verteilten Ausführung auf 2 Rechnern mit der Optimierung *opt2* wurden 3 Worker-Threads zusammen mit dem Canvas-Objekt auf demselben Rechner platziert. Daher fand die Kommunikation dieser 3 Worker mit dem Canvas dann lokal statt. Dies führt zur Einsparung von etwa 98% Kosten an entfernten Methodeaufrufen. Mit Ausnahme von diesem sind die anderen inneren Balken für die Kommunikationszeit fast gleich hoch. Dies entspricht auch der Tatsache, dass bei Verteilung auf mehreren Rechnern auch soviel Daten verschickt werden.

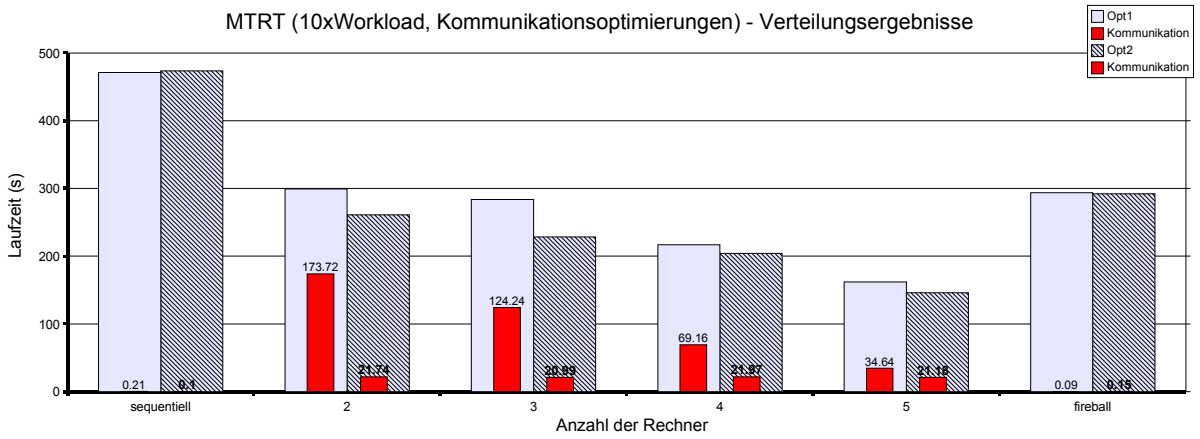

Abbildung 8.15.: Verteilungsergebnisse mit Kommunikationsoptimierung *opt1* und *opt2* und 10fach Workload für mtrt

Diese Ergebnisse zeigen, dass die entfernte Kommunikation durch entfernte Methodenaufrufe bei der verteilten Ausführung eines mehrsträngigen Programms schwergewichtig ist. Wenn man aber nur den eigentlichen Workload für das Rendern in Abbildung 8.14 betrachtet, also den restlichen Teil ohne die Kommunikationszeit bei allen Balken, dann ist diese Zeit kürzer als die bei der sequentiellen Ausführung. Dies kommt dadurch, dass die Parallelität

im verteilten Fall besser ausgenutzt wurde als die verzahnte, aber sequentielle Ausführung auf einem Rechner. Es lohnt sich daher der Fall zu untersuchen, bei dem der Workload mehr Gewicht als die Kommunikation aufweist. Deshalb wurde zu diesem Zweck der Workload für das Rendern zusätzlich um das 10fache erhöht. Dieser Zusatzaufwand entspricht etwa mehr Berechnungen zusätzlicher Informationen für jedes Pixel wie etwa Farben, Lichtreflexion und 10fach Antialiasing usw. Die Verteilungsergebnisse dafür werden in Abbildung 8.15 dargestellt.

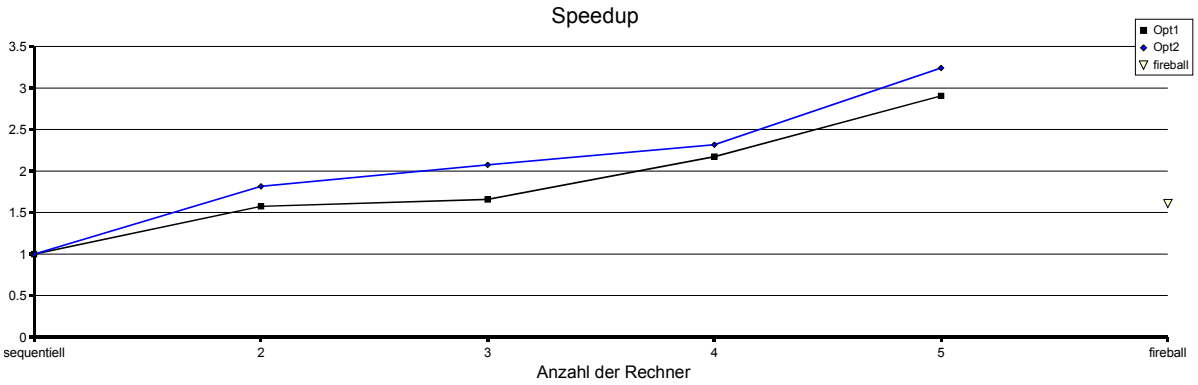

Abbildung 8.16.: Speedup bei mtrt mit 10fach Workload und Optimierungen *opt1*, *opt2*

Daran erkennt man, dass das Verschicken der gesamten Kachel (*opt2*) im verteilten Fall gleich bleibt (die inneren Balken bei *opt2*). Zum Vergleich mit den Ergebnissen über Kommunikationskosten bei der Optimierung *opt1* in Abbildung 8.14 sinken die inneren Balken für Kommunikationskosten hierbei mit steigender Rechnerzahl. Der Grund dafür ist, dass ohne Erhöhung des Workloads die Worker-Threads schnell fertig berechnen konnten und ihre Daten gleichzeitig auf den Canvas aktualisieren. Da das Canvas-Objekt auf einem Rechner liegt, entsteht dort ein Engpass zur Entgegennahme der entfernten Aufrufe. Denn auf dieser Seite gibt es nur einen einzigen Thread, der von RMI erzeugt wurde und das Skeleton des Canvas-Objekts repräsentiert. Er muss alle diese entfernten Aufrufe nach und nach abarbeiten. Durch die Erhöhung des Workloads haben die Worker viel mehr zu tun als zu kommunizieren. Die einzelnen Messungen (wurde schon in der Grafik zusammengefasst) zeigten, dass je mehr ein Worker Zeit zum Rendern braucht, desto weniger Zeit benötigt er zur Kommunikation. Je mehr Rechner sich daran beteiligen, um so gleichmäßiger wird der Workload auf alle Workern verteilt, so dass der Kommunikationsaufwand insgesamt reduziert wurde. Bei diesem

Performanzvergleich wurde das Messergebnis auf der fireball-Maschine einbezogen. So kann man deutlich erkennen, dass allein auf 2 Rechnern schon mehr Performanz erreicht wurde als auf der Mehrprozessor-Maschine. Wenn man nur den Rechenanteil ohne Kommunikation in Betracht zieht, dann wird etwa 2,4fache mehr Performanz beim Cluster mit 5 Rechnern erreicht als bei der Mehrprozessor-Maschine.

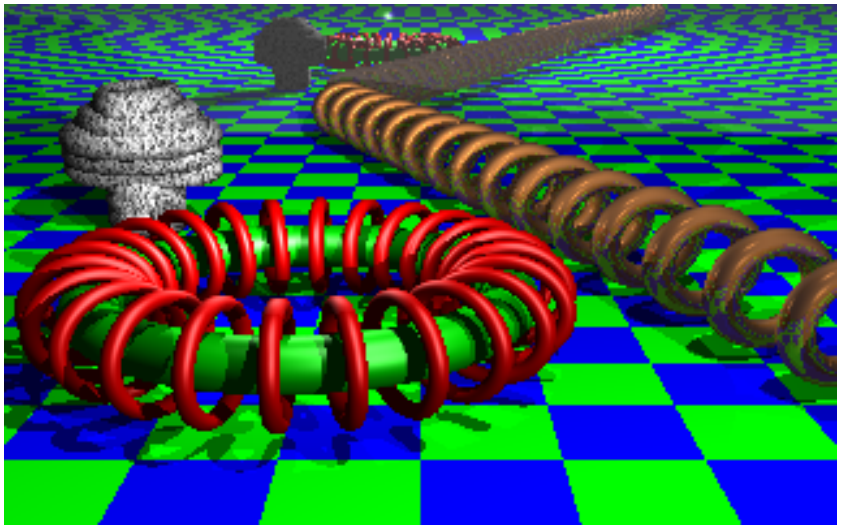

Abbildung 8.17.: Die gerenderte Szene

Abbildung 8.16 stellt die Speedup-Daten dar. So kann eine verteilte Ausführung mit geschickten Verteilungsstrategien sehr gute Performanz erreichen. Diese kann aber nur erzielt werden, wenn das Programm sehr viel mehr Zeit mit Berechnungen verbringt als mit Kommunikation. Sonst können intelligente Strategien zur Verteilung auch nicht zur Geschwindigkeitssteigerung beitragen. Zusätzlich zur Gültigkeitsprüfung der gerenderten Daten wurde das Ergebnis des Programms bei der verteilten Ausführung mit dem der sequentiellen Ausführung der originalen Version verglichen. Das sind die Pixeldaten, die sich aus dem Rendering ergeben. Das Ergebnis läßt sich in Abbildung 8.17 sehen.

Performanz-Messungen auf dem Arminius-Cluster (PC^2)

Für dieses Benchmark-Programm wurden ebenfalls Messungen zur automatischen Verteilung auf dem Arminius-Cluster des PC^2 durchgeführt. Wie bei der *Grocery*-Simulation wurden die Verteilungsstrategien verwendet, die auf

dem normalen Pool-Cluster die besten Ergebnisse ermittelten. Abbildung 8.18 zeigt die Messwerte zum Raytracer *mtrt*, die sich aus der empirischen Untersuchung auf dem Pool-Cluster und auf dem Cluster des PC^2 ergaben. Hierbei wurden auch 5 Knoten zur Berechnung genommen, die jeweils 2 CPUs als Dual-System besitzen.

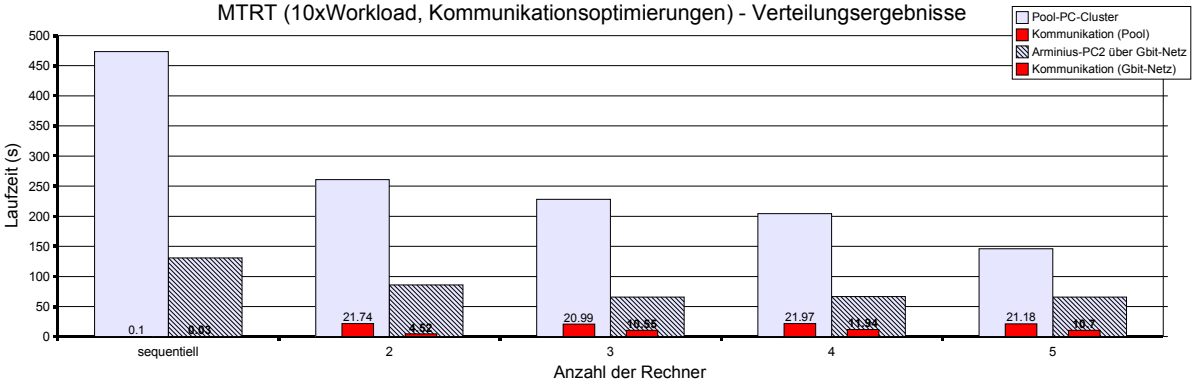

Abbildung 8.18.: Verteilungsergebnisse - Messwerte auf normalem PC-Cluster und auf Arminius-Cluster

Schon beim sequentiellen Ablauf des Raytracers erkennt man den großen Laufzeitgewinn (etwa 3.6fach schneller) durch mehr Rechenleistung, die ein Knoten mit 2 CPUs anbietet. Die verteilte Ausführung auf mehr als einem Knoten des Arminius-Clusters brachte dann wieder etwas mehr Performanz als die sequentielle auf einem Arminius-Knoten. Hier haben die geschickten Strategien, die Rechenleistung und die schnellere Verbindung ihren Beitrag. So konnte die Verteilung des Programms auf 3 Knoten etwa 23,7% mehr Performanz als auf 2 Knoten und 49.9% mehr als auf einem Knoten erzielen. Die Geschwindigkeitssteigerung durch Teilnahme von 4 und 5 Knoten änderte sich dann aber nicht mehr. Es ist sogar etwas langsamer. Der Grund ist der, dass das Raytracer-Programm insgesamt nur 5 Worker-Threads zum Rendern enthält. Die 3 Knoten des Arminius-Clusters besitzen bereits 6 CPUs, also optimal für die parallele Berechnung der Worker. Das heißt, dass 2 Worker-Threads auf einem Knoten parallel ablaufen können und die Kommunikation zwischen ihnen zum Schluß beim Schreiben der zugeteilten Kachel vom Canvas fast Null ist. Es entstand nur entfernter Kommunikationsaufwand dort, wo die Worker auf anderen Knoten ihre Daten verschicken.

Da die Rechnerarchitektur bei der Verteilungsberechnung (noch) nicht berücksichtigt wurde, wurde die Information nicht ausgenutzt, dass ein Knoten ein Dual-System mit 2 CPUs ist und 2 Worker-Threads auf einen Knoten

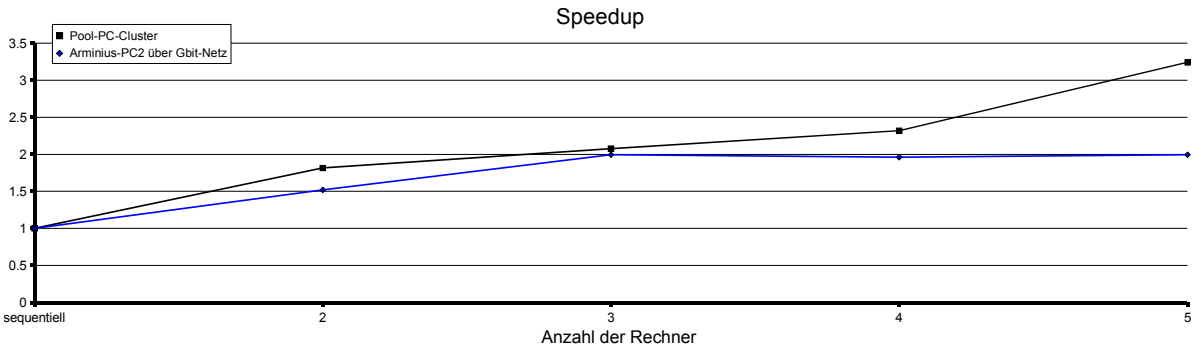

Abbildung 8.19.: Speedup-Werte

platziert werden sollten. Stattdessen wurden die 5 Worker einfach zyklisch auf die verfügbaren Knoten verteilt, was bei 4 und 5 Knoten zur Verschwendung der Rechnerleistung führte. Damit wurde aber noch mehr entfernter Kommunikationsaufwand verursacht. Das läßt sich deutlich an den anteiligen Kommunikationskosten der inneren Balken erkennen. Abbildung 8.19 zeigt die Speedup-Werte aller drei Untersuchungen.

Im Gegensatz zur Grocery-Simulation besitzt das *mtrt*-Programm reguläre Berechnungsstruktur, bei der die Worker ihre eigenen Berechnungsanteile ausführen und in einer bestimmten Phase miteinander kommunizieren. Die Parallelität derartiger Anwendungen skaliert sehr gut mit zunehmender Anzahl der beteiligten CPUs. Durch mehr Rechenleistung und eine schnellere Verbindung, welche der Arnimius-Cluster zur Verfügung stellt, konnten sowohl die Rechen- als auch Kommunikationszeiten bei der verteilten Ausführung verkürzt werden. Insgesamt erzielte man erheblich Geschwindigkeitssteigerung.

Speichernutzung

Bei der Evaluation des mtrt-Programms wurde erkannt, dass sehr viel Speicher zur Ausführung erforderlich ist. Das liegt auch daran, dass jeder Worker-Thread ein eigenes `Scene`-Objekt besitzt. Eine Szene erzeugt während des Renderns, also innerhalb der `mtrt.Scene.RenderScene()`-Methode, sehr viele lokale Objekte, z.B. Vector-Objekte (nicht vom Typ `java.util.Vector`, sondern eine Struktur zur Darstellung mathematischer Vektoren) oder Point-Objekte. Daher habe ich noch einen anderen Aspekt untersucht, und zwar den Speicherverbrauch bei solchen Programmen im Zu-

sammenhang mit der Verteilung. So wurde Profiling für das ursprüngliche mtrt-Programm mit derselben Eingabe durchgeführt. Die Anwendung wurde sequentiell auf einem Powerbook G4 mit 1,5 GB Hauptspeicher ausgeführt, wobei das Programm `jprofiler` [Gmb] (Version 4.1.2) für Profiling benutzt wurde.

Abbildung 8.20 zeigt die absteigend sortierte Anzahl der erzeugten Instanzen und den Speicherverbrauch dafür. In Abbildung 8.21 finden sich die Profiling-Daten darüber, in welchen aufgerufenen Methoden wie viele Instanzen insgesamt erzeugt werden und wieviel Speicher (in Bytes) jeweils gelegt wird. Die Methoden werden als Aufrufbaum zusammen mit ihrem anteiligen Speicherverbrauch und der Anzahl der allokierten Objekten dargestellt. Diese Daten gelten für das ganze Programm, also für alle Worker-Threads. Die Angabe zu erzeugten Instanzen bezieht sich auf die während des gesamten Programmablaufs erzeugten Objekte. Das bedeutet, dass man etwa 2,5GB Speicher für die Ausführung des mtrt-Programms benötigen würde, wenn der Garbage-Collector nicht aktiv wäre. Mit Garbage-Collector brauchte jeder Worker-Thread etwa 25MB Speicher ständig während seiner gesamten Ausführung. Mit den hier verwendeten Eingabedaten, also mit 5 Worker-Threads und Canvas-Größe 805×805, musste das Programm mit einer initialen Heap-Größe von 250MB starten. An den Ergebnissen in Abbildung 8.20 erkennt man, dass sehr viele `mtrt.Vector`-Objekte zum Rendern erzeugt wurden. Zusammen mit der Information aus Abbildung 8.20 kann man daraus schließen, dass 98.4% alle dieser Objekte innerhalb der `mtrt.Scene.RenderScene()`-Methode erzeugt wurden, insbesondere durch die Vielzahl der Aufrufe der `mtrt.Scene.Shade()`-Methode.

Aus dieser Speicheranalyse, insbesondere aus den Daten in Abbildungen 8.21 und 8.22 kann man erkennen, dass der Einsatz der geschickten Verteilungsstrategien in diesem Fall sehr nützlich war. Denn ohne die Lokalitätsoptimierung mittels der verwendeten ODFs, die die statischen Programmeigenschaften `isThreadLocal`, `isProxy`, `isPeer` und `isStagewiseImmutable` berechnen, müssten sehr viele Objekte entfernt erzeugt werden. Dies würde dann zu sehr viel zusätzlichen Kommunikationen führen. Durch diese statischen ODFs, insbesondere die zur `isThreadLocal`-Eigenschaft, wurden viele Objekte innerhalb der `mtrt.Scene`-Klasse lokal zusammen mit dem zugehörigen Thread erzeugt. Daher konnten alle Methodenaufrufe innerhalb des erzeugten `Scene`-Objekts direkt lokal ausgeführt werden. Zusätzlich dazu leistete die ODF $f_{\text{THREADWORKCOMM}}$ zur Berechnung von Workload und Kommunikationskosten zusammen mit der hybriden Analyse einen großen

Beitrag. Durch die Bekanntgabe der Laufzeitkonstanten zur Breite und Höhe des Canvas konnten Verteilungsentscheidungen für die in der geschachtelten Schleife zu Rendern entstehenden Objekte erneut berechnet werden, so dass diese zusammen mit dem sie erzeugenden Worker-Thread platziert werden. Das sind die meisten Objekte, die durch viele Aufrufe der `mtrt.Scene.Shade()`-Methode in der inneren Schleife entstanden.

In Abbildung 8.22 kann man erkennen, dass 96.8% der gesamten Laufzeit durch die Ausführung der `mtrt.Scene.Shade()`-Methode verursacht wurde. Diese Zeit ergabt sich aus den Ausführungszeiten verschiedener Methoden, die sich in den Unterbaumknoten in der Grafik finden lassen: z.B. 36.3% von `mtrt.Scene.FindLightBlock`, 29.3% von `mtrt.IntersectPt.FindNearestIsect`, 25.2% der `Shade()`-Methode selbst mit 18.0% Anteil von `mtrt.IntersectPt.FindNearestIsect`. Würden beispielsweise die `mtrt.ItersectPt`-Objekte vom `mtrt.Scene`-Objekt entfernt platziert, hätte dies sehr viel zusätzlichen Kommunikationsaufwand durch *remote* Aufrufe verursacht. Diese Profiling-Daten bestätigen also nochmals die Güte der getroffenen Verteilungsentscheidungen.

Aus der Untersuchung des Speicherbedarfs derartiger Anwendungen ergibt sich die Erkenntnis, dass durch die verteilte Ausführung nicht nur die Performanz verbessert werden kann, sondern automatisch auch mehr Speicher zur Verfügung gestellt wird. Die Skalierbarkeit kann hier in zwei Hinsichten genutzt werden. In bestimmten Anwendungsgebieten, wie z.B. Load-Balancierung bei verteilten Servern, wo die verteilten Programme endlos laufen, spielt der Speicherbedarf eine wichtigere Rolle als die Geschwindigkeitssteigerung.

Name	Instance count	Size
mtrt.Vector	64,712,257	1,553,094,168
int[ ]	16,544,920	399,673,208
mtrt.Point	5,091,992	122,207,808
float[ ]	3,244,246	77,861,936
<class>[ ]	3,186,876	147,635,128
mtrt.Color	3,033,787	72,810,888
mtrt.IntersectPt	2,847,968	91,134,976
mtrt.Ray	2,435,193	58,444,632
mtrt.OctNode	1,026,169	42,437,936
java.lang.String	180,112	4,322,688
mtrt.ObjNode	152,370	3,656,880
char[ ]	131,187	6,391,520
mtrt.Face	89,790	2,154,960
java.lang.FloatingDecimal	64,295	2,571,800
java.lang.Float	64,270	1,028,320
mtrt.CacheIntersectPt	7,060	282,400
mtrt.TriangleObj	6,975	502,200
java.lang.StringBuffer	1,191	28,584
java.util.HashMap	145	5,800
java.util.HashSet	145	2,320
short[ ]	142	11,160
byte[ ]	84	430,064
mtrt.PolygonObj	60	3,840
java.lang.Class	59	3,776
java.net.URL	58	3,248
java.net.URLClassLoader$1	58	928
java.lang.ClassNotFoundException	58	1,856
mtrt.Material	40	1,600
mtrt.MaterialNode	40	960
java.lang.ref.Finalizer	35	1,120
java.nio.HeapCharBuffer	33	1,584
java.io.FileInputStream	30	480
java.io.FileDescriptor	30	480
java.io.File	30	480
sun.misc.URLClassPath$7	29	696
java.security.CodeSource	29	696
java.security.PrivilegedActionException	29	928
mtrt.SphereObj	25	1,200
java.lang.ThreadLocal$ThreadLocalMap$Entry	10	320
mtrt.LightNode	10	240
mtrt.Light	10	240
java.security.AccessControlContext	8	192
mtrt.Scene	5	360
mtrt.Runner	5	520
mtrt.Camera	5	160
java.io.BufferedInputStream	5	160
java.io.DataInputStream	5	120
java.lang.ThreadLocal$ThreadLocalMap	5	120
mtrt.io.FileInputStream	5	160
java.io.ByteArrayInputStream	4	96
java.lang.reflect.Method	4	224
double[ ]	3	312
java.util.Hashtable$Entry	2	48
java.lang.Thread	2	144
mtrt.Canvas	1	24
java.util.Hashtable	1	40

Abbildung 8.20.: Profilingdaten für mtrt: Anzahl erzeugter Objekte mit dafür benötigtem Speicher

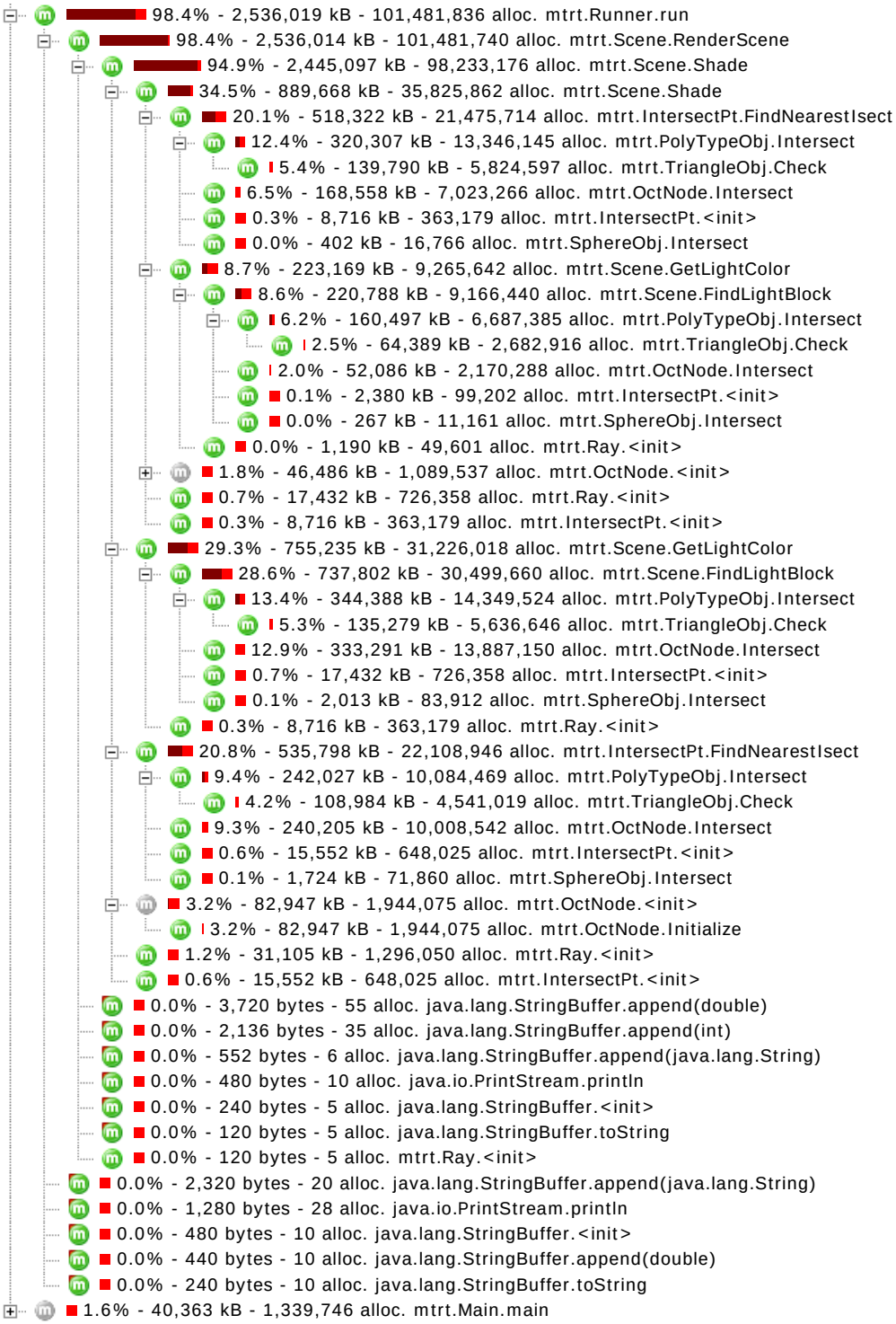

Abbildung 8.21.: Profilingdaten für mtrt: Aufrufbaum mit Anzahl erzeugter Objekte zusammen mit belegtem Speicher

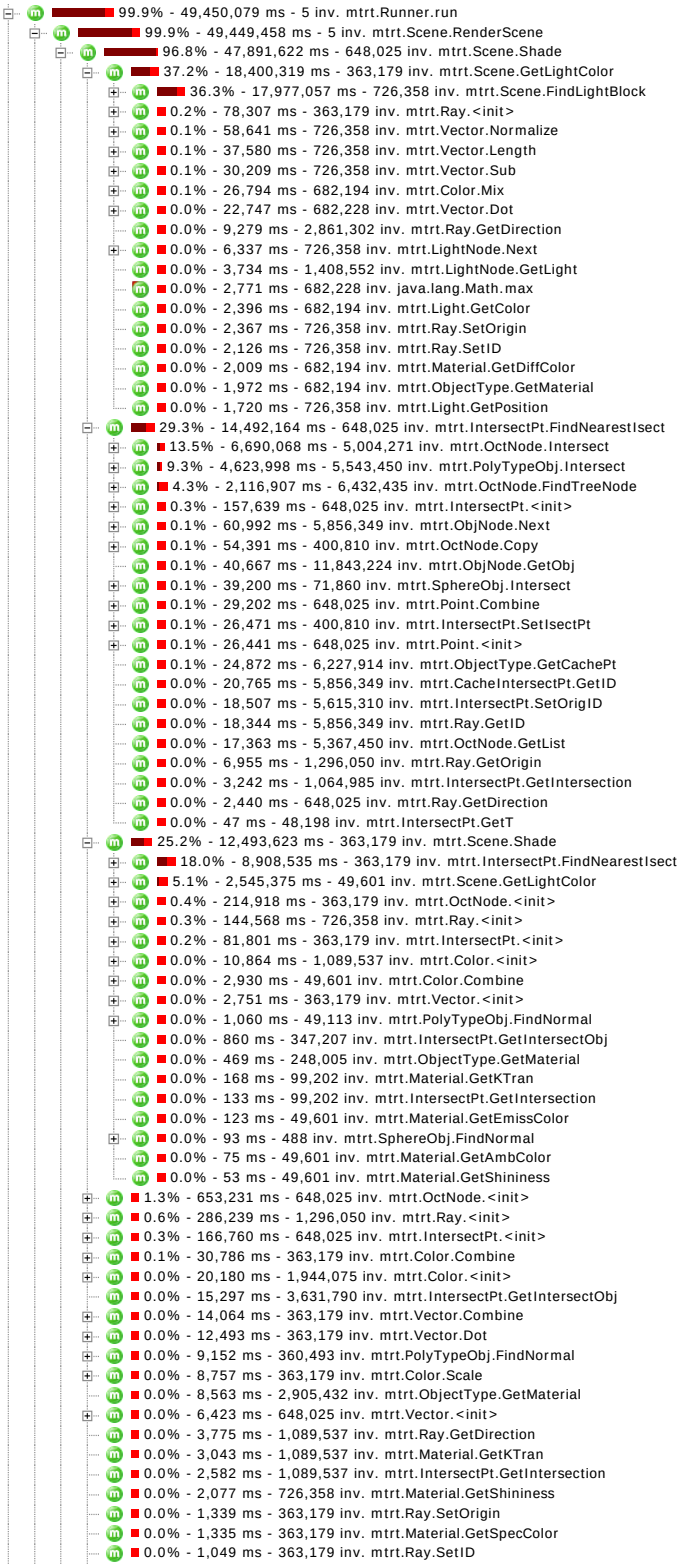

Abbildung 8.22.: Profilingdaten für den Aufrufbaum des mtrt-Programms

8.3.3. Das *Traveling Salesperson*-Problem (TSP)

Das TSP-Programm von IBM wurde mit dem Branch-and-Bound-Verfahren zur Berechnung der Wege implementiert. Außerdem wurde ein Load-Balancierungsmechanismus bei der Abarbeitung der Routen der Worker-Threads verwendet. Zur Lösung der Aufgabe werden 5 Worker-Threads verwendet. 15 Städte wurden für die Routen-Berechnung aufgenommen. Für das ganze Programm wurde ein `tsp.BestTour`-Objekt erzeugt, das die bisher beste gefundene Lösung präsentiert. Ein Worker bekommt eine Tour von der WorkQueue, ebenfalls ein Objekt vom Typ `tsp.Tour`, und fängt damit an die Route zu berechnen. Der hier verwendete Algorithmus zur Lösung wurde rekursiv implementiert: Wenn alle Städte einer Tour schon besucht wurden, wird die berechnete Länge der Tour mit der bisher besten gefundenen Lösung im `BestTour`-Objekt verglichen und ggf. die beste Tour aktualisiert. Sonst werden neue Städte in die Tour aufgenommen und die Berechnung wird für jeden der nächsten Kandidaten rekursiv durchgeführt.

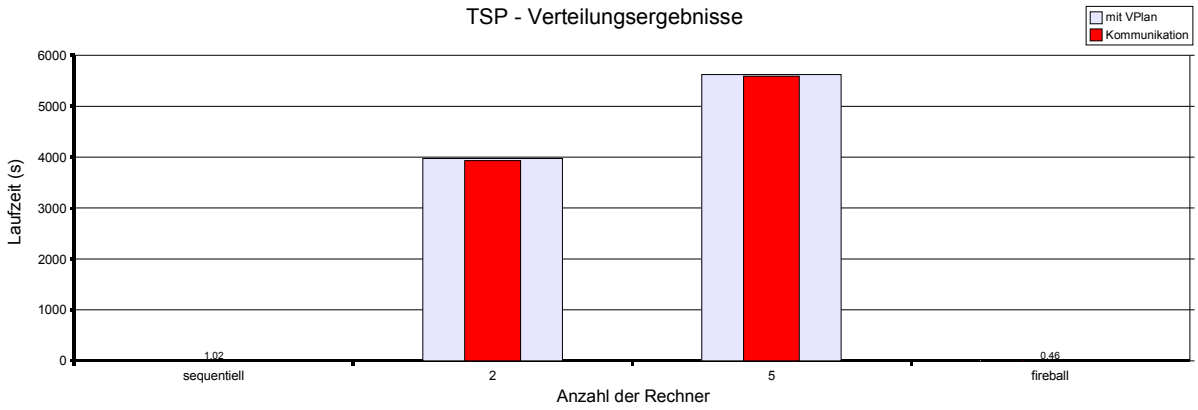

Abbildung 8.23.: Verteilungsergebnis für TSP

Aufgrund dieser Implementierungsstruktur wurde die zum eigentlichen Lösen des Problems zuständige `Worker.solve(Tour t)`-Methode bei jedem Worker-Thread in der sequentiellen Ausführung durchschnittlich 137786 mal, auf der fireball-Maschine 121116 mal und im verteilten Fall (auf 2 und 5 Rechnern) 254143 mal rekursiv ausgeführt. Dies entspricht also der Rekursionstiefe. Für die verteilte Ausführung wurden die gleichen Strategien angewandt, die bei der Evaluation des mtrt-Programms zum Einsatz kamen. Hiermit wurden die in dieser Methode erzeugten `Tour`-Objekte repliziert. Trotz der Anwendung der geschickten Strategien, welche eine positive Auswirkung auf

die Laufzeit bei anderen Programmen aufzeigen, wirkt die verteilte Ausführung in diesem Fall sehr schlecht, wie die Messergebnisse in Abbildung 8.23 zeigen. Zum Vergleich mit der Ausführungsdauer im sequentiellen Fall von 1.02s sowie auf der fireball-Maschine von 0.46s ist die verteilte Ausführung auf 2 und 5 Rechnern viel zu langsam. Wie man wohl am Balkendiagramm erkennt, wird diese Geschwindigkeitsverlangsamung durch die entstehende entfernte Kommunikation verursacht. Das sind entfernte Methodenaufrufe, die die fünf Worker auf das `tsp.BestTour`-Objekt durchführten. Diese dominieren fast die gesamte Laufzeit des Programms.

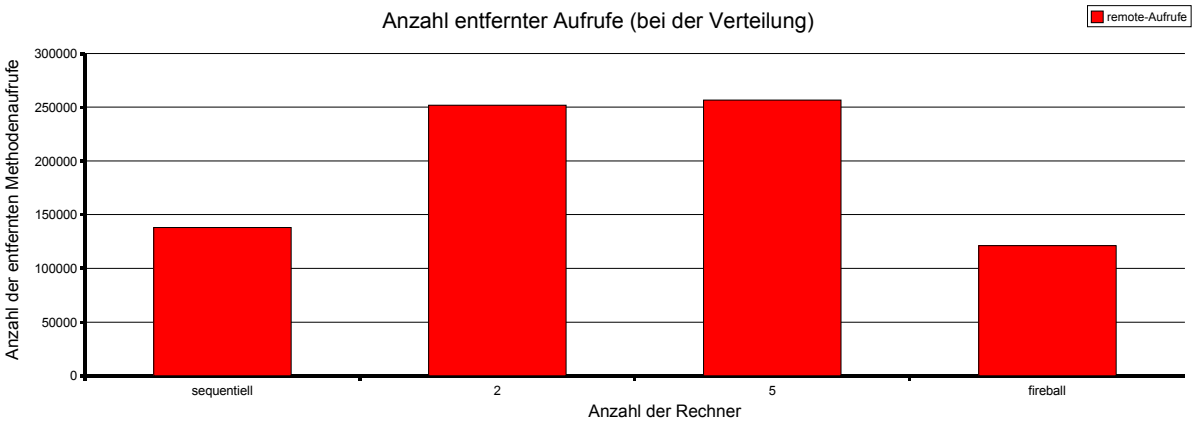

Abbildung 8.24.: Anzahl der entfernten Methodenaufrufe jedes Worker-Threads

Abbildung 8.24 zeigt uns nochmals die Anzahl der entfernten Methodenaufrufe bei jedem Worker-Thread. Die Ausführungsdauer der Methodenaufrufe im sequentiellen Fall sowie in der Mehrprozessor-Maschine kostet fast keine Zeit, während ein *remote*-Aufruf im verteilten Fall sehr teuer ist. Daher ist es sehr ungünstig, dieses Programm mit diesem Konzept verteilt auszuführen. Mit diesem Beispiel wird nochmals bestätigt, dass viel entfernte Kommunikation die gesamte Laufzeit eines verteilten Programms sehr beeinträchtigt.

Es gibt andere Systeme wie cJVM von IBM [AFT99], [AFT+00] und Hyperion [A+01], die eine eigene JVM implementieren (siehe Abschnitt 4.1.5) und sehr schnelle Kommunikation unterstützen (nicht über RMI). Zur Verteilung der Objekte wird hierbei neben der Technik der Lastbalancierung ebenfalls auch die Last der Rechner-Knoten zur Laufzeit mit berücksichtigt. So konnte für das TSP-Problem mit 14 Städten bei 4 Threads unter cJVM ein Speedup von 3 erreicht werden. Der verwendete Cluster ist eine Gruppe von *IBM Intellistations*, die untereinander via Myrinet verbunden sind.

8.4. Die Erkenntnisse

Durch die empirische Untersuchung von drei Programmen aus drei verschiedenen Kategorien kann man die Erkenntnisse gewinnen, dass unter dem Einsatz der geschickten Verteilungsstrategien viel Performanz erzielt werden kann. Dies wurde durch die Messergebnisse bei der Grocery-Simulation und dem mtrt-Programm bestätigt. Der Vergleich zwischen der zyklischen Verteilung und einem Verteilungsplan mit intelligenten Strategien zeigt ganz deutlich den entscheidenden Beitrag der automatisch eingehenden Analysetechniken. Neben den verwendeten statischen Analysetechniken zur Erkennung von Lokalitätseigenschaften der entstehenden Objekte können mittels der *stagewise-immutable* Eigenschaft viele zusätzliche Objekte repliziert werden, was zur Performanzsteigerung beiträgt. Für diesen Zweck wurde auch die neuartige statische *stagewise-immutable* Analyse entworfen. Dies ist ein sehr aufwändiges Analyseverfahren, wird aber vor der Laufzeit angewandt und hat deshalb keine negative Wirkung auf die Laufzeit des Programms. Zur Verbesserung der ungenauen statischen Abschätzung der Workload sowie der Kommunikation wurde der hybride Ansatz zur Kombination von Laufzeitdaten mit statischen Analyseergebnissen in einem Beispiel erfolgreich evaluiert.

Außerdem wurde gezeigt, wie wichtig die Reihenfolge zur Anwendung der verschiedenen Strategien innerhalb eines Verteilungsplans ist. So konnten Programmeigenschaften berechnet werden, die durch Anwendung vorheriger Strategien nicht entdeckt werden konnten. Die automatische Verteilung mit geschickten Strategien funktioniert im Allgemeinen nur gut für mehrsträngige Anwendungsprogramme, die insgesamt sehr viel aufwändige Berechnungen durchführen, aber wenig miteinander kommunizieren. Beispiele dafür sind diskrete ereignisbasierte Simulationen und Programme mit regulären Strukturen wie das mtrt-Programm, wobei innerhalb eines Threads auch viel lokale Datenobjekte zur Eigenberechnung erzeugt werden können. Der nicht zu vernachlässigende Aspekt dabei ist der Speicherverbrauch: Wenn ein Rechner sehr wenig Speicher zur Verfügung hat, können Anwendungen mit sehr hohem Speicherbedarf darauf nicht ausgeführt werden oder sie laufen wegen des notwendigen Swappings sehr langsam. Dieses Problem kann gut mit der automatischen Verteilung auf mehreren Rechnern gelöst werden. Ein weiterer Vorteil der automatischen Verteilung ist auch die Skalierbarkeit, die bei Mehrprozessor-Maschinen sehr beschränkt ist.

Der Einsatz des *JScatter*-Systems und die verteilte Ausführung der untersuchten Programme auf dem normalen PC-Cluster und auf dem Arminius-Cluster des PC^2 zeigten, wie einfach es ist, das System auf unterschiedlichen Cluster-Architekturen zu benutzen. Die Messergebnisse auf dem Cluster des PC^2 bestätigen, dass nicht nur durch die geschickt angewandten Strategien, sondern auch durch hohe Rechenleistung mehr Geschwindigkeit erreicht werden kann. Die Messergebnisse zeigten, dass die verteilte Ausführung irregulärer Anwendungen wie der Grocery-Simulation auf dem Arminius-Cluster keine sehr gute Performanz erreichte. Das liegt daran, dass die Charakteristiken solcher diskreten ereignisbasierten Simulationen nicht typisch für *High-Performance-Computing* sind. Im Gegensatz dazu konnte sehr viel Performanz beim *mtrt*-Raytracer auf dem Cluster des PC^2 erzielt werden: Mit den geschickten Verteilungsstrategien, mehr Rechenleistung und schnellerer Verbindung konnte dieses Programm bis auf einen Faktor 3 schneller verteilt ausgeführt werden, als die verteilte Ausführung auf einem normalen Pool-Cluster.

Speziell beim TSP-Programm und allgemein bei Programmen mit viel Kommunikationen lohnt sich die verteilte Ausführung nur, wenn entfernte Kommunikationskosten sehr gering sind. Für einen Fall wie bei TSP kann mein Konzept wie folgt erweitert werden: Zuerst ist mittels Programmanalyse herauszufinden, ob im Eingabeprogramm die Kommunikationsanteile erheblich größer als Rechenanteile sind. Für die Programmanalyse ist es technisch möglich, solche Abschätzungen vor der Laufzeit durchzuführen, insbesondere bei rekursiven Strukturen schärfere Schranken zur Abschätzung zu verwenden. Wenn dies der Fall ist, dann wird entschieden, das Programm oder Teile davon besser nicht verteilt auf den zur Verfügung stehenden JVMs auszuführen, sondern nur in einer JVM lokal laufen zu lassen.

Zur Evaluationszeit wurden die Ergebnisse einiger Analysen, wie die *stagewise-immutable* Analyse und die hybride Analyse, noch nicht automatisch direkt zur Berechnung der Verteilungsentscheidung integriert, sondern manuell eingebunden. Dies erfordert noch einiges an Implementierung im JScatter-System. Trotzdem wurde das Konzept zur automatischen Verteilung, insbesondere mit der *stagewise-immutable* Eigenschaft und der hybriden Analyse, durch die empirische Untersuchung erfolgreich evaluiert.

9. Zusammenfassung

Inhalt

In dieser Arbeit habe ich ein Konzept zur automatischen Verteilung mehrsträngig formulierter Java-Anwendungen repräsentiert. Über einen Verteilungsplan können alle strategischen Entscheidungen über Transformation und Verteilung der Objekte getroffen werden. Mit automatisch eingehenden Programmanalysen können die für die Verteilung relevanten Programmeigenschaften bestimmt werden. Sie dienen als Grundlage der Verteilungsberechnung vor der Laufzeit und zur Laufzeit. Insbesondere zählen hierzu die *stagewise-immutable* Eigenschaft zur Objektreplikation und die Anwendung der hybriden Analyse. Das ist eine Kombination statischer Analyseergebnisse mit dynamischer Analyse. Sie wird zur Verbesserung der Workload-Eigenschaft und somit der Verteilungsentscheidung zukünftig zu erzeugender Objekte verwendet. Im Folgenden werden die Kernpunkte des gesamten Konzepts zusammengefasst.

9.1. Wichtige Aspekte in Kürze

Verteilungsplan-Konzept Ein Verteilungsplan liefert vor der Laufzeit die Entscheidung über die Transformation der Erzeugungsstellen und berechnet für als *remote* gekennzeichnete Objektrepräsentanten initiale Platzierungsentscheidungen. Zur Laufzeit wird mittels eines Planes die Entscheidung zur Verteilung unmittelbar vor der Erzeugung getroffen. Mit diesem Konzept können verschiedene in einem Plan verwendete Strategien zur geschickten Verteilung der Objekte unabhängig von der Anwendung formuliert werden. So werden allgemeine Programmeigenschaften mittels der Programmanalysen ermittelt, die innerhalb spezieller Objektverteilungsfunktionen (ODFs) zur Verteilungsberechnung verwendet werden. Bestimmte ODFs zusammen bilden eine Strategie. Mit diesem Modell hat man den Vorteil, neue ODFs formulieren zu können, verschiedene Varianten der Strategien anwenden oder neue entwickeln zu können und somit unterschiedliche Pläne zur Verteilung eines Programms einsetzen zu können. Durch die Modellierung der Programmeigenschaften und deren Verwendung in den ODFs wird die eigentliche Programmanalyse von der Berechnung der Verteilungsfunktion getrennt. So können durch Einsatz spezieller Analysetechniken Analyseergebnisse zur Verteilung verbessert werden.

Die *stagewise-immutable* Eigenschaft und hybride Analyse Zur Objektreplikation habe ich eine neuartige Analyse entwickelt, die die *stagewise-*

immutable Eigenschaft der Objekte vor der Laufzeit erkennt. *Pure-immutable* Objekte können direkt nach ihrer Initialisierung repliziert werden. *Postponed-immutable* Objekte werden nach der Initialisierungsphase noch geschrieben, aber ab einem späteren Zeitpunkt zur Laufzeit nur noch gelesen. Sie können dann auch repliziert werden. Nach der Replikation finden alle solche Zugriffe dann lokal statt, so dass viel Kommunikationsaufwand gespart werden kann, wie dies bei der empirischen Untersuchung gezeigt wurde.

Zur Verbesserung der unpräzisen Abschätzung der Workload sowie der Kommunikationskosten vor der Laufzeit wurde der Ansatz zur hybriden Analyse erprobt. Hierbei wurden die Laufzeitdaten mittels dynamischer Analyse mit den statischen Analyseergebnissen kombiniert. Dadurch konnten die tatsächlichen verwendeten Grenzwerte der äquivalenten Schleifen berechnet werden, was zur genaueren Workload-Berechnung führt. Somit konnten Verteilungsentscheidungen verbessert werden, die dann zur Platzierung zukünftig zu erzeugender Objekte benutzt werden. Der Vorteil des hybriden Ansatzes gegenüber nur dynamischer Analyse liegt darin, dass vor der Laufzeit die statische Struktur, insbesondere die interprozedurale Information des Programms zur Verfügung steht. Der Beschaffungsaufwand statischer Ergebnisse hat keinen Einfluss auf die Laufzeit des Programms. Die dynamische Analyse wird gezielt eingesetzt, um die noch fehlenden Laufzeitdaten zu ermitteln.

Ergebnisse Die vorgestellten Konzepte wurden prototypisch im *JScatter*-System sowie die Analysen in der Analyseumgebung PAULI realisiert. Sie wurden dann auf 8 Anwendungsprogramme und Benchmarks angewandt, um die *stagewise-immutable* Eigenschaft zu evaluieren. Das Konzept zur automatischen Verteilung wurde auf 3 Programmen aus drei verschiedenen Kategorien empirisch erprobt. Als komplettes (vollständiges) Programm konnten sehr viele Erzeugungsstellen mit *stagewise-immutable*, insbesondere der *pure-immutable* Eigenschaft, entdeckt werden. So besitzen etwa 44% der `NEW`-Stellen bei allen 8 untersuchten Anwendungen die *pure-immutable* und die *postponed-immutable* Eigenschaft. Dadurch können bis zu 15524 (etwa 6,97% aller statisch gezählten) Lesezugriffe nach der Replikation lokal werden, so dass erheblich viel entfernter Kommunikationsaufwand reduziert werden kann.

Bei der Untersuchung der Verteilungsgüte konnte gezeigt werden, dass viel Performanz unter dem Einsatz der geschickten Verteilungsstrategien erzielt

wurde. Bei der verteilten Ausführung auf 9 Rechnern einer diskreten ereignisbasierten Simulation mit Replikation der *stagewise-immutable* Objekte konnte ein Speedup von etwa 3.8 erreicht werden. Außerdem wurde dabei auch erkannt, dass die automatische Verteilung mit intelligenten Strategien im Allgemeinen nur gut für Anwendungen funktioniert, die insgesamt viel aufwändige Berechnungen durchführen, aber wenig Kommunikation verursachen. Dies wurde deutlich durch Messungen am *mtrt*-Programm gezeigt: Durch den künstlich eingefügten Workload sowie Kommunikationsoptimierung konnte ein Geschwindigkeitsgewinn bei der verteilten Ausführung erzielt werden. Messergebnisse der TSP-Anwendung, die über 98% der gesamten Laufzeit mit Kommunikation verbringt, zeigten auch deutlich, dass man durch die verteilte Ausführung derartiger Anwendungsprogramme mit dem *JScatter*-System auf PC-Cluster mit Standard-Ethernet-Vernetzung keine gute Performanz erreichen kann.

Durch viel Rechenleistung und schnellere Kommunikation des Arminius-Clusters im PC^2 konnten die Programme mit geschickten Strategien bis auf einen Faktor 3 (über Ethernet-Gbit-Verbindung) schneller verteilt ausgeführt werden, als die verteilte Ausführung auf einem normalen Pool-Cluster.

Zusammenfassung Mit diesem Konzept kann ein mehrsträngiges Java-Programm benutzertransparent automatisch verteilt werden. Alle notwendigen und aufwändigen Schritte, die ein Programmierer manuell durchführen muss, um ein sequentielles multithreaded Java-Programm in eine verteilte Version mit RMI umzuschreiben und diese auf mehreren Rechnern verteilt auszuführen, werden automatisiert. Durch die geschickten Verteilungsstrategien mit automatisch eingehender Programmanalyse werden die Objekte automatisch auf die beteiligten JVMs platziert, so dass der Benutzer sich nicht darum kümmern muss. Hierbei erhält man eine gewünschte Verteilung der Programmobjekte, durch die für bestimmte Anwendungsgebiete eine Geschwindigkeitssteigerung mittels der verteilten Ausführung auf mehreren Rechnern erzielt wird. Da das *JScatter*-Systems vollständig in Java realisiert ist, kann man es plattformunabhängig verwenden. Dies wurde gezeigt, indem die Evaluation nicht nur auf einem Cluster von Standard-PCs, sondern auch auf dem Arminius-Cluster des Paderborn Center for Parallel Computing (PC^2) durchgeführt wurde. Außerdem stehen einem durch verteilte Ausführung auf einem Cluster auch automatisch mehr Ressourcen wie z.B. Speicher zur Verfügung.

Dieser Ansatz kann allerdings nicht mit denjenigen konkurrieren, bei denen ein Experte eine spezielle Lösung für den im Programm verwendeten Algorithmus programmiert und danach die Objekte verteilt. Hierbei kann viel mehr Performanz erreicht werden. Der Vorteil meines Ansatzes ist aber, dass alles vollständig automatisch durchgeführt wird. Außerdem erhält man automatisch die Skalierbarkeit, die Mehrprozessor-Maschinen nicht zur Verfügung stellen können. Der Benutzer braucht für die verteilte Ausführung nur noch die Rechner zu bestimmen, die sich an der parallelen Lösung des Problems beteiligen. Außerdem kann bei bestimmten Anwendungen (TSP) keine gute Performanz mit *JScatter*-System erreicht werden, wie mit anderen Systemen (*cJVM* von IBM Research Laboratory in Haifa [AFT99], [AFT+00]). Das liegt daran, dass bei *JScatter* die verteilte Laufzeitumgebung aus den Standard-JVMs entsteht und RMI als Kommunikationsplattform verwendet wird. Alle Komponenten des *JScatter*-Systems sind also in Java implementiert. *cJVM* ist dagegen eine speziell implementierte JVM, die einen Cluster virtualisiert, die Load-Balancierungstechnik zur Objektverteilung verwendet und ein anderes Objekt- sowie Speichermodell benutzt. Außerdem kommunizieren die Rechner miteinander über eine sehr schnelle Verbindung mit einer Latenz im Bereich von Microsekunden.

9.2. Ausblick

Neben den Vorschlägen zur Erweiterung des Ansatzes zur hybriden Analyse, die im Abschnitt 6.3.3 beschrieben wurden, kann ein weiteres Konzept dazu untersucht werden:

Verfeinerung statischer Analyseergebnisse durch dynamische Analyse Mittels der Objektgraph-Analyse [Spi99b] kann die Laufzeitstruktur eines Programms statisch modelliert werden. Die Knoten im Graphen repräsentieren die zur Laufzeit entstehenden Objekte und die Kanten die Beziehungen zwischen ihnen. Die interessanten Kanten im Objektgraphen sind diejenigen, bei denen jeder der Endknoten mehrere Laufzeitobjekte darstellt, wie das in Abbildung 9.1 gezeigt wird.

Es gibt einen gemeinsamen Knoten für die Instanzen der Klasse `MyThread` und einen Knoten für die `Vector`-Objekte. Diese beiden Knoten werden aber durch eine einzige Kante miteinander verbunden. Man kann statisch nicht bestimmen, wie die Beziehung zwischen den $2n$ Objekten genau aussieht. Ziel

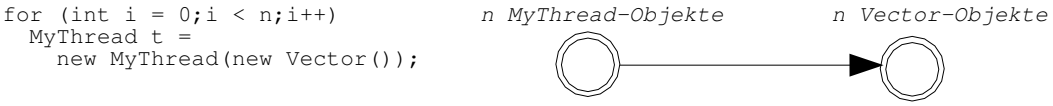

Abbildung 9.1.: n:n-Beziehung zwischen Laufzeitobjekten im Objektgraphen

ist es, genau die 1:1-Beziehung für diese *n* Paare von Laufzeitobjekten festzustellen. Dadurch erhält man die Information, dass zur Laufzeit jedes `Vector`-Objekt genau einem *MyThread*-Objekt zugeordnet wird. Somit ist keine Synchronisation beim Zugriff auf die `Vector`-Objekte nötig.

Als Lösung kann eine statische Analyse entwickelt werden, die auf einem Objektgraphen die Knotenpaare nach obigem Muster erkennt und die betroffenen Programmstellen markiert. An solchen markierten Stellen kann eine dynamische Analyse das Programmverhalten beobachten und eliminiert überflüssige Thread-Synchronisation, wenn 1:1-Beziehungen zwischen den Laufzeitobjekten vorliegen.

Beseitigung unnötiger Synchronisation von *stagewise-immutable* Objekten Eine weitere Optimierungsanwendung für Objekte mit *stagewise-immutable* Eigenschaft ist das Entfernen unnötiger Thread-Synchronisation. Dies kann man für *pure-immutable* Objekte direkt nach ihrer Initialisierung anwenden. Für *postponed-immutable* Objekte kann die Beseitigung der Synchronisation an geeigneten Stellen nach einem guten Schnitt durchgeführt werden.

Analyse zur Erkennung anwendungsspezifischer Muster Zu der als `Peer` zusammengefassten Programmeigenschaft gehören verschiedene elementare Eigenschaften, die jeweils ein anwendungsspezifisches Muster darstellen. Einige Beispiele dafür sind Muster zur Benutzerinteraktion mittels Eingabe und Ausgabe, Muster zur Benutzung von speziellen Hardware-Ressourcen wie Hardware- oder Grafik-Beschleuniger etc. So können spezielle Analysetechniken dafür gezielt eingesetzt werden, um solche Muster im Programm vor der Laufzeit zu entdecken. Bei der automatischen Verteilung können Objekte, die solche Muster enthalten, nur dann dort erzeugt werden, wo die benötigten Ressourcen benutzt werden.

Im Prinzip stellt die hybride Analyse eine ideale Ausgewogenheit der statischen und dynamischen Analyse dar. Damit können die Vorteile der beiden Welten genutzt werden, um die Eigenschaften des Programms so genau wie möglich zu berechnen. Außerdem kann durch Hinzunahme externer Information wie z.B. der aktuellen CPU-Load in die Verteilungsentscheidung mehr Performanz durch eine verteilte Ausführung erzielt werden.

Anhang

A. Verzeichnisse

Literaturverzeichnis

[A+01] G. Antoniu et al. The Hyperion System: Compiling Multithreaded Java Bytecode for Distributed Execution. *Parallel Computing*, 27(10):1279–1297, September 2001.

[AB] G. Antoniu and L. Bougé. Implementing Multithreaded Protocols for Release Consistency on Top of the Generic DSM-PM Platform. *Lecture Notes in Computer Science*, 2326.

[AB01] G. Antoniu and L. Bouge. DSM-PM2: A Portable Implementation Platform for Multithreaded DSM Consistency Protocols. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01)*, pages 104–104, Los Alamitos, CA, April 23–27 2001. IEEE Computer Society.

[ABH+a] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Compiling Multithreaded Java Bytecode for Distributed Execution (Distinguished Paper). *Lecture Notes in Computer Science*, 1900.

[ABH+b] G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. Implementing Java Consistency Using a Generic, Multithreaded DSM Runtime System. *Lecture Notes in Computer Science*, 1800.

[ACG00] I. Attali, D. Caromel, and R. Guider. A Step Toward Automatic Distribution of Java Programs, 2000. FMOODS 2000: 141-162.

[AFT99] Y. Aridor, M. Factor, and A. Teperman. cJVM: A Single System Image of a JVM on a Cluster. In *International Conference on Parallel Processing*, pages 4–11, 1999.

[AFT+00] Y. Aridor, M. Factor, A. Teperman, T. Eilam, and A. Schuster. A High Performance Cluster JVM Presenting a pure Single System Image. In *Java Grande*, pages 168–177, 2000.

[AH01] G. Antoniu and P. Hatcher. Remote Object Detection in Cluster-Based Java. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS-01)*, pages 108–108, Los Alamitos, CA, April 23–27 2001. IEEE Computer Society.

[Ant] INRIA Sophia Antipolis. ProActive PDC - Parallel, Distributed and Concurrent Computing in Java with Mobility. `http://www-sop.inria.fr/sloop/javall`.

[Ass] InfiniBand Trade Association. InfiniBand HCA. `http://www.infinibandta.org/home`.

[B. 00] B. Joy and G. Steele and J. Gosling and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition edition, 2000.

[BBH+96] Henri E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Orca: a Portable User-Level Shared Object System. Technical Report IR-408, Vrije University, Amsterdam, June 1996.

[BBH+98] H. E. Bal, R. Bhoedjang, R. Hofman, C. Jacobs, K. Langendoen, T. Rühl, and M. F. Kaashoek. Performance Evaluation of the Orca Shared-Object System. *ACM Transactions on Computer Systems*, 16(1):1–40, February 1998.

[BCCH95] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-Insensitive Interprocedural Alias Analysis in the Pressence of Pointers. In David Gelertner, Alexandru Nicolau, and David Padua, editors, *Lecture Notes in Computer Science, 892*. Springer-Verlag, 1995.

[BCFK] M. Baker, B. Carpenter, G. Fox, and S. H. Koo. mpiJava: An Object-Oriented Java Interface to MPI. *Lecture Notes in Computer Science*, 1586.

[BCP96] B. Blount, S. Chatterjee, and M. Philippsen. Irregular Parallel Algorithms in Java. Technical Report IR-408, Vrije University, Amsterdam, June 1996.

[BG97] A. J. C. Bik and D. B. Gannon. Automatically Exploiting implicit Parallelism in Java. *j-CPE*, 9(6):579–619, June 1997.

[BG98] A. J. C. Bik and D. B. Gannon. A Prototype Bytecode Parallelization Tool. *Concurrency: Practice and Experience*, 10(11–13):879–885, September 1998. Special Issue: Java for High-performance Network Computing.

[BH99] J. Bogda and U. Hölzle. Removing unnecessary Synchronization in Java. *ACM SIGPLAN Notices*, 34(10):35–46, 1999.

[BK93] H. E. Bal and M. F. Kaashoek. Object Distribution in Orca using Compile-time and Run-time Techniques. In Andreas Paepcke, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 28, pages 162–177, New York, NY, 1993. ACM Press.

[BLS97] G. Brose, K. P. Löhr, and A. Spiegel. Java Does not Distribute. Technical Report B-97-07, Freie Universität Berlin, October 1997.

[Bor02] J. Borrmann. Replikation statischer Konstanten in JavaParty. Technical report, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe. Studienarbeit, February 2002.

[Cen] The Edinburgh Parallel Computing Centre. The Java Grande Forum Multi-threaded Benchmarks. `http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/`.

[CGS+99] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape Analysis for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[CJR92] R. Clark, D.E. Jensen, and F.D. Reynolds. An Architectural Overview of the Alpha Real-Time Distributed Kernel. In *Proceedings of the USENIX Workshop on Microkernel and Other Kernel Architectures*, April 1992.

[CKV98] D. Caromel, W. Klauser, and J. Vayssière. Towards Seamless Computing and Metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, September 1998. Special Issue: Java for High-performance Network Computing.

[Cor] The Standard Performance Evaluation Corporation. SPEC JVM98. `http://www.spec.org/osg/jvm98`.

[CZF+98] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. *HPJava*: Data Parallel Extensions to Java. *Concurrency: Practice and Experience*, 10(11–13):873–877, September 1998. Special Issue: Java for High-performance Network Computing.

[Daha] M. Dahm. BCEL: The Byte Code Engineering Library. `http://bcel.sourceforge.net`.

[Dahb] M. Dahm. Doorastha A Step towards Distribution Transparency. `http://citeseer.nj.nec.com/dahm00doorastha.html`.

[Dah00] M. Dahm. The Doorastha System. Technical report B-1-2000, Freie Universität Berlin, 2000.

[Dow98] T. B. Downing. *Java RMI: Remote Method Invocation*. IDG Books, San Mateo, CA, USA, January 1998.

[DPW01] T. Dowling, J. Power, and J. Waldron. Relating Static and Dynamic Measurements for the Java Virtual Machine Instruction Set. In *Symposium on Mathematical Methods and Computational Techniques in Electronic Engineering*, Athens, Greece, December 29-31 2001.

[DS84] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, 1984.

[FFF+02] M. Faust, M. Freund, J. Friedrich, M. Stiller, M. Weißenborn, and H. Wessels. Abschlussbericht der Projektgruppe Verteilung von parallelen Java-Anwendungen. Abschlussbericht, November 2002.

[FHA99] E. F., S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, June 1999.

[Gmb] EJ-Technologies GmbH. JProfiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[Hau98] B. Haumacher. Lokalitätsoptimierung durch statische Typanalyse in JavaParty. Master's thesis, Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe, 1998.

[HBCC99] M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural Pointer Alias Analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.

[HCU91] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches, 1991.

[HD04] C. Herder and J. J. Dujmović. Workload Characterization Using Metrics Based on Instruction Grouping. *International Journal of Computer and Information Science (IJCIS)*, Vol. 5, No. 1, Mar 2004.

[HL98] D. Hagimont and D. Louvegnies. Javanaise: Distributed Shared Objects for Internet Cooperative Applications. In *Middleware'98*, The Lake District, England, 1998.

[HMP] B. Haumacher, T. Moschny, and M. Philippsen. KaRMI: Efficient RMI for Java. `http://www.ipd.uka.de/JavaParty/KaRMI/index.html`.

[HMRT03] B. Haumacher, T. Moschny, J. Reuter, and W. F. Tichy. Transparent Distributed Threads for Java. In *Proceedings of 5th International Workshop on Java For Parallel and Distributed Computing (in conjunction with JPDPS 2003)*, April 2003.

[Hoe] J. Hoenicke. JODE - Java Optimize and Decompile Environment. `http://jode.sourceforge.net/`.

[HP99] B. Haumacher and M. Philippsen. More Efficient Object Serialization. *Parallel and Distributed Processing*, (number 1586 in Lecture Notes in Computer Science):718–732, Puerto Rico, April 12 1999.

[HP01] B. Haumacher and M. Philippsen. Exploiting Object Locality in JavaParty, a Distributed Computing Environment for Workstation Clusters. In *9th Workshop on Compiler for Parallel Computers (CPC, Edingburgh, Scotland,UK 2001)*, 27-29 Juni 2001.

[HRP] B. Haumacher, J. Reuter, and M. Philippsen. JavaParty: A distributed Companion to Java. `http://wwwipd.ira.uka.de/JavaParty`.

[Höl94] U. Hölzle. Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming, 1994.

[KLK02] K. Klohs, D. K. Le, and U. Kastens. Automatic Distribution of Multithreaded Java Programs using Distribution Plans. Technischer Bericht, Reihe Informatik tr-ri-02-234, Universität Paderborn - Institut für Informatik, 2002.

[Klo02] K. Klohs. Analyse nebenläufiger Java-Programme zur Berechnung von Verteilungsstrategien. Master's thesis, Fachbereich 17, Mathematik-Informatik, Universität Paderborn, 2002.

[Le00] D. K. Le. Referenzziel-Analyse für Java-Bibliotheken und Wiederverwendung ihrer Ergebnisse. Master's thesis, Universität Paderborn, 2000.

[LP97] P. Launay and J.-L. Pazat. A Framework for Parallel Programming in Java. Technical Report 1154, IRISA, Intitut de Recherche en Informatique et Systèmes Aléatoires, Dezember 1997.

[LP98] P. Launay and J.-L Pazat. Generation of Distributed Parallel Java Programs. *Lecture Notes in Computer Science*, 1470:729–32, September 1998.

[Mic] Sun Microsystems. Java RMI. `http://java.sun.com/products/jdk/rmi/`.

[Mic98] Sun Microsystems. Java Remote Method Invocation – Distributed Computing for Java. White Paper, March 1998.

[MKB00] J. Maassen, T. Kielmann, and H. E. Bal. Efficient Replicated Method Invocation in Java. In *Java Grande*, pages 88–96, 2000.

[MKB01] J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 13(8–9):681–712, 2001.

[MPA99] Sun Microsystems and CA P. Alto. *Java Remote Method Invocation: Specification*. December 1999.

[Muc97] S. S. Muchnick. *Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.

[MWL00] M. J. M. Ma, C.-L. Wang, and F. C. M. Lau. JESSICA: Java-Enabled Single-System-Image Computing Architecture. *Journal of Parallel and Distributed Computing*, 60(10):1194–1222, 2000.

[MZ96] T. Mowbray and R. Zahavi. *The Essential CORBA: Systems Intergration Using Distributed Objects*. Wiley, 1996.

[NS96] N. Nagaratnam and A. Srinivasan. Remote objects in Java, 1996. In IASTED International Conference on Networks, 1996.

[OW99] S. Oaks and H. Wong. *Java Threads*. Oreilly, 2nd edition edition, 1999.

[P. 97] P. Orbek. *Trust and Dependence Analysis*. PhD thesis, 1997. University of Aarhus, Denmark.

[P. 01] P. W. Sathyanathan. *Interprocedural Dataflow Analysis - Alias Analysis*. PhD thesis, 2001. Stanford University's Computer Science Department, USA.

[Pad] Universität Paderborn. The Paderborn Center for Parallel Computing, PC^2. `http://wwwcs.uni-paderborn.de/pc2/`.

[PH00] M. Philippsen and B. Haumacher. Locality Optimization in JavaParty by means of static Type Analysis. *Concurrency: Practice and Experience*, 2000.

[Phi98] M. Philippsen. Data Parallelism in Java. In J. Schaefer, editor, *High Performance Computing Systems and Applications*, pages 85–99. Kluwer Academic Publishers, Boston, Dordrecht, London, 1998.

[PS91] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 146–161, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[PZ97] M. Philippsen and M. Zenger. JavaParty — Transparent Remote Objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997. Special Issue: Java for computational science and engineering — simulation and modeling II.

[Rin01a] M. C. Rinard. Analysis of Multithreaded Programs. In *Proceedings of the 8th Static Analysis Symposium (Paris, France)*, July 2001.

[Rin01b] M. C. Rinard. Analysis of Multithreaded Programs. In *Static Analysis Symposium*, pages 1–19, 2001.

[RR99] R. Rugina and M. Rinard. Pointer Analysis for Multithreaded Programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 77–90, Atlanta, Georgia, May 1–4, 1999.

[Ryd03] B. Ryder. Dimensions of Precision in Reference Analysis of Object-oriented Languages, 2003. In CC 2003, pages 126–137.

[SH97] M. Shapiro and S. Horwitz. Fast and Accurate Flow-insensitive Points-to Analysis. In *Proceedings of the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[SH98] M. Schroeder and F. Hauck. Juggle: Eine verteilte virtuelle Maschine fuer Java. Techn. Report TR-I4-98-03, Univ. of Erlangen-Nuernberg, 1998. german.

[Spi99a] A. Spiegel. Pangaea: An Automatic Distribution Front-End for Java. *Lecture Notes in Computer Science*, 1586:93–99, 1999.

[Spi99b] A. Spiegel. Object Graph Analysis. Technical report, Technical Report B-99-11, FU Berlin, FB Mathematik und Informatik, July 1999.

[Spi99c] A. Spiegel. Automatische Verteilung in Pangaea. In *Java Informations-Tage (JIT '99), Düsseldorf*, September 1999.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. In *ACM TOPLAS*, pages 20(1): 1–50, January 1998.

[Ste96] B. Steensgaard. Points-to Analysis by Type Inference of Programs with Structures and Unions. In *Proceedings of the 6th International Conference on Compliler Construction*, pages 136–150, April 1996.

[StJOG01] Y. Smaragdakis and the J-Orchestra Group. Apllication Partitioning without Programming (a White-Paper and Future Work Proposal, 2001. Colege of Computing, Georgia Institute of Technology.

[T. 99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition edition, 1999.

[TAS98] B. Topol, M. Ahamad, and J. T. Stasko. Robust State Sharing for Wide Area Distributed Applications. In *International Conference on Distributed Computing Systems*, pages 554–561, 1998.

[Thi01] M. Thies. *Combining Static Analysis of Java Libraries with Dynamic Optimization*. PhD thesis, 2001. Universität Paderborn, Germany.

[TP00] F. Tip and J. Palsberg. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '00)*, October 2000.

[TRV+00] E. Truyen, B. Robben, B. Vanhaute, T. Coninx, W. Joosen, and P. Verbaeten. Portable Support for Transparent Thread Migration in Java. In *Proceedings of International Symposium on Agent Systems and Applications/Mobile Agents (ASA/MA'2000), Zurich, Suisse*, 2000.

[TS01] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Apllication Partitioning, 2001. Colege of Computing, Georgia Institute of Technology.

[TS02] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning, June 2002. European Conference on Object-Oriented Programming (ECOOP 2002), Malaga.

[VHBB01] R. Veldema, R. F. H. Hofman, R. Bhoedjang, and H. E. Bal. Runtime Optimizations for a Java DSM Implementation. In *Java Grande*, pages 153–162, 2001.

[vNMB+99] R. v. Nieuwpoort, J. Maassen, H. E. Bal, T. Kielmann, and R. Veldema. Wide-area Parallel Computing in Java. In *Proceedings of the ACM Java Grande Conference*, New York, NY, June 1999. ACM Press.

[Wei03] M. Weißenborn. Strategien zur automatischen Objektmigration auf Grundlage statischer und dynamischer Analyse. Master's thesis, Universität Paderborn, 2003.

[Wes04] H. Wessels. Optimierung dynamischer Methodenbindung zum Entfernen ungenutzter Programelemente durch Kombination statischer Analyseverfahren. Master's thesis, Universität Paderborn, 2004.

[WR99] J. Whaley and M. Rinard. Compositional Pointer and Escape Analysis for Multithreaded Java Programs. Technical report, Massachusetts Institute of Technology Cambrige, MA 02139, November 1999.

[WTV02a] D. Weyns, E. Truyen, and P. Verbaeten. Distributed Threads in Java. In *In Proceedings of the International Symposium on Parallel and Distributed Computing (ISPDC'2002)*, 2002.

[WTV02b] D. Weyns, E. Truyen, and P. Verbaeten. Serialization of a Distributed Execution-state in Java. In *accepted for Net.ObjectDays 2002*, 2002.

[YC97] W. Yu and A. L. Cox. Java/DSM: A Platform for Heterogeneous Computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.

[YSP+98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: a High-Performance Java Dialect. *Concurrency: Practice and Experience*, 10(11–13):825–836, September 1998. Special Issue: Java for High-performance Network Computing.

Abbildungsverzeichnis

Tabellenverzeichnis