Architekturanalyse der Java Virtual Machine unter z/OS und Linux

Diplomarbeit

vorgelegt von

Marc Beyerle
Matrikelnummer: 1830910

Betreuer:
Prof. Dr.-Ing. Wilhelm G. Spruth
Dr. Bernhard Dierberger

Eberhard-Karls-Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Technische Informatik

29. September 2003

Erklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum Unterschrift

Danksagung

Hiermit möchte ich mich bei allen Personen bedanken, die zur Erstellung dieser Diplomarbeit beigetragen haben. An erster Stelle stehen dabei meine Eltern, die mich während meines gesamten Studiums unterstützt haben. Für ihr Verständnis und ihre Geduld, die sie während der Diplomarbeit aufbrachte, möchte ich meiner Freundin Gudrun Wolters danken. Mein besonderer Dank gilt ebenfalls meinen Betreuern Prof. Dr.-Ing. Wilhelm G. Spruth, Dr. Bernhard Dierberger und Erich Amrehn. Weiterhin möchte ich mich auch bei folgenden Personen der Firma IBM bedanken (in alphabetischer Reihenfolge): Dr. Nikolaus Breuer, Uwe Denneler, Axel Lonzer, Hans Dieter Mertiens, Elisabeth Puritscher.

Zusammenfassung

In der vorliegenden Arbeit wird die Architektur der *Java Virtual Machines* (Java VMs) unter den Betriebssystemen z/OS und Linux analysiert. Darauf aufbauend werden die Leistungsunterschiede der Java VMs unter z/OS und *Linux for zSeries* (zLinux) untersucht.

Der größte Unterschied zwischen den Java VMs besteht in der ausschließlich unter z/OS verfügbaren *Persistent Reusable Java Virtual Machines* (PRJVM) Technologie. Die PRJVM stellt einen Ansatz dar, Java in Transaktionsverarbeitungssystemen einzusetzen.

Basierend auf der PRJVM wurde das *Basic Online-Banking System* (BOBS) nach dem TPC-A Standard entwickelt. Die Transaktionen wurden in Java implementiert. Die nachfolgende Tabelle verdeutlicht den Leistungsunterschied zwischen der PRJVM-Technologie und dem bisherigen Ansatz für Transaktionsverarbeitung mit einer Java Virtual Machine. Die Werte sind in Transaktionen pro Sekunde angegeben.

Transaktionstyp	zLinux	z/OS, bisher	z/OS, PRJVM	z/OS, Verhältnis
Full TPC-A	1,223	0,796	261,04	1 : 327,94
Reduced TPC-A	1,315	0,805	511,77	1 : 635,74

BOBS wurde zusätzlich in einer zweiten Version implementiert. Diese bietet nicht den hohen Isolationsgrad der ersten. Dadurch wurde der Overhead bestimmt, welchen das Programmier- und Ausführungsmodell der PRJVM unter z/OS verursacht. Leistungsmessungen ergaben, daß dieser stark abhängig vom Transaktionstyp ist. Ebenfalls wurde mit dieser Variante eine höhere Leistung der Java VM unter zLinux ermittelt (in Abhängigkeit vom Transaktionstyp bis zu einem Faktor von circa 1,5).

Weiterhin werden die *Best Practices* vorgestellt. Diese wurden während der Implementierung und den Performance-Tests ermittelt. Zusätzlich werden die Ergebnisse der *Microbenchmarks* diskutiert. Diese Benchmarks messen bestimmte Leistungsaspekte einer Java VM und wurden ebenfalls im Rahmen dieser Arbeit implementiert.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Motivation für die vorliegende Diplomarbeit waren Anfragen von Kunden des *Technical Marketing Competence Center* (TMCC) der IBM Deutschland Entwicklung GmbH bezüglich der Unterschiede zwischen den Java Virtual Machines unter z/OS und Linux. Aufgrund der Themenstellung muß ausdrücklich darauf hingewiesen werden, daß sich diese Arbeit nicht mit *Java2 Enterprise Edition* (J2EE) Anwendungen beschäftigt. Die vorliegende Arbeit befaßt sich nahezu ausschließlich mit den Themengebieten Transaktionsverarbeitung und Vergleich der untersuchten Plattformen. Die Ergebnisse der durchgeführten Performance-Messungen, sowie das Kapitel über die Best Practices bilden ebenfalls einen großen Teil dieser Arbeit.

Zwar schließen sich die Themen J2EE und Transaktionsverarbeitung nicht aus, es existiert jedoch bis heute kein Web Application Server, welcher von den Möglichkeiten der Persistent Reusable Java Virtual Machines Technologie Gebrauch macht. So wird beispielsweise unter dem *WebSphere Application Server for z/OS* das Management von Transaktionen mit Hilfe von Betriebssystemmitteln durchgeführt (siehe [Litt03]). Die einzigen auf der PRJVM-Technik basierenden Software-Produkte sind der *CICS Transaction Server V2.2* und *IMS Version 7* (siehe [CICS02] und [IMS02]).

Am Kontext dieser Produkte ist zu erkennen, daß ein weiterer Themenbereich in dieser Arbeit eine große Rolle spielt: Das Mainframe-Betriebssystem OS/390 von IBM, sowie die zugrunde liegende S/390-Architektur und ihre neueren Ausprägungen z/OS und zSeries. Sofern nicht anders angegeben, werden die Begriffe hier synonym verwendet. Eine Einführung in dieses Thema würde an dieser Stelle zu weit gehen, und daher sei auf [Herr02] verwiesen.

Der Inhalt dieser Arbeit ist wie folgt aufgebaut: Zuerst wird die Rolle der Programmiersprache Java auf der Serverseite diskutiert (Kapitel 2), danach folgt die Analyse der unterschiedlichen Architekturaspekte in Kapitel 3. Kapitel 4 befaßt sich mit dem Hauptunterschied der Java Virtual Machines: die Persistent Reusable Java Virtual Machines Technologie. Im anschließenden Kapitel 5 wird das Basic Online-Banking System eingeführt, das zur praxisnahen Demonstration der Unterschiede in den Java VMs dient. Um die beiden Java VMs nicht nur unter Verwendung von BOBS bezüglich des Leistungsverhaltens zu untersuchen, werden in Kapitel 6 die ebenfalls bei den

Messungen eingesetzten Microbenchmarks beschrieben. An dieses Kapitel schließen sich die Best Practices an (Kapitel 7). Danach folgt die Diskussion der Benchmark-Ergebnisse in Kapitel 8. In Anhang A wird der Inhalt der beigefügten CD aufgelistet. Den Abschluß bilden die Ergebnisse der Microbenchmarks (Anhang B), sowie die Ergebnisse der Benchmarks mit dem Online-Banking System (Anhang C).

Kapitel 2

Java auf der Serverseite

Wird heute der Begriff *Java* diskutiert, so wird dieser in der Regel nicht mehr nur mit der Programmiersprache *per se* in Verbindung gebracht, sondern mit einer Vielzahl an Spezifikationen, Technologien und Implementierungen. Rund um Java entstand in den letzten Jahren eine Art "Technologielandschaft", die für einen außenstehenden Betrachter praktisch unüberschaubar geworden ist. Das vorliegende Kapitel hebt die Java-spezifischen Unterschiede zwischen Desktop- und Server-Computing hervor, skizziert die verschiedenen Szenarien auf der Serverseite und geht auf die diesbezüglichen Anforderungen an Java ein. Der letzte Abschnitt faßt die Anstrengungen zusammen, die für die *Java Virtual Machine* (Java VM oder auch JVM) unter OS/390 unternommen wurden, um diese Anforderungen zu erfüllen.

2.1 Anforderungen

Während man auf dem Desktop[1] (neben anderen Dingen) besonderen Wert auf multimediale Ausstattung wie zum Beispiel ein *Graphical User Interface* (GUI) oder Video- und Soundausgabe legt, werden an einen Server ganz andere Anforderungen gestellt. Die Tatsache, daß Java in mehreren Ausprägungen für unterschiedliche Einsatzszenarien zur Verfügung steht, untermauert diese Aussage - wenigstens was Java betrifft. Ein Blick auf die *Application Programming Interfaces* (APIs) der clientseitigen *Java2 Standard Edition* (J2SE) und der Ausgabe für Geschäftsanwendungen, der *Java2 Enterprise Edition* (J2EE), genügt, um die verschiedenen Anforderungen hervorzuheben. Während sich bei der J2SE mehr als die Hälfte der Java Packages in den Bereich GUI bzw. Multimedia einordnen lassen, sind die APIs der J2EE auf typische Server-Anwendungen ausgerichtet: *Enterprise JavaBeans* (EJB), *Java Servlets* und *Java ServerPages* (JSP), um nur die bekanntesten Vertreter zu nennen.

[1] Im folgenden *Client* genannt.

2.1.1 Reliability, Availability, Serviceability (RAS)

Vielfach faßt man die Eigenschaften und somit auch die Anforderungen, die für einen reibungslosen (und ausfallsicheren) Betrieb eines serverseitigen Dienstes erforderlich sind, unter dem Akronym RAS (*Reliability, Availability, Serviceability*) zusammen.

Zuverlässigkeit ist bei zSeries-Systemen seit Generationen in praktisch allen Schichten implementiert. Angefangen bei der redundanten Auslegung des Prozessorboards über Fehlerprüfungsalgorithmen in der Speicherarchitektur bis hin zur Integration von Transaktionsservices im Betriebssystem[2].

Der Grundgedanke der *Verfügbarkeit* besteht in der Fähigkeit eines Systems, einen bestimmten Dienst ununterbrochen verrichten zu können. Die auf Desktop-Systemen weit verbreitete Vorgehensweise, den Rechner nach einem nicht behebbaren Fehler neu zu starten, ist für einen Server-Betrieb inakzeptabel. Daher wurden die zugrunde liegenden (Server-) Technologien kontinuierlich verbessert, so daß diese Situation so gut wie nie eintritt. So kann man beispielsweise für zSeries-Systeme mit Hilfe der *Parallel Sysplex* Technologie eine Verfügbarkeit von 99,999 Prozent garantieren. Anders ausgedrückt stellt dies einen Systemausfall von fünf Minuten im Jahr dar - für geplante und ungeplante Unterbrechungen des Rechnerbetriebs.

Unter *Wartbarkeit* versteht man schließlich nicht nur die Instandhaltung der Hard- und Software eines Rechnersystems, sondern auch die für das Mainframe-Computing und insbesondere für die zSeries-Rechner einzigartigen Eigenschaften wie die (unterbrechungsfreie) Austauschbarkeit von Hardware-Komponenten während des laufenden Betriebs (*Hot Swap*).

2.1.2 Quality of Service (QoS)

Durch die zunehmende Bedeutung des Internets für das Geschäftsleben ist ein weiterer Begriff entstanden, welcher heute in seiner Bedeutung dem erwähnten RAS in nichts nachsteht: *Quality of Service* (QoS).

Im allgemeinen versteht man darunter die Priorisierung von Transaktionen hinsichtlich verschiedener nicht-funktionaler Aspekte, wie zum Beispiel die Priorisierung von bestimmten Internet-Diensten gegenüber anderen oder die Erfüllung bestimmter *Service Level Agreements* (SLAs). Auf der Seite von Internet-Serviceanbietern bedeutet QoS unter anderem auch die Bevorzugung von bestimmten Kunden. Realisiert wird dies durch die Unterscheidung zwischen gewöhnlichen Transaktionen und solchen, die einen höheren Umsatz generieren.

Überschneidungen zwischen QoS und RAS finden sich in technischen Zielen wieder, wie der Minimierung der Antwortzeiten, Maximierung des Transaktionsdurchsatzes und ähnlichem.

[2] OS/390 Transaction Management and Recoverable Resource Manager Services (OS/390 RRS)

2.1.3 Anforderungen an die Programmiersprache

All dies stellt besondere Anforderungen an die Software und somit auch an die zu ihrer Herstellung verwendete Programmiersprache. Auf der einen Seite stehen dabei aus der Softwaretechnik bekannte Eigenschaften wie Zuverlässigkeit (hierzu zählen Stabilität, Korrektheit, Robustheit), Wirtschaftlichkeit (Entwicklungskosten, Effizienz, Portabilität) und Wartbarkeit (Kompatibilität, Lesbarkeit, Anpaßbarkeit). Auf der anderen Seite ist es notwendig, daß Standards zur Verfügung stehen. Diese tragen ebenfalls dazu bei, die eingangs erwähnten Anforderungen zu erfüllen.

Die Programmiersprache Java hat sich in den letzten Jahren als Quasi-Standard etabliert, nicht nur für die Entwicklung von Client-, sondern auch zur Herstellung von Server-Software. Mögliche Gründe hierfür sind die freie Verfügbarkeit des *Java Development Kit* (JDK) für nahezu alle Plattformen, die umfangreiche Dokumentation, die im Vergleich zu anderen Sprachen leichte Erlernbarkeit und die große Anzahl an frei verfügbaren Bibliotheken. Auch die Menge an Entwicklungstools und die Tatsache, daß immer mehr Hochschulabgänger im Laufe ihres Studiums Erfahrungen mit Java gesammelt haben, tragen ihren Teil zu Javas Popularität bei. Welche Rolle dabei Java auf der Serverseite zukommt, wird im nächsten Abschnitt näher ausgeführt.

2.2 Szenarien

Java, das ursprünglich für sogenannte *Embedded Devices* entwickelt wurde, hat sich in der Zwischenzeit auch auf der Serverseite weit verbreitet. So wird beispielsweise in einem Strategiepapier der Firma IBM (siehe [Java00]) die Rolle Javas als die lang ersehnte “Universalsprache” hervorgehoben: Das als explosionsartig zu bezeichnende Wachstum des Internet und die damit verbundenen Möglichkeiten für e-business Anwendungen führten dazu, daß die Eignung der Sprache Java nicht nur für kleinere Internet-, sondern in besonderem Maße auch für komplexe Geschäftsanwendungen erkannt wurde. Ein weiterer Schritt in diese Richtung war die Entwicklung von sogenannten *Konnektoren*, welche die Verbindung zu Datenbankmanagement- und Transaktionsverarbeitungssystemen und somit zu den *Key Assets* vieler IT-Firmen ermöglichen. Beispiele für solche Konnektoren sind SQLJ und JDBC für DB2, das *CICS Transaction Gateway* für das *Customer Information Control System* (CICS) oder der *IMS Connector for Java* für das *Information Management System* (IMS). Auch Javas “write once, run anywhere” Paradigma und die damit verbundene Möglichkeit, Java-Programme auf unterschiedliche Plattformen zu portieren, leisten ihren Beitrag zur Verbreitung von Java auf der Serverseite. So existiert heute ein äußerst breit gefächertes Spektrum an Möglichkeiten, Java auf der Serverseite einzusetzen, welches in den folgenden Unterabschnitten näher ausgeleuchtet wird.

2.2.1 Java Servlets und JavaServer Pages

Die bekanntesten Vertreter dieses Spektrums sind Java Servlets und JavaServer Pages. Beide Technologien haben sich in der Praxis bewährt und werden von vielen Firmen in der Produktion eingesetzt. Servlets sind laut Sun Javas "komponentenbasierte, plattformunabhängige Methode, World Wide Web-basierte Anwendungen zu entwickeln, ohne an die Performance-Beschränkungen von *Common Gateway Interface* (CGI) Programmen gebunden zu sein". Sie befolgen dabei das aus dem Client/Server-Paradigma abgeleitete Prinzip des *Hypertext Transfer Protocol* (HTTP) Request/Response-Verfahrens. Die Laufzeitumgebung für Servlets wird von sogenannten Servlet *Containern* bereitgestellt.

JavaServer Pages sind eine Erweiterung der Servlet-Technologie, die es erlauben, ohne tiefgehende Java-Programmierkenntnisse dynamische Webseiten zu erstellen. Dabei kommen *Extensible Markup Language* (XML)-artige *Tags* zum Einsatz, welche sowohl das Aussehen als auch das Verhalten der Seiten bestimmen. Die Homepages in Tabelle 2.1 enthalten weiterführende Informationen.

Komponenten-Technologie	Homepage
Enterprise JavaBeans (EJB)	http://java.sun.com/products/ejb
Java Servlet	http://java.sun.com/products/servlets
JavaServer Pages (JSP)	http://java.sun.com/products/jsp

Tabelle 2.1: J2EE: Komponenten-Technologien

2.2.2 Enterprise JavaBeans (EJB)

Sun Microsystems bezeichnet die beiden gerade erwähnten Programmiermodelle als komponentenbasiert. Die ebenfalls in der J2EE vorhandenen *Enterprise JavaBeans* (EJB) sind jedoch die eigentliche Technologie, auf welche dieses Attribut vom softwaretechnischen Standpunkt her zutrifft. EJBs existieren in unterschiedlichen Ausprägungen: Session Beans, Entity Beans und Message-Driven Beans, wobei man Session Beans weiter in *stateful* und *stateless* unterteilt. Im folgenden werden die einzelnen Beans kurz vorgestellt.

Ein *Session Bean* repräsentiert einen Client innerhalb eines J2EE-Servers und die mit dem Client verbundene Sitzung, während ein *Entity Bean* ein geschäftsbezogenes Objekt innerhalb eines persistenten Speichermechanismus darstellt. Ein Session Bean verrichtet eine bestimmte Aufgabe innerhalb eines J2EE-Servers im Auftrag seines ihm zugeordneten Clients, beispielsweise das Hinzufügen eines Artikels in einen Warenkorb. Auf der Seite der Entity Beans wären Kunden, Bestellungen und Produkte als Beispiele für geschäftsbezogene Objekte zu nennen.

Ein *Message-Driven Bean* ist schließlich ein Bean, welches es J2EE-Anwendungen erlaubt, Nachrichten asynchron zu verarbeiten. Dabei verhält es sich ähnlich wie ein

aus der J2SE bekannter `EventListener`, mit dem Unterschied, daß es auf Nachrichten und nicht auf Ereignisse wartet. Eine sehr gute Einführung in EJBs findet man unter [J2EETut].

2.2.3 Java2 Enterprise Edition (J2EE)

Nachdem die Java2 Enterprise Edition mehrfach erwähnt wurde, soll an dieser Stelle ein kurzer Überblick über J2EE gegeben werden. Pragmatisch betrachtet ist J2EE eine Erweiterung der J2SE um eine Vielzahl an Spezifikationen, Designrichtlinien, Vorgehensweisen und APIs. Es wird ein Modell für verteilte, mehrschichtige Anwendungen definiert, welches die verschiedenen Aspekte einer J2EE-Anwendung auf unterschiedliche logische Schichten verteilt. So kann man zum Beispiel in der *Client* Schicht Java Applets einsetzen, in der *Web* Schicht Java Servlets und JSPs, in der *Business* Schicht EJBs und schließlich am hinteren Ende (auf englisch: *back-end*) sogenannte *Enterprise Information Systems* (EIS). In Abbildung 2.1 ist diese Aufteilung in unterschiedliche Schichten dargestellt.

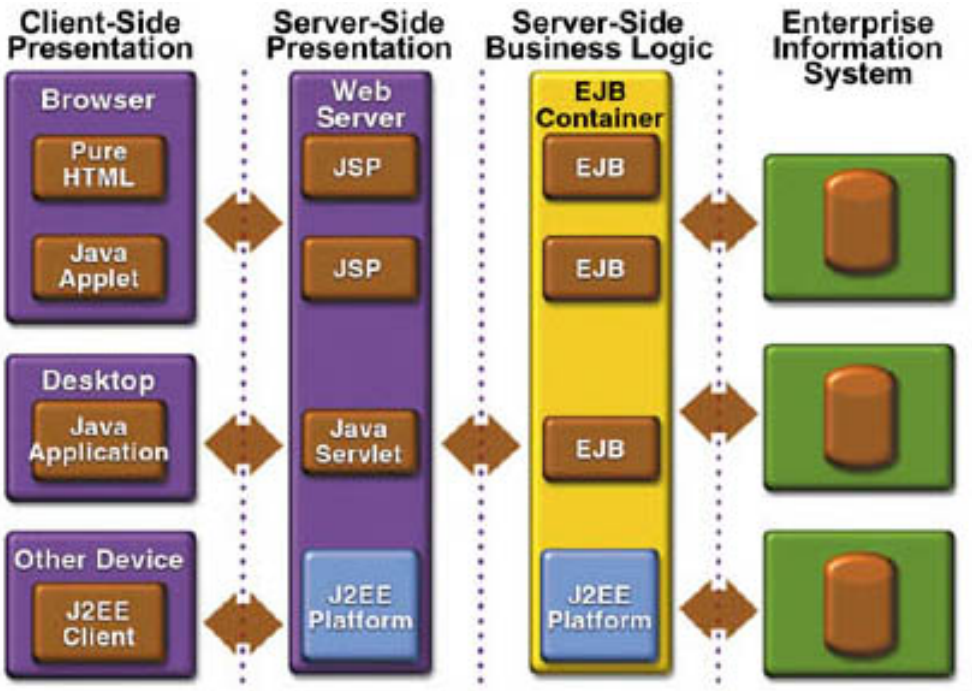

Abbildung 2.1: J2EE-Anwendungsmodell

Zusätzlich zu den bereits in der Java2 Standard Edition enthaltenen APIs, wie zum Beispiel JDBC[3], *Remote Method Invocation* (RMI), *Java Naming and Directory Interface* (JNDI), *Java Cryptography Extension* (JCE) usw. sind in der Java2 Enterprise

[3]Eine Bemerkung am Rande: JDBC stand einst für *Java Database Connectivity*, heute ist es ein eigenständiges Markenzeichen der Firma Sun Microsystems.

Edition weitere enthalten, wie die *Java Transaction API* (JTA), welche den Zugriff auf Transaktionsmanager ermöglicht. Dabei müssen diese Transaktionsmanager die *Java Transaction Service* (JTS) Spezifikation erfüllen. Ebenfalls sind noch die *Java Message Service* (JMS) API, *JavaMail* und die weiter vorne in diesem Kapitel erwähnten Enterprise JavaBeans, Java Servlets und JavaServer Pages zu erwähnen. Abbildung 2.2, welche die bekannteste Abbildung im Zusammenhang mit J2EE ist, faßt diese Aufzählung zusammen.

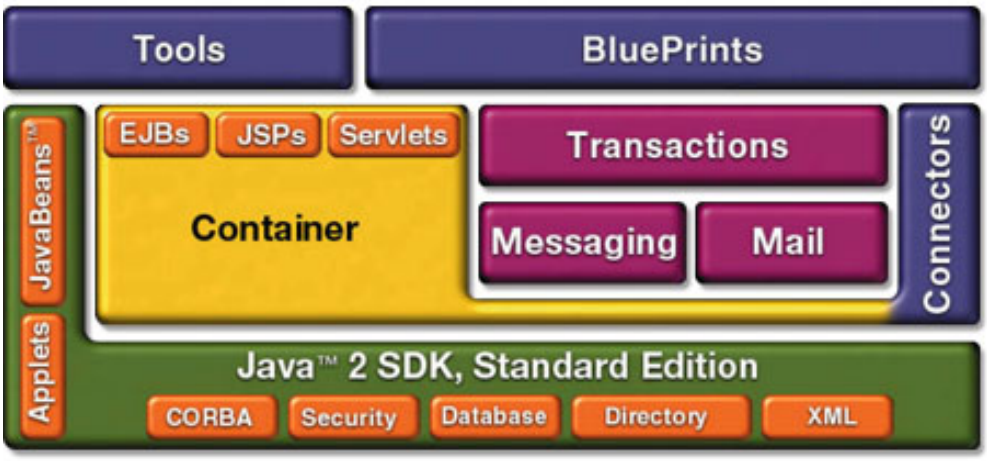

Abbildung 2.2: J2EE-Architektur

2.2.4 Web Services

Eine weitere Möglichkeit, Java auf der Serverseite einzusetzen, sind *Web Services*. Im Grunde genommen stellen Web Services keine Neuentwicklung dar, sie kombinieren lediglich vorhandene Technologien und erweitern diese um einige Spezifikationen. Ein Web Service im wörtlichen Sinne ist die zur Verfügungstellung eines Dienstes über das World Wide Web. Dabei kommt für XML-basierte Web Services als Protokoll typischerweise das *Simple Object Access Protocol* (SOAP) über HTTP zum Einsatz. Ein oft zitiertes Beispiel für ein Web Service-Szenario ist die Abfrage eines Aktienkurses. Dabei schickt ein Client eine Anfrage an einen Aktienservice, welcher diese bearbeitet und den aktuellen Kurs als Antwort zurückschickt.

Das Haupteinsatzgebiet solcher Dienste sind sogenannte *Business-to-Business* Anwendungen (B2B-Anwendungen). XML spielt dabei als Sprache zum Datenaustausch eine zentrale Rolle, da bei B2B-Interaktionen oft unterschiedliche Plattformen zum Einsatz kommen, die in der Regel über eigene Formate zur Datenspeicherung und -auswertung verfügen. Für Java-Anwendungen existieren eine ganze Reihe von APIs, um Web Services mit XML zu kombinieren. Angefangen von der *Java API for XML Processing* (JAXP) zum Parsen und Bearbeiten von XML-Dokumenten über die *Java API for XML-based RPC* (JAX-RPC) und *Java API for XML Messaging* (JAXM)

zum Versenden von SOAP-Anfragen und Empfangen der Antworten bis hin zur *Java API for XML Registries* (JAXR) zum einheitlichen Zugriff auf sogenannte *Business Registries*[4] werden von Java alle Hilfsmittel bereitgestellt, um eigene Web Services zu erstellen und zu nutzen.

2.2.5 Standalone Java Applications

Ein weiteres Beispiel für serverseitiges Java ist das Schreiben eigener, sogenannter *standalone* Java-Applikationen, die von einer Kommandozeile aus gestartet werden können. Beispiele für solche Anwendungen sind Stapelverarbeitungsprogramme (*Batch*), Kommunikations-Server für asynchrone Nachrichtenverarbeitung oder Programme zur Transaktionsverarbeitung. Das in Kapitel 5 vorgestellte Basic Online-Banking System gehört in den letzteren dieser Bereiche, wodurch auch die eingangs erwähnte Positionierung dieser Diplomarbeit verdeutlicht wird.

2.2.6 Java und Transaktionsverarbeitung

Bisher litt Java unter dem Vorurteil, zu langsam für den Einsatz in Transaktionsverarbeitungssystemen zu sein. Der Grund für dieses Vorurteil war folgender: Damit eine Transaktion als einwandfrei abgearbeitet betrachtet werden kann, müssen bestimmte Voraussetzungen erfüllt sein, die in den allgemein anerkannten ACID[5] Eigenschaften festgeschrieben wurden. Um diese zu erreichen, mußte die Java Virtual Machine bisher für jede Transaktion neu gestartet werden. Dieser Neustart war für die Folgetransaktion zwingend notwendig, um eventuelle sicherheitskritische Reste der vorhergehenden Transaktion zu beseitigen. Beispiele hierfür sind überschriebene statische Variablen, geladene native Bibliotheken usw.

Das Hoch- und Herunterfahren der Java VM stellt einen immensen Zeitaufwand dar, da bei jedem Starten der Java VM mehrere hundert Klassen geladen, Speicherbereiche alloziert und Java VM-interne Datenstrukturen angelegt werden müssen. Daher erreicht man mit dieser Vorgehensweise nur sehr geringe Transaktionsraten, wie aus Tabelle 8.5 auf Seite 92 zu entnehmen ist. Die in Kapitel 4 vorgestellte Architektur der Persistent Reusable Java Virtual Machines stellt einen neuen Ansatz zur Umgehung dieser Problematik dar. Mit Hilfe dieser Technologie können wesentlich höhere Durchsatzraten erzielt werden (Faktor 327,94 und höher, siehe Abschnitt 8.3.5 und Anhang C.1).

Das Standardwerk zum Thema Transaktionsverarbeitung ist [Gray93]. Hier werden alle relevanten Problemstellungen der Transaktionsverarbeitung behandelt, man findet dort auch ausführliche Erläuterungen zu den weiter oben erwähnten ACID-Eigenschaften.

[4] eine Art elektronische “Gelbe Seiten”

[5] ACID steht für *Atomicity, Consistency, Isolation and Durability*.

2.3 Was bisher geschah

Java auf der Serverseite bedeutet nicht nur den Einsatz von Java auf PC-basierten oder Midrange-Servern, es schließt auch den Einsatz auf Mainframes mit ein. Dieser Abschnitt gibt die Anstrengungen wieder, die von einem Team von Experten unternommen wurden, um Java unter dem OS/390-Betriebssystem (zu dieser Zeit existierte noch kein z/OS) mit einer akzeptablen Performance verfügbar zu machen. Der Artikel, auf den hier bezug genommen wird, ist unter [Dill00] zu finden.

2.3.1 Portierung der Java VM auf OS/390

Als die Java VM zum ersten Mal auf OS/390 portiert wurde, mußte man feststellen, daß der Prototyp 60100 Mal langsamer als die zugrunde liegende Referenzimplementierung lief, welche von Sun für UNIX-Plattformen zur Verfügung gestellt wurde. Ein weiteres Problem bestand darin, daß die Performance des Prototyps nicht mit der Anzahl der vorhandenen Prozessoren skalierte. Obwohl der Prototyp noch keinen *Just-In-Time Compiler* (JIT-Compiler oder JIT) besaß und im *debug* Modus lief, war der Umfang des Portierungsaufwands bereits zu diesem Zeitpunkt ersichtlich.

Das hauptsächliche Augenmerk bei der Portierung richtete sich (aufgrund der genannten Problematik) neben der Integration von OS/390-spezifischer Funktionalität[6] auf die Steigerung der Performance. Einige Gebiete des Prototyps wirkten sich in besonderem Maße negativ auf die Performance aus: Die Initialisierung der Java VM, die Locking-Techniken, die Speicherverwaltung, zu große Pfadlängen der Service-Aufrufe, zu viele Abfragen nach Systeminformationen und ineffiziente Behandlung von Datentypen.

2.3.2 Locking

Im Locking-Bereich wurden die beiden am meisten angeforderten Java VM-internen Locks weitestgehend dadurch eliminiert, daß für eine der Tabellen, die durch diese Locks geschützt wurde, nur-lesende Funktionen eingeführt wurden und die Reihenfolge der Locking-Aufrufe für die andere Tabelle optimiert wurde.

Im IBM Developer Kit for OS/390, Java Technology Edition, Version 1.1.4, wurde ein neues Locking-Schema nach dem *Compare-And-Swap* Verfahren eingeführt, was ebenfalls zu einer Performance-Verbesserung führte. Details zu den übrigen Verbesserungen im Locking-Bereich sind in Tabelle 1 des Artikels aufgeführt.

2.3.3 Garbage Collection

Im Gebiet der Garbage Collection, Javas automatischer Speicherbereinigung, wurde im JDK 1.1.4[7] ein *concurrent* Garbage Collector implementiert. Dieser ermöglicht es

[6] z.B. Unterstützung für EBCDIC

[7] In diesem Abschnitt ist immer das JDK von IBM für OS/390 gemeint.

den sogenannten Java *Worker Threads,* ihre Arbeit während einer laufenden Speicherbereinigung fortzuführen. Dies steht im Gegensatz zu dem Garbage Collector des Prototypen, der nach dem *Stop-The-World* Prinzip arbeitete und die Ausführung aller aktiven Worker Threads aussetzte, sobald eine Garbage Collection durchgeführt werden mußte. Für Programme, die Javas Multithreading-Möglichkeiten benutzen, können sich hierdurch Performance-Verbesserungen ergeben. Im allgemeinen wird die Garbage Collection als ein besonders performancekritisches Gebiet angesehen, so daß hier immer wieder Fortschritte erzielt werden konnten.

2.3.4 Pfadlängen

Ein Beispiel für eine Verbesserung der Pfadlänge bei Service-Aufrufen ist folgendes: Die Identität und der Zustand eines Java-Threads müssen von der Java VM unabhängig von dem darunterliegenden Betriebssystem-Thread verwaltet werden. Ein Service-Aufruf, der zu diesem Zweck sehr oft ausgeführt wird, ist `sysThreadSelf()`. Dieser liefert die Adresse des Java Control Blocks für den gerade ausführenden Thread zurück. Man kann für dessen Implementierung einen API-Aufruf (`pthread_getspecific()`) benutzen, um einen vorher mittels `pthread_setspecific()` gesetzten Wert auszulesen. Eine der Änderungen im JDK 1.1.4 war die Einführung einer Assembler-Schnittstelle, welche den Zeiger auf einen Java-Thread in einem UNIX-bezogenen Thread Control Block ablegte. Diese Optimierung hatte eine Performance-Verbesserung von ungefähr zehn Prozent zur Folge.

2.3.5 EBCDIC

Als eine der größten Hürden bei der Portierung des C-Teils der Referenzimplementierung erwies sich die Tatsache, daß OS/390 intern EBCDIC benutzt, um Character-Daten zu kodieren. So wurde entschieden, die C-Strings in ASCII zu kodieren, welches von den meisten (anderen) Plattformen verwendet wird. Dieser Ansatz machte es notwendig, eine Art “virtuelle Grenze” um die Java VM zu ziehen: Alle Character-Daten innerhalb der Java VM werden in ASCII kodiert und nur dann nach EBCDIC konvertiert, wenn sie die virtuelle Grenze verlassen, um zum Beispiel auf dem Terminal ausgegeben zu werden. Dies ermöglichte es der Java VM, so zu operieren, als ob sie auf einer ASCII-basierten Plattform zum Einsatz käme.

2.3.6 Erweiterte Funktionalität

Schließlich wurde die Referenzimplementierung um einige OS/390-spezifische Funktionen erweitert, um die zum Teil einzigartigen und seit Jahrzehnten in die OS/390-Plattform integrierten Komponenten für Java-Programme zugänglich zu machen. So ist es zum Beispiel möglich, jedem Java-Thread eine eigene *Security Identity* zuzuweisen. Diese Funktionalität wird dann benötigt, wenn nicht alle Threads einer Anwendung einer einzigen Security Identity zugeordnet werden sollen. Auch im Bereich der

Diagnoseinformationen wurden Erweiterungen vorgenommen, um für die Java VM ähnliche Diagnosemöglichkeiten zur Verfügung zu haben wie für andere Software-Produkte unter OS/390.

2.3.7 Just-In-Time Compiler

Neben der Garbage Collection existiert ein weiterer Teilbereich der Java VM, der als besonders performancekritisch gilt: der Just-In-Time Compiler. Auf der einen Seite mußte der JIT-Compiler angepaßt werden, da er als solcher zur Laufzeit Maschinencode für eine Hardware-Plattform erzeugt. Auf der anderen Seite wurden im Just-In-Time Compiler der Java VM unter OS/390 eine große Anzahl an Verbesserungen durchgeführt, die zum Teil auf Optimierungstechniken vorhandener IBM-Compiler beruhen. Hier alle Maßnahmen der verschiedenen Compile-Stufen wie z.B. *Loop Optimization* oder *Inlining* im sogenannten *Front End* oder *Instruction Ordering* im *Back End* einzeln zu nennen, würde mehrere Seiten füllen, und deshalb sei auf den entsprechenden Abschnitt in [Dill00] verwiesen.

Der in diesem Zusammenhang oft genannte *High Performance Compiler for Java for OS/390* (HPCJ/390) verfolgt einen anderen Ansatz als ein Just-In-Time Compiler. Während ein JIT-Compiler Maschinencode zur Laufzeit erzeugt, wird ein Java-Programm mit dem HPCJ/390 vor dessen Ausführung in Maschinencode übersetzt - entweder als *Executable* oder als *Dynamic Link Library* (DLL). In dieser Hinsicht entspricht der HPCJ/390 einem “gewöhnlichen” Compiler, wie er zum Beispiel für C-Programme existiert. Ein Vorteil dieses Ansatzes ist, daß das resultierende Programm unabhängig von einer Java VM ausgeführt wird und somit beispielsweise direkt als CICS-Transaktion oder DB2 *Stored Procedure* aufgerufen werden kann[8]. Ein Nachteil ist, daß die Plattformunabhängigkeit von Java-Programmen nicht mehr gegeben ist, sobald diese mit dem HPCJ/390 Compiler übersetzt wurden.

2.3.8 Zusammenfassung

Im Rückblick auf die beschriebenen Änderungen wird deutlich, daß sich die meisten von ihnen auch auf andere Server-Plattformen übertragen lassen. Anders ausgedrückt ist der Großteil von ihnen nicht OS/390-spezifisch. Am Ende des Artikels [Dill00] wird eine Architektur für eine Java VM eingeführt, welche die Grundlage für die in Kapitel 4 vorgestellte Persistent Reusable Java Virtual Machines Technologie bildete. Die Besonderheit dabei ist, daß diese Architektur bisher nur unter OS/390 respektive z/OS implementiert wurde. Mit sehr großer Wahrscheinlichkeit wird diese Technologie in absehbarer Zukunft auch auf andere Server-Plattformen übertragen werden.

[8] Um Java-Programme ausführen zu können, die nicht mit dem HPCJ/390 übersetzt wurden, muß CICS bzw. DB2 eine (oder mehrere) Java VMs starten und verwalten.

Kapitel 3

Analyse der unterschiedlichen Architekturaspekte

Führt man eine Architekturanalyse eines bestehenden Systems durch, so ist es vorteilhaft, wenn man die gefundenen Unterschiede kategorisiert, um eine bessere Übersichtlichkeit zu erhalten. Eine der Zielsetzungen dieser Diplomarbeit war es, eine Architekturanalyse der Java Virtual Machine unter z/OS und Linux durchzuführen. Im vorliegenden Kapitel sind die Ergebnisse dieser Analyse festgehalten, wobei die einzelnen Unterschiede den logischen Schichten Hardware (Abschnitt 3.1), Betriebssystem (Abschnitt 3.2) und schließlich der Java Virtual Machine selbst (Abschnitt 3.3) zugeordnet werden.

Die Motivation dabei ist nicht, alle Hardware- oder Betriebssystem-Unterschiede zwischen zSeries und IA-32 Intel Architecture beziehungsweise z/OS und Linux zu diskutieren, da dies den Rahmen dieser Arbeit sprengen würde. Vielmehr sollen nur diejenigen in der Hardware und dem Betriebssystem gefundenen Unterschiede vorgestellt werden, die eine direkte Auswirkung auf die Java VM haben. Die praktische Relevanz dieser Unterschiede wird in Kapitel 7 deutlich.

Ein weiterer Aspekt einer Architekturanalyse ist die Untersuchung eines Systems unter dem Gesichtspunkt der Performance. Diesem Sachverhalt wird in Kapitel 8 sowie im Anhang Rechnung getragen.

3.1 Hardware

Der größte Hardware-Vorteil eines zSeries-Rechners in bezug auf die Java Virtual Machine ist der sogenannte *Shared Second Level Cache* (Shared L2 Cache) dieser Architektur. *Shared* steht in diesem Zusammenhang für die Tatsache, daß sich die Hälfte aller Prozessoren eines zSeries-Rechners einen L2 Cache teilen. Der Vorteil, der sich hierdurch für Java-Anwendungen ergibt, liegt in Javas Programmiermodell begründet. Dieses ist explizit für das Schreiben von *multithreaded* Anwendungen ausgelegt. Verfügt ein Rechner über mehrere Prozessoren, so können die einzelnen Ausführungs-

einheiten[1] auf die verschiedenen Prozessoren verteilt werden.

Im Gegensatz zu anderen Multiprozessor-Architekturen liegt der L2 Cache wie eine Art "gigantischer Switch" zwischen den Prozessoren und dem Hauptspeicher. Abbildung 3.1 stammt aus [Harr02] und zeigt den Aufbau eines z900 *Processor Cage.* Deutlich ist die hohe Verdrahtungsdichte zwischen den Prozessoren und dem L2 Cache zu erkennen. Diese ist neben der hohen Geschwindigkeit des Interface zum L2 Cache dafür verantwortlich, daß sich die zSeries-Architektur durch eine herausragende Multiprozessor-Performance auszeichnet. Zudem wird eine Cacheline grundsätzlich nur dann invalidiert, wenn es sich nicht vermeiden läßt, was zu sehr guten Cache-Miss-Raten führt. Das Ersetzungsverfahren für Cache-Einträge beruht auf einer *Least Recently Used* (LRU) Strategie, die dazu führt, daß sich der L2 Cache gewissermaßen "selbst organisiert". Damit ist gemeint, daß ohne zusätzliche Schaltungs- oder Software-Logik immer die am meisten verwendeten Einträge im Cache zu finden sind.

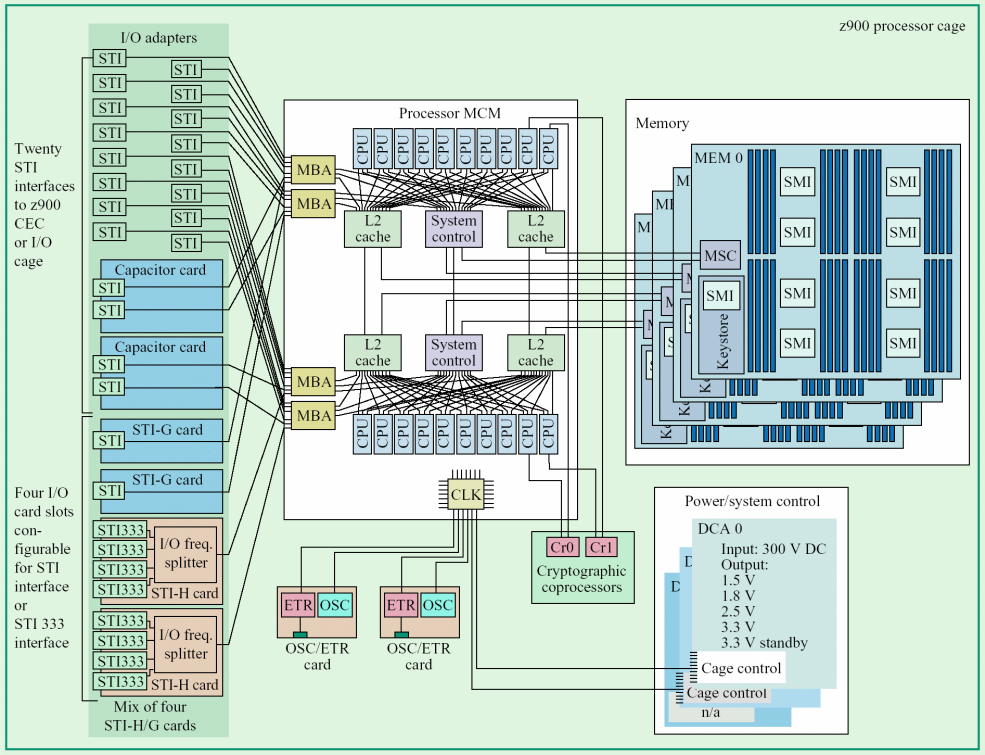

Abbildung 3.1: z900 Processor Cage

Es wird allgemein anerkannt, daß die zSeries-Hardware durch diese Eigenschaften eine außerordentlich gute Skalierbarkeit für Anwendungen erreicht, welche die Mehrprozessor-Eigenschaften der darunterliegenden Hardware ausnutzen können. Zu diesen Anwendungen zählen wie eingangs erwähnt auch Java-Anwendungen - vorausgesetzt, sie sind entsprechend implementiert.

[1] Näheres hierzu siehe Abschnitt 3.2.5, “Threads”.

Leser, die an einem einführenden Vergleich zwischen der IA-32 Intel Architecture und der S/390-Architektur interessiert sind, seien auf [Amre00], Anhang A, “Intel architecture, S/390 architecture” verwiesen.

3.2 Betriebssystem

In bezug auf die Java VM liegt der Hauptunterschied auf Betriebssystemebene in den verschiedenen Speicher- und Prozeßmodellen von z/OS und Linux. Linux auf Intel-basierten Systemen und *Linux for zSeries* (zLinux) unterscheiden sich in dieser Hinsicht nicht.

Um die Unterschiede hinreichend zu erklären, muß etwas weiter ausgeholt werden. Die Literaturreferenzen, welche die Grundlage für diesen Abschnitt bilden, sind [ABCs00a], [ACSR02], [PortGuide] und [Coms02].

3.2.1 UNIX System Services

Im Zusammenhang mit einer Java-Diskussion unter z/OS müssen die *z/OS UNIX System Services* (z/OS UNIX) erwähnt werden. Der Grund hierfür ist unter anderem die Tatsache, daß Java-Programme unter z/OS immer Teile der UNIX System Services benutzen. Historisch betrachtet wurde die in Abschnitt 2.3.1 erwähnte Referenzimplementierung der Java Virtual Machine auf z/OS UNIX portiert.

Zunächst einmal muß geklärt werden, was unter dem Begriff UNIX System Services zu verstehen ist. Die Kurzfassung einer möglichen Erklärung ist: Die UNIX System Services stellen unter z/OS eine UNIX-Umgebung zur Verfügung. Dazu gehören unter anderem die Integration eines UNIX-Kernels in das *Base Control Program* (BCP)[2], das *Language Environment* (LE), das *Hierarchical File System* (HFS), eine Shell, ein C/C++ Compiler und weitere Komponenten. Die UNIX System Services sind an vielen Stellen in das z/OS-Betriebssystem integriert, wie Abbildung 3.2 auf Seite 20 verdeutlicht (entnommen aus [ABCs00b]).

Bei der Zusammenführung zweier so verschiedenartiger Betriebssysteme wie z/OS und UNIX entsteht folgendes Problem: Auf der einen Seite existieren unterschiedliche Vorgehensweisen für die Ausführung Betriebssystem-typischer Aufgaben. Auf der anderen Seite ist häufig zu beobachten, daß für ein Konzept, das in beiden Betriebssystemen existiert, zwei verschiedene Begriffe verwendet werden. Die folgenden Unterabschnitte gehen auf diese Problematik ein und versuchen, etwas Licht in dieses Dunkel zu bringen.

3.2.2 Address Spaces

Unter z/OS existiert das Konzept des *Address Space*, eines virtuellen Adreßraums, dessen Größe abhängig von der Hardware-Generation ist (zum Beispiel 2 Gigabyte bei

[2] Das BCP ist unter Verwendung von UNIX-Terminologie der “Kernel” des z/OS-Betriebssystems.

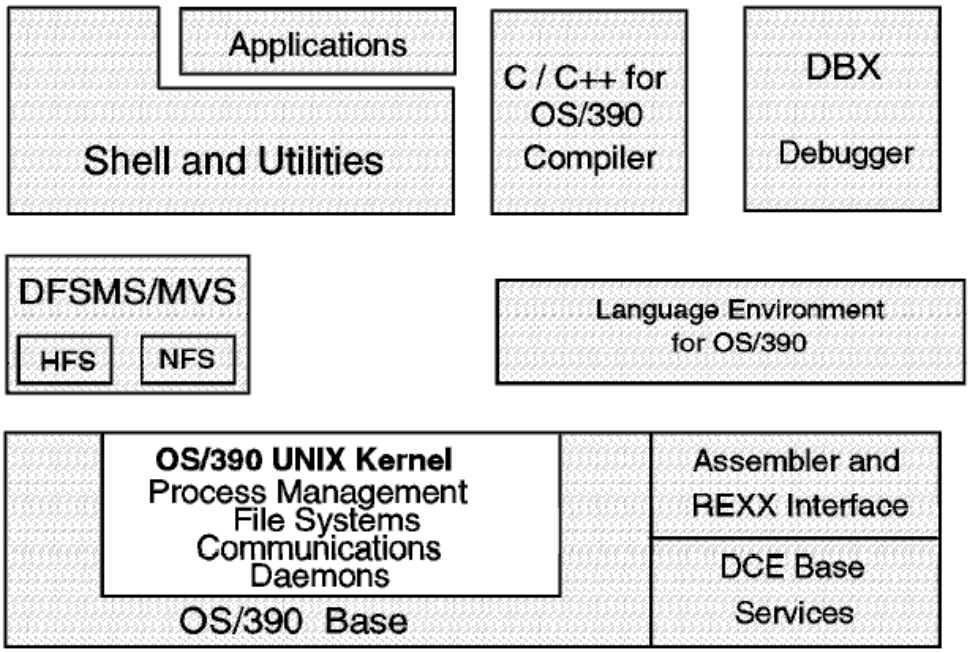

Abbildung 3.2: UNIX System Services

31-Bit Adressierung). Moderne zSeries-Rechner unterstützen zusätzlich 64-Bit Adressierungsmöglichkeiten. Die resultierende Größe des Address Space beträgt somit 16 Exabytes.

Das Konzept des virtuellen Adreßraums existiert auch unter allen UNIX-Dialekten (einschließlich Linux), jedoch sind virtuelle Adreßräume dort fest mit einem einzigen UNIX-Prozeß verbunden. So ist es unter UNIX nicht möglich, daß sich mehrere Prozesse einen virtuellen Adreßraum teilen. Eine Ausnahme bilden Threads (zum Unterschied zwischen Prozessen und Threads sei auf Abschnitt 3.2.5 verwiesen).

Unter z/OS ist es möglich, daß sich mehrere Prozesse einen virtuellen Adreßraum teilen, was bei der Implementierung von Mehrprozeß-Anwendungen berücksichtigt werden muß. Am deutlichsten wird diese z/OS-spezifische Einrichtung bei der Verwendung des `spawn()` Systemaufrufs, siehe Abschnitt 3.2.4.

Sieht man von den Unterschieden zwischen Prozessen und Threads ab, so sind für die Verwendung mehrerer Ausführungseinheiten in einem virtuellen Adreßraum dieselben Vorsichtsmaßnahmen zu treffen. In [ACSR02] heißt es dazu “When the parent and child processes are sharing the same address space, special consideration must be given to the resources that are shared in this environment.” So muß man unter anderem darauf achten, daß ein Prozeß (beziehungsweise ein Thread) nicht unkontrolliert Teile des Speicherbereichs eines anderen überschreibt, denn sonst kann das aufrufende Programm fehlerhafte Ergebnisse produzieren oder abstürzen. Weitere Beispiele für solche “special considerations” finden sich in der genannten Literaturreferenz.

Tabelle 3.1 auf Seite 21 gibt einen Überblick über die verschiedenen Möglich-

keiten der Kombination von virtuellem Adreßraum und Ausführungseinheit. Für eine Erklärung der verwendeten Begriffe *Prozeß*, *Task* und *Thread* sei auf die folgenden Unterabschnitte verwiesen.

Konzept	z/OS	UNIX
1 Prozeß in 1 Adreßraum	1 Address Space mit 1 (kernel-level) Task	1 Prozeß
Mehrere Prozesse in 1 Adreßraum	1 Address Space mit mehreren (kernel-level) Tasks und mehreren PIDs	-
Mehrere Threads in 1 Adreßraum	1 Address Space mit mehreren (kernel-level) Tasks und mehreren TIDs	Mehrere (kernel-level) Threads
Mehrere Threads in 1 Adreßraum	CICS	Mehrere (user-level) Threads

Tabelle 3.1: Virtuelle Adreßräume und Ausführungseinheiten

Als *kernel-level* Threads bezeichnet man solche Threads, welche dem Kernel des Betriebssystems bekannt sind und dadurch vom Scheduler kontrolliert werden. Im Gegensatz dazu sind *user-level* Threads dem Kernel nicht bekannt. Das Scheduling muß in diesem Fall durch das Programm selbst im Benutzermodus erfolgen.

3.2.3 Prozesse und Tasks

Unter OS/390 wurde der Begriff des *Prozeß* mit den UNIX System Services eingeführt. Ein UNIX System Services Prozeß entspricht der Kombination aus einem Address Space und einem sogenannten *Task Control Block* (TCB)[3]. Eine Task ist unter z/OS die kleinste Ausführungseinheit, auf englisch “smallest dispatchable unit of work”. Sie wird durch einen ihr zugeordneten Control Block - dem TCB - repräsentiert. Die kleinste Ausführungseinheit unter UNIX sind die in Abschnitt 3.2.5 vorgestellten Threads.

Neu angelegte Address Spaces, in denen zu einem späteren Zeitpunkt z/OS UNIX-Prozesse gestartet werden sollen, sind zunächst gewöhnliche z/OS Address Spaces und werden vom Betriebssystem entsprechend behandelt. Beginnt die erste Task in einem Address Space, Teile der z/OS UNIX System Services zu benutzen[4], so wird die Task ge-*dubbed*. Aus der UNIX-Perspektive bedeutet dies, daß die Task für den z/OS UNIX-Kernel als neuer UNIX-Prozeß gekennzeichnet ist.

Beim Dub-Vorgang wird zuerst überprüft, ob für den ausführenden Benutzer ein UNIX-Segment definiert ist. Ist für den Benutzer kein UNIX-Segment definiert, schlägt

[3] Sowie einiger zusätzlicher Kontrollblöcke, die für diese Betrachtung jedoch vernachlässigt werden können.

[4] Dies entspricht dem Aufruf eines z/OS UNIX Service, z.B. BPX1RED für `read()`.

die Dub-Anfrage fehl. Anschließend werden dem Prozeß eine *Prozeß Identifizierung* (PID) zugewiesen und verschiedene Kontrollblöcke angelegt. Eine Übersicht über diese Kontrollblöcke ist in [Coms02] zu finden.

3.2.4 `fork()`, `exec()` und `spawn()`

Unter UNIX existiert der `fork()` Systemaufruf, mit dem ein neuer Kind-Prozeß gestartet werden kann. Durch `exec()` wird das laufende Prozeß-Image mit einem anderen ausführbaren Programm überschrieben. In der Regel wird `exec()` dazu benutzt, nach einem `fork()` Aufruf ein anderes Programm auszuführen als dasjenige, welches den `fork()` Aufruf beinhaltet. Der `spawn()` Aufruf kombiniert `fork()` und `exec()` in einem Systemaufruf.

Benutzt man `fork()` unter z/OS UNIX, so wird wie bei anderen UNIX-Betriebssystemen ein neuer Adreßraum erzeugt. Da durch den `fork()` Aufruf selbst feststeht, daß der virtuelle Adreßraum Teile der UNIX System Services benutzt, hat sich die Redewendung vom Dubben des *Address Space* durchgesetzt[5].

Der `spawn()` Aufruf verhält sich unter z/OS UNIX in dieser Hinsicht anders als ein `fork()` mit anschließendem `exec()`, da man zur Laufzeit festlegen kann, ob ein neuer Address Space angelegt werden soll oder nicht. Mit anderen Worten kann das Programm, das durch den `spawn()` Aufruf geladen wird, im gleichen Address Space ausgeführt werden wie dasjenige Programm, welches den `spawn()` Aufruf beinhaltet. Dieses Verhalten wird durch eine Umgebungsvariable festgelegt: `_BPX_SHAREAS`. Gültige Werte sind `YES`, `NO`, `MUST` und `REUSE`. Eine Beschreibung dieser Werte befindet sich [ACSR02], Kapitel 2, “Callable services descriptions“, unter “spawn (BPX1SPN)”.

3.2.5 Threads

Unter einem Thread versteht man zunächst einen sequentiellen Kontrollfluß durch ein Programm. In traditionellen UNIX-Systemen waren lediglich Prozesse implementiert, wobei jeder Prozeß einen (einzigen) sequentiellen Kontrollfluß besaß, mit anderen Worten einen *Thread Of Control*.

Das Programmieren mit mehreren Threads - auch *Multithreaded Programming* genannt - unterscheidet sich von dem Ansatz, mehrere UNIX-Prozesse zu benutzen, durch folgendes: Alle Threads teilen sich den gleichen virtuellen Adreßraum, sowie einige andere Prozeß-spezifische Systemressourcen, wie zum Beispiel die *File Descriptor Table* (Tabelle der Dateideskriptoren) und die Umgebungsvariablen. Deshalb werden Threads auch als *Lightweight Processes* (LWP) bezeichnet, da ihr privater Kontext im Vergleich zu einem Prozeß erheblich reduziert ist. Multithreaded Programming ermöglicht es einem Programm, gleichzeitig mehrere Kontrollflüsse zu besitzen, daher ist es eine Form des *Parallel Programming*.

[5] Man beachte: In Abschnitt 3.2.3 wurde das Dubbing für *Tasks* vorgestellt.

Unter z/OS UNIX wird jeder Thread wie ein Prozeß durch einen TCB repräsentiert. Im Gegensatz zu einem Prozeß erhält ein Thread keine PID, sondern eine *Thread Identifizierung* (TID). Diese kann beispielsweise mit Hilfe des `ps` Kommandozeilentools sichtbar gemacht werden (siehe das entsprechende Kapitel in [USSCR02]). Dies führt zu folgender Aussage: Die z/OS UNIX-Threads, die im übrigen dem POSIX-Standard folgen, sind dem MVS *Subtasking* sehr ähnlich. Es existiert jedoch ein Unterschied: MVS Subtasks werden in der Regel dazu benutzt, Maschinencode auszuführen, der auch unabhängig von der Mutter-Task ausgeführt werden kann. Unter Verwendung von UNIX-Terminologie sind Subtasks also eigenständige Programme (*Executables*). Alle POSIX-Threads eines Prozesses sind im Gegensatz dazu Teil desselben *Load Module*.

Java-Threads werden von der Java Virtual Machine auf Betriebssystem-Threads abgebildet. Unter z/OS UNIX bedeutet dies, daß Java-Threads POSIX-Threads sind. Somit wird jedem Java-Thread ein TCB zugeordnet und vom Scheduler im Betriebssystem wie ein POSIX-Thread behandelt.

Unter Linux sind Threads momentan (Kernel 2.4.x) nach dem *One-to-One* Modell implementiert. Das bedeutet, daß jeder Thread im Kernel auf einen Prozeß abgebildet wird. Jeder Thread wird durch eine PID repräsentiert, teilt sich jedoch den virtuellen Adreßraum und die anderen erwähnten Systemressourcen mit dem Prozeß, zu dem er gehört, sowie mit den anderen Threads, die ebenfalls zu diesem Prozeß gehören. Diese Vorgehensweise beinhaltet einige Vorteile, wie zum Beispiel die einfache und robuste Implementierung, da die meiste kritische Arbeit in den Kernel-Scheduler verlagert ist. Dabei wird jeder Thread vom Scheduler wie ein regulärer Prozeß behandelt, siehe die FAQ bei [LinuxThreads]. Der größte Nachteil sind die teuren Kontextwechsel bei *Mutex* und *Condition* Operationen, die dadurch zustande kommen, daß diese Operationen im Kernel durchgeführt werden müssen.

Die *Next Generation POSIX Threads* (NGPT), die bei IBM entwickelt wurden (siehe [NGPT]), verfolgen einen anderen Ansatz. Es wird ein M:N Modell eingeführt, bei dem M Threads im Benutzermodus auf N Threads im Kernel abgebildet werden. Der Nachteil bei diesem in vielen kommerziellen UNIX-Systemen enthaltenen Ansatz ist die Komplexität der Implementierung.

Unter Linux werden Java-Threads ebenfalls auf Betriebssystem-Threads abgebildet. Ein Unterschied der Java VMs unter Linux und z/OS besteht in der soeben beschriebenen Thread-Implementierung unter Linux. Java-Threads erscheinen somit unter Linux im Kernel als Prozesse mit eigener PID.

3.2.6 XPLink

Der letzte Unterschied, der in die Betriebssystemebene einzuordnen ist, ist die Verwendung von *OS/390 Extra Performance Linkage* (XPLink) unter z/OS ab dem IBM SDK for z/OS, Java 2 Technology Edition, Version 1.4. XPLink ist eine neue Linkerkonvention für OS/390. Sie wurde entworfen, um Performance-Steigerungen für Programme zu erreichen, die eine große Anzahl an Funktions- bzw. Methodenaufrufen aufwei-

sen, wie zum Beispiel C und C++ Programme. Dies wird durch ein neues Stacklayout und durch eine Veränderung der Registerkonventionen erreicht. Im einzelnen führen folgende Auswirkungen dieser Veränderungen zu einer Steigerung der Performance:

- schnelleres Entdecken von *Stack Overflows*
- schnelleres Speichern von Registern
- schnellere Allokation von funktionslokalem Speicher
- verbesserte Registerallokation im *Funktionskörper* (*function body*)
- verbesserter Gebrauch von Registern für Funktionsargumente und -rückgabewerte

Bei speziell angepaßten Codefragmenten erreicht man dadurch eine Performance-Steigerung von über 30 Prozent. In synthetischen Tests mit COBOL-C Mischanwendungen, die einen hohen Anteil an sehr kleinen C-Funktionen aufweisen, erreicht man Performance-Steigerungen von über 40 Prozent. Programme mit sehr großen Funktionen oder einer geringen Häufigkeit an Funktionsaufrufen können von XPLink nicht profitieren. Technische Details zu XPLink sind in [XPLink00] beschrieben.

3.3 Java Virtual Machine

Der Hauptunterschied zwischen den beiden Java Virtual Machines besteht in der in Abschnitt 2.3.8 erwähnten und in Kapitel 4 beschriebenen Technologie der *Persistent Reusable Java Virtual Machines*. Da sich Kapitel 4 ausführlich mit dieser Thematik beschäftigt, wird an dieser Stelle nicht näher darauf eingegangen.

Ein weiterer Unterschied zwischen den beiden Java Virtual Machines sind die unterschiedlichen *Default Encodings*. Während die Java VM unter Linux ASCII benutzt (ISO-8859-1), verwendet die Java VM unter z/OS standardmäßig EBCDIC (CP1047), um zum Beispiel Zeichenketten in eine Datei zu schreiben. Erwähnenswert in diesem Zusammenhang ist, daß ca. 60 Prozent aller geschäftsrelevanten Daten weltweit in EBCDIC und nicht im ASCII-Format vorliegen.

Dieser Unterschied tritt dann zutage, wenn eine Anwendung eine Ausgabe auf einer Plattform erzeugt, und diese Ausgabe auf einer anderen Plattform ausgewertet wird. Java-Programme, die implizit von einer ASCII-Kodierung ausgehen oder ASCII-Konstanten “hartkodiert” enthalten[6], müssen genau untersucht werden. In der Regel muß man hierauf nicht achten, da Zeichenketten-Konstanten in Java in Unicode repräsentiert werden.

[6]Ein Beispiel hierfür ist `byte[] myByteArray = { 0x61 }; String myString = new String(myByteArray);` für einen `String`, der den Buchstaben `'a'` enthält.

Anders stellt sich die Situation dar, wenn man das *Java Native Interface* (JNI) verwendet. Da die Java VM die Zeichenketten in Unicode kodiert, muß man die Java `Strings`, wenn man sie beispielsweise unter Linux in einem C-Programm weiterverarbeiten will, zunächst mit der JNI-Funktion `GetStringUTFChars()` nach UTF-8 konvertieren. Weiterhin ist durch Überprüfungen im Programm sicherzustellen, daß die resultierende UTF-8 Zeichenkette ausschließlich 7-Bit ASCII-Zeichen enthält. Anschließend kann sie für den Aufruf einer C-Funktion wie zum Beispiel `strcmp()` benutzt werden. Unter z/OS müßte man die so gewonnenen Zeichenketten noch einmal konvertieren (mit Hilfe der `atoe()` Funktion nach EBCDIC). Dies ist nicht notwendig, da im JDK für z/OS eine JNI-Funktion vorhanden ist, welche Java `Strings` direkt in EBCDIC umwandeln kann: `GetStringPlatform()`. Diese JNI-Funktion ist z/OS-spezifisch und somit unter anderen Plattformen nicht verfügbar.

Kapitel 4

Architektur der PRJVM

Wie in Abschnitt 2.2.6 geschildert wurde, existiert die weit verbreitete Meinung, Java sei zu langsam für den Einsatz in Transaktionsverarbeitungssystemen. Diese stellen jedoch extrem hohe Ansprüche - sowohl an die Zuverlässigkeit als auch an die Performance der Transaktionen, da sie das Rückgrat vieler moderner e-business Lösungen bilden. Dabei spielt die Programmiersprache, die zur Implementierung der Transaktionen benutzt wird, eine entscheidende Rolle.

In diesem Kapitel wird die Architektur eines neuen Ansatzes vorgestellt, welcher die Performance von Java in Transaktionsverarbeitungssystemen deutlich erhöht: Die *Persistent Reusable Java Virtual Machines* (PRJVM) Technologie, beschrieben in [Borm01] und [PRJVM01]. Kernideen dieses neuen Konzepts sind die Möglichkeit, die Java VM in einen Anfangszustand zurückzusetzen (zu *resetten*), die Aufteilung in System-, Middleware- und Applikationsklassen und eine sehr schnelle Garbage Collection.

Abschnitt 4.1 gibt einen kurzen Überblick über die PRJVM, während der Rest dieses Kapitels die Einzelheiten der Implementierung diskutiert: Die Reset-Fähigkeit (Abschnitt 4.2), die Rolle des Java Native Interface (Abschnitt 4.3), die *Trusted Middleware* und *Application* Klassen (Abschnitt 4.4), die *Split Heaps* und Heap-spezifische Garbage Collection (Abschnitt 4.5) und schließlich die Aufgabe der *Class Loader* (Abschnitt 4.6).

4.1 Überblick

Die in [Borm01] vorgestellte Implementierung der PRJVM basiert auf einem Prototypen, der von Dillenberger *et al.* entworfen und am Ende des Artikels [Dill00] vorgestellt wurde. Der Prototyp enthielt bereits den Großteil der Eigenschaften der PRJVM, auch wenn sie zum Teil noch anders benannt wurden[1]. Diese Eigenschaften werden nun kurz eingeführt, mit Verweisen auf diejenigen Abschnitte, welche tiefergehende Informationen zu den einzelnen Techniken enthalten.

[1] So wurde zum Beispiel die *Master* JVM als *resource-owning* JVM bezeichnet, s.u.

Um die in der Einleitung erwähnte Zuverlässigkeit zu erreichen, wird in der PRJVM jede Transaktion in einer eigenen Java Virtual Machine ausgeführt. Dies resultiert in einem hohen Grad an Isolation. Eine andere Möglichkeit ist durch die parallele Ausführung mehrerer Transaktionen in einer Java VM unter Verwendung mehrerer Java-Threads gegeben. Dieser Ansatz kann in Transaktionsverarbeitungssystemen nicht benutzt werden, da ein Absturz der Java VM den Verlust aller gerade in der Java VM ausführenden Transaktionen zur Folge hat. Außerdem wird durch den erstgenannten Ansatz vermieden, daß sich die Transaktionen gegenseitig beeinflussen. Ein Beispiel hierfür ist folgende Situation: Transaktion 1 könnte einen globalen Zustand der Java VM verändern (beispielsweise durch das Überschreiben einer globalen statischen Variable in einer Systemklasse). Die folgende Transaktion 2 wäre dann diesem veränderten Zustand ausgesetzt. Diese Situation kann zu einem unvorhersehbaren Verhalten der Transaktion führen, im schlimmsten Fall könnte Transaktion 2 fehlschlagen. Auch das Laden von nativen Bibliotheken ist problematisch, da Fehler in diesen Bibliotheken von der Java VM nicht abgefangen werden können. Da diese Bibliotheken in Maschinencode vorliegen, ist eine Bytecode-Überprüfung durch die Java VM ausgeschlossen.

Abgesehen von der Möglichkeit, eine einzige Java Virtual Machine im *resettable* Modus (siehe Abschnitt 4.2) zu starten, kann man in einem z/OS Address Space[2] ein sogenanntes *JVM Set* anlegen. Alle Java VMs teilen sich dabei einen *System Heap* (siehe Abschnitt 4.5). Dadurch wird die Startzeit einer einzelnen Java VM erheblich verkürzt, da später gestartete Java VMs den Großteil der Java-Systemklassen bereits im System Heap vorfinden. Außerdem reduziert diese Anordnung den Hauptspeicherbedarf, da der Speicher für die Systemklassen nur einmal alloziert werden muß. Die erste Java VM übernimmt dabei die Rolle der *Master JVM* und kontrolliert das JVM Set in zweierlei Hinsicht:

- Sie stellt den System Heap zur Verfügung, welcher von allen *Worker JVM*s benutzt wird.
- Sie richtet die *Class Loader* Umgebung ein, die zum Laden der *Primordial* und *Shareable* Klassen (siehe Abschnitt 4.4) benötigt wird.

Hat die Master JVM diese Arbeiten verrichtet, ist ihre Aufgabe im JVM Set praktisch erledigt. Nachdem sie den System Heap und diejenigen Java VM Optionen, welche für das JVM Set relevant sind (zum Beispiel die verschiedenen Class Loader-Einstellungen), externalisiert hat, wird die eigentliche (transaktionsbezogene) Arbeit den Worker JVMs übertragen. Diese neue Eigenschaft der Java Virtual Machine unter z/OS wird durch eine Java VM Option kontrolliert: `-Xjvmset`. Warum dies zum Starten eines JVM Sets nicht auf der Kommandozeile benutzt werden kann, wird in Abschnitt 4.3 erläutert.

[2] Man erinnere sich an Abschnitt 3.2.2, “Address Spaces”.

4.2 Reset-Fähigkeit

Das Vorgehensmodell, eine Transaktion isoliert in einer eigenen Java VM auszuführen, hat die Notwendigkeit zur Folge, die Java VM nach jeder Transaktion in einen "sauberen" Ausgangszustand zu überführen. Dies kann - wie in Abschnitt 2.2.6 erwähnt - entweder dadurch erreicht werden, daß man die Java VM nach jeder Transaktion neu startet oder durch die in der PRJVM implementierte Fähigkeit der Java VM, nach einer erfolgreichen Transaktion einen *Reset* durchzuführen. Diese Eigenschaft wird in den eingangs erwähnten Literaturreferenzen auch als *Serial Reusability* bezeichnet. Der Begriff *seriell* ist deshalb berechtigt, weil in *einer* Java VM zu *einem* Zeitpunkt nur *eine* Transaktion ausgeführt wird.

Aus der Sicht eines Anwendungsentwicklers ist die Reset-Fähigkeit in der PRJVM durch eine neue Java Native Interface Funktion realisiert: `ResetJavaVM()`. Aktiviert wird diese Erweiterung wie bei einem JVM Set durch eine zusätzliche Java VM Option: `-Xresettable`. Die einzelnen Schritte, die bei einem Reset durchgeführt werden, sind:

- Der Middleware wird die Möglichkeit eingeräumt, sich selbst zu resetten (siehe Abschnitt 4.4) und es wird überprüft, ob die Java VM zurückgesetzt werden kann.
- Die *Application Class Loader* (siehe Abschnitt 4.6) werden dereferenziert.
- Der eigentliche Reset wird durchgeführt: Der *Transient Heap* (siehe Abschnitt 4.5) wird einer Garbage Collection unterzogen.
- Die Application Class Loader werden neu instanziiert.

Als erstes muß nach einer Transaktion sichergestellt werden, daß keine Java `Exceptions` ausstehen. Dieses wird typischerweise von einem sogenannten *Launcher Subsystem* erledigt (mehr dazu in Abschnitt 4.3). Danach werden in den Middleware-Klassen spezielle Methoden aufgerufen, in denen die "Aufräumarbeiten" für die Middleware stattfinden.

Bevor der eigentliche Reset ausgeführt werden kann, ist es notwendig zu überprüfen, ob der Zustand der Java VM dies zuläßt. Eine Java VM kann unter Umständen nach einer Transaktion als *dirty* markiert werden. Intuitiv führen all diejenigen Vorgänge zu solch einem "unsauberen" Zustand, welche die nachfolgende Transaktion in irgendeiner Weise beeinflussen könnten. Ein Beispiel für solche als *Unresettable Events* bezeichneten Ereignisse ist das Überschreiben globaler statischer Variablen in Java-Systemklassen oder das Laden von nativen Bibliotheken. Verallgemeinert kann man sagen, daß hierbei ungefähr die gleichen Restriktionen zur Geltung kommen, wie sie für Enterprise JavaBeans festgelegt wurden. So dürfen zum Beispiel von den Application-Klassen keine Java-Threads gestartet werden. Eine detaillierte Auflistung aller Einschränkungen befindet sich in [PRJVM01], Kapitel 5, "Developing applications".

Eine weitere Bedingung für einen erfolgreichen Reset ist, daß keine Referenzen von persistenten Objekten in den Transient Heap zu Application-Objekten existieren. Unter *persistenten* Objekten versteht man dabei solche Objekte, die sich im System Heap oder im Middleware Heap befinden. Deshalb ist es für Middleware-Objekte unbedingt notwendig, alle Referenzen zu Application-Objekten zu entfernen (siehe auch Abschnitt 4.5).

4.3 Die Rolle des Java Native Interface

Um die Erweiterungen der PRJVM benutzen zu können, ist es notwendig, ein sogenanntes *Launcher Subsystem* zu entwickeln. Der Pseudocode für solch ein Launcher-Programm sieht wie folgt aus (stark verkürzt):

```
...
javaVMOptions[i].optionString = “-Xresettable”;
...
jniResult = JNI_CreateJavaVM(javaVM, jniEnvironment,
                             javaVMInitArguments);
...
while (resetFailed == 0) {
   ...
   getWorkForJavaVM(className, methodName, arguments);
   myClass = FindClass(jniEnvironment, className);
   myMethod = GetStaticMethodID(jniEnvironment, myClass,
                                methodName, ...);
   CallStaticVoidMethod(jniEnvironment, myClass,
                        myMethod, arguments);
   ...
   /* check for Java Exceptions */
   ...
   resetFailed = ResetJavaVM(javaVM);
}
...
```

Abbildung 4.1: Minimales Launcher Subsystem

Wie man im Code erkennen kann, wird dabei in besonderem Maße ein bestimmter Teil des Java Native Interface benutzt, das sogenannte *Invocation Interface*. In der Regel verwendet man das Java Native Interface, um aus Java-Applikationen heraus native Programme (zum Beispiel C-Programme) aufzurufen. Bei einem Launcher Subsystem für die PRJVM muß man jedoch den umgekehrten Weg gehen (hierbei wird das Invocation Interface eingesetzt): Die Java Virtual Machine wird aus einem C-Programm heraus initialisiert. Anschließend können an die Java VM Daten zur Ausführung übergeben werden.

Experimente während der Entwicklung des Online-Banking Systems haben gezeigt, daß ein Reset auch aus einem Java-Programm heraus möglich ist[3]. Will man jedoch ein JVM Set initialisieren, muß man wie in Abbildung 4.1 angedeutet verfahren. Dies liegt daran, daß bei der Initialisierung der Worker JVMs ein sogenanntes *Token* übergeben werden muß, das einen Verweis auf die von der Master JVM externalisierte Information darstellt. Ein Launcher Subsystem zu schreiben ist die einzige Möglichkeit, dieses Token auszulesen.

Um an dieser Stelle keinen falschen Verdacht aufkommen zu lassen, muß erwähnt werden, daß die Persistent Reusable Java Virtual Machines Technologie eine *fully compliant* Java-Implementierung ist. Das bedeutet, daß die Java VM auch "ganz gewöhnlich" genutzt werden kann, ohne die Reset-Funktionalität und die Verwendung eines Launcher Subsystems. Etwas einfach formuliert kann man weiterhin alle bisherigen Java-Applikationen auf der Kommandozeile mittels `java MyClass <arguments>` starten. Will man die neuen Techniken benutzen, so kommt man nicht umhin, das Java Native Interface einzusetzen.

4.4 *Trusted Middleware* und *Application* Klassen

Damit der in Abschnitt 4.2 beschriebene Reset schnell durchgeführt werden kann, ist es notwendig, daß die zu einer Transaktion gehörige Datenmenge so gering wie möglich gehalten wird. Diese Daten müssen gelöscht werden, damit die Folgetransaktion eine "saubere" Java VM vorfindet. Die Trennung in Daten, die zur Transaktion gehören, und solche, die über eine Transaktion hinaus existieren dürfen, wird durch die Unterscheidung in *System*-, *Middleware*- und *Applikationsklassen* realisiert. Zu jeder dieser Klassenkategorien gehört ein Abschnitt im Java Heap, was im nächsten Abschnitt dargestellt wird.

Zu den Systemklassen (auch *Primordial* Klassen genannt) zählen die elementaren Java-Klassen wie `String`, `Thread`, `Integer` usw., sowie diejenigen, die durch Javas *Standard Extensions* Mechanismus geladen werden. Diese Klassen sind für die gesamte Lebensdauer einer Java VM gültig, da ihre Class Loader niemals dereferenziert werden.

Die Middleware-Klassen (auch als *Trusted Middleware* bezeichnet) sind innerhalb der Java VM (abgesehen von den Systemklassen) diejenigen Klassen, denen man vertraut (daher der Begriff *Trusted* Middleware). Middleware-Klassen dürfen auch solche Aktionen durchführen, welche die Java VM in einen *dirty* Zustand versetzen, wenn sie von *Application* Klassen initiiert werden. Middleware-Klassen sind Teil der Infrastruktur eines Transaktionsverarbeitungssystems und dürfen ihren inneren Zustand über die Grenzen einer Transaktion hinaus aufrechterhalten. Ein typisches Beispiel für solche Middleware-Klassen sind die Klassen des SQLJ/JDBC-Treibers für das IBM DB2 Datenbankmanagementsystem.

[3] Hierfür muß man den "gewöhnlichen" JNI-Weg gehen.

Für Middleware-Klassen ist es möglich, aber nicht zwingend vorgeschrieben, zwei spezielle Methoden zu implementieren:

- `private static boolean ibmJVMTidyUp();`
- `private static void ibmJVMReinitialize();`

Die `ibmJVMTidyUp()` Methode wird von der PRJVM bei einem Reset für alle Middleware-Klassen aufgerufen, die bei der Ausführung der vorhergehenden Transaktion referenziert wurden. Dies räumt einer Middleware-Klasse die Möglichkeit ein, zwischen zwei Transaktionen "aufzuräumen". In erster Linie müssen dabei Referenzen zum Transient Heap entfernt werden. Weiterhin müssen auch solche Ressourcen freigegeben werden, welche von der Middleware-Klasse angefordert wurden und für die nächste Transaktion nicht mehr zur Verfügung stehen dürfen.

Die Aufgabe der `ibmJVMReinitialize()` Methode ist es, einer Middleware-Klasse die Option zur Verfügung zu stellen, sich für die Ausführung einer neuen Transaktion vorzubereiten. So kann eine Middleware-Klasse in dieser Methode Systemressourcen für die neue Transaktion anfordern, globale statische Variablen neu initialisieren usw.

Die letzte Kategorie sind die *Application* Klassen. Diese sind "normale" Java-Klassen, die in einem Transaktionsverarbeitungssystem die eigentlichen Transaktionen darstellen. Hier wird derjenige Teil der *Business Logic* ausgeführt, dessen Parameter für jede Transaktion einzigartig sind und nicht in die Middleware verlagert werden kann. So sollten alle Daten, die aufgrund der ACID-Eigenschaften nach einer erfolgreichen Transaktion gelöscht werden müssen, in die Application-Klassen hinein modelliert werden. Typischerweise benutzen Application-Klassen Teile der Middleware, um Daten in Enterprise Information Systems zu aktualisieren. Dies liegt unter anderem an Performance-Gründen, was am charakteristischen Beispiel eines Connection Pools für Datenbankverbindungen ersichtlich ist. Diese Einrichtungen besitzen implizit einen Zustand und müssen daher in der Middleware positioniert werden, da nur diese ihren Zustand über eine Transaktion hinaus aufrechterhalten darf.

4.5 *Split Heaps* und Heap-spezifische Garbage Collection

Ein weiteres Konzept in der PRJVM ist es, die Java-Objekte auf verschiedene Heaps zu verteilen, wodurch *a priori* eine Aufteilung in transaktionsbezogene (und somit kurzlebige) und langlebigere Daten vorgenommen wird. Dies ist - wie im letzten Abschnitt geschildert - notwendig, um den Reset zwischen zwei Transaktionen so schnell wie möglich durchzuführen. Die Lebensdauer der verschiedenen Objekte wird durch eine eigene *Garbage Collection Policy* für jeden Abschnitt des Heaps festgelegt. Abbildung 4.2 auf Seite 34 stammt aus [PRJVM01] und verdeutlicht die Aufteilung in

die verschiedenen Abschnitte sowie die unterschiedlichen Garbage Collection Vorgehensweisen für jeden Teil des Heaps.

Der *System Heap* enthält nur solche Objekte, die über die gesamte Lebensdauer der Java VM hinweg existieren. Dazu zählen die Klassenobjekte[4] für die Systemklassen und die *Shareable* Middleware-Klassen. Ein separater Teil des System Heap, der *Application-class* System Heap enthält die sogenannten *Shareable* Application-Klassen. *Shareable* bedeutet dabei, daß diese Klassen von allen Java VMs in einem JVM Set gemeinsam benutzt werden können und deshalb nur einmal in den entsprechenden Teil des Heaps geladen werden müssen. Zwar sind auch *Nonshareable* Application-Klassen denkbar, in [PRJVM01] heißt es jedoch "It is anticipated that most, if not all, application classes will be loaded as shareable using the SAC loader." Der System Heap wird niemals einer Garbage Collection unterzogen, da die Objekte für die gesamte Lebensdauer einer Java VM gültig sind. Die statischen Initialisierungsblöcke der Klassen im System Heap werden nur einmal aufgerufen (da die Klassen nur einmal geladen werden). Diese Blöcke müssen in denjenigen Klassen, welche im *Application-class* System Heap liegen, nach jedem Reset durchlaufen werden. Dies ist notwendig, damit die Shareable Application-Klassen in einen definierten Ausgangszustand versetzt werden.

Der *Nonsystem Heap* ist in den *Middleware Heap* und den *Transient Heap* aufgeteilt, die (wie in Abbildung 4.2 angedeutet) einander entgegenwachsen. Der Middleware Heap enthält *Nonshareable* Middleware-Klassen, Middleware-Objekte und solche Objekte, die im Kontext von Middleware-Klassen erzeugt wurden, wie zum Beispiel Primordial und Array Objekte. Typischerweise ist die Allokationsrate in diesem Bereich des Heap nicht besonders hoch, so daß die PRJVM hier ein Standardverfahren für die Garbage Collection einsetzt[5]. Die Garbage Collection für den Middleware Heap wird nicht automatisch nach jedem Reset durchgeführt, sondern muß vom Launcher Subsystem in regelmäßigen Abständen initiiert werden.

Im Transient Heap befinden sich schließlich diejenigen Objekte, deren Gültigkeit mit dem Ende einer Transaktion erlischt. Beispiele für solche Objekte sind Application-Objekte, Nonshareable Application-Klassen und solche Objekte, die von Application-Klassen erzeugt und nicht im Middleware Heap plaziert wurden. Am Ende einer Transaktion sollten alle diese Objekte von den persistenteren Teilen des Heaps aus nicht mehr erreichbar sein. Die Garbage Collection *Policy* für diesen Teil des Heaps ist recht einfach: Er wird bei jedem `ResetJavaVM()` vollständig gelöscht. Damit dies gelingt, ist es wie geschildert unbedingt notwendig, daß keine Referenzen in diesen Teil des Heaps mehr existieren. Sollten am Ende einer Transaktion doch Referenzen übrig sein, muß die PRJVM in einer teuren Operation (`TraceForDirty()`) feststellen, ob eine dieser Referenzen noch aktiv ist. Wenn der Algorithmus eine aktive Referenz findet, wird die Java VM als *dirty* markiert.

[4] Es gilt der Sinnspruch "Everything in Java is an Object": Erfolgreich geladene Klassen sind für den Anwendungsentwickler durch Objekte vom Typ `Class` erreichbar.

[5] das sogenannte *Mark/Sweep* Verfahren

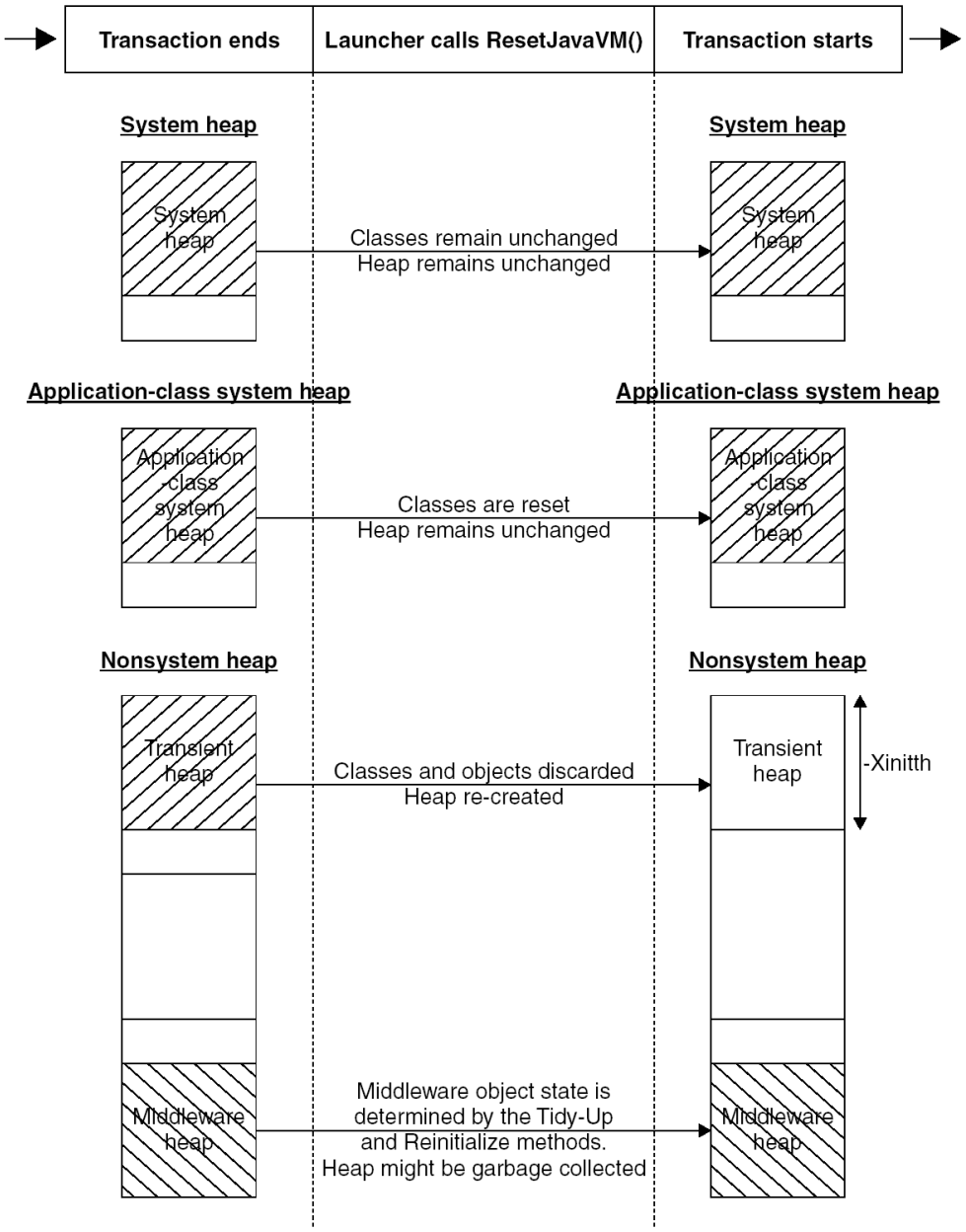

Abbildung 4.2: Split Heaps

Das vorgestellte Split Heaps Design ist dem in einigen Java Virtual Machines realisierten *Generational Garbage Collection* Ansatz ähnlich. Bei diesem wird der Heap ebenfalls in verschiedene Bereiche unterteilt, die als *Generations* bezeichnet werden. Der in der Regel *Nursery*[6] genannte Teil enthält dabei neu instanziierte Objekte, die nach einiger Zeit in die älteren Abschnitte des Heaps verlagert werden. Objekte, die über einen sehr langen Zeitraum existieren, kommen in einen speziellen Teil des Heaps, der nicht mehr einer regelmäßigen Garbage Collection unterzogen wird. Im Vergleich zu diesem Ansatz liegt ein Vorteil der PRJVM darin, daß man die erwartete Lebensdauer der Objekte *a priori* bestimmen kann (durch die im nächsten Abschnitt vorgestellte Technik verschiedener Class Loader). Somit wird ein Verschieben der Objekte im Speicher unnötig. Außerdem fehlt in der Generational Garbage Collection Technik der hohe Grad an Isolation, den die PRJVM bietet.

4.6 Class Loader

Ein Sachverhalt, der in der Regel bei der Entwicklung von Java-Applikationen nicht direkt zutage tritt, ist folgender: Jede Java-Klasse wird von einem *Class Loader* in die Java Virtual Machine geladen. Bei der PRJVM ist es entscheidend, welcher Class Loader den Ladevorgang für eine Klasse übernimmt, denn durch die Wahl des Class Loaders wird der Status der Java-Klasse in der PRJVM bestimmt. So wird eine Klasse beispielsweise zu einer Middleware-Klasse, indem sie durch den *Trusted Middleware Class Loader* (TMC) geladen wird. Der andere Class Loader, der einzigartig für die PRJVM ist, ist der *Shareable Application Class Loader* (SAC).

```
Bootstrap
\
 Standard Extensions
 \
  Trusted Middleware (*)
  \
   Shareable Applications (*)
   \
    Nonshareable Applications
```

Abbildung 4.3: Class Loader Hierarchie

Die PRJVM erweitert den Java2 Class Loader Mechanismus, wie Abbildung 4.3 verdeutlicht. Dabei sind die mit (*) gekennzeichneten Class Loader die in der PRJVM eingeführten Erweiterungen. In dem Class Loader Modell wird nach dem *Parent Delegation* Prinzip verfahren: Wenn ein Class Loader eine Klasse laden soll[7], überprüft er zuerst, ob er sie bereits geladen hat. Wenn dies nicht zutrifft, überträgt er die Anfrage

[6] auf deutsch: Kinderstube

[7] Das Standard-Beispiel hierfür: Ein Objekt einer noch nicht geladenen Klasse wird instanziiert.

an seinen Vater usw. Kommt die Anfrage an der Wurzel an, versucht jeder Class Loader auf dem Weg nach unten, die Klasse zu laden. Der Vorgang ist erfolgreich, wenn die Klasse gefunden und vom richtigen Class Loader geladen wurde.

Wie in jeder anderen Java Virtual Machine ist in der PRJVM der Standard Class Loader für einen ausführenden Thread (der sogenannte *Context Class Loader*) der Nonshareable Application Class Loader. Dadurch wird das Laden aller Kategorien von Klassen ermöglicht, da die Suche ganz unten in der Hierarchie beginnt.

Zu jedem Class Loader gehört ein eigener `CLASSPATH`, so daß die Einteilung in Middleware- und Application-Klassen auch eine Frage der Konfiguration ist (und nicht nur eine der Implementierung). Die verschiedenen Pfade werden der PRJVM in je einer Java *System Property* übergeben:

- `-Dibm.jvm.trusted.middleware.class.path=<path>` für den Trusted Middleware Class Loader
- `-Dibm.jvm.shareable.application.class.path=<path>` für den Shareable Application Class Loader
- `-Djava.class.path=<path>` für den Nonshareable Application Class Loader (dieses entspricht dem “normalen” Java `CLASSPATH`)

Der TMC und der SAC sind Teil der PRJVM, ein Class Loader für Nonshareable Middleware-Klassen ist nicht enthalten, da diese Art von Klassen eher als exotisch zu bezeichnen ist. Bei Bedarf können jedoch eigene Class Loader implementiert werden. Kapitel 4 in [PRJVM01], “Writing middleware”, enthält einen Abschnitt hierzu (“Creating a class loader”).

Abschließend sei noch der Bezug zu der in 4.2 vorgestellten Reset-Fähigkeit der PRJVM hergestellt: Die Systemklassen werden durch den Bootstrap Class Loader geladen, die Middleware-Klassen durch den Trusted Middleware Class Loader. Beide Arten von Klassen bleiben gültig, bis die Java VM beendet wird. Die Shareable Application-Klassen werden durch den SAC geladen und werden ungültig, wenn eine Transaktion zu Ende ist. Bei jedem Reset werden sie zurückgesetzt und ihre statischen Initialisierungsblöcke werden durchlaufen, wenn sie im Zuge einer neuen Transaktion zum ersten Mal referenziert werden. So wird für die neue Transaktion dieselbe Ausgangssituation geschaffen, die entsteht, wenn die Klassen aus dem Speicher entfernt und wieder neu geladen werden.

Kapitel 5

BOBS - Basic Online-Banking System

Um die während der Architekturanalyse ermittelten Unterschiede möglichst praxisrelevant darzustellen, mußte ein Weg gefunden werden, diese anschaulich zu demonstrieren. Da die gesamte Architektur der PRJVM und die ihr zugrunde liegenden Designentscheidungen nur auf ein Ziel ausgerichtet sind (Transaktionsverarbeitung) fiel die Entscheidung auf den Entwurf und die Implementierung eines Online-Banking Systems, da Online-Banking traditionell zu den Hauptanwendungsgebieten der kommerziellen Transaktionsverarbeitung zählt. Das vorliegende Kapitel stellt das Ergebnis dieser Anstrengungen vor: BOBS, das *Basic Online-Banking System*.

Abschnitt 5.1 gibt einen Überblick über BOBS, während sich die restlichen Abschnitte mit Details des Designs und der Implementierung (Abschnitt 5.3ff), sowie dem Zusammenhang des Online-Banking Systems mit der TPC-A Spezifikation beschäftigen (Abschnitt 5.2).

5.1 Überblick

Da mit BOBS primär nur die Java-spezifischen Unterschiede zwischen z/OS und Linux aufgezeigt werden sollen, wurde an vielen Stellen Wert auf ein möglichst einfaches Design gelegt, um den Einfluß von Seiteneffekten so gering wie möglich zu halten. So existieren in BOBS beispielsweise keine Authentifizierungsmechanismen, was in einem realen Online-Banking System völlig undenkbar ist. Um dennoch einen gewissen Grad an Funktionalität aufweisen zu können, orientiert sich BOBS an einer Referenzimplementierung für Online-Banking Systeme, die in der Zwischenzeit von ihren Entwicklern als *obsolete* eingestuft wurde: Der TPC-A Benchmark des *Transaction Processing Performance Council* (TPC), dessen Spezifizierung öffentlich zugänglich ist, siehe [TPC94] und Abschnitt 5.2. Mehr Informationen zum TPC erhält man auf dessen World Wide Web Präsenz (siehe [TPC]).

BOBS besteht aus einer Menge an verschiedenartigen Software-Komponenten, die zusammengenommen ein grundlegendes[1] Online-Banking nach dem *Debit/Credit* Ver-

[1] Daher kommt der Begriff *Basic* in BOBS.

fahren ermöglichen. Um einen Überblick über BOBS zu geben, ist eine *bottom-up* Herangehensweise am besten geeignet. Abbildung 5.1 skizziert die Einordnung von BOBS in die logischen Schichten eines Rechnersystems am Beispiel eines zSeries-Rechners mit dem z/OS-Betriebssystem. Würde man dieselbe Abbildung für Linux erstellen, so wären einige Unterschiede zu erkennen: Die mit *UNIX System Services* und *Language Environment* beschrifteten Teile wären überflüssig, da deren Funktionalität im Linux-Betriebssystem selbst (im Kernel bzw. in der Standard C Library) zu finden ist. An der Stelle der PRJVM steht das IBM Java Developer Kit for Linux, Java2 Technology Edition und das verwendete Datenbankmanagementsystem ist *DB2 Universal Database for Linux* anstelle von *DB2 for z/OS and OS/390*.

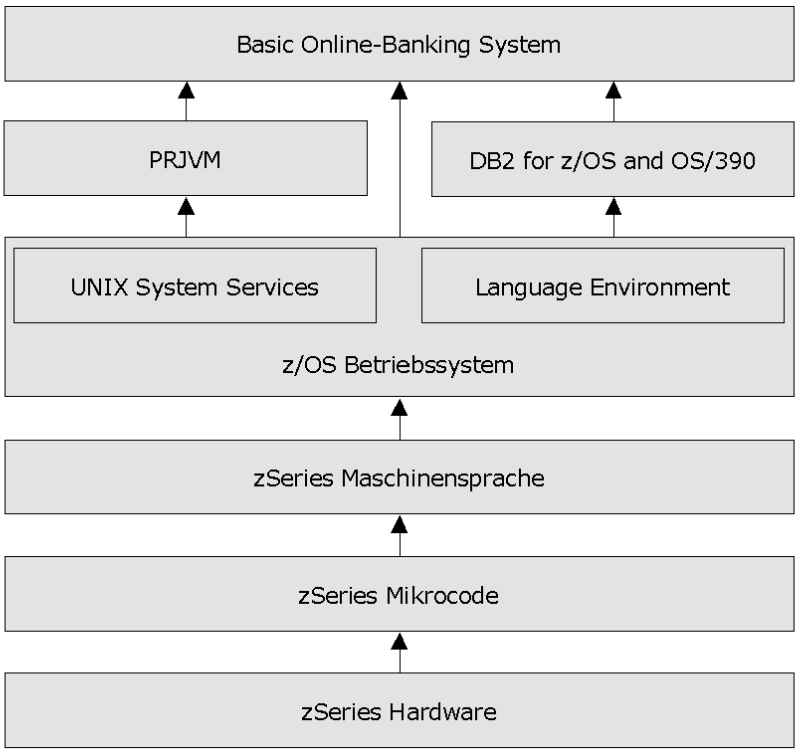

Abbildung 5.1: Einordnung von BOBS

Am unteren Ende der Hierarchie findet man die Hardware und andere Komponenten[2], auf die an dieser Stelle nicht näher eingegangen werden soll. Darüber befindet sich das Betriebssystem z/OS mit seinen Komponenten z/OS UNIX System Services und dem Language Environment. BOBS greift an vielen Stellen auf die Funktionalität der beiden genannten z/OS-Komponenten zurück. Auf dem Betriebssystem setzt das Datenbankmanagementsystem *DB2 for z/OS and OS/390* auf, in dem BOBS seine

[2]Die Abbildung zeigt zwar einen zSeries-Rechner, für einen IA-32 Intel Architecture basierten Rechner ergibt sich jedoch dasselbe Bild.

Online-Banking bezogenen Daten hinterlegt. Die Relationen in der DB2-Datenbank sind nach der TPC-A Vorgabe modelliert.

Ungefähr auf derselben Ebene wie DB2 ist die Persistent Reusable Java Virtual Machines Technologie anzusiedeln, deren transaktionsrelevante Eigenschaften wie zum Beispiel die Reset-Fähigkeit (siehe Abschnitt 4.2) in BOBS benutzt werden. Die PRJVM wiederum benutzt die z/OS UNIX System Services und das Language Environment, wie es praktisch alle Hochsprachen-Programme tun, die unter z/OS erstellt werden.

Ganz oben in der Hierarchie steht schließlich BOBS selbst, das ein “hybrides” Programm darstellt, bestehend aus in C und Java geschriebenen Komponenten. Dabei sind es die Java-Komponenten, welche auf die in DB2 hinterlegten Daten zugreifen. Die Kommunikation mit einem angeschlossenen Netzwerk wird von einer in C geschriebenen Komponente realisiert (TCP/IP-Server). Eine Auflistung der einzelnen Komponenten und ihrer Aufgaben befindet sich in Abschnitt 5.3.

5.2 Bezug zu TPC-A

Bevor die Architektur des Online-Banking Systems eingeführt wird, sei der Zusammenhang mit der TPC-A Spezifikation hergestellt. Die vollständige Bezeichnung dieses Benchmarks lautet *TPC Benchmark A, Standard Specification*. Er wurde zur Performance-Messung von sogenannten *Online Transaction Processing* (OLTP) Systemen entwickelt. Dabei liegt das Hauptaugenmerk auf solchen OLTP-Systemen, die im Datenbank-Teil besonders updateintensiv sind.

Der Durchsatz wird in *tpsA* gemessen, was für *Transaktionen pro Sekunde im TPC Benchmark A* steht. Zum Einsatz kommt dabei lediglich ein Transaktionstyp[3], der zur Lasterzeugung auf dem *System Under Test* (SUT) dient. Die Transaktionen werden von einem *Remote Terminal Emulator* (RTE) initiiert, welcher sogenannte *Online-Terminals* simuliert.

Der Benchmark beruht auf dem Modell einer hypothetischen Bank, die eine oder mehrere Zweigstellen (auf englisch: *branches*) hat. Zu jeder Zweigstelle gehören mehrere Kassierer (*tellers*). Die Bank hat Kunden, von denen jeder ein eigenes Konto (*account*) führt. In der Datenbank werden die Bilanzen für jede Entität (Zweigstelle, Kassierer und Konto) festgehalten. Der einzige Transaktionstyp im TPC-A spezifiziert die für einen Einzahlungs- bzw. Auszahlungsvorgang notwendigen Schritte (*Debit/Credit*). Die Transaktion wird von einem Kassierer einer Zweigstelle ausgeführt, wobei der Kassierer eines der erwähnten Online-Terminals bedient.

Der TPC-A legt fest, wie die einzelnen Schritte der Transaktion zu verrichten sind. BOBS hält sich an diese Vorgaben, wie man in Abschnitt 5.4, “Transaktionsfluß”, nachvollziehen kann. Auch der Aufbau der Datenbank ist vorgeschrieben, siehe dazu Abschnitt 5.6. Weiterhin definiert die TPC-A Spezifikation, wie die Antwortzeit

[3]Im Gegensatz zu beispielsweise dem TPC-C, welcher einen Mix aus verschiedenen Typen von Transaktionen beinhaltet.

(*Response Time*) und der Durchsatz (*Throughput*) zu messen sind. Auch darauf wurde bei der Entwicklung des Online-Banking Systems geachtet.

Nichtsdestotrotz haben die in Kapitel 8 und im Anhang genannten Ergebnisse nicht die Einheit tpsA, da zur Veröffentlichung eines offiziellen Benchmark-Ergebnisses die vollständige Prüfung der verwendeten Hard- und Software durch einen sogenannten *Auditor* nötig wäre. Dieser muß vom Transaction Processing Council zertifiziert sein. Der für diese Prozedur notwendige zeitliche und finanzielle Aufwand liegt außerhalb dessen, was im Rahmen einer Diplomarbeit im Bereich des Möglichen liegt.

5.3 Architektur

Um die Funktionsweisen der einzelnen Komponenten und ihr Zusammenspiel besser zu verstehen, ist es hilfreich, sich zunächst mit Hilfe von Abbildung 5.2 einen Überblick über die Architektur von BOBS zu verschaffen.

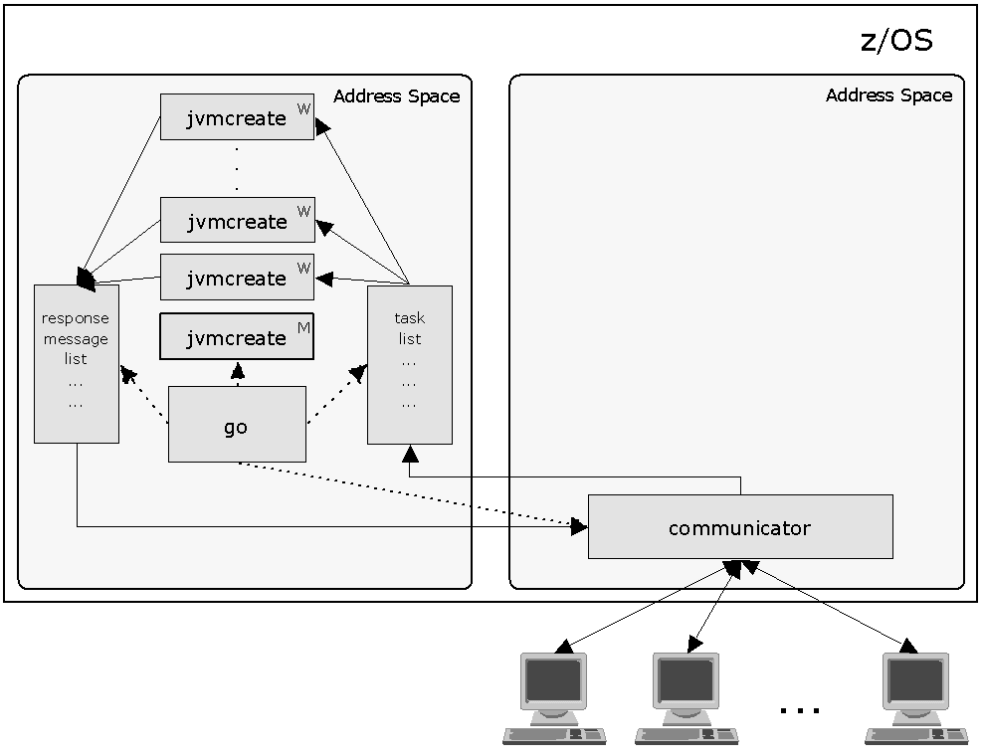

Abbildung 5.2: Die Architektur des Basic Online-Banking Systems

Die abgerundeten, großen Rechtecke stellen die Address Space Grenzen innerhalb des z/OS-Betriebssystems dar. Die Terminals in der unteren rechten Ecke stehen für die simulierten Online-Terminals. Die durchgezogenen Linien zwischen den Terminals und der mit *communicator* beschrifteten Komponente stehen für die von außen

kommenden TCP/IP-Verbindungen. Über diese werden die Parameter für die Transaktionen übertragen. Die eckigen, kleineren Rechtecke, die mit *go*, *jvmcreate* und *communicator* beschriftet sind, stellen einzelne z/OS UNIX System Services Prozesse dar. Die anderen beiden Rechtecke (*task list* und *response message list*) sind zwei auf C-Arrays basierende Listen, die in einem Shared Memory Segment liegen und durch mehrere Semaphore geschützt und verwaltet werden. Shared Memory Segmente sind Bereiche des realen Hauptspeichers, die mittels Seitenabbildungsverfahren in diejenigen Adreßräume eingeblendet werden, die auf diese Bereiche zugreifen. Aus diesem Grund müßten die beiden genannten Listen auch in den Address Space des *communicators* eingetragen werden. Um die Übersichtlichkeit zu wahren und nicht den Eindruck zu erwecken, es handele sich um verschiedene Listen, wurde in der Abbildung auf sie verzichtet.

Bevor BOBS gestartet werden kann, ist es notwendig, eine z/OS UNIX System Services Konsole zu öffnen und einige Language Environment Einstellungen vorzunehmen (siehe Abschnitt 7.1.1 auf Seite 58). Anschließend wird das Online-Banking System durch die in C geschriebene Komponente *go* gestartet. Durch die Language Environment Einstellungen wird *go* in einem von der Konsole unabhängigen z/OS Address Space gestartet und mittels des einsetzenden *Dubbing* Vorgangs[4] zu einem z/OS UNIX System Services Prozeß.

go dient lediglich dazu, das Online-Banking System zu initialisieren, angedeutet durch die gestrichelten Linien in Abbildung 5.2. Hat es diese Arbeit verrichtet, ist seine Aufgabe in BOBS praktisch erledigt. Es wartet nur noch mittels der C `wait()` Funktion auf das Ende eines Kind-Prozesses und dessen Status-Information. Bevor es in diesen Wartezustand verfällt, hat *go* folgende Arbeiten verrichtet:

- Auslesen der BOBS-Einstellungen aus der `go.prp` Konfigurationsdatei
- Anlegen einer globalen Logdatei für die Fehlerdiagnose
- Erzeugen des Shared Memory Segments und des Semaphore Sets
- Starten des *communicator* Prozesses
- Hochfahren des JVM Sets (eine Master und konfigurierbar viele Worker JVMs)

In Abbildung 5.2 ist ersichtlich, daß dem *communicator* Prozeß ein eigener Address Space zugewiesen wird. Diese Designentscheidung wurde getroffen, damit die vom *communicator* verbrauchte CPU-Zeit bei der Performance-Analyse getrennt von der CPU-Zeit des Java-Teils ausgewertet werden kann. Der *communicator* Prozeß ist in sich multithreaded programmiert: Für jede ankommende TCP/IP-Verbindung wird ein eigener POSIX-Thread unter z/OS UNIX System Services gestartet. Zusätzlich existieren konfigurierbar viele als *Response Processor* bezeichnete Threads innerhalb des

[4] Man erinnere sich an Abschnitt 3.2.4, “fork(), exec() und spawn()”.

communicator Prozesses, deren Aufgabe in Abschnitt 5.4 beschrieben wird. Die Aufgabe des *communicator* Prozesses als ganzem ist es, die eingehenden Parameter für die Transaktionen anzunehmen, an die PRJVM zur Ausführung zu übergeben und die Ergebnisse zurückzuschicken (mehr dazu im nächsten Abschnitt).

Ein *jvmcreate* Prozeß implementiert die Funktionalität eines Launcher Subsystems. Dieser Prozeß startet je nach Konfiguration entweder eine Master oder eine Worker JVM. In der Abbildung ist die Master JVM mit einem *M* markiert, die Worker JVMs mit einem *W*. Das in Abschnitt 4.3 beschriebene Token wird nach der Initialisierung der Master JVM in einen Bereich des Shared Memory Segments kopiert, so daß es von den Worker JVMs ausgelesen werden kann. Die Worker JVMs sind diejenigen z/OS UNIX-Prozesse, in denen die transaktionsbezogene Arbeit des Online-Banking Systems verrichtet wird. Sie werten die einzelnen Parameter der Transaktionen aus, führen mittels SQLJ (wahlweise auch JDBC) die Updates in der Datenbank durch und erzeugen die Rückgabewerte für die Terminals. Nicht zu sehen in der Abbildung sind die dabei involvierten Java-Klassen, sie werden in einem eigenen Abschnitt (siehe Abschnitt 5.5) vorgestellt.

Abschließend sei bemerkt, daß diese Architektur nicht von der TPC-A Spezifikation vorgeschrieben ist, sondern eine Eigenentwicklung des Autors darstellt. Nichtsdestotrotz wurde beim Entwurf darauf geachtet, daß das Online-Banking System die Regeln des TPC-A nicht verletzt.

5.4 Transaktionsfluß

Um die Arbeitsweise von BOBS zu verdeutlichen, wird in diesem Abschnitt der Weg einer Transaktion durch das Online-Banking System beschrieben und die Aufgaben der einzelnen dabei involvierten Komponenten erläutert. Als Orientierung dienen dabei die in Abbildung 5.2 eingezeichneten, durchgezogenen Linien zwischen den einzelnen Komponenten des Online-Banking Systems, die an einem Ende mit einem Pfeil versehen sind.

Der Transaktionsfluß beginnt damit, daß der *communicator* eine von außen ankommende TCP/IP-Verbindung auf einem zuvor konfigurierten Port akzeptiert. Zur weiteren Bearbeitung der Anfrage wird ein neuer POSIX-Thread gestartet und die `main()` Routine des *communicator* wartet auf die nächste Verbindung. Der neu gestartete Thread liest den Transaktions-Request aus, der folgendem Muster entsprechen muß:

```
request := className ':' classArguments
```

Der `className` ist dabei der in ISO-8859-1 kodierte Name derjenigen Java-Klasse, die zu einem späteren Zeitpunkt in der PRJVM als Transaktion abgearbeitet wird[5].

[5] In [Borm01] wird die Kombination aus Klasse, Methode und Argumenten für die Methode als *Unit Of Work* (UOW) bezeichnet. Mit anderen Worten besteht eine Transaktion aus dem Abarbeiten einer Methode einer Java-Klasse.

Nach dem Klassennamen folgen die durch einen Doppelpunkt getrennten Argumente für die Transaktion. Die Argumente sind eine ebenfalls in ISO-8859-1 kodierte Zeichenkette, welche die für eine nach dem TPC-A Standard modellierte Transaktion benötigten Parameter enthalten (Account ID, Teller ID, Branch ID und Delta, mehr dazu in Abschnitt 5.6). Der Klassenname und die Argumente werden zusammen als neue *task* in das Shared Memory Segment kopiert und der Thread wartet, bis der Klient die nächste Anfrage schickt.

Die *jvmcreate* Prozesse warten (durch ein Semaphore synchronisiert) inaktiv, bis eine Anfrage in die *task list* gestellt wird. Einer der Prozesse erwirbt daraufhin das Recht, exklusiv auf die *task list* zuzugreifen. Er liest die Anfrage aus und gibt die *task list* nach dem Entfernen der Anfrage aus der Liste wieder frei. Nun wird die in `className` festgelegte Java-Klasse mittels einer Java Native Interface Routine lokalisiert und automatisch durch die PRJVM-Mechanismen[6] initialisiert. Eine Konvention in BOBS ist es, daß jede Java-Klasse, die als Transaktion ausgeführt werden soll, folgende Methode implementieren muß:

```
public [final] String doTransaction(String arguments);
```

Dies kann jedoch bei Bedarf in *jvmcreate* abgeändert werden. Nach dem Lokalisieren der Klasse und der `doTransaction()` Methode verzweigt der Kontrollfluß in *jvmcreate* an dieser Stelle für einige Zeit in die PRJVM hinein, indem die `doTransaction()` Methode mit ihrem dazugehörigen Argument[7] aufgerufen wird. Dieser Methodenaufruf ist selbst durch den Aufruf einer Java Native Interface Routine realisiert (`CallObjectMethod()`). War der bisherige Teil des Transaktionsflusses praktisch vollständig unter der Kontrolle der in C geschriebenen Teile des Online-Banking Systems, so wird der Transaktionsfluß nun von einer in Java geschriebenen Klasse bestimmt.

Eine Instanz dieser Klasse, die in Anlehnung an die TPC-A Spezifikation `TPCATransaction` genannt wurde, extrahiert die Parameter aus der `arguments` Zeichenkette, welche nach folgendem Muster aufgebaut sein muß:

```
arguments := accountID ':' tellerID ':' branchID ':'
             delta ':'
```

Nach diesem Auslesevorgang werden bestimmte, durch die Middleware festgelegte Methoden in einem Middleware-Objekt aufgerufen. Diese Methoden nehmen die Updates in der Datenbank vor. Abhängig vom Erfolg oder Mißerfolg der Datenbankoperationen wird ein Java `String` erzeugt und als Rückgabewert der `doTransaction()` Methode übergeben. An dieser Stelle kehrt der Kontrollfluß in den C-Teil des

[6]Gemeint ist hier das Abarbeiten der statischen Initialisierungsblöcke der Application-Klassen, wenn diese bei einer neuen Transaktion zum ersten Mal referenziert werden.

[7]Das einzige Argument der `doTransaction()` Methode ist ein Java `String`, welcher aus der `classArguments` Zeichenkette aus dem Shared Memory Segment konstruiert wird.

Online-Banking Systems zurück. Der Java `String`, welchen die `doTransaction()` Methode zurückliefert, wird ausgelesen und durch ein Semaphore synchronisiert als Antwort in die *response message list* kopiert.

Wie bereits in Abschnitt 5.3 kurz angedeutet, besitzt der *communicator* Prozeß einen oder mehrere als *Response Processor* bezeichnete Threads. Deren Aufgabe ist es, solange (inaktiv) zu warten, bis eine Antwort in die *response message list* gestellt wird. Wie bei der *task list* erwirbt nun einer dieser *Response Processor* Threads das Recht, exklusiv auf die *response message list* zuzugreifen. Dieser Thread liest die Antwort aus, entfernt sie aus der Liste und gibt die *response message list* wieder frei. Hier erfolgt der letzte Schritt des Transaktionsflusses: Die Antwort wird von besagtem *Response Processor* Thread an das Terminal zurückgeschickt, von dem die ursprüngliche Transaktionsanfrage kam. Sobald das Terminal die Antwort ausgewertet hat, kann der Transaktionsfluß erneut beginnen.

5.5 Middleware- und Applikationsklassen

In Abschnitt 4.4 wurde der Begriff der *Trusted Middleware* und der *Application* Klassen eingeführt. Der vorliegende Abschnitt stellt diejenigen Java-Klassen des Basic Online-Banking Systems vor, welche dieser Kategorie angehören.

5.5.1 Die `MiddleWare` Klasse

Die einzige Java-Klasse, welche der Trusted Middleware zuzuordnen ist, ist die `MiddleWare` Klasse. Sie existiert in zwei Varianten: `MiddleWare.sqlj` repräsentiert die SQLJ-Implementierung der für BOBS notwendigen Datenbankoperationen und `MiddleWare.java` ist die auf JDBC basierende Variante, welche dieselbe Funktionalität bietet. Beide lesen zur Laufzeit die Systemumgebung aus, bauen anhand der daraus gewonnenen Informationen die Datenbankverbindung auf und erhalten diese bis zum Herunterfahren des Online-Banking Systems aufrecht. Das Aufrechterhalten der Datenbankverbindung ist deshalb notwendig, da die Herstellung der Verbindung zur Datenbank (abgesehen von der `prepare statement` Anweisung) die teuerste Operation im Zusammenhang mit SQLJ und JDBC ist. Dies haben Performance-Untersuchungen ergeben, die sich mit SQLJ und JDBC beschäftigt haben. Ein gut dokumentiertes Beispiel für solch eine Untersuchung findet sich in dem IBM Redbook [Brun02], Kapitel 4, “Java support”. Das Aufrechterhalten der Datenbankverbindung ist auch der Grund, weshalb diese Funktionalität in die Middleware verlagert wurde[8].

Neben dem Aufbauen der Verbindung zur Datenbank stellt die `MiddleWare` Klasse auch Methoden für die Datenbankoperationen zur Verfügung, die von der TPC-A Spezifikation gefordert sind. Dies sind im einzelnen folgende:

[8] Man erinnere sich an Abschnitt 4.4, “*Trusted Middleware* und *Application* Klassen”.

- `public boolean updateAccount()` zum Aktualisieren des Kontostands eines Kunden
- `public BigDecimal selectABalance()` zum Auslesen des Kontostands eines Kunden
- `public boolean updateTeller()` zum Aktualisieren des Umsatzes eines Kassierers
- `public boolean updateBranch()` zum Aktualisieren des Umsatzes einer Zweigstelle
- `public boolean updateHistory()` zum Festhalten des Buchungsvorgangs in der History
- `public boolean commit()` zum Festschreiben der Veränderungen an der Datenbank
- `public boolean rollback()` zum Rücksetzen der Veränderungen an der Datenbank

Die `MiddleWare` Klasse enthält keine der in Abschnitt 4.4 vorgestellten speziellen Middleware-Methoden, da die einzige Systemressource, die für eine Transaktion benötigt wird, die Datenbankverbindung ist. Da diese Verbindung aufrechterhalten wird und keine anderen kritischen Systemressourcen benutzt werden, muß sie auch nicht beendet (`ibmJVMTidyUp()`) bzw. neu aufgebaut werden (`ibmJVMReinitialize()`). Auch unterhält die `MiddleWare` Klasse keine Referenzen zu Objekten im Transient Heap, was eine "Aufräum-" Methode zwingend notwendig machen würde.

5.5.2 Die `TPCATransaction` Klasse

In Abschnitt 5.4 kurz vorgestellt, ist die `TPCATransaction` Klasse die einzige, welche in die *Application* Kategorie einzuordnen ist. Ihre Aufgabe ist es, die Parameter einer Transaktion aus dem ihr übergebenen Java `String` auszulesen, auf ihre Gültigkeit hin zu überprüfen und die Datenbankmethoden der Middleware in der richtigen Reihenfolge aufzurufen. Wenn eine der Operationen fehlschlägt, veranlasst die `TPCATransaction` Klasse ein Rollback der bis zu diesem Zeitpunkt vorgenommenen Änderungen. Je nach Art des Fehlers gibt die einzige Methode der Klasse, die `doTransaction()` Methode, eine Meldung zurück, welche die Ursache des Fehlers beschreibt. Im Erfolgsfall gibt sie neben den Parametern der Transaktion den aktuellen Kontostand des betroffenen Kunden zurück.

5.6 Die Datenbank

Die Relationen[9], welche die Online-Banking Daten enthalten, sind in einer DB2-Datenbank zusammengefaßt - sowohl unter z/OS als auch unter Linux. Sie entsprechen der Vorgabe der TPC-A Spezifikation und werden im einzelnen nun kurz vorgestellt. Die dabei verwendete Darstellung verzichtet auf den Relationenbegriff und benutzt den anschaulicheren Begriff der Tabelle.

Spaltenname	Spaltentyp	Genauigkeit / Länge
BID	DECIMAL	9
BBALANCE	DECIMAL	10
FILLER	CHARACTER	88

Tabelle 5.1: Die BRANCHES Tabelle

BID steht für *Branch ID*, BBALANCE für *Branch's Balance* und die mit FILLER bezeichnete Spalte ist in der TPC-A Spezifikation vorgeschrieben, damit eine Zeile einer Tabelle mindestens eine Länge von 100 Bytes besitzt. BID ist fett gedruckt, da es sich bei diesem Wert um den sogenannten Primärschlüssel (*Primary Key)* der Tabelle handelt.

Spaltenname	Spaltentyp	Genauigkeit / Länge
TID	DECIMAL	9
BID	DECIMAL	9
TBALANCE	DECIMAL	10
FILLER	CHARACTER	84

Tabelle 5.2: Die TELLERS Tabelle

In der TELLERS Tabelle steht TID für *Teller ID* und TBALANCE für *Teller's Balance*. BID ist unterstrichen, da es sich hierbei um einen Fremdschlüssel (*Foreign Key*) handelt. Dieser bezieht sich auf einen gültigen Eintrag in der BRANCHES Tabelle. FILLER erfüllt die gleiche Aufgabe wie in der BRANCHES Tabelle.

Spaltenname	Spaltentyp	Genauigkeit / Länge
AID	DECIMAL	9
BID	DECIMAL	9
ABALANCE	DECIMAL	10
FILLER	CHARACTER	84

Tabelle 5.3: Die ACCOUNTS Tabelle

[9]Relationen werden oft aus Gründen der Anschaulichkeit als Tabellen bezeichnet. *Tabelle* steht jedoch im Grunde für die Darstellung einer Ausprägung einer Relation zu einem bestimmten Zeitpunkt.

In der ACCOUNTS Tabelle steht der Primärschlüssel AID für *Account ID* und ABALANCE für *Account Balance*. Die mit BID und FILLER bezeichneten Spalten erfüllen dieselbe Funktionalität wie in der TELLERS Tabelle. Ein Kunde ist einer Zweigstelle[10] und nicht einem Kassierer zugeordnet. Daher ist ein Eintrag für einen Kassierer, also eine Spalte mit einem Fremdschlüssel, der sich auf einen gültigen Eintrag in der TELLERS Tabelle bezieht, nicht vorgesehen.

Spaltenname	Spaltentyp	Genauigkeit / Länge
TID	DECIMAL	9
BID	DECIMAL	9
AID	DECIMAL	9
DELTA	DECIMAL	10
TIME	TIMESTAMP	N/A
FILLER	CHARACTER	22

Tabelle 5.4: Die HISTORY Tabelle

Die letzte Tabelle, die im TPC-A Benchmark vorgesehen ist, ist die HISTORY Tabelle. In ihr wird für jeden Buchungsvorgang die Teller ID, die Branch ID, die Account ID, der Betrag (DELTA) und der Zeitstempel (TIME) festgehalten. Auf die Definition eines Primärschlüssels wurde verzichtet, da der TPC-A Standard einen solchen in der HISTORY Tabelle nicht vorschreibt. Zudem verursacht ein Primärschlüssel zusätzlichen Zeitaufwand beim Einfügen einer neuen Zeile, da das Datenbankmanagementsystem die Einzigartigkeit des Werts sicherstellen muß.

5.7 Umsetzung in reines Java (Pure Java)

Wie in der Einleitung zu Kapitel 3 beschrieben, besteht ein Gesichtspunkt einer Architekturanalyse in der Untersuchung der Performance des betrachteten Systems. Um den zusätzlichen Aufwand (*Overhead*) des Reset-Vorgangs[11] der PRJVM zu untersuchen, wurde das Basic Online-Banking System in einer zweiten Variante implementiert, bei der *ausschließlich* Java (und kein C bzw. Java Native Interface) verwendet wurde - deshalb auch der Ausdruck *rein*. Die folgenden Abschnitte stellen diese Variante und die dabei getroffenen Designentscheidungen vor.

5.7.1 Gemeinsamkeiten

Für ein Terminal, das sich mit dem Online-Banking System verbindet, besteht zunächst kein Unterschied, ob es sich mit der hybriden oder der reinen Variante verbindet. Stellt

[10] repräsentiert durch die mit BID gekennzeichnete Spalte

[11] Sowie der anderen Eigenheiten der PRJVM, siehe hierzu Abschnitt 8.3.6, “Pure Java unter z/OS (Ergebnis D)”.

man sich beide Systeme als je eine *Black Box* vor, so verhalten sie sich von außen betrachtet gleich.

Auch wurde bei der Umsetzung von BOBS in reines Java darauf geachtet, daß die grundlegende Architektur und der Transaktionsfluß bei beiden Versionen gleich ist, um einen möglichst präzisen Vergleich durchführen zu können. So existiert für jede Komponente, die in Abbildung 5.2 auf Seite 40 zu sehen ist, ein entsprechendes in Java geschriebenes Gegenstück: Der in C geschriebene *communicator* entspricht der `Communicator` Klasse, die *go* Komponente der `RunMe` Klasse, die *task list* der `TaskList` Klasse und die *response message list* der `ResponseMessageList` Klasse. Exakt gleich sind die beiden bereits in der hybriden Variante benutzten Java-Klassen `MiddleWare` und `TPCATransaction`. Abbildung 5.3 illustriert diese Aufzählung.

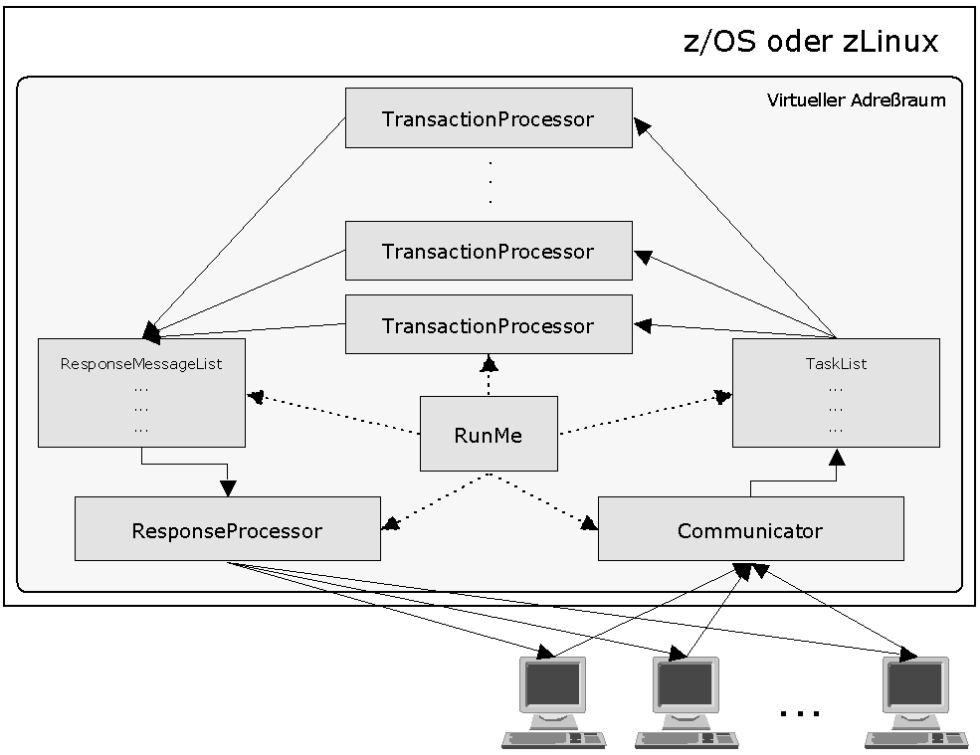

Abbildung 5.3: Die Architektur der reinen Java-Version

Um die Liste der Gemeinsamkeiten zu vervollständigen, sei an dieser Stelle erwähnt, daß die `Communicator` Klasse für jede ankommende TCP/IP-Verbindung einen eigenen Thread startet, welcher die weitere Kommunikation mit dem Klienten übernimmt. Dieses Verhalten entspricht der Vorgehensweise des *communicator* Prozesses der hybriden Variante.

5.7.2 Unterschiede

Da in der reinen Java-Variante auf die Verwendung des Java Native Interface und damit verbundene C-Programme verzichtet wurde, ergeben sich zwangsläufig einige, zum Teil gravierende Unterschiede.

So werden alle Transaktionen, die zu einem Zeitpunkt aktiv sind, in einer einzigen Java Virtual Machine abgearbeitet. Die Ursache für diesen Sachverhalt liegt in Javas Programmiermodell begründet, das eine Interaktion zwischen zwei Java VMs nur mittels Java RMI, CORBA oder Java Sockets zuläßt. Da die Architektur der beiden Versionen so ähnlich wie nur möglich gestaltet wurde, existiert in der reinen Java-Lösung nur ein `TaskList` Objekt, auf das alle `TransactionProcessor` Threads[12] zugreifen. Der Zugriff auf die *task list* in der hybriden Variante erfolgt über ein Shared Memory Segment, was ungleich schneller als eine der genannten Kommunikationsmöglichkeiten ist und somit einen Vergleich nicht aussagefähig erscheinen läßt. Deshalb wurde ein *multithreaded* Ansatz gewählt, der im übrigen auch die gewöhnliche Herangehensweise an Java-Probleme dieser Art darstellt.

Die tiefgreifendste Konsequenz dieser Lösung ist das ungeschützte Abarbeiten der Transaktionen nebeneinander, sowie das Fehlen eines Resets am Ende einer Transaktion. Die reine Java-Lösung kann niemals in einem produktiven Umfeld eingesetzt werden, das höchste Ansprüche an die Transaktionssicherheit stellt. Der Sinn und Zweck dieser Version besteht lediglich in der erwähnten Ermittlung des Overheads, den die PRJVM mit sich bringt.

In Abbildung 5.3 auf Seite 48 sind die bereits genannten Unterschiede deutlich zu erkennen: So erzeugt die reine Java-Version des Basic Online-Banking Systems nur einen einzigen virtuellen Adreßraum[13], da nur ein z/OS UNIX System Services Prozeß (bzw. ein Linux-Prozeß) gestartet wird (die Java Virtual Machine). Die mit `TransactionProcessor` und `Communicator` beschrifteten Objekte sind von `java.lang.Thread` abgeleitet und werden somit von der Java VM zur Laufzeit auf Betriebssystem-Threads abgebildet und (im Falle von z/OS) nicht auf z/OS UNIX System Services Prozesse wie ihre Pendants in Abbildung 5.2 auf Seite 40.

Ein weiterer Unterschied, der sich aus der Tatsache ergibt, daß in der reinen Java-Variante keine Eigenschaften der PRJVM ausgenutzt werden, ist das Fehlen einer Aufteilung in Master und Worker `TransactionProcessor` Threads[14]. Die Dedizierung eines Threads als Master-Thread ist nicht notwendig, da die Funktionalität der Master JVM (beispielsweise das Externalisieren des System Heap) nicht nachgebildet werden muß: Alle `TransactionProcessor` Threads gehören zu einer einzigen Java Virtual Machine und teilen sich somit einen gemeinsamen Heap und die Class Loader. Dadurch wird das in der PRJVM notwendige Hochfahren einer Master-Ausführungseinheit hinfällig.

[12] Ein `TransactionProcessor` Thread bildet die Funktionalität einer Worker JVM nach.

[13] angedeutet durch das abgerundete, große Rechteck

[14] Zur Erinnerung: Die `TransactionProcessor` Threads entsprechen den mit *jvmcreate* und einem *W* beschrifteten Rechtecken in Abbildung 5.2 auf Seite 40.

Weiterhin ist in der Abbildung noch ein `ResponseProcessor` Thread zu sehen, der keine Entsprechung in Abbildung 5.2 auf Seite 40 hat. Dies liegt daran, daß die `ResponseProcessor` Threads in der hybriden Version Teil des *communicator* Prozesses sind und somit aus Gründen der Übersichtlichkeit nicht in der Abbildung dargestellt wurden. Wie bereits in der hybriden Version existiert auch in der reinen Java-Version eine konfigurierbar große Anzahl an `ResponseProcessor` Threads, die jedoch aus denselben Gründen in der Abbildung auf einen Thread reduziert wurden. Somit ist dieser Sachverhalt lediglich in die Kategorie Darstellungs-Unterschied einzuordnen und nicht etwa als Design-Unterschied zu sehen.

Kapitel 6

Microbenchmarks

Wie in der Einleitung zu Kapitel 3 erwähnt, besteht ein Aspekt einer Architekturanalyse eines Systems in der Untersuchung der Performance. Das in Kapitel 5 vorgestellte Basic Online-Banking System bildet die Grundlage für einen Teil der Performance-Untersuchungen. Das Hauptaugenmerk liegt dabei auf der Leistung des Gesamtsystems. Die Java Virtual Machine ist hierbei nur für einen Teil der Gesamtleistung verantwortlich[1]. Will man die Performance der Java Virtual Machine isoliert untersuchen, so eignet sich BOBS nur bedingt. Um diesen Bereich ebenfalls abzudecken, wurden die sogenannten *Microbenchmarks* entwickelt, welche in diesem Kapitel vorgestellt werden.

In Abschnitt 6.1 wird definiert, was im Kontext dieser Diplomarbeit unter dem Begriff Microbenchmark zu verstehen ist und welche Teilaspekte der Java Virtual Machine dadurch abgedeckt werden. Abschnitt 6.2 stellt die einzelnen Microbenchmarks vor und geht auf Eigenheiten ein, die bei ihrer Ausführung beachtet werden müssen.

6.1 Definitionen

Um den Unterschied zwischen den Begriffen *Benchmark* und *Microbenchmark* deutlich zu machen, muß zunächst die Bedeutung des Begriffs Benchmark klargestellt werden. [IEEE90] definiert beispielsweise Benchmark als “A standard against which measurements or comparisons can be made”. Da dies für die spätere Diskussion der Microbenchmarks zu allgemein formuliert ist, sei an dieser Stelle der Begriff Benchmark konkreter wie folgt definiert:

Definition Ein Benchmark ist ein Computerprogramm, welches die Leistung einer Anwendung oder eines Systems in einer reproduzierbaren Art und Weise mißt und dabei eine Metrik verwendet, die einen Vergleich mit anderen Anwendungen bzw. Systemen zuläßt.

[1] Dieser Teil ist zwar nicht unerheblich, es spielen aber auch andere Faktoren wie zum Beispiel das verwendete Datenbankmanagementsystem eine gewichtige Rolle.

Beispiele für Benchmarks, welche die Performance der Java Virtual Machine messen können, sind der SPECjvm98 und der SPECjbb2000 der *Standard Performance Evaluation Corporation* (siehe [SPEC]). Die Bereiche der Java Virtual Machine, welche durch diese Benchmarks abgedeckt werden, sind für die vorliegende Arbeit nicht relevant[2]. Daher wurde die Entscheidung getroffen, nur die Leistung der Kernfunktionalitäten der Java VM auszumessen. Diese Bereiche sind gemäß [Trip01]:

- Methodenaufrufe
- Objekterzeugung
- Locking

Da Benchmarks für diese Bereiche ausschließlich dafür bestimmt sind, spezifische Teilaspekte der Leistung einer Java Virtual Machine zu messen, sind sie Beispiele für *Microbenchmarks*. Anhand dieser Beispiele kann der Begriff Microbenchmark eingeführt werden:

Definition Ein Microbenchmark ist ein Benchmark, der einen spezifischen Teil der Gesamtleistung einer Anwendung oder eines Systems ausmißt.

Obwohl die im Rahmen dieser Diplomarbeit erstellten Microbenchmarks nur sehr spezifische Leistungsmerkmale der Java VM messen, können sie dennoch Anhaltspunkte für die Obergrenzen der Leistung einer Java Virtual Machine geben. Dies wird dadurch erreicht, daß die drei oben erwähnten Teilgebiete die grundlegenden Vorgänge jeder (objektorientierten) Java-Anwendung abdecken.

Kennzeichnend für die objektorientierte Programmierung ist unter anderem eine hohe Anzahl an *Methodenaufrufen*. Insbesondere im Zusammenhang mit J2EE sind oft Methoden zu beobachten, die lediglich aus einer oder zwei Java-Anweisungen bestehen. Meistens handelt es sich dabei um sogenannte `getter` und `setter` Methoden. Aus diesem Grund ist die Anzahl der Methodenaufrufe, die in einem bestimmten Zeitintervall durchgeführt werden können, eine wichtige Kenngröße für die Beurteilung der Performance einer Java VM.

Ebenfalls charakteristisch für die Objektorientierung ist die Kapselung von primitiven Daten (`int`, `double` usw.) in Objekten. Da solche Objekte ständig verworfen und wieder neu instanziiert werden, besteht ein weiteres wichtiges Maß in der Geschwindigkeit, mit der eine Java VM Objekte instanziieren kann (*Objekterzeugung*).

Oftmals kann die zur Verfügung stehende CPU-Zeit von einer Ausführungseinheit (zum Beispiel einem Thread) nicht vollständig ausgenutzt werden. Daher wurde in Java bereits bei der Einführung der Sprache die Möglichkeit geschaffen, in einem Programm mehrere Kontrollflüsse (Threads) zu benutzen. In der Regel müssen die

[2] So sind z.B. im SPECjvm98 verschiedene Benchmarks zu Komprimierungsverfahren enthalten. Diese Benchmarks werden jedoch in der Regel für die Leistungsmessung sogenannter “Number Cruncher” Rechner benutzt und sind in einer Transaktionsverarbeitungsumgebung weniger von Bedeutung.

Threads auf ein Datum sowohl lesend als auch schreibend zugreifen. Daher ist die Geschwindigkeit, mit der die Synchronisation (*Locking*) erfolgt, ebenfalls ein Maß für die Performance einer Java Virtual Machine.

6.2 Die Benchmarks im Detail

Alle Microbenchmarks beruhen auf demselben Prinzip: Die zu messende Operation wird in einer Schleife konfigurierbar oft hintereinander ausgeführt. Vor dem Eintritt in diese *Innere Schleife* und nach dem Austritt aus derselben wird die Zeit gemessen und die Anzahl der Operationen pro Sekunde bestimmt. Bevor der eigentliche Benchmark stattfindet, wird die Zeit gemessen, die für die innere Schleife selbst benötigt wird. Dieser Betrag wird von der Meßzeit abgezogen, um nur diejenige (eigentliche) Zeitdauer zu erhalten, welche für die gemessenen Operationen verbraucht wurde.

Weiterhin ist sicherzustellen, daß die Meßergebnisse nicht von Javas Garbage Collection oder dem Just-In-Time Compiler verfälscht werden. Beide könnten während der Messung CPU-Zeit in Anspruch nehmen. Einer dieser beiden Problemfälle wird dadurch ausgeschlossen, daß die Methoden vor dem Meßvorgang so oft aufgerufen werden, daß sie vom JIT-Compiler zum Zeitpunkt der Messung auf jeden Fall übersetzt sind. Der andere Fall[3] tritt nur beim `AllocateObjects` Microbenchmark auf und spielt daher bei den anderen keine Rolle. Im entsprechenden Abschnitt (siehe Abschnitt 6.2.2 auf Seite 55) wird die Vorgehensweise beschrieben, um bei besagtem Microbenchmark auch dieses Problem auszuschließen.

Jeder Microbenchmark wird zehn Mal hintereinander ausgeführt, um am Ende der Meßreihe ein gesichertes Ergebnis zu erhalten. Weicht eines der Ergebnisse um mehr als fünf Prozent vom arithmetischen Mittel der Meßwerte ab, so ist das Ergebnis als unbrauchbar anzusehen. Diese Vorgehensweise ist die bei Benchmarks übliche Methode, statistisch einwandfreie Resultate zu erhalten.

Außerdem sind die Microbenchmarks so auszuführen, daß eine Einzelmessung mindestens fünf Sekunden dauert. Die hier vorgestellten Microbenchmarks orientieren sich an der *jMocha* Benchmark-Suite von IBM (siehe [jMocha]). Dort wird zwar nur eine Dauer von einer Sekunde gefordert, je höher aber die Dauer einer Einzelmessung gewählt wird, desto gesicherter (und somit reproduzierbarer) ist das Ergebnis.

Alle Microbenchmarks müssen mit der gleichen Java Virtual Machine Konfiguration gestartet werden:

```
$> java -verbose:gc -Xgcpolicy:optthruput -Xnoclassgc
        -Xms32M -Xmx32M <Name des Microbenchmarks>
```

Abbildung 6.1: Gemeinsame Java VM Optionen für die Microbenchmarks

Die `-verbose:gc` Option dient dazu, die Java Garbage Collection zu beobachten und somit deren Einfluß auszuschließen. Die beiden Parameter `-Xgcpolicy:`

[3] also der Einfluß der Garbage Collection

`optthruput` und `-Xnoclassgc` beschleunigen lediglich die Garbage Collection etwas, eine Auswirkung auf die Resultate der Microbenchmarks haben sie jedoch nicht.

Beim `AllocateObjects` Microbenchmark müssen aufgrund der Natur des Benchmarks (Objekte auf dem Java Heap allozieren) die Parameter für die minimale (`-Xms`) und maximale (`-Xmx`) Java Heap Größe angepaßt werden. Setzt man beide Parameter wie in Abbildung 6.1 gezeigt auf denselben Wert, fordert die Java VM während der Ausführung des Benchmarks keinen zusätzlichen Speicher an (und gibt keinen frei). Speicherallokation würde sich negativ auf das Resultat auswirken, da zusätzliche CPU-Zeit verbraucht wird.

6.2.1 Methodenaufrufe

Bei diesem `CallMethods` getauften Microbenchmark werden verschiedene Typen von Java-Methoden aufgerufen. Diese sind im einzelnen in Tabelle 6.1 aufgeführt. Die verwendete Maßeinheit für ein Ergebnis in diesem Benchmark ist *Methodenaufrufe pro Sekunde*.

Methodentyp	Deklaration in Java
Klassenmethode	`static`
Finale Klassenmethode	`final static`
Instanzenmethode	N/A
Finale Instanzenmethode	`final`
Abgeleitete Instanzenmethode	In einer abgeleiteten Klasse wird eine Methode deklariert, welche dieselbe Signatur aufweisen muß wie die abzuleitende Methode der übergeordneten Klasse.
Spezialfall einer abgeleiteten Instanzenmethode	Wie oben, es wird lediglich explizit via `super.<methodName>` die übergeordnete Methode aufgerufen.
Rein virtuelle Instanzenmethode	In Anlehnung an die C++ Namensgebung wird eine Methode in einer abgeleiteten Klasse aufgerufen, die ausschließlich in der übergeordneten Klasse deklariert ist.

Tabelle 6.1: Methodenaufrufe im `CallMethods` Microbenchmark

Der Benchmark wird von der Konsole aus gestartet, wie in Abbildung 6.2 auf Seite 55 dargestellt ist (aus Platzgründen wurden die Java VM Optionen aus Abbildung 6.1 weggelassen).

Wie bereits erwähnt, beeinflußt die Garbage Collection das Ergebnis dieses Microbenchmarks nicht. Es werden lediglich bei der Initialisierung des Benchmarks einige Objekte erzeugt, deren Methoden dann in der Messung benutzt werden. Abgesehen

```
$> java CallMethods <Anzahl der Methodenaufrufe>
```

Abbildung 6.2: Starten des `CallMethods` Microbenchmark

von einigen `String` Objekten, die zur Ausgabe des Ergebnisses auf der Konsole nötig sind, alloziert der `CallMethods` Microbenchmark keine weiteren Objekte zur Ausführungszeit.

6.2.2 Objekterzeugung

Dieser Microbenchmark dient dazu, die Leistung einer Java VM im Hinblick auf die Objekterzeugung zu messen. In Anlehnung an seine Aufgabenstellung wurde er `AllocateObjects` genannt. Um das Verhalten der Java Virtual Machine bei unterschiedlicher Objektgröße zu analysieren, enthält die Klasse, von welcher die Objekte instanziiert werden, einen internen Puffer vom Typ `byte[]`. Beim Starten des Benchmarks kann die Größe dieses internen Puffers wie folgt konfiguriert werden:

```
$> java AllocateObjects <Anzahl der zu allozierenden
       Objekte> <Größe des internen Puffers>
```

Abbildung 6.3: Starten des `AllocateObjects` Microbenchmark

Wie in Abbildung 6.2 wurde auch in Abbildung 6.3 auf die Angabe der gemeinsamen Java VM Argumente verzichtet. Es bleibt zu erwähnen, daß die Java Heap Größe so gewählt werden muß, daß während eines Benchmark-Durchlaufs keine Garbage Collection außer der vom Benchmark selbst initiierten auftritt. Diese von `AllocateObjects` selbst initiierte Garbage Collection muß zwischen zwei Durchläufen durchgeführt werden, damit der Java Heap für den nächsten Durchgang leer ist und wieder mit Objekten besetzt werden kann. Durch die `-verbose:gc` Option wird sichergestellt, daß alle Garbage Collection Vorgänge auf der Konsole ausgegeben werden. Abschließend sei noch die Einheit für das Meßresultat aufgeführt: *Objekte pro Sekunde*.

6.2.3 Threads und Locking

Der letzte Microbenchmark in dieser Reihe wurde `AcquireLocks` benannt. Durch diesen Benchmark wird die Locking-Performance einer Java VM bestimmt. Die Einheit für ein Benchmark-Resultat ist *Locks pro Sekunde*. `AcquireLocks` startet eine konfigurierbare Anzahl an Threads[4], die alle auf eine mit `synchronized` deklarierte Methode eines Objekts zugreifen. Der Zugriff auf diese Methode wird von der Java VM mit einem aus der Informatik bekannten *Monitor* Konstrukt gelöst. Ein Monitor regelt den Zugriff auf eine Ansammlung aus Daten und Prozeduren derart, daß sich

[4] In Anlehnung an ihre Rolle im Benchmark wurden sie mit `LockGrabber` bezeichnet.

zu einem Zeitpunkt nur ein einziger Prozeß im Monitor befinden kann. In dieser Hinsicht erfüllt ein Monitor die Aufgaben eines Locks, jedoch mit dem Unterschied, daß ein Monitor grobgranularer als ein Lock ist: Mit einem Lock kann man einen beliebig kleinen (beziehungsweise großen) Teil eines Programms synchronisieren, ein Monitor in Java verwaltet ein ganzes Objekt. Mit anderen Worten: Es existiert ein Monitor pro Objekt. Für weiterführende Informationen zu dieser Thematik sei auf [Lind99] verwiesen.

Wie beim `CallMethods` Microbenchmark spielt die Garbage Collection hier keine Rolle. Es werden nur die anfänglich allozierten Objekte benutzt und der Benchmark instanziiert während der Messung keine neuen (abgesehen von den zehn sehr kleinen `LockGrabber` Objekten pro Benchmark-Durchlauf). Gestartet wird er von der Kommandozeile wie folgt:

```
$> java AcquireLocks <Anzahl der konkurrierenden Threads>
        <Anzahl der Locks>
```

Abbildung 6.4: Starten des `AcquireLocks` Microbenchmark

Kapitel 7

Best Practices

Neben der Architekturanalyse und der praktischen Demonstration der Unterschiede der Java Virtual Machines unter z/OS und Linux war es ebenfalls eine Zielsetzung dieser Diplomarbeit, die während der Programmierung und den Performance-Tests ermittelten *Best Practices* der jeweiligen Umgebung zu dokumentieren. Das vorliegende Kapitel gibt die Erfahrungen wieder, die während dieser Phasen der Diplomarbeit gemacht wurden, und stellt einige Eigenheiten der verschiedenen Systeme vor.

Die Best Practices sind in unterschiedliche Kategorien aufgeteilt. So findet man in Abschnitt 7.1 Hinweise, die sich auf die Persistent Reusable Java Virtual Machines Technologie beziehen (Konfigurationsparameter und Programmierung). Abschnitt 7.2 enthält die Best Practices, die sich bei der Umsetzung des Online-Banking Systems in reines Java herausstellten (Java-Entwicklung). Abschnitt 7.3 beschäftigt sich mit datenbankspezifischen Optimierungen.

7.1 PRJVM

Die Anwendungsentwicklung für die PRJVM-Technologie läßt sich in zwei Teilbereiche gliedern: Auf der einen Seite steht die Implementierung eines Launcher Subsystems[1], auf der anderen Seite die Programmierung von Java-Anwendungen für die PRJVM. Während die Entwicklung eines Launcher Subsystems nahe am Betriebssystem erfolgt (erkennbar an der Verwendung von Shared Memory und Semaphore-Techniken), ist die Erstellung von Java-Anwendungen für die PRJVM auf einer höheren Abstraktionsstufe anzusiedeln. Um diesem Sachverhalt Rechnung zu tragen, wurde der folgende Text in mehrere Unterabschnitte aufgeteilt.

Die ersten beiden Unterabschnitte behandeln die unteren Schichten: Die Programmierung in C und die Konfiguration des Online-Banking Systems und der PRJVM. Der dritte Unterabschnitt geht auf die erwähnte höhere Stufe in Form der Java-Klassen ein.

[1] Man erinnere sich an Abschnitt 4.3, “Die Rolle des Java Native Interface”.

7.1.1 C und das Java Native Interface

Das Basic Online-Banking System wurde unter einem auf der IA-32 Intel Architecture basierenden Linux entwickelt. Dadurch lag der gravierendste Unterschied bei der Programmierung in C in den verschiedenen Speicher- und Prozeßmodellen, was in Abschnitt 3.2 behandelt wurde.

Eine anfängliche Version des Online-Banking Systems verzichtete auf einen separaten *communicator* Address Space, so daß die gesamte Arbeit in einem (einzigen) Adreßraum verrichtet wurde. Dadurch konnte das Shared Memory Segment unter z/OS so gestaltet werden, daß die Parameter für eine Transaktion nicht in dem Shared Memory Segment selbst, sondern an einem beliebigen Platz im Address Space positioniert wurden. Dieser Platz wurde dem Online-Banking System zur Laufzeit vom z/OS Language Environment zugewiesen. Im Shared Memory Segment wurden lediglich Zeiger auf diese Speicherorte gesetzt, welche die ankommenden Transaktions-Requests enthielten. Dieser Ansatz bietet den Vorteil, daß die *task list* aus Abbildung 5.2 auf Seite 40 nicht überlaufen kann (einzige Ausnahme: der gesamte für den Heap verfügbare Bereich des z/OS Address Space ist bereits vergeben). Trotz eingehender Prüfung des Quellcodes auf *Memory Leaks*[2] gelang es anfänglich nicht, diese Version von BOBS stabil am Laufen zu halten. Erst das Aktivieren eines alternativen Verwaltungsalgorithmus für den Heap, der durch das Language Environment kontrolliert wird, ermöglichte das fehlerfreie Abarbeiten der Requests über mehrere Millionen Transaktionen hinweg. Dieses Aktivieren erfolgt auf einer z/OS UNIX System Services Konsole durch das Setzen einer sogenannten *Language Environment Run-Time Option*.

```
$> export _CEE_RUNOPTS="HEAPPOOLS(ON),"$_CEE_RUNOPTS
```

Abbildung 7.1: Setzen der `HEAPPOOLS` Run-Time Option

In [LECus02] sind alle Run-Time Optionen für das Language Environment beschrieben. Es existiert eine weitere (wichtige) Run-Time Option: `RPTSTG` ("RePorT SToraGe"). Mit Hilfe dieser Option werden die optimalen Speichereinstellungen für eine Anwendung ermittelt. Dazu muß man die Option wie für `HEAPPOOLS` gezeigt auf `ON` setzen und die Anwendung ablaufen lassen. Nach Beendigung der Anwendung erhält man einen Report der einzelnen Speicherbereiche (STACK, HEAP) und kann diese über weitere Run-Time Optionen auf ihre optimalen Werte einstellen.

Weiterhin konnte das Launcher Subsystem nicht mehr gestartet werden, nachdem das Online-Banking System von einem zSeries-Rechner mit einer älteren Version des z/OS-Betriebssystems auf eine neuere Hardware mit einem aktuelleren z/OS kopiert wurde. Der in Abschnitt 3.2.4 beschriebene `spawn()` Aufruf schlug bei jedem Hochfahren des Online-Banking Systems fehl. Er konnte erst wieder benutzt werden, nachdem in einem C `struct` namens `inheritance`, das für diesen Aufruf benötigt wird, die `flags` Komponente explizit auf `0` gesetzt wurde.

[2] auf deutsch: Speicherlecks

Die z/OS C-Funktionen `__atoe()` und `__etoa()` werden zur Konvertierung von ISO-8859-1 nach EBCDIC und umgekehrt benötigt. Diese können nur dann ohne Fehler aufgerufen werden, wenn man die Zeichenketten, auf denen die Funktionen operieren, vorher dynamisch auf dem Heap mittels `malloc()` alloziert. Zu diesem Zweck können auch andere, `malloc()` verwandte Funktionen wie zum Beispiel `strdup()` verwendet werden. Zeichenketten, die als Teil der lokalen Variablen einer C-Funktion deklariert sind, liegen auf dem Stack. Der Versuch einer Konvertierung solcher Zeichenketten mit den genannten Funktionen resultiert in einem Absturz des aufrufenden Programms.

Ferner ist es für spätere Performance-Messungen notwendig, daß das Launcher Subsystem nicht in dem z/OS Address Space gestartet wird, in dem die z/OS UNIX System Services Konsole ausgeführt wird. Nur so können die verbrauchten CPU-Zeiten voneinander getrennt werden. Um dies zu erreichen, muß man die in Abschnitt 3.2.4 eingeführte Umgebungsvariable `_BPX_SHAREAS` benutzen und sie wie in folgender Abbildung setzen:

```
$> export _BPX_SHAREAS=NO
```

Abbildung 7.2: Setzen der `_BPX_SHAREAS` Umgebungsvariable

Die Persistent Reusable Java Virtual Machines Technologie erfordert den Einsatz des Java Native Interface. Dadurch werden sogenannte *Lokale Referenzen* im Launcher Subsystem erzeugt, die am Ende jeder Transaktion mittels der JNI-Funktion `DeleteLocalRef()` freigegeben werden müssen. Dies ist einerseits notwendig, um Memory Leaks zu vermeiden, da die Ressourcen von der Java Virtual Machine nicht automatisch freigegeben werden. Andererseits muß man insbesondere solche lokalen Referenzen freigeben, die auf Klassen bzw. Objekte im Transient Heap[3] zeigen, denn sonst ist die Java Virtual Machine als *dirty* markiert und muß für die Folgetransaktion neu gestartet werden.

7.1.2 Konfiguration des Online-Banking Systems

Bevor die eigentlichen Performance-Messungen mit dem Online-Banking System durchgeführt werden konnten, mußten die Konfigurationsparameter von BOBS (und der PRJVM) hinsichtlich ihres Einflusses auf das Meßergebnis untersucht werden. Es stellte sich heraus, daß einige dieser Einstellungen immense Auswirkungen auf die Durchsatzrate haben.

Ehe auf die einzelnen Parameter eingegangen werden kann, muß die Methodik eingeführt werden, die beim Ermitteln der optimalen Werte für diese Parameter angewandt wurde. Abbildung 7.3 auf Seite 60 stammt aus [Chow03] und zeigt die gängige Methode, welche für diese Art von Tuningprozessen benutzt wird, um das optimale Resultat zu erhalten.

[3] Zur Erinnerung: Der Transient Heap ist eine spezieller Abschnitt des Heaps der PRJVM, siehe Abschnitt 4.5, “Split Heaps und Heap-spezifische Garbage Collection”.

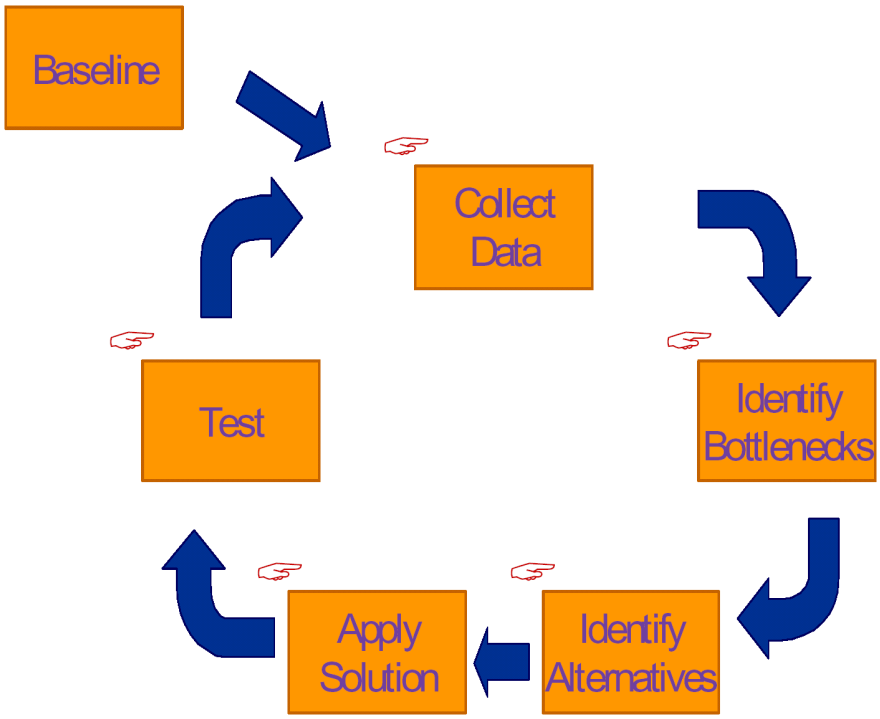

Abbildung 7.3: Iterativer Tuningprozeß

Mit *Baseline* bezeichnet man im allgemeinen einen Ausgangswert, auf den man sich bei einer Meßreihe beziehen kann. Dadurch kann später festgestellt werden, ob eine Parameter- oder sonstige Veränderung einen positiven oder negativen Einfluß auf die Performance hatte.

Treten bei einer Messung sogenannte *Bottlenecks*[4] auf, so meint man damit, daß ein Teil eines Systems den Rest derart "ausbremst", daß die Gesamt-Performance des Systems alleinig durch diesen Teil beschränkt wird. Das typische Beispiel für solche Bottlenecks sind I/O-Bottlenecks, die durch Peripheriespeicher oder das Netzwerk verursacht werden. Weiterhin können Bottlenecks im Software-Design existieren. Diese kommen zum Beispiel dadurch zustande, daß der für eine bestimmte Aufgabe ausgewählte *Abstrakte Datentyp* (ADT) eine unzureichende Laufzeitkomplexität aufweist und somit die gesamte Anwendung verlangsamt. Letztere Bottlenecks sind naturgemäß schwerer zu finden als die erstgenannten.

Nicht immer treten Bottlenecks auf, so daß diese Stufe in der Abbildung in vielen Fällen übersprungen werden kann. Die Alternative (in der Abbildung mit *Identify Alternatives* bezeichnet) besteht darin, den Wert eines Parameters zu erhöhen oder zu erniedrigen und den Testlauf mit der neuen Einstellung zu wiederholen. Im anderen Fall muß der Bottleneck behoben werden, bevor mit dem Tuningprozeß fortgefahren werden kann.

[4] auf deutsch: Flaschenhals

Zu erwähnen bleibt, daß es bei dem gesamten Tuningprozeß äußerst wichtig ist, jeweils nur einen Parameter isoliert auf seinen optimalen Wert hin zu untersuchen. Wird der Fehler begangen, mehrere Einstellungen gleichzeitig zu variieren, kann am Ende einer Tuningphase unter Umständen nicht mehr nachvollzogen werden, welcher Parameter im einzelnen welche Veränderung verursachte.

Der erste Parameter des Online-Banking Systems, der im Zuge eines Tuningvorgangs untersucht werden muß, ist die Anzahl der Worker JVMs. Da es praktisch unmöglich ist, diesen Parameter rechnerisch zu bestimmen, muß er experimentell ermittelt werden (mehr dazu später). Um den optimalen Wert zu bestimmen, wurde das Online-Banking System mit einer linear ansteigenden Zahl an Worker JVMs gestartet und die Durchsatzrate über mehrere Minuten hinweg gemessen. Da der Wert für jede Art von Transaktion unterschiedlich ist, wurde der gesamte Vorgang für alle im Anhang aufgeführten Meßreihen separat durchgeführt. Dies stellt zwar einen großen zeitlichen Aufwand dar, macht sich jedoch im Benchmark-Ergebnis jeder Konfiguration deutlich bemerkbar. Tabelle 7.1 zeigt exemplarisch die Performance-Steigerung für den *Full TPC-A* Transaktionstyp[5] bei Verwendung einer CPU. Die Angabe der Performance-Steigerung in Prozent bezieht sich dabei auf diejenige *Baseline*, welche der Verwendung einer Worker JVM entspricht.

Anzahl Worker JVMs	Performance-Steigerung
2	35%
3	46%
4	52%
5	51%
6	45%

Tabelle 7.1: Performance-Steigerung bei unterschiedlicher Anzahl Worker JVMs

Wie man erkennen kann, ist das Optimum bei vier Worker JVMs erreicht. Ein Erhöhen des Werts führt zu keiner Steigerung der Leistung mehr, er verringert diese sogar. Dies liegt in folgendem Sachverhalt begründet: Die zur Verfügung stehende CPU-Zeit kann von einer (einzigen) Worker JVM nicht vollständig ausgenutzt werden. In der Zeit, in der eine Worker JVM beispielsweise auf die Fertigstellung einer Datenbankoperation wartet, kann eine andere Worker JVM die in der Zwischenzeit untätige CPU für eine Berechnung heranziehen. Auf diese Art und Weise werden bei einer Erhöhung der Anzahl der Worker JVMs alle CPU-Zyklen, in denen eine Worker JVM aus irgendeinem Grund warten muß, für andere Worker JVMs zur Verfügung gestellt. Ab einem bestimmten Wert fangen die Worker JVMs jedoch an, sich gegenseitig “an der Arbeit zu hindern”, indem sie um die vorhandenen CPU-Zyklen konkurrieren. Diese Konkurrenz wirkt sich negativ auf den Transaktionsdurchsatz aus, da zusätzliche CPU-Zeit verbraucht wird. Sinnvoll kann dieser Punkt nur experimentell bestimmt werden.

[5] Mehr zu den verschiedenen Transaktionstypen ist in Abschnitt 8.3.3 zu finden.

Der Parameter mit dem größten Einfluß auf die Performance ist das Garbage Collection Intervall für den Middleware Heap des Online-Banking Systems. Wie in [PRJVM01], Kapitel 2, Abschnitt "Required garbage collection" beschrieben, muß die Garbage Collection für den Middleware Heap der PRJVM vom Launcher Subsystem in regelmäßigen Abständen aufgerufen werden. Bei einer gewöhnlichen Java Virtual Machine findet die Garbage Collection immer dann statt, wenn die Java VM nicht mehr genügend zusammenhängenden Speicherplatz für Objekte auf dem Java Heap vorfindet. Bei Verwendung des *resettable* Modus der PRJVM (`-Xresettable`) muß die Garbage Collection vom Launcher Subsystem programmatisch aufgerufen werden. Um die Bedeutung dieser Einstellung hervorzuheben, ist in Tabelle 7.2 die Performance-Steigerung anhand des *Reduced TPC-A* Transaktionstyps bei Verwendung einer CPU dargestellt. Die Performance-Steigerung bezieht sich wie bei der letzten Tabelle auf eine Baseline. Diese entspricht hier der Verwendung des Werts `1` für das Garbage Collection Intervall. Der Wert `1` bedeutet, daß die Garbage Collection nach jedem Reset durchgeführt wird.

Garbage Collection Intervall	Performance-Steigerung
25	765%
50	923%
75	981%
100	1016%
125	1033%
150	1042%
175	1072%
200	1044%
225	1053%
250	1026%
275	1023%

Tabelle 7.2: Performance-Steigerung bei unterschiedlichem Garbage Collection Intervall

Die größte Performance-Steigerung wurde bei diesem Transaktionstyp bei einem Garbage Collection Intervall von 175 ermittelt. Die in der Tabelle aufgeführten 1072 Prozent entsprechen einem Zuwachs um den Faktor 11,72[6]. Mit anderen Worten: Wenn man die Garbage Collection bei diesem Transaktionstyp nach jedem 175. Reset durchführt, ist die Gesamtleistung des Online-Banking Systems 11,72 Mal höher als bei der Baseline-Einstellung, bei welcher die Garbage Collection nach jedem Reset durchgeführt wird. Wiederum bleibt zu erwähnen, daß sich dieser Wert nicht vernünftig rechnerisch bestimmen läßt. Man muß ihn experimentell ermitteln, denn er ist abhängig von der Anzahl der allozierten Java-Objekte pro Transaktion.

[6]Um Mißverständnissen entgegenzuwirken: Ein Zuwachs um 100 Prozent entspricht einem Faktor von 2.

Man kann die Garbage Collection beobachten, indem man die `-verbose:gc` Option in der `go.prp` Konfigurationsdatei des Online-Banking Systems für eine Worker JVM aktiviert. So läßt sich beobachten, daß die Zeitdauer für eine Garbage Collection bei wachsendem Intervallwert anfänglich kaum ansteigt. Erst ab einem bestimmten Wert für das Intervall steigt auch die Zeitdauer der Garbage Collection merklich. Eine Erklärung hierfür wäre, daß die Garbage Collection eine gewisse Grundkomplexität zu beinhalten scheint, deren Einfluß auf den Gesamtdurchsatz erst ab einer bestimmten Anzahl von allozierten Objekten auf dem Java Heap überdeckt wird. In dem Beispiel aus der Tabelle wäre dieser Punkt also ab derjenigen Anzahl der Objekte erreicht, welche nach 175 Transaktionen auf dem Heap alloziert wurde. Anders formuliert: Ob man jede zehnte oder jede hundertste Iteration eine Garbage Collection aufruft, spielt für die *Dauer* der Garbage Collection keine Rolle. Für den *Gesamtdurchsatz* ist es jedoch von großer Bedeutung, denn in einem Fall wird die Garbage Collection nach jeder zehnten, im anderen Fall erst nach jeder hundertsten Transaktion durchgeführt.

Ein weiterer Parameter, der untersucht werden muß, ist die Größe des Transient Heap innerhalb des Nonsystem Heap der PRJVM. Diese wird durch die `-Xinitth` Option der Java VM festgelegt. Je größer dieser Teil des Heaps ist, desto länger dauert der Reset-Vorgang nach einer erfolgreichen Transaktion[7]. Tabelle 7.3 zeigt den Zusammenhang zwischen der Größe des Transient Heap und deren Auswirkung auf die Performance anhand des Full TPC-A Transaktionstyps. Die Baseline bei dieser Messung ist die Standardeinstellung für die Größe des Transient Heap: 512 KB. Der angegebene relative Durchsatz bezieht sich somit auf den Transaktionsdurchsatz, der bei der Standardeinstellung erzielt wurde. Außerdem muß noch angemerkt werden, daß die Anzahl der Objekte, die auf dem Transient Heap allozierten werden, gleich bleibt. Dies bedeutet, daß lediglich die Größe des Transient Heap, nicht jedoch dessen Inhalt verändert wurde.

Größe des Transient Heap	Relativer Durchsatz
1024 KB	99%
2048 KB	98%
4096 KB	94%
8192 KB	91%
16384 KB	84%

Tabelle 7.3: Auswirkung der Größe des Transient Heap auf den Transaktionsdurchsatz

Da bei den Performance-Tests mit BOBS die minimale Größe für den Transient Heap (512 KB) ausreichend ist, konnte hier keine Optimierung vorgenommen werden (die PRJVM läßt einen Wert kleiner als 512 KB nicht zu). Im Online-Banking System können jedoch verschiedenartige Transaktionen (Transaktions-Mix) aufgerufen

[7] Zur Erinnerung: Die Garbage Collection *Policy* für den Transient Heap ist es, den gesamten Transient Heap zu löschen.

werden, welche unter Umständen länger dauern können als der Full TPC-A Transaktionstyp. In diesem Fall können (abhängig vom Programmcode der Transaktionen) so viele Objekte im Transient Heap alloziert werden, daß die Minimaleinstellung für dessen Größe nicht ausreichend ist. Tritt diese Situation ein, so muß der Parameter auf seinen für diesen speziellen Mix optimalen Wert hin analysiert werden. Wie man aus der Tabelle entnehmen kann, ist es unvorteilhaft, in Erwartung eines größeren Bedarfs an Speicher im Transient Heap einen zu hohen Wert einzustellen.

7.1.3 Middleware- und Applikationsklassen

Pragmatisch betrachtet sind die im Online-Banking System benutzten Middleware- und Applikationsklassen "gewöhnliche" Java-Klassen. Beim Entwurf und der Implementierung dieser Art von Klassen sind jedoch einige Eigenheiten zu berücksichtigen, die auf die Architektur der Persistent Reusable Java Virtual Machines Technologie zurückzuführen sind. In diesem Unterabschnitt werden einige Richtlinien vorgestellt, welche für die Programmierung dieser Klassen hilfreich sind.

Beim gesamten Software-Entwicklungsprozeß muß berücksichtigt werden, daß die Funktionalität einer Gesamtanwendung in zwei Kategorien aufzuteilen ist: Auf der einen Seite stehen die Daten und der Programmcode, die fest mit einer (einzigen) Transaktion verbunden sind (diese sind der Kategorie Applikationsklassen zuzuordnen). Auf der anderen Seite sind diejenigen Daten inklusive Programmcode zu positionieren, welche über die Dauer einer Transaktion hinweg existieren (Trusted Middleware). Sind die verschiedenen Komponenten bezüglich ihrer Rolle im Gesamtsystem identifiziert, ist die Grundlage für ein erfolgreiches Design geschaffen.

Eine weitere grundsätzliche Regel ist es, besonders performancekritische Teile der Anwendung in die Middleware zu verlagern. Der Grund hierfür ist folgender: Diese Komponenten (hierbei handelt es sich um Java-Objekte) können über die Grenzen einer Transaktion hinweg existieren. Werden die Klassen der entsprechenden Objekte als Applikationsklassen deklariert, so muß das Launcher Subsystem diese Klassen nach jedem Durchlauf mittels der Java Native Interface Funktion `FindClass()` neu lokalisieren. Dieser Vorgang verursacht einen Overhead, welcher durch die Deklaration als Middleware vermieden werden kann. Dabei darf man jedoch unter keinen Umständen den Sicherheitsaspekt aus den Augen verlieren: Alle Daten, die fest mit einer Transaktion verbunden sind, müssen im Anschluß an die Transaktion zuverlässig gelöscht werden. Beim Online-Banking System sind dies unter anderem die Kontonummer und der Betrag des Debit- bzw. Credit-Vorgangs. Eine in böswilliger Absicht geschriebene Folgetransaktion könnte sonst unter Umständen an diese Informationen gelangen. Bildlich gesprochen müssen alle Spuren der vorherigen Transaktion beseitigt werden, bevor eine neue beginnen kann. Dieses "Beseitigen der Spuren" wird durch den Reset-Vorgang sichergestellt, welcher einzigartig für die Persistent Reusable Java Virtual Machines Technologie ist.

Bei der Entwicklung des Online-Banking Systems stand von Beginn an fest, daß ein großer Teil der für eine Transaktion benötigten CPU-Zeit für den Datenbankzugriff

aufgewendet werden muß. Daher wurde der gesamte Datenbank-Teil von BOBS in die Middleware verlagert. Wie in Abschnitt 5.5.1 erwähnt, ist die teuerste Operation im Zusammenhang mit SQLJ und JDBC das Aufbauen der Verbindung zur Datenbank[8]. Deshalb wird diese Verbindung nur einmal (beim Initialisieren eines `MiddleWare` Objekts) aufgebaut und solange aufrechterhalten, bis das Online-Banking System beendet wird. Damit der gesamte Datenbankverkehr nicht über eine (einzige) logische Verbindung erfolgen muß, wird jeder Worker JVM beim Initialisieren ein eigenes `MiddleWare` Objekt (und somit auch eine eigene Datenbankverbindung) zugewiesen. Da die Verbindung durch dieses Design zur Trusted Middleware deklariert wird, muß sie am Ende einer Transaktion nicht beendet werden. Abbildung 7.4 stammt aus [Brun02] und zeigt deutlich, wieso der Aufbau der Verbindung (in der Abbildung mit `getconnection` bezeichnet) nur dann erfolgen sollte, wenn es sich nicht vermeiden läßt.

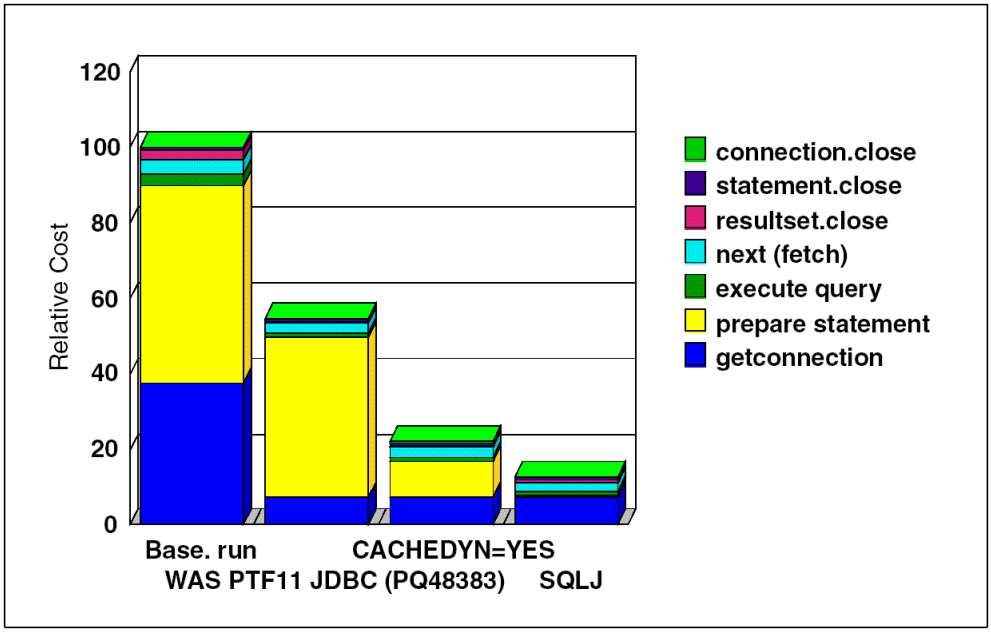

Abbildung 7.4: Kosten der Datenbankoperationen

Weiterhin ist zu erkennen, daß neben dem Aufbau der Datenbankverbindung auch das Vorbereiten eines SQL-Statements (in der Abbildung durch `prepare statement` gekennzeichnet) eine sehr teure Operation ist. Diese Kosten treten nur bei der Verwendung von JDBC auf (repräsentiert durch die linken drei Balken in der Abbildung). Durch SQLJ realisierte Datenbankoperationen (der rechte Balken) sind *per definitionem* statisch. Somit müssen sie nicht mehr vorbereitet werden, da sie bereits zum Zeitpunkt des Bindens des Zugriffsplans für die Datenbank feststehen. Für einen

[8]Um keine Mißverständnisse aufkommen zu lassen: Diese Datenbankverbindung ist keine Netzwerkverbindung, sondern eine durch Betriebssystemmittel wie Shared Memory realisierte logische Verbindung zur Datenbank.

eingehenden Vergleich zwischen SQLJ und JDBC sei auf [Brun02] oder [Vila02] verwiesen.

Auf der Seite der Applikationsklassen lassen sich ebenfalls einige grundlegende Richtlinien angeben. In Tabelle 7.3 auf Seite 63 wurde deutlich, daß die Dauer eines Reset-Vorgangs mit der Größe des Transient Heap steigt. Daher sollte ein Applikationsobjekt zur Laufzeit so wenige Objekte wie möglich erzeugen. Jedes zusätzliche Objekt vergrößert den Inhalt des Transient Heap und erhöht so die Dauer eines Resets. Sehr vorsichtig sollte man sein, wenn man aus diesem Grund versucht, Applikationsobjekte in der Middleware zwischenzuspeichern (zu *cachen*), um sie für eine Folgetransaktion zu benutzen. Sind am Ende einer Transaktion nicht alle Referenzen vom Middleware Heap in den Transient Heap beseitigt worden, so muß die PRJVM in dem erwähnten (sehr teuren) Vorgang `TraceForDirty()` kontrollieren, ob eine dieser Referenzen noch aktiv ist. Im ungünstigsten Fall (eine dieser Referenzen ist noch *live*) wird die Worker JVM als *dirty* markiert und muß heruntergefahren werden. Andererseits ist es legal, Middleware-Objekte in der Middleware zu cachen, und deren Daten mittels sogenannter `getter` und `setter` Methoden für die Applikationsobjekte zugänglich zu machen.

Weiterhin müssen alle in [PRJVM01], Kapitel 5, “Developing applications” beschriebenen Aktionen vermieden werden, die dazu führen, daß eine Worker JVM *unresettable* wird. Dazu zählen zum Beispiel das Starten bzw. Stoppen eines Java-Threads, die Benutzung von *Abstract Window Toolkit* (AWT) Klassen zum Zugriff auf das Display oder die Tastatur, das Umleiten der Standardein- und -ausgabe mittels `java.lang.System.setIn()` und `java.lang.System.setOut()` und viele weitere, die im angegebenen Kapitel in [PRJVM01] beschrieben sind.

7.2 Umsetzung in reines Java (Pure Java)

Bevor der vorliegende Abschnitt auf die Best Practices eingeht, die während der Umsetzung des Basic Online-Banking Systems in reines Java gefunden wurden, muß an dieser Stelle etwas weiter zum Thema *Java Performance* ausgeholt werden.

Die Anstrengungen ganzer Generationen von Software-Entwicklern, die Performance von Computerprogrammen zu steigern, existieren nicht erst seit Einführung der Programmiersprache Java. So wurden alle möglichen Tricks versucht, den betroffenen Codezeilen immer noch mehr Leistung abzugewinnen. Viele dieser Tricks lassen sich auch auf Java-Programme anwenden. Die größten Steigerungen werden jedoch nicht mit dem Schreiben kryptischer Einzelanweisungen, sondern mit der Verbesserung des zugrunde liegenden Designs und der Auswahl der Algorithmen erzielt.

Abbildung 7.5 auf Seite 67 zeigt ein Beispiel für hochoptimierten Code. Dieses Programm ist allerdings einer der Gewinner des *International Obfuscated C Code Contest* (siehe [IOCCC]) und somit (zugegebenermaßen) weniger auf Leistung als auf Verwirrung des Lesers optimiert. Nichtsdestotrotz spiegelt es die Kernaussage des letzten Absatzes wieder.

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'//'//'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Abbildung 7.5: Eine “Hello, World!” Version

Es existiert die weit verbreitete Meinung, Java-Programme müßten besonders gut optimiert sein, um eine akzeptable Performance zu erreichen. Um diese Meinung zu entkräften, sei auf die zahlreichen Untersuchungen verwiesen, die in diesem Bereich durchgeführt wurden. Viele dieser Untersuchungen kommen zu dem Ergebnis, daß Java-Programme durch die modernen JIT-Compilertechnologien eine ähnliche Performance wie C-Programme erreichen. Als Beispiel sei [Bull01] aufgeführt, in dem die Autoren zu folgendem Schluß kommen: “[...] the performance gap is small enough to be of little or no concern to programmers”.

Zurück zur eigentlichen Thematik dieses Abschnitts: Es existiert eine sehr umfangreiche Menge an Literatur zum Thema *Java Performance* und ebenso viele Seiten im Internet, welche sich mit dieser Thematik auseinandersetzen. Diejenigen Quellen, welche das Thema ernsthaft behandeln, kommen zu derselben Kernaussage, die in der Einleitung dieses Abschnitts vorgestellt wurde: Die größten Performance-Steigerungen werden mit der Verbesserung des zugrunde liegenden Designs und der Auswahl der Algorithmen erzielt. Tips in der Art “Man muß Methoden als `static` deklarieren, dann wird das Programm schneller” sind nicht nur schlichtweg falsch, wie die Ergebnisse der Microbenchmarks zeigen. Sie sind auch insofern schädlich, als daß sie ein unter Softwaretechnik-Gesichtspunkten als durchdacht zu bezeichnendes Design vollständig verwässern können. Als einzige Literaturangabe, die sich sowohl tiefgehend als auch seriös mit der Thematik befaßt, sei auf [Shir03] und die damit verbundene Internetpräsenz [JavaPerf] verwiesen.

Um ein konkretes Beispiel für eine Optimierung zu nennen, wurde während der Umsetzung des Online-Banking Systems in reines Java folgende Designrichtlinie angewandt: Die Anzahl der Instruktionen in einer mit `synchronized` deklarierten Methode muß auf ein Minimum begrenzt werden. Der Hintergrund dieser Optimierung ist folgender: Durch die Deklaration mit dem `synchronized` Schlüsselwort wird der Zugriff auf eine Methode so synchronisiert, daß sich zu einem Zeitpunkt nur ein Thread in der Methode[9] aufhalten darf. Dies ist notwendig, wenn mehrere Threads auf dasselbe Datum sowohl lesend als auch schreibend zugreifen. Je größer die Anzahl der Instruktionen in solch einer Methode ist, desto länger müssen diejenigen Threads warten, die sich gerade nicht in der Methode befinden. Um wieder den Bezug zur Einleitung dieses Abschnitts herzustellen, muß erwähnt werden, daß diese Richtlinie keine spezifische Java-Optimierung darstellt, sondern sich auf viele Programmiersprachen umsetzen läßt. So würde zum Beispiel die entsprechende Richtlinie in der Programmiersprache C wie folgt lauten: Die Anzahl der Instruktionen zwischen einer `pthread_mutex_lock()` und einer `pthread_mutex_unlock()` Anweisung

[9] Genaugenommen müßte man eigentlich sagen, daß sich zu einem Zeitpunkt nur ein Thread in einem *Objekt* aufhalten kann, siehe auch Abschnitt 6.2.3, “Threads und Locking”.

muß auf ein Minimum reduziert werden, wenn man in C Programme unter Verwendung von POSIX-Threads implementiert.

Ein weiterer grundsätzlicher Ansatz, besonders teure Methodenaufrufe zu vermeiden, ist das *Cachen* von Ergebnissen solcher Aufrufe. Da die Umsetzung von BOBS in reines Java genauso flexibel auf einkommende Transaktions-Requests reagieren muß wie die hybride Variante, war es notwendig, Teile der Java Reflection API zu benutzen. Diese ist für ihren negativen Einfluß auf die Gesamt-Performance von Java-Anwendungen bekannt. Diese API dient (kurz gesagt) dazu, in Java-Klassen dynamisch zur Laufzeit Informationen über die in den Klassen enthaltenen Konstruktoren, Methoden und Variablen zu erhalten und diese aufzurufen bzw. auszulesen oder zu setzen. Um eine Performance-Steigerung zu erzielen[10], genügt es im Fall der reinen Java-Variante des Online-Banking Systems, einmal lokalisierte Java-Methoden einer Klasse zwischenzuspeichern und diese beim Abarbeiten des nächsten Transaktions-Requests aus dem Cache abzurufen. Diese Vorgehensweise war lediglich bei der reinen Java-Version möglich, da die hybride Variante durch diesen Vorgang ein *Unresettable Event* hervorrief. Dieses Caching-Prinzip ist unbedingt auch auf Objekte anzuwenden, deren Erzeugung sehr teuer ist. Ein Beispiel hierfür ist das in Abschnitt 7.1.3 vorgestellte Erzeugen einer Datenbankverbindung, welches in einem `java.sql.Connection` Objekt resultiert.

Als sehr nützlich hat sich auch folgendes herausgestellt: Für die Java Virtual Machine existiert eine Kommandozeilenoption, mit Hilfe der man diejenigen Teile des Programmcodes isolieren kann, in welchen die meiste CPU-Zeit verbraucht wird: `-Xrunhprof:cpu=times`. Diese Option aktiviert den HPROF *Profiler Agent*, der als Teil einer Java Development Kit Installation mitgeliefert wird. Ein Profiler Agent ist ein Teil der *Java Virtual Machine Profiler Interface* (JVMPI) Spezifikation, die ab dem JDK Version 1.2 als experimenteller Zusatz integriert wurde. Das JVMPI ist ein sogenanntes *Function Call Interface* zwischen der Java Virtual Machine und einem Profiler Agent. Es erlaubt solch einem Agenten, bestimmte Ereignisse der Java VM wie zum Beispiel die Allokation von Speicher auf dem Heap oder das Starten und Stoppen eines Threads mitzuverfolgen. Läßt man eine Java-Anwendung mit oben aufgeführter Option ablaufen, erhält man nach Beendigung der Applikation eine Aufstellung der verbrauchten CPU-Zeit für jede aufgerufene Methode. So können diejenigen Methoden identifiziert werden, in denen die meiste CPU-Zeit verbraucht wurde.

Es ist ausreichend, die Aufmerksamkeit beim Optimieren ausschließlich auf diese zu richten. Verbesserungen für Methoden, die einen Anteil im unteren einstelligen Prozentbereich der Gesamtausführungszeit der Anwendung beanspruchen, führen zu keiner meßbaren Performance-Steigerung. In den allermeisten Fällen ist es ausreichend, einige derjenigen Methoden zu untersuchen, die in der Rangliste der CPU-intensivsten Methodenaufrufe ganz oben stehen. Dies steht im Einklang mit der aus der Informatik bekannten sogenannten *90/10 Regel*, welche besagt, daß 90 Prozent der insgesamt

[10] Im Performance-Test mit dem Reduced TPC-A Transaktionstyp kann die Gesamt-Performance hierdurch um bis zu fünf Prozent gesteigert werden.

verbrauchten Rechenzeit eines Programms auf 10 Prozent des Codes zurückzuführen sind.

Während der Entwicklung des Online-Banking Systems ergab sich durch dieses Monitoring ein anschauliches Beispiel für die erwähnte Nutzlosigkeit der Optimierung von Einzelanweisungen. Der vom TPC-A Standard vorgeschriebene Wertebereich für einige numerische Spalten der Datenbanktabellen kann mit einer Integer-Variablen (`int`) nicht abgedeckt werden. Daher mußten `java.math.BigDecimal` Objekte benutzt werden. Aus der Anwendung ergab sich der geradlinigste Weg der Erzeugung eines `BigDecimal` Objekts wie in Variante 1 in Abbildung 7.6. Da sich dieser als teuer erwies, wurde er durch die Variante 2 ersetzt.

```
/*
   “stringValue” ist der Wert einer Zahl in
   String-Darstellung
*/
String stringValue = “9999999999”;

/* Variante 1 */
BigDecimal myBigDecimalOne = new BigDecimal(stringValue);

/* Variante 2 */
BigDecimal myBigDecimalTwo = BigDecimal.valueOf(
   Long.parseLong(stringValue), 0);
```

Abbildung 7.6: Erzeugung eines `BigDecimal` Objekts

Obwohl Variante 2 drei Mal schneller als Variante 1 ist und pro Transaktionsdurchlauf vier Mal aufgerufen werden muß, ergibt sich für den Gesamtdurchsatz eine Performance-Steigerung von weniger als einem Prozent. So macht sich diese Optimierung im Online-Banking System praktisch nicht bemerkbar.

Abschließend sei erwähnt, daß die Konfiguration des Online-Banking Systems in der reinen Java-Version (ähnlich wie bei der hybriden) so verändert werden kann, daß sich dies positiv auf den Gesamtdurchsatz auswirkt. Wie in Abschnitt 5.7.2 beschrieben, existiert für jede Worker JVM ein Gegenstück in der reinen Java-Version, die sogenannten `TransactionProcessor` Threads. Ähnlich wie bei den Worker JVMs in der hybriden Version kann man auch in der reinen Java-Version die Anzahl der `TransactionProcessor` Threads variieren und so eine meßbare Performance-Steigerung erzielen. Auch hier trifft die am Ende von Abschnitt 7.1.2 getroffene Aussage zu: Der optimale Wert für diesen Parameter kann nur experimentell sinnvoll bestimmt werden.

7.3 Datenbankspezifische Optimierungen

Für den aufmerksamen Leser muß zunächst die Frage geklärt werden, wieso in einer Arbeit, deren Hauptthema die Java Virtual Machine ist, datenbankspezifische Optimierungen aufgeführt werden. Die Antwort darauf ist durch die Tatsache gegeben, daß das zur praktischen Demonstration der Unterschiede in den Java VMs verwendete Online-Banking System die Datenbankzugriffe unter Java durchführt. Um die Resultate der untersuchten Plattformen miteinander vergleichen zu können, muß gewährleistet sein, daß alle Komponenten des Basic Online-Banking Systems (und somit auch die Datenbank) optimal arbeiten. Die datenbankspezifischen Optimierungen, welche hier als *Best Practices* vorgestellt werden, wurden als Teil der eingangs erwähnten Performance-Test-Phase der Diplomarbeit ermittelt.

Der erste nun folgende Unterabschnitt geht auf die Verwendung von SQLJ und JDBC in BOBS ein, während im zweiten einige abschließende Bemerkungen zum Datenbankdesign sowie zu den untersuchten Datenbankparametern aufgeführt sind.

7.3.1 SQLJ und JDBC

Bevor die Performance-Vergleiche zwischen SQLJ und JDBC durchgeführt werden konnten, mußte zunächst die Umgebung eingerichtet werden. Unter Linux und im *non-resettable* Modus der PRJVM bereitete dies keine Schwierigkeiten, es müssen lediglich die Anweisungen in den DB2-Handbüchern genau befolgt werden. Im *resettable* Modus unter z/OS konnte der SQLJ/JDBC-Treiber zu Beginn nicht benutzt werden (der Treiber lieferte ausschließlich `SQLException`s zurück). Erst ein Modifizieren des `CLASSPATH` für die Middleware löste das Problem. So muß unbedingt das Verzeichnis zum Middleware `CLASSPATH` hinzugefügt werden, in dem sich das sogenannte *serialisierte Profil* für den Zugriff auf das Datenbankmanagementsystem befindet. Auf dem z/OS-System, das für die Performance-Tests benutzt wurde, war dies `/SYSTEM/local/db2/db2v7/db2/db2710/classes`. Zu beachten ist, daß dieser Pfad unabhängig von der `.zip` Datei mit dem SQLJ/JDBC-Treiber hinzuzufügen ist. Dieser Sachverhalt war in keiner Literatur verzeichnet. Erst das Auffinden eines sogenannten *Problem Management Record* (PMR)[11], der ein ähnliches Problem behandelte, führte zum Erfolg.

Die Performance-Vergleiche zwischen SQLJ und JDBC wurden mit der in Abschnitt 7.1.2 eingeführten Methodik (*Iterativer Tuningprozeß*) durchgeführt (mehr dazu später). Die Untersuchungen brachten folgendes zutage: Unter z/OS wird die höchste Performance durch SQLJ mit einigen Modifizierungen erzielt, während unter Linux das optimale Resultat mit Hilfe von JDBC und `PreparedStatement`s erreicht wurde. Eine Erklärung hierfür könnte in der Tatsache begründet liegen, daß in den SQLJ/JDBC-Treibern für das jeweilige Datenbankmanagementsystem unterschiedli-

[11] Ein Problem Management Record ist eine Art "Report" und wird immer dann angelegt, wenn ein Kunde einen Software-Fehler in einem IBM-Produkt vermutet.

che Teilgebiete optimiert wurden (sowohl *DB2 for z/OS and OS/390* als auch *DB2 Universal Database for Linux* werden mit einem eigenen SQLJ/JDBC-Treiber ausgeliefert). Die folgenden Unterabschnitte behandeln die beiden Plattformen getrennt.

SQLJ unter z/OS

Für die Performance-Messungen unter z/OS wurde SQLJ mit einigen Modifizierungen benutzt, da diese Variante die höchste Durchsatzrate liefert. Es muß jedoch bemerkt werden, daß SQLJ und JDBC (zumindest in dieser Arbeit) ungefähr dieselbe Performance aufweisen, wenn keine Optimierungen vorgenommen werden (eine ausführliche Erklärung hierfür befindet sich einige Absätze weiter unten). Das ist auf den ersten Blick erstaunlich, denn: Programme, die auf SQLJ basieren, sind statisch gegen die Datenbank gebunden, während der Zugriffsplan für ein mittels JDBC abgesetztes SQL-Statement dynamisch zur Laufzeit berechnet werden muß. Dieses Berechnen verbraucht zusätzliche CPU-Zeit. Dies ist zum Beispiel dann notwendig, wenn das SQL-Statement (dynamisch) zur Ausführungszeit konstruiert wird. Außerdem müssen die Zugriffsrechte für die betroffenen Datenbanktabellen im Falle von JDBC zur Laufzeit überprüft werden. Diese Kontrolle wird bei SQLJ zum Zeitpunkt des Bindens vorgenommen. Abbildung 7.7 stammt aus [Brun02] und verdeutlicht diese Zusammenhänge. Dabei entspricht JDBC dem mit *Dynamic SQL* überschriebenen Teil der Abbildung und SQLJ dem mit *Static SQL* beschrifteten.

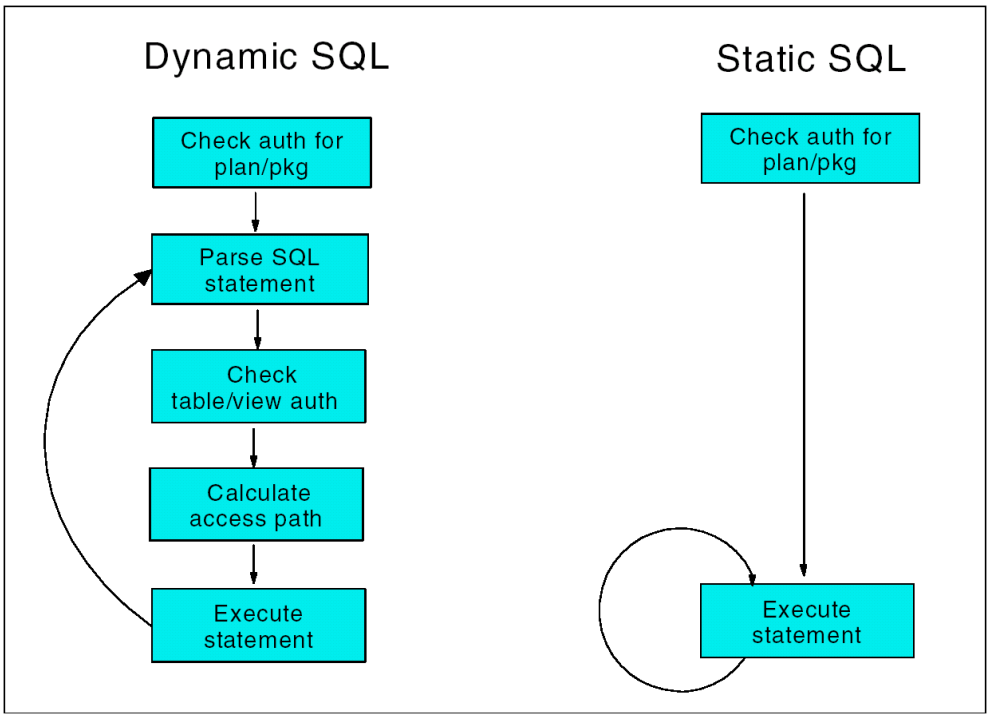

Abbildung 7.7: Dynamisches versus Statisches SQL

Um eine Entscheidung zwischen den beiden Varianten herbeizuführen, wurde das Online-Banking System mit unterschiedlichen Versionen der `MiddleWare` Klasse gestartet und die Performance über mehrere Minuten hinweg gemessen. Wie in Abschnitt 5.5.1 erwähnt, existiert diese Java-Klasse in zwei Versionen (eine für SQLJ und eine für JDBC). Sie läßt sich einfach austauschen, da beide Varianten dieselbe Schnittstelle und Funktionalität aufweisen. Unter Verwendung von Softwaretechnik-Terminologie würde man diese Schnittstelle mit *Exportschnittstelle* bezeichnen. Ein Anpassen der übrigen Teile des Online-Banking Systems ist nicht notwendig.

Tabelle 7.4 faßt die Ergebnisse der Untersuchungen für den Full TPC-A Transaktionstyp zusammen. Dabei ist deutlich zu erkennen, daß die Variante mit modifiziertem SQLJ und veränderten `BIND` Parametern die höchste Performance-Steigerung zur Folge hat. Die Baseline für diese Messungen entspricht der Verwendung der `MiddleWare` Klasse in der JDBC-Variante ohne `PreparedStatement`s. Die SQLJ-Modifizierungen werden weiter unten beschrieben, während die `BIND` Parameter unabhängig von SQLJ und JDBC zu sehen sind[12] und deshalb in Abschnitt 7.3.2 beschrieben werden.

Datenbankzugriff	Performance-Steigerung
JDBC mit `PreparedStatement`s	444%
SQLJ ohne Modifizierungen	505%
SQLJ mit Modifizierungen	516%
SQLJ mit Modifizierungen und veränderten `BIND` Parametern	599%

Tabelle 7.4: SQLJ versus JDBC - Performance

Weiterhin wird durch diese Tabelle der eingangs erwähnte Sachverhalt bestätigt: SQLJ und JDBC weisen ohne Optimierungen ungefähr dieselbe Performance auf (444 zu 505 Prozent). Diese Aussage trifft jedoch nur auf JDBC mit `PreparedStatement`s zu. Seit deren Einführung (und Implementierung) im SQLJ/JDBC-Treiber wurden viele Optimierungen vorgenommen, um die Performance von JDBC `PreparedStatement`s zu steigern. Die größte Performance-Steigerung wurde durch die Einführung eines *Dynamic Statement Cache* für SQL-Statements in der Version 5 der *DB2 for z/OS and OS/390* erreicht. Das Prinzip ist recht einfach: Wenn ein SQL-Statement von DB2 ge-*parsed* und anhand der ermittelten Informationen ein Zugriffsplan erstellt wird, legt DB2 eine Kontrollstruktur an, die zur Ausführung der Anfrage benutzt wird. Diese Kontrollstruktur wird zwischengespeichert und falls dieselbe Anfrage noch einmal an die Datenbank gestellt wird, muß DB2 für die Ausführung dieses Statements nur die entsprechende Struktur im Cache lokalisieren. Durch die ständige Weiterentwicklung dieser Caching-Strategie erreichen die eigentlich dynamischen

[12] Diese Parameter kann man unter z/OS für jede Art von *Database Request Module* (DBRM) angeben, die gegen eine Datenbank gebunden werden.

SQL-Statements, die einmal *prepared*[13] wurden, annähernd die Performance der statischen.

In manchen Fällen können die dynamischen sogar schneller sein. Der Grund hierfür ist folgender: Wenn ein SQL-Statement bei einer nahezu leeren Datenbank statisch gegen diese gebunden wird, ist der für diese Anfrage optimale Zugriffsplan anders als zu einem späteren Zeitpunkt, bei dem die Datenbank dichter besetzt ist. Ein dynamisches `PreparedStatement` kann von den veränderten Rahmenbedingungen profitieren, während das statische noch mit den alten Informationen ausgeführt wird. Deshalb ist es notwendig, die statischen SQL-Statements in regelmäßigen Abständen neu gegen die Datenbank zu binden. Der optimale Zeitpunkt für diesen Vorgang ist nach dem Ausführen der `REORG` und `RUNSTATS` Utility-Programme, welche die angegebenen Tabellen neu organisieren (`REORG`) und die Statistiken über die in den Tabellen enthaltenen Werte aktualisieren (`RUNSTATS`). Die Syntax und die verschiedenen Möglichkeiten zur Ausführung der beiden Utilities sind in [DB2UGR02] beschrieben.

Ein Nachteil bei SQLJ-Programmen ist der im Vergleich zu JDBC erhöhte Entwicklungsaufwand. Dieser ergibt sich durch einen zusätzlichen sogenannten *Precompile* Schritt und dem Binden des Zugriffsplans. Abbildung 7.8 auf Seite 74 stammt aus [DB2Java02] und stellt den hierfür notwendigen Prozeß dar.

Ein SQLJ-Programm muß zunächst durch einen Precompiler übersetzt werden, welcher die im Java-Sourcecode direkt eingebetteten SQL-Statements ersetzt (in der Abbildung mit *SLQJ Translator* gekennzeichnet). Erst danach kann das entstandene Java-Programm durch den "normalen" `javac` Compiler in Bytecode übersetzt werden. Weiterhin erzeugt der Precompiler ein sogenanntes *serialisiertes Profil* (in der Abbildung mit *Serialized Profile* bezeichnet). Dieses enthält Informationen über die eingebetteten SQL-Statements und muß durch den *DB2 SQLJ Profile Customizer* an die Datenbank angepaßt werden. Als letzter Schritt folgt dann das Binden des Zugriffsplans. Unter z/OS wird dieser Vorgang durch einen in der *Job Control Language* (JCL) geschriebenen Job durchgeführt. Unter Linux ist der letztgenannte Schritt nicht notwendig, da er bereits vom Profile Customizer durchgeführt wird.

Die erwähnten SQLJ-Modifizierungen beziehen sich auf den nach dem Precompile-Schritt entstandenen Java-Sourcecode. In dem durch den Precompiler generierten Code wird für jedes eingebettete SQL-Statement ein eigenes `RTStatement` Objekt erzeugt, welches nach dem Datenbankzugriff wieder freigegeben wird. Das ständige Erzeugen und wieder Freigeben verursacht einen unnötigen Overhead. Dieser kann dadurch vermieden werden, daß das `RTStatement` Objekt nur einmal pro SQL-Statement generiert wird und für den Rest der Lebensdauer des `MiddleWare` Objekts nicht dereferenziert wird. Diese Optimierungsmöglichkeit kann unter [Pool02] nachgelesen werden.

Weiterhin wird das `RTStatement` Objekt in einem mit `synchronized` deklarierten Block innerhalb der betroffenen Methoden ausgeführt. Dies ist im Online-Banking System unnötig, da in einer Worker JVM zu einem Zeitpunkt nur eine Instanz

[13] Daher kommt der Name der Java-Klasse für solche SQL-Statements: `PreparedStatement`.

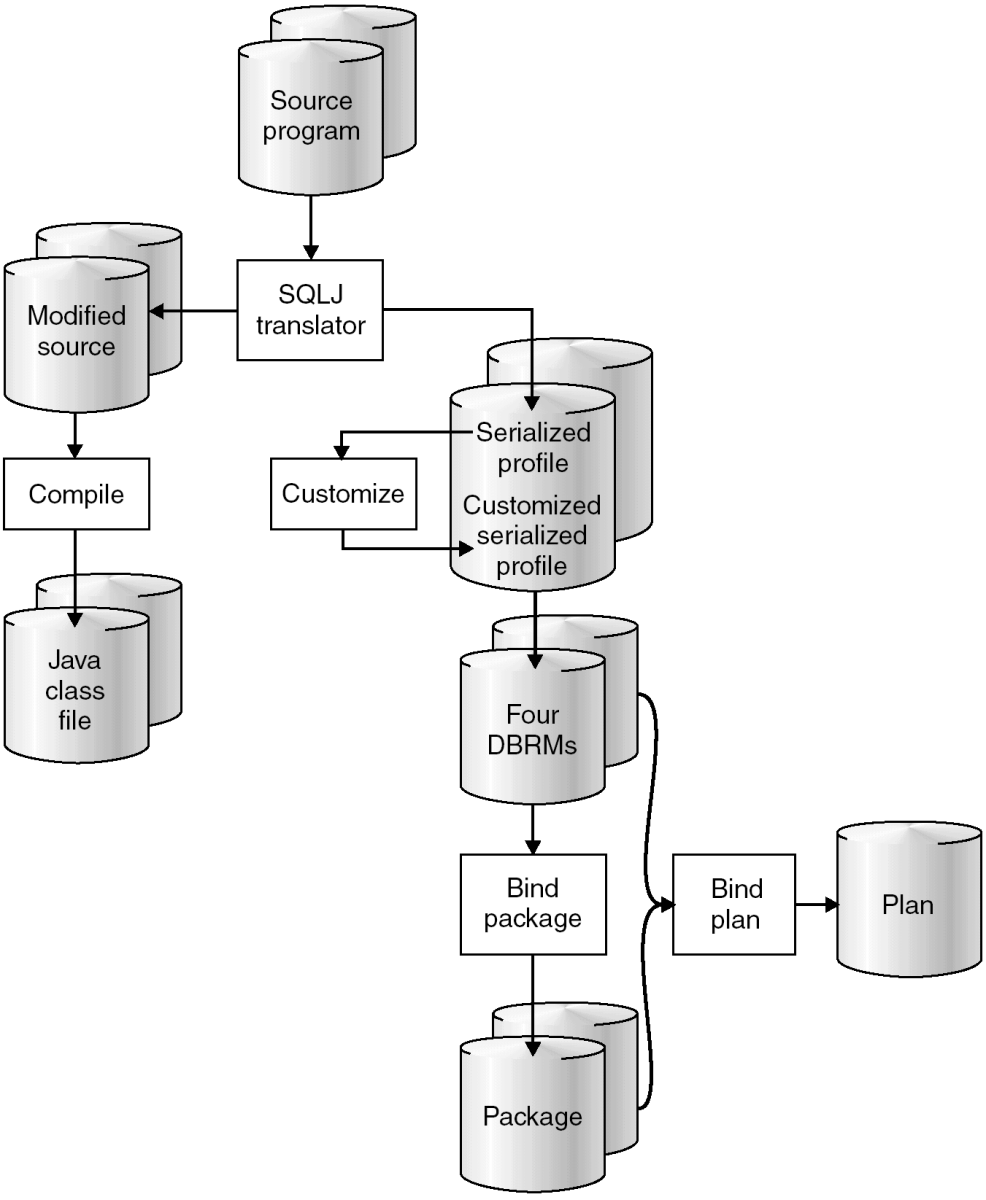

Abbildung 7.8: Der SQLJ-Entwicklungsprozeß

eines `TPCATransaction` Objekts auf das `MiddleWare` Objekt[14] zugreifen kann. Andere Objekte des Online-Banking Systems greifen auf das `MiddleWare` Objekt nicht zu und somit ist es bei BOBS ausgeschlossen, daß sich zwei Threads gleichzeitig in besagtem synchronisierten Block aufhalten. In der Konsequenz kann das `synchronized` Schlüsselwort, welches zusätzliche CPU-Zeit verbraucht, inklusive der Klammern für den Block entfernt werden. Dies stellt eine einfache und effektive Möglichkeit dar, Synchronisationsprobleme durch Software-Design zu umgehen.

JDBC unter Linux

Wie erwähnt wurde unter Linux die höchste Performance mit Hilfe von JDBC `PreparedStatement`s erzielt. Hierfür wurde ebenfalls eine Optimierungsmöglichkeit ermittelt, welche eine Performance-Steigerung im unteren einstelligen Prozentbereich zur Folge hat. Diese wurde mit Hilfe des in Abschnitt 7.2 beschriebenen Monitoring durch den HPROF Profiler Agent ermittelt. Um diese Optimierung erläutern zu können, müssen zunächst einige Bemerkungen zu JDBC `PreparedStatement`s gemacht werden. Bevor ein JDBC `PreparedStatement` ausgeführt werden kann, müssen (neben einigen anderen Dingen) zuerst dessen Parameter gesetzt werden. Abbildung 7.9 zeigt die grundlegenden Schritte, die für diesen Vorgang nötig sind. Um Platz zu sparen, wurde auf Anweisungen zur Fehlerbehandlung verzichtet. In der Abbildung ist zu erkennen, daß die mit "?" markierten Platzhalter im betroffenen SQL-Statement zur Laufzeit mit Werten besetzt werden.

```
/* Erzeugen des Statements */
PreparedStatement myStatement = databaseConnection.
   prepareStatement("UPDATE ACCOUNTS SET ABALANCE =
   ABALANCE + ? WHERE AID = ?");

/* Setzen der Parameter */
myStatement.setBigDecimal(1, <Wert>);
myStatement.setBigDecimal(2, <Wert>);

/* Ausführen des Statements */
myStatement.executeUpdate();
```

Abbildung 7.9: Ausführen eines `PreparedStatement`

Unter Linux war durch das Profiling[15] zu beobachten, daß der SQLJ/JDBC-Treiber den `BigDecimal` Wert aus der Abbildung in einen `String` Wert umwandelt, bevor dieser an die Datenbank geschickt wird. Dieses Verhalten wurde durch folgende Beobachtung festgestellt: Die `toString()` Methode der `BigDecimal` Klasse wird innerhalb einer Klasse des SQLJ/JDBC-Treibers aufgerufen. Auf der Grundlage

[14] Zur Erinnerung: Es existiert ein `MiddleWare` Objekt pro Worker JVM.

[15] Genauer: Durch die *Stack Traces* in der Ausgabedatei, die durch den HPROF Profiler Agent erzeugt wird.

dieser Beobachtung wurde folgende Optimierung vorgenommen: Die `BigDecimal` Werte in der Abbildung werden durch `String` Werte ersetzt, da sie in der Anwendung ohnehin in `String` Form vorliegen. Weiterhin müssen in der Abbildung alle `setBigDecimal()` Methoden durch entsprechende `setString()` Methoden ersetzt werden. Es bleibt noch zu bemerken, daß diese Optimierung lediglich für den SQLJ/JDBC-Treiber unter Linux angewendet werden kann. Unter z/OS erzeugt der Treiber beim Versuch, die entsprechende Spalte in der Datenbanktabelle mit Hilfe der `setString()` Methode zu setzen, eine `SQLException`.

Als Abschluß dieses Unterabschnitts bleibt für JDBC noch folgendes festzuhalten: In der JDK-Dokumentation wird darauf hingewiesen, daß Veränderungen an bestimmten `java.sql.Statement` Einstellungen einen Einfluß auf die Performance besitzen. Diese verursachten jedoch im Falle des Online-Banking Systems keine meßbaren Performance-Verbesserungen. So waren durch die Methoden `setFetchSize()`, `setMaxFieldSize()` und `setMaxRows()` keine Performance-Steigerungen festzustellen.

7.3.2 Datenbankeinstellungen und -design

Der grundlegende Aufbau der Datenbank wird durch die TPC-A Spezifikation festgelegt, wie in Abschnitt 5.6, “Die Datenbank”, nachzulesen ist. Nichtdestotrotz ergeben sich für die unterschiedlichen Plattformen einige Optimierungsmöglichkeiten, die im folgenden vorgestellt werden.

Optimierungen für beide Plattformen

Grundsätzlich gilt zunächst für alle Datenbanktabellen folgende Aussage: Das Anlegen eines Index ist für diejenigen Spalten einer Tabelle vorteilhaft, welche in SQL-Statements zum Lokalisieren eines bestimmten Eintrags benutzt werden. Beispiele für solche Spalten sind die Spalten der BRANCHES, TELLERS und ACCOUNTS Tabellen, welche den Primärschlüsseln entsprechen (BID, TID, AID).

Die SQL `UPDATE` Statements für die BOBS-Tabellen sind durchgängig wie folgt strukturiert: “`UPDATE ... WHERE <Primärschlüssel> = ...`”. Bei SQL-Statements dieser Art wird eine höhere Performance erreicht, wenn die Position des betroffenen Tabelleneintrags mittels eines Index berechnet wird (die Alternative besteht im sequentiellen Durchsuchen der gesamten Tabelle). Andererseits muß der Index angepaßt werden, wenn der Inhalt der Tabelle wächst. Dieses Anpassen benötigt CPU-Zeit. Aus diesem Grund wurde bei der HISTORY Tabelle auf einen Index verzichtet, da in diese Tabelle lediglich Einträge eingefügt, jedoch nie lokalisiert werden.

Optimierungen unter Linux

Für das DB2 Datenbankmanagementsystem unter Linux existiert aus diesem Grund die in Abbildung 7.10 auf Seite 77 gezeigte Tabellenoption, welche in dieser Form

unter z/OS nicht vorhanden ist. Durch diese Option werden neue Einträge am Ende der Tabelle eingefügt. Die sonst übliche DB2-Vorgehensweise, zuerst nach einem freien Platz zwischen zwei vorhandenen Einträgen zu suchen, wird hierdurch umgangen. Dies eignet sich in besonderem Maße für Tabellen wie die HISTORY Tabelle, in die ständig nur neue Einträge eingefügt werden.

```
ALTER TABLE HISTORY APPEND ON;
```

Abbildung 7.10: Setzen des `APPEND` Modus für eine Tabelle

Es existiert eine weitere Möglichkeit für Tabellen wie die HISTORY Tabelle, die Performance von SQL `INSERT` Statements zu steigern: Das Aktivieren der sogenannten *Multi-page File Allocation* unter Linux. Dieses Aktivieren erfolgt mit Hilfe des `db2empfa` Kommandozeilentools, das als Teil einer DB2-Installation mitgeliefert wird. Diese Option führt dazu, daß der Festplattenplatz für einen *Tablespace*[16] nicht seiten-, sondern *Extent*-weise vergrößert wird, wenn der Tablespace durch eine der Tabellen an seine Kapazitätsgrenze stößt. Die Größe für einen Extent kann bei der Definition der Tabelle angegeben werden.

Neben den bereits genannten Einstellungen existieren noch eine Reihe weiterer sogenannter *Konfigurationsschlüssel* für eine DB2-Datenbank unter Linux, sowie für das Datenbankmanagementsystem selbst. Die Vorgehensweise zur Ermittlung der jeweils optimalen Werte dieser Parameter entspricht der in Abschnitt 7.1.2 eingeführten Methodik (*Iterativer Tuningprozeß*). Im einzelnen wurde dabei wie folgt vorgegangen:

1. Die Datenbanktabellen erstellen.
2. Die BRANCHES, TELLERS und ACCOUNTS Tabellen mit dem `PopulateDataBase` Utility füllen, welches Teil des Online-Banking Systems ist.
3. Beim ersten Durchlauf: Den zu untersuchenden Parameter auf einen Ausgangswert (*Baseline*) einstellen. Bei Folgedurchläufen: Den Parameter auf den nächst höheren Wert einstellen.
4. Das Online-Banking System starten.
5. Das `TestDriver` Lasterzeugungs-Programm mindestens zehn Minuten lang mit zehn Terminal-Threads laufen lassen und die Durchsatzrate aufzeichnen.
6. Das Online-Banking System beenden.
7. Die Tabellen löschen.
8. Mit Schritt 1 fortfahren.

[16] Ein Tablespace ist eine Ansammlung von *Containern*. Ein Container beinhaltet eine oder mehrere Tabellen auf der Festplatte .

Dieser Prozeß stellt einen großen zeitlichen Aufwand dar. Die Performance-Steigerungen, welche sich durch diese Prozedur ergeben, rechtfertigen jedoch die Anstrengungen, wie Tabelle 7.5 zu entnehmen ist. Durch Veränderungen an den Einstellungen für das Datenbankmanagementsystem wurden keine Performance-Steigerungen festgestellt. Daher sind in der Tabelle nur die Datenbank-Konfigurationsschlüssel aufgelistet, die durch geeignete Veränderung zu einer Performance-Steigerung führen. Die Baseline ist dabei die Standardkonfiguration einer Datenbank, die beim Anlegen einer neuen Datenbank durch DB2 vorgenommen wird. Die angegebenen Performance-Steigerungen in Prozent beziehen sich auf diejenige Performance, welche durch Setzen aller vorherigen optimalen Werte erzielt wurde.

Konfigurationsschlüssel	Wert	Performance-Steigerung
BUFFPAGE	2000	1%
NUM_IOCLEANERS	3	1%
MINCOMMIT	12	81%
CHNGPGS_THRESH	70	2%
LOGBUFSZ	32	1%

Tabelle 7.5: Konfigurationsschlüssel für DB2-Datenbanken

Der BUFFPAGE Parameter gibt die Größe des Speicher-Pufferpools in Seiten an. Dabei ist darauf zu achten, daß im DB2-Systemkatalog SYSCAT.BUFFERPOOLS für denjenigen Pufferpool, welcher von der Datenbank benutzt wird, ein Wert von -1 für den NPAGES Parameter eingetragen wird. In der Standardeinstellung ist dies IBMDEFAULTBP. In der Regel können durch diesen Parameter höhere Performance-Steigerungen erzielt werden, denn: Je größer die Anzahl der Seiten im Puffer ist, desto höher ist die Wahrscheinlichkeit, daß DB2 einen Tabelleneintrag im Puffer findet und diesen nicht auf der Festplatte lokalisieren muß. Da die Online-Banking Tabellen jedoch (bis auf die HISTORY Tabelle) klein sind, ist der gewählte Puffer groß genug, um alle Einträge im Speicher zu halten. Dadurch konnte an dieser Stelle keine weitere Performance-Steigerung erreicht werden. Die einzige Tabelle, deren Inhalt kontinuierlich vergrößert wird, ist die HISTORY Tabelle. Da in diese lediglich Einträge eingefügt, jedoch niemals lokalisiert werden, spielt ein Puffer zum Lokalisieren von Einträgen für diese Art von Tabellen offensichtlich keine Rolle.

Der Wert für NUM_IOCLEANERS bezeichnet die Anzahl der sogenannten *asynchronen Seitenlöschfunktionen* und wurde wie auch CHNGPGS_THRESH (Schwellenwert für geänderte Seiten) und LOGBUFSZ (Protokollpuffergröße) zum Teil aus [DB2AG01] und zum Teil aus [An01] entnommen und experimentell verifiziert. Dort ist auch die Bedeutung dieser Parameter beschrieben, was an dieser Stelle zu weit gehen würde.

Wie aus der Tabelle zu entnehmen ist, verursachte eine Veränderung des MINCOMMIT Parameters den größten Performance-Schub. Dieser Parameter steht für die sogenannte *Anzahl der Gruppenfestschreibungen*. Damit ist gemeint, daß mehrere COMMIT

Punkte zusammen auf die Festplatte geschrieben werden. Der Standardwert dieses Parameters ist `1`. Dies bedeutet, daß jeder `COMMIT` Vorgang einzeln auf die Festplatte geschrieben wird.

Optimierungen unter z/OS

Um unter z/OS ebenfalls eine Performance-Steigerung für die HISTORY Tabelle zu erreichen, kann der gesamte für die Tabelle angelegte Tablespace vorformatiert werden. Hierfür muß bei einem Aufruf des erwähnten `REORG` Utility-Programms die `PREFORMAT` Option angegeben werden. Die Standard-Vorgehensweise in DB2 ist es, die Seiten in dem betroffenen Tablespace erst dann zu formatieren, wenn die Anzahl der freien Seiten für die in dem Tablespace enthaltenen Tabellen nicht mehr ausreicht. Die `PREFORMAT` Option veranlaßt das `REORG` Utility-Programm, alle Seiten des Tablespace im voraus zu formatieren.

Als Abschluß dieses Unterabschnitts werden in Tabelle 7.6 auf Seite 80 die optimalen Werte für die sogenannten `BIND` Parameter aufgelistet, die von der Standardkonfiguration abweichen. Bei der Ermittlung dieser Werte wurde folgende Methodik angewandt:

1. Die Datenbanktabellen erstellen.
2. Die BRANCHES, TELLERS und ACCOUNTS Tabellen mit dem `PopulateDataBase` Utility füllen, welches Teil des Online-Banking Systems ist.
3. Beim ersten Durchlauf: Den SQLJ-Plan mit den Ausgangswerten (*Baseline*) binden. Bei Folgedurchläufen: Den zu untersuchenden Parameter auf den nächst höheren (bei `CACHESIZE`) bzw. einen noch nicht benutzten (bei `VALIDATE`, `RELEASE` und `KEEPDYNAMIC`) Wert einstellen.
4. Das Online-Banking System starten.
5. Das `TestDriver` Lasterzeugungs-Programm mindestens zehn Minuten lang mit zehn Terminal-Threads laufen lassen und die Durchsatzrate aufzeichnen.
6. Das Online-Banking System beenden.
7. Die Tabellen löschen.
8. Mit Schritt 1 fortfahren.

Wiederum muß der große zeitliche Aufwand erwähnt werden, der für diese Prozedur notwendig ist. Die Baseline entspricht der Verwendung der Standardwerte, die von DB2 unter z/OS für das Binden eines Zugriffsplans benutzt werden.

Auf der Grundlage dieser Parameter wurde die in Tabelle 7.4 auf Seite 72 festgehaltene Untersuchung durchgeführt, die zur Ermittlung der optimalen Variante der `MiddleWare` Klasse unter z/OS diente.

Parameter	Optimaler Wert
VALIDATE	BIND
RELEASE	DEALLOCATE
KEEPDYNAMIC	YES
CACHESIZE	4096

Tabelle 7.6: DB2-Parameter für den BIND Vorgang

Während des Bindens eines Plans kann es vorkommen, daß bestimmte Objekte und Privilegien (noch) nicht vorliegen. Durch die VALIDATE(BIND) Option wird festgelegt, daß der BIND Vorgang fehlschlägt, wenn nicht alle diese Objekte und Privilegien zum Zeitpunkt des Bindens vorhanden sind. Dadurch kann DB2 einige Überprüfungen vermeiden, die ansonsten zur Laufzeit erfolgen müssen.

Der RELEASE Parameter legt fest, zu welchem Zeitpunkt DB2 die von einer Anwendung angeforderten Systemressourcen wieder freigibt. Der DEALLOCATE Wert bewirkt, daß diese Ressourcen erst nach dem Beenden der Anwendung wieder freigegeben werden (und nicht nach jedem COMMIT Punkt). Dadurch wird ein ständiges Anfordern und wieder Freigeben vermieden.

Durch das Setzen des KEEPDYNAMIC Parameters auf YES wird DB2 veranlaßt, dynamische SQL-Statements über COMMIT Punkte hinaus im sogenannten *Prepared Statement Cache* vorzuhalten. Andernfalls muß ein dynamisches SQL-Statement nach einem COMMIT Vorgang erneut *prepared* werden, auch wenn es in seiner Struktur nicht verändert wurde.

Der Wert des CACHESIZE Parameters (hier 4096) gibt die Größe des sogenannten *Authorization Cache* an. In diesem Cache werden User IDs gespeichert, die zur Ausführung des DB2-Plans berechtigt sind. Dadurch können *Catalog Lookups* vermieden werden.

Kapitel 8

Diskussion der Benchmark-Ergebnisse

Im vorliegenden Kapitel werden die Ergebnisse der Microbenchmarks und die Ergebnisse der Benchmarks mit dem Basic Online-Banking System diskutiert. Der Inhalt dieses Kapitels ist wie folgt aufgebaut: Abschnitt 8.1 gibt einen Überblick über die verschiedenen Benchmarks. Während sich die Ergebnisse der Microbenchmarks in Abschnitt 8.2 befinden, enthält Abschnitt 8.3 die Ergebnisse der Benchmarks mit dem Basic Online-Banking System.

8.1 Überblick

Einige (numerische) Meßergebnisse der Benchmarks wurden als vertraulich (*IBM Confidential*) eingestuft. Daher werden in diesem Kapitel in der Regel relative beziehungsweise qualitative Ergebnisse vorgestellt. Die quantitativen Ergebnisse befinden sich in den Anhängen B und C. Die Benchmarks sind im einzelnen:

- **Microbenchmarks** (Ergebnis **A**): Diese wurden unter z/OS und zLinux durchgeführt und dienen zur Leistungsmessung der Kernfunktionalitäten einer Java VM. Details zu den Microbenchmarks (Methodik, Ausführung) befinden sich in Kapitel 6.

- **Basic Online-Banking System** (Hoch- und Herunterfahren der Java VM, Ergebnis **B**): Dieser Benchmark wurde ebenfalls unter z/OS und zLinux durchgeführt. Das Hoch- und Herunterfahren der Java VM für jede Transaktion (siehe Abschnitt 2.2.6, “Java und Transaktionsverarbeitung”) stellt die bisherige Vorgehensweise dar, Transaktionsverarbeitung mit der Programmiersprache Java zu betreiben.

- **Basic Online-Banking System** (PRJVM im resettable Modus, Ergebnis **C**): Da die PRJVM-Technologie ausschließlich unter z/OS zur Verfügung steht, wurde dieser Benchmark lediglich unter diesem Betriebssystem durchgeführt. Details zur PRJVM-Technologie befinden sich in Kapitel 4.

- **Basic Online-Banking System** (Pure Java, Ergebnisse **D** und **E**): Dieser Benchmark wurde sowohl unter z/OS als auch unter zLinux durchgeführt. Er dient einerseits dazu, den Overhead des PRJVM-Programmiermodells unter z/OS zu bestimmen (Ergebnis **D**). Andererseits wurde er dazu benutzt, die Java VMs unter z/OS und zLinux mit einer praxisnahen Java-Applikation bezüglich der Performance zu vergleichen (Ergebnis **E**).

Um den Überblick über dieses Kapitel zu vervollständigen, werden im folgenden die untersuchten Software-Plattformen aufgeführt (die verwendete zSeries-Hardware war für alle Benchmarks dieselbe, siehe Abschnitt 8.3.1): Unter z/OS wurde das IBM Developer Kit for OS/390, Java 2 Technology Edition, SDK 1.3.1 in der Ausgabe vom 04. Februar 2003 benutzt, unter zLinux das IBM Developer Kit for Linux, Java 2 Technology Edition, SDK 1.3.1 vom 02. November 2002[1]. Diese Java VMs wurden für alle Benchmarks benutzt (sowohl für die Microbenchmarks als auch für das Online-Banking System).

8.2 Ergebnisse der Microbenchmarks (Ergebnis A)

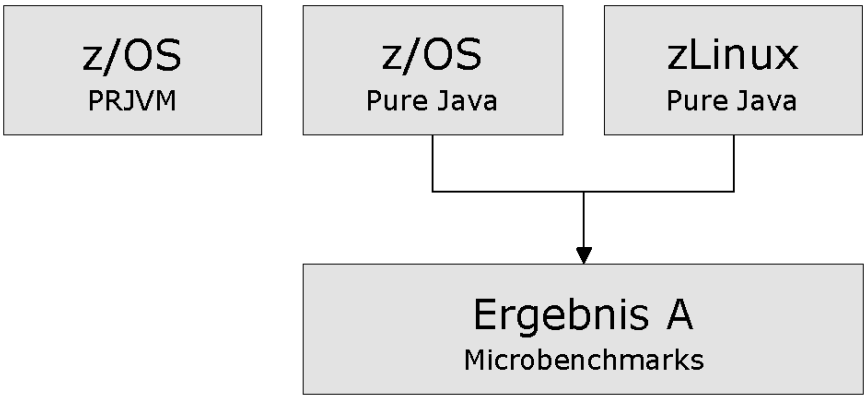

Abbildung 8.1: Einordnung der Microbenchmarks

Der vorliegende Unterabschnitt (sowie die folgenden) wurde mit einem Übersichts-Diagramm versehen, um die Einordnung dieses Benchmarks in die Auflistung aus dem Überblick (Ergebnis **A** bis **E**) graphisch darzustellen. Die Bezeichnung *Pure Java* in Abbildung 8.1 steht für die Java VM unter zLinux sowie für den "gewöhnlichen" (non-resettable) Modus der Java VM unter z/OS.

[1] Sämtliche spätere Versionen der SDK 1.3.1-Reihe, die noch (kurz) getestet werden konnten, hatten keine meßbaren Performance-Steigerungen zur Folge.

8.2.1 Methodenaufrufe

Mit Hilfe des ersten Microbenchmarks (`CallMethods`) wird die Anzahl der Methodenaufrufe gemessen, die von einer Java VM in einem vorgegebenen Zeitintervall durchgeführt werden können (*Methodenaufrufe pro Sekunde*, in der Abbildung: *calls/sec*). Eine Erklärung zu den unterschiedlichen Methodentypen (in der Abbildung: *method call type*) befindet sich in Tabelle 6.1 auf Seite 54. In Tabelle 8.1 ist das Performance-Verhältnis zwischen den beiden Java VMs festgehalten.

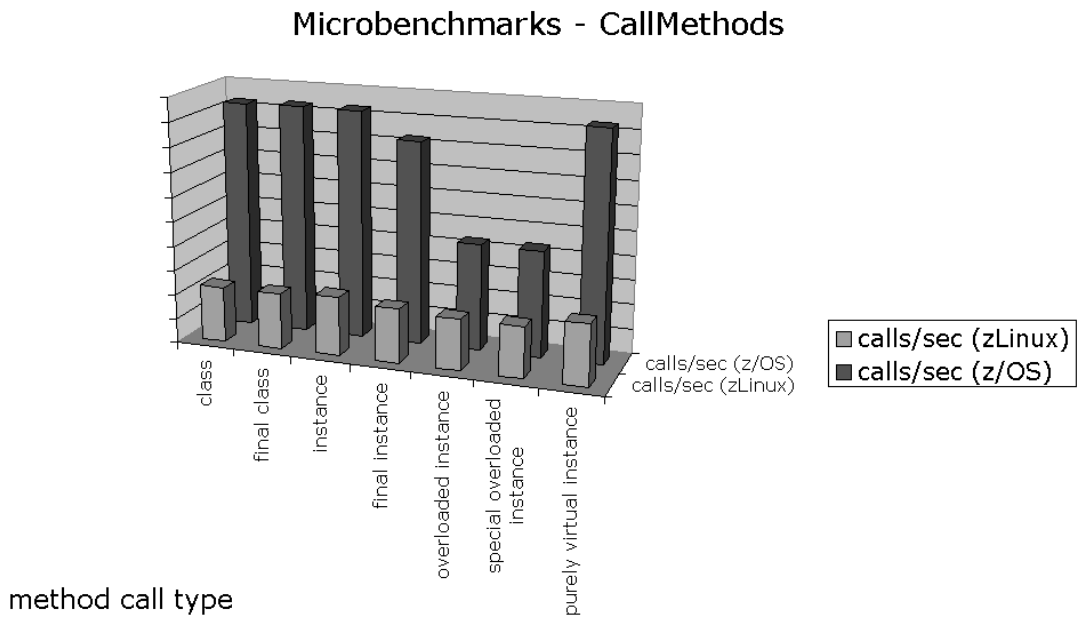

Abbildung 8.2: `CallMethods` Vergleichsdiagramm (relativ)

Methodentyp	Verhältnis z/OS : zLinux
class	4,11 : 1
final class	4,12 : 1
instance	3,88 : 1
final instance	3,76 : 1
overloaded instance	2,09 : 1
special overloaded instance	2,08 : 1
purely virtual instance	3,76 : 1

Tabelle 8.1: `CallMethods` Ergebnistabelle (relativ)

Auch nach Unterhaltung mit Experten konnte die Ursache für dieses Ergebnis nicht vollständig geklärt werden. Eine mögliche Erklärung besteht darin, daß der JIT-

Compiler unter z/OS bei Methodenaufrufen besser zu optimieren scheint als der JIT-Compiler unter zLinux. Der Grund für diese Vermutung ist folgender: Der Just-In-Time Compiler beeinflußt das Ergebnis dieses Microbenchmarks wesentlich. Wird er deaktiviert, so werden beispielsweise unter z/OS Werte erzielt, die im arithmetischen Mittel um den Faktor 56,23 niedriger liegen als die Werte, die mit aktiviertem JIT-Compiler erreicht werden.

8.2.2 Objekterzeugung

Abbildung 8.3 und Tabelle 8.2 auf Seite 85 enthalten die Ergebnisse für den `AllocateObjects` Microbenchmark. Mit Hilfe dieses Benchmarks werden bei steigender Objektgröße (in der Abbildung: *size [bytes]*) die Anzahl der Objekte gemessen, die eine Java VM in einem vorgegebenen Zeitintervall erzeugen kann (*Objekte pro Sekunde*, in der Abbildung: *objects/sec*). Da diese Java-Objekte auf dem Java Heap alloziert werden, ist dieser Benchmark Hauptspeicher-intensiv.

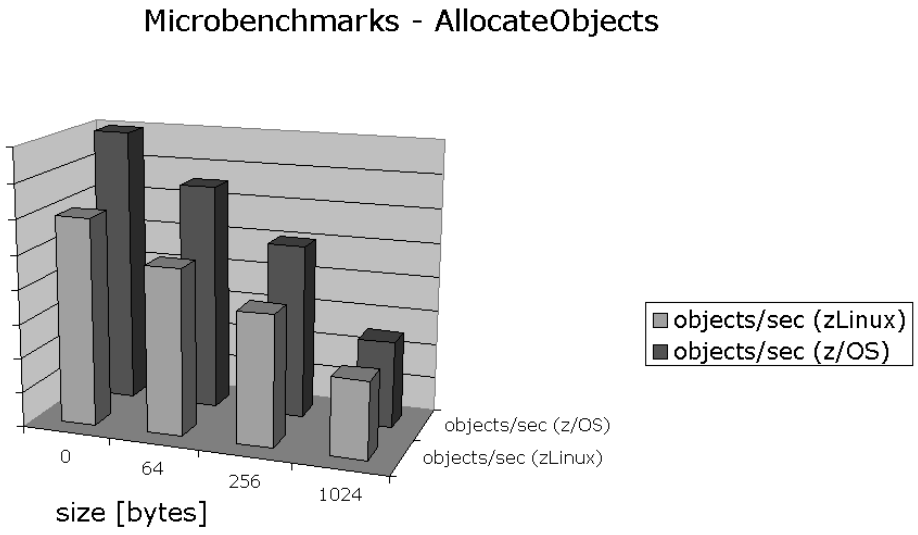

Abbildung 8.3: `AllocateObjects` Vergleichsdiagramm (relativ)

Aus der Tabelle ist ersichtlich, daß die beiden Java VMs bei diesem Microbenchmark annähernd dieselbe Performance erreichen. Auch hier konnte die Ursache für das (geringfügig) höhere Benchmark-Ergebnis der Java VM unter z/OS nicht geklärt werden. Ein Einfluß der Garbage Collection ist ausgeschlossen[2]. Daher besteht eine Vermutung für die Ursache dieses Ergebnisses darin, daß die Speicherverwaltung der Java VM unter z/OS besser optimiert ist als unter zLinux.

[2]Siehe hierzu Abschnitt 6.2, “Die Benchmarks im Detail”.

Größe des Objekts in Byte	Verhältnis z/OS : zLinux
0	1,33 : 1
64	1,36 : 1
256	1,33 : 1
1024	1,12 : 1

Tabelle 8.2: `AllocateObjects` Ergebnistabelle (relativ)

8.2.3 Threads und Locking

Die Ergebnisse des letzten Microbenchmarks (`AcquireLocks`) sind in Abbildung 8.4 und Tabelle 8.3 auf Seite 86 festgehalten. Mit Hilfe dieses Benchmarks werden bei steigender Anzahl konkurrierender Threads (in der Abbildung: *Threads*) die Anzahl der Locks gemessen, die eine Java VM in einem vorgegebenen Zeitintervall verwalten kann (*Locks pro Sekunde*, in der Abbildung: *locks/sec*). Auch hier müssen weitere Untersuchungen zur Klärung des Ergebnisses durchgeführt werden.

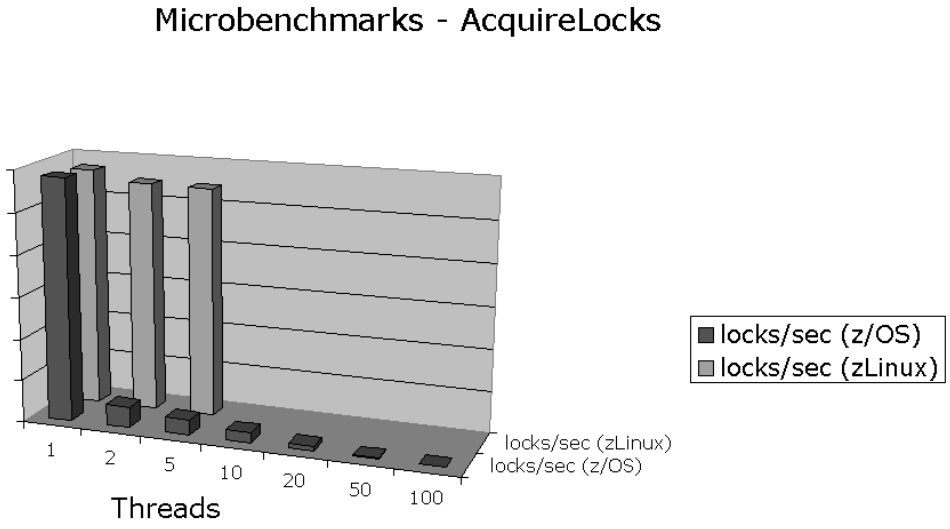

Abbildung 8.4: `AcquireLocks` Vergleichsdiagramm (relativ)

Auffällig ist in der Abbildung der große Abfall der Performance unter z/OS beim Übergang von einem auf zwei konkurrierende Threads. Wird der JIT-Compiler unter z/OS bei diesem Benchmark deaktiviert, so ist folgendes zu beobachten: Die erzielten Werte sind (bis auf den Wert für einen Thread) bei deaktiviertem JIT circa um den Faktor zwei niedriger als bei aktiviertem JIT. Bei der Messung mit einem Thread unter z/OS liegt der Wert um den Faktor 20 (bei aktiviertem JIT) höher als der Wert bei

Anzahl der Threads	Verhältnis z/OS : zLinux
1	1,01 : 1
2	0,09 : 1
5	0,07 : 1

Tabelle 8.3: `AcquireLocks` Ergebnistabelle (relativ)

deaktiviertem JIT. Daher besteht eine Vermutung für die Ursache dieses Benchmark-Ergebnisses darin, daß der JIT-Compiler unter z/OS bei einem Thread eine (Locking-) Optimierung vornimmt, die nicht auf zwei konkurrierende Threads übertragen werden kann.

Unter zLinux konnten bei Messungen mit mehr als fünf konkurrierenden Threads keine nachvollziehbaren Ergebnisse erzielt werden, da einzelne Ergebnisse um mehr als fünf[3] Prozent vom arithmetischen Mittel der Meßwerte abwichen. Daher bricht die Meßreihe nach fünf konkurrierenden Threads ab. Eine Vermutung für die Ursache dieses Verhaltens geht auf den Scheduler des zLinux-Betriebssystems zurück. So wäre es möglich, daß der Scheduler die zur Verfügung stehende Rechenzeit (bei einer Messung mit mehr als fünf konkurrierenden Threads) nicht zu gleichen Teilen auf die einzelnen Threads verteilen kann. Ebenso könnte die sogenannte *Time Slice* zu grobgranular sein. Als Time Slice bezeichnet man die Zeitscheibe, in welcher der Scheduler die CPU einem Prozeß (ununterbrochen) zuweist.

Eine andere Interpretation ergibt sich für die Meßwerte unter z/OS: Der Scheduler des z/OS-Betriebssystems verteilt die zur Verfügung stehende CPU-Zeit auf die einzelnen Threads bei allen Meßpunkten zu gleichen Teilen.

8.2.4 Zusammenfassung

Die Ergebnisse der Microbenchmarks lassen sich wie folgt zusammenfassen:

Ergebnis A Die Resultate der Microbenchmarks unter z/OS und zLinux unterscheiden sich um weniger als eine Größenordnung. Daher sind die beiden Java Virtual Machines unter Performance-Gesichtspunkten als vergleichbar zu bezeichnen.

Das Verhältnis der gemessenen Methodenaufrufe pro Sekunde (`CallMethods` Microbenchmark) beträgt im arithmetischen Mittel 3,4 : 1 (z/OS : zLinux). Aufgrund der wesentlichen Unterschiede bei `AcquireLocks` kann für diesen Microbenchmark kein Verhältnis angegeben werden (siehe die Diskussion im letzten Abschnitt). Bei `AllocateObjects` beträgt das Verhältnis der gemessenen Objekte pro Sekunde im arithmetischen Mittel 1,28 : 1 (z/OS : zLinux). Es muß darauf hingewiesen werden, daß sich diese Ergebnisse nicht bedingungslos auf alle Arten von Java-Anwendungen übertragen lassen (siehe Abschnitt 8.3.7, sowie die Einleitung zu Kapitel 6).

[3] Siehe hierzu Abschnitt 6.2, “Die Benchmarks im Detail”.

8.3 Basic Online-Banking System: Ergebnisse der Benchmarks (Ergebnisse B bis E)

Bevor die Ergebnisse **B** bis **E** vorgestellt werden können, müssen zunächst der Testaufbau (Abschnitt 8.3.1) und der Testablauf (Abschnitt 8.3.2) vorgestellt werden. Die meisten Performance-Ergebnisse mit dem Online-Banking System sind stark abhängig vom eingesetzten Transaktionstyp. Daher werden die beiden in dieser Arbeit benutzten Typen ebenfalls eingeführt (Abschnitt 8.3.3).

8.3.1 Der Testaufbau

Beide Systeme liefen als Gäste unter dem z/VM-Betriebssystem und hatten je eine[4] *dedizierte* CPU eines z900-Rechners (Modell 2064-109) zur Verfügung. Durch die Dedizierung ist sichergestellt, daß den z/VM-Gästen exakt die Rechenleistung einer physikalischen CPU zugewiesen wird. Während dem z/OS-Gast zwei GB Hauptspeicher zur Verfügung standen, wurden den zLinux-Gästen jeweils ein GB zugewiesen. Da die Benchmarks mit dem Online-Banking System primär auf die Erzeugung von CPU-Last ausgelegt sind, spielt die unterschiedliche Größe des zugewiesenen Hauptspeichers für deren Ergebnisse keine Rolle.

Weiterhin wurde ein *Enterprise Storage System* (ESS) vom Typ ESS 2105-F20 verwendet. Von diesem System wurden 19 Festplatten zu je 2,8 GB benutzt. Die Verbindung zum ESS wurde durch acht *Enterprise System Connectivity* (ESCON) Channels realisiert. Da die I/O-Last (ähnlich wie der Hauptspeicher) bei den folgenden Benchmarks keine zentrale Rolle spielt, wurde lediglich die CPU-Last aufgezeichnet und dokumentiert. Es existiert eine Ausnahme, die im entsprechenden Abschnitt analysiert wird.

Für eine Erläuterung des softwareseitigen Testaufbaus ist es vorteilhaft, sich zunächst mit Hilfe von Abbildung 8.5 auf Seite 88 einen Überblick über den Testaufbau zu verschaffen. Auf der soeben beschriebenen zSeries-Hardware wurde für die Benchmarks eine *Logical Partition* (LPAR) reserviert. In der LPAR wurde ein z/VM-Betriebssystem installiert. Unter z/VM wurden drei Gäste definiert, ein z/OS und zwei zLinux-Systeme.

Auf dem z/OS-Gast wurde je nach Benchmark-Konfiguration entweder das Online-Banking System in der hybriden Version oder die Variante in reinem Java gestartet. In der Abbildung entspricht die hybride Version der Beschriftung *PRJVM* und bezeichnet den *resettable* Modus der Java VM unter z/OS. *Pure Java* steht für die Variante des Online-Banking Systems in reinem Java und entspricht dem *non-resettable* Modus der Java VM unter z/OS.

Ein zLinux-Gast wurde zur Lasterzeugung benutzt (in der Abbildung: *TestDriver*). Unter Verwendung von TPC-A-Terminologie entspricht dieser zLinux-Gast dem

[4] In einer Benchmark-Konfiguration wurden dem z/OS-Betriebssystem zwei CPUs zugewiesen.

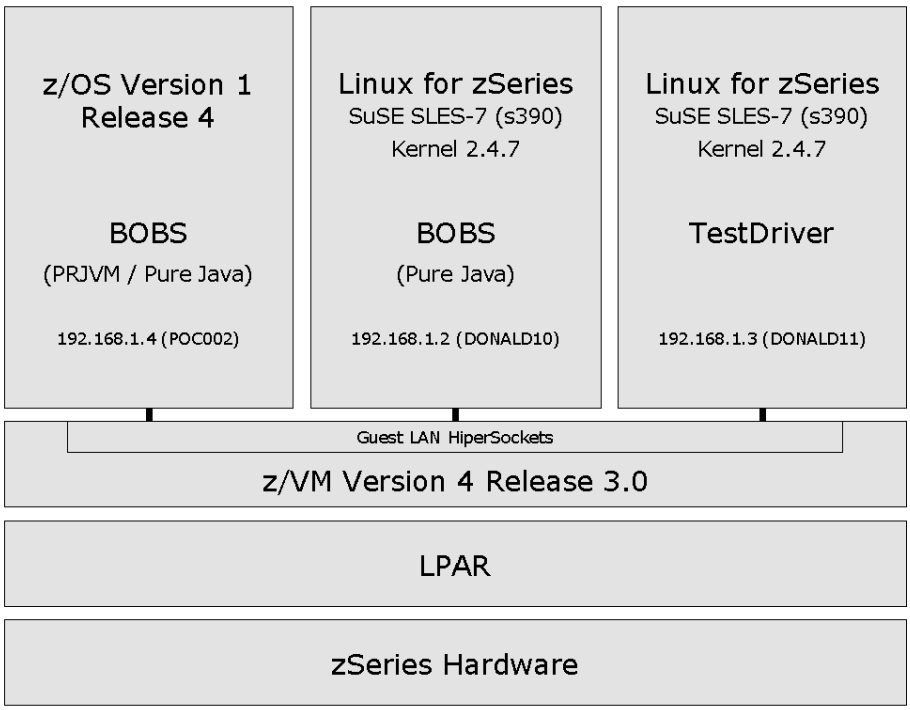

Abbildung 8.5: Der Testaufbau

Remote Terminal Emulator[5], während das z/OS-System dem *System Under Test* entspricht.

Neben dem z/OS-Gast wurde der zweite zLinux-Gast ebenfalls als System Under Test benutzt (in der Abbildung: *BOBS (Pure Java)*). Da der resettable Modus der Java VM (`-Xresettable`) unter zLinux nicht existiert, ist diese Variante im Testaufbau nicht aufgeführt.

Allen drei Gästen wurden lokale IP-Adressen zugewiesen (in der Abbildung: *192.168.1.x*). Die Gäste kommunizierten über sogenannte *HiperSockets*. Diese wurden im Testaufbau durch die *Guest LAN* Technologie des z/VM-Betriebssystems realisiert. HiperSockets ist eine mit der zSeries-Architektur eingeführte Microcode-Funktion, die unter Verwendung des Hauptspeichers eine sehr schnelle TCP/IP-Kommunikation zwischen zwei Servern ermöglicht (siehe hierzu [Whit02]). Diese Option wurde verwendet, um Bottlenecks auszuschließen, die durch das Netzwerk verursacht werden können.

8.3.2 Der Testablauf

Im folgenden wird der Ablauf eines Benchmarks mit dem Basic Online-Banking System beschrieben. In Tabelle 8.4 auf Seite 89 sind die Konfigurationsmöglichkeiten für

[5] Siehe hierzu Abschnitt 5.2, “Bezug zu TPC-A”.

die Benchmarks mit dem Basic Online-Banking System zusammengefaßt.

Modus	Full TPC-A	Reduced TPC-A
Hoch- und Herunterfahren der Java VM (z/OS, zLinux)	Ergebnis **B**, Abschnitt 8.3.4	Ergebnis **B**, Abschnitt 8.3.4
PRJVM im resettable Modus (ausschließlich z/OS)	Ergebnis **C**, Abschnitt 8.3.5 und Anhang C.1.1	Ergebnis **C**, Abschnitt 8.3.5 und Anhang C.1.2
Pure Java (z/OS)	Ergebnis **D**, Abschnitt 8.3.6 und Anhang C.2.1	Ergebnis **D**, Abschnitt 8.3.6 und Anhang C.2.2
Pure Java (zLinux)	Ergebnis **E**, Abschnitt 8.3.7 und Anhang C.3.1	Ergebnis **E**, Abschnitt 8.3.7 und Anhang C.3.2

Tabelle 8.4: Basic Online-Banking System: Benchmark-Konfigurationen

Der erste Schritt besteht im Hochfahren des Online-Banking Systems in der zu untersuchenden Konfiguration. Im zweiten Schritt erfolgt ein Probedurchlauf von 20000 Transaktionen. Damit wird sichergestellt, daß alle Java-Methoden vom Just-In-Time Compiler übersetzt wurden. Zu Beginn dieses Probedurchlaufs ist eine schwankende Durchsatzrate zu beobachten. Der Grund hierfür ist folgender: Die Java Virtual Machine kann in der Zeit, in welcher der JIT-Compiler die Methoden übersetzt, weniger Transaktionen durchführen als zu einem späteren Zeitpunkt, bei dem die Methoden übersetzt sind. Dieses Schwanken ist für die Messungen inakzeptabel und deshalb muß ein Probedurchlauf durchgeführt werden.

Nach dem Probedurchlauf erfolgen die eigentlichen Messungen. Das Testtreiber-Programm, das auf dem mit *TestDriver* beschrifteten zLinux-Gast aus Abbildung 8.5 läuft, wird mit einer (exponentiell) wachsenden Anzahl an Benutzern gestartet. Diese wurde wie folgt festgesetzt: 1, 2, 5, 10, 20, 50, 100, 200, 500 Benutzer.

Der Begriff *Benutzer* steht an dieser Stelle für eine TCP/IP-Verbindung, welche durch den Testtreiber aufgebaut und während eines Benchmark-Durchlaufs aufrechterhalten wird. Um den Bezug zur TPC-A Spezifikation herzustellen, kann der Ausdruck *Benutzer* mit *Terminal-Verbindung* oder *Terminal* gleichgesetzt werden.

Nach dem Aufbauen der TCP/IP-Verbindungen versucht der Testtreiber, auf jeder Verbindung so viele Transaktionen wie möglich (hintereinander) anzustoßen. Bei den Performance-Messungen mit BOBS existiert somit keine *Think Time*[6], wie sie bei einigen Benchmarks zur Simulation eines zufälligen Benutzerverhaltens üblich ist. Die TPC-A Spezifikation schreibt an dieser Stelle vor, daß der Testtreiber zwischen zwei Transaktionen so lange zu warten hat, bis das System Under Test die Antwort auf den Transaktions-Request zurückliefert. Diese Bedingung wird vom Testtreiber erfüllt.

[6] auf deutsch: Bedenkzeit

Jeder Benchmark-Durchgang dauert mindestens zwölf Minuten. Dadurch stehen den Monitoring-Programmen (siehe unten) mindestens zehn Minuten zur Datenauswertung zur Verfügung. Dies ist notwendig, da diese Programme auf Sampling-Basis und mit einem Intervall von 60 bzw. 100 Sekunden arbeiten. Das jeweils erste Intervall enthält somit Sampling-Daten, zu deren Zeitpunkt noch keine Last auf dem System Under Test erzeugt wurde. Es stellt also eine Art "Übergangsintervall" dar und wird daher für die spätere Auswertung nicht berücksichtigt.

Ist ein Benchmark-Durchgang vorüber, wird die Anzahl der gemessenen Transaktionen pro Sekunde auf der Konsole ausgegeben. Diese Ausgabe wird zu Auswertungszwecken gespeichert. Danach müssen die Messungen für zwei Sampling-Intervalle ausgesetzt werden. Dadurch wird das erste Sampling-Intervall eines neuen Benchmark-Durchgangs vom letzten des vergangenen Durchgangs getrennt. Anschließend wird der Testtreiber mit der nächst höheren Anzahl an Benutzern gestartet und die Messungen fortgesetzt.

Monitoring-Programme

Unter z/OS wurde die *Resource Measurement Facility* (RMF) und unter z/VM die *Full Screen Operator Console and Graphical Real Time Performance Monitor* (FCONX) eingesetzt. Beide Programme werden von einem Personal Computer aus mit Hilfe eines sogenannten *3270 Emulators* bedient.

FCONX eignet sich in besonderem Maße dazu, die *CPU Load* und *Virtual IO/s* Werte für alle in einem z/VM-Betriebssystem installierten Gast-Betriebssysteme auf einem (einzigen) *3270 Screen* anzuzeigen. Dieser Bildschirm (FCX112) wurde für jedes Sampling-Intervall jedes Benchmark-Durchgangs aufgezeichnet. Realisiert wurde dieses Aufzeichnen durch einen automatisierten Screenshot-Mechanismus[7]. Die so gewonnenen Screenshots wurden zur späteren Datenauswertung (CPU-Auslastung) benutzt.

Mit Hilfe von RMF wurde die verbrauchte CPU-Zeit für die vom Online-Banking System erzeugten Address Spaces aufgezeichnet. Der 3270 Screen hierfür ist *Processor Delays* und der zu beobachtende Wert *USG%* (Usage Percent). Wiederum wurde ein automatisierter Screenshot-Mechanismus benutzt, um jedes Sampling-Intervall jedes Benchmark-Durchgangs festzuhalten. Mit den so gewonnenen Daten konnte später die CPU-Auslastung für jede Komponente des Online-Banking Systems (die *jvmcreate* Prozesse, der *communicator* Prozeß, sowie DB2) getrennt analysiert werden. Mit Hilfe des *Delay Report* Bildschirms von RMF wurde die weiter unten beschriebene I/O-Problematik entdeckt. Für weiterführende Informationen zu RMF sei auf [RMFPMG02] und [RMFUG03] verwiesen.

Da zLinux (wie z/OS) als Gast-Betriebssystem unter z/VM installiert wurde, konnte FCONX ebenfalls für zLinux benutzt werden. Die FCONX-Daten (CPU Load im

[7] Hierfür wurde *IrfanView* verwendet, ein frei verfügbares Grafik-Programm für Windows-Betriebssysteme (siehe [IrfanView]).

FCX112-Screen) wurden auch im Falle von zLinux zur Auswertung der CPU-Auslastung des Systems benutzt.

Unter zLinux wurde für die verschiedenen Komponenten des Online-Banking Systems eine ähnliche Aufteilung der CPU-Last wie unter z/OS vorgenommen. Hierfür wurde das `top` Utility-Programm benutzt. Dieses bietet jedoch im Vergleich zu RMF lediglich rudimentäre Auswertungsmöglichkeiten der CPU-Aktivitäten. Weiterhin ist zu erwähnen, daß das `top` Programm wesentlich mehr CPU-Zeit benötigt als RMF (RMF wurde so konzipiert, daß es selbst kaum CPU-Zyklen benötigt). Das `top` Utility-Programm ist daher nur bedingt vergleichbar mit RMF.

8.3.3 Die Transaktionstypen

Manche Benchmarks beinhalten eine Mischung aus verschiedenen Transaktionstypen. So werden beispielsweise im TPC-C Benchmark fünf verschiedene Transaktionstypen definiert. Für die Benchmarks mit dem Basic Online-Banking System wurden zwei Transaktionstypen verwendet, die im folgenden vorgestellt werden.

Der *Full TPC-A* Transaktionstyp führt alle in der TPC-A Spezifikation geforderten Schritte durch. Die Reihenfolge der Schritte ist nicht vorgeschrieben ("The order of the data manipulations within the transaction is immaterial, [...]"). Im Online-Banking System werden die einzelnen Schritte wie folgt durchgeführt: Zuerst wird das betroffene Konto mit dem angegebenen Betrag belastet (Debit) bzw. der angegebene Betrag wird auf dem Konto gutgeschrieben (Credit). Dieser Vorgang erfolgt durch ein SQL `UPDATE` Statement. Danach wird der (neue) Kontostand durch ein SQL `SELECT` Statement ausgelesen. Anschließend wird der Tabelleneintrag des betroffenen Kassierers sowie der Zweigstelle aktualisiert (zwei SQL `UPDATE` Statements). Als letzter Schritt wird ein Eintrag in die HISTORY Tabelle vorgenommen (ein SQL `INSERT` Statement). Durch diesen wird der Buchungsvorgang mit allen Parametern festgehalten und mit einem Zeitstempel versehen. Wurden alle Einzelschritte ohne Fehler abgearbeitet, wird ein `COMMIT` durchgeführt und die Transaktion ist (erfolgreich) beendet. Andernfalls wird ein `ROLLBACK` durchgeführt und eine entsprechende Fehlermeldung für das Terminal generiert, von dem die Transaktionsanfrage initiiert wurde.

Beim Full TPC-A Transaktionstyp werden insgesamt fünf SQL-Statements durchgeführt[8]. Dies stellt eine hohe Aktivität auf der Seite der Datenbank dar. Um den Datenbank-Anteil einer Transaktion zu reduzieren (und somit den Java-Anteil zu erhöhen), wurde der *Reduced TPC-A* Transaktionstyp eingeführt. Bei diesem Transaktionstyp wird ausschließlich das oben beschriebene SQL `SELECT` Statement durchgeführt. Modifizierende (schreibende) Zugriffe auf die Datenbank finden somit nicht statt. Durch die Einführung dieses Transaktionstyps wurden einige (zusätzliche) Beobachtungen ermöglicht (siehe die folgenden Diskussionen).

[8] Wird der `COMMIT` Vorgang in diese Betrachtung einbezogen, erhöht sich die Anzahl der Statements auf sechs.

8.3.4 Hoch- und Herunterfahren der Java VM (Ergebnis B)

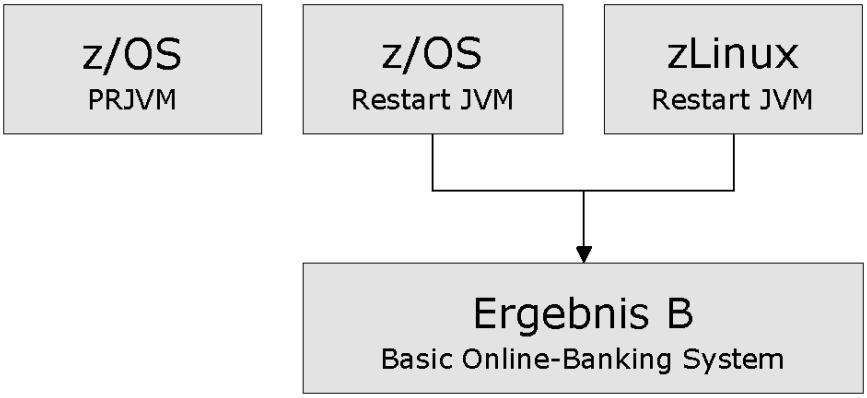

Abbildung 8.6: Hoch- und Herunterfahren der Java VM: Einordnung

Abbildung 8.6 dient zur Einordnung dieser Benchmark-Konfiguration in die Auflistung aus dem Überblick (Ergebnis **B**). Die Beschriftung *Restart JVM* bezeichnet dabei die Vorgehensweise des Online-Banking Systems für diese Benchmark-Konfiguration: Die Java VM wird für jede Transaktion neu gestartet.

In Tabelle 8.5 sind die Ergebnisse für diese Benchmark-Konfiguration festgehalten. Die erzielten Transaktionen pro Sekunde (tx/sec) sind bei diesem Benchmark unabhängig von der Anzahl der Benutzer. Weiterhin ist aus der Tabelle ersichtlich, daß die Performance für beide Plattformen nahezu identisch ist. Mit Nachdruck muß darauf hingewiesen werden, daß die ermittelten Werte extrem niedrig sind. Der Grund hierfür ist folgender: Die Pfadlänge für das Starten einer Java VM beträgt zwischen 20 und 100 Millionen Instruktionen (siehe [Borm01]), abhängig von der untersuchten Plattform. Transaktionspfadlängen sind dagegen typischerweise um ein bis zwei Größenordnungen niedriger (siehe [Borm01]). Die (enorm) hohen Kosten für das Starten einer Java Virtual Machine führen zu den angegebenen Werten.

Transaktionstyp	[tx/sec] z/OS	[tx/sec] zLinux
Full TPC-A	0,796	1,223
Reduced TPC-A	0,805	1,315

Tabelle 8.5: Restart JVM: [tx/sec] für z/OS und zLinux

Die Vorgehensweise, die Java VM für jede Transaktion neu zu starten, stellt somit keine Alternative für den CICS/COBOL-Ansatz zur Transaktionsverarbeitung dar. Die Werte aus Tabelle 8.5 verdeutlichen diesen Sachverhalt. Zusammenfassend läßt sich für diese Benchmark-Konfiguration folgendes festhalten:

Ergebnis B Der *Restart JVM* Ansatz für Java in Transaktionsverarbeitungssystemen ist für performancekritische Anwendungen inakzeptabel.

8.3.5 PRJVM im resettable Modus (Ergebnis C)

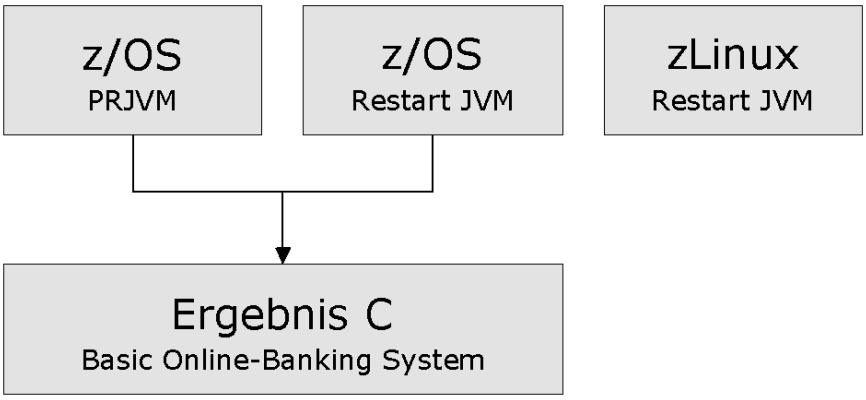

Abbildung 8.7: PRJVM im resettable Modus: Einordnung

Abbildung 8.8 auf Seite 94 und Tabelle 8.6 zeigen den Transaktionsdurchsatz für den Full TPC-A Transaktionstyp in Abhängigkeit von der Anzahl der Benutzer. Die Werte sind dabei in Transaktionen pro Sekunde (tx/sec) angegeben. Dem z/OS-Betriebssystem wurde in dieser Konfiguration eine CPU zugewiesen. Die Anzahl der Benutzer wurde wie in Abschnitt 8.3.2 beschrieben exponentiell erhöht.

Benutzer	[tx/sec] PRJVM	[tx/sec] Restart JVM	Verhältnis
1	149,08	0,796	187,29 : 1
2	223,65	0,796	280,97 : 1
5	261,04	0,796	327,94 : 1
10	251,68	0,796	316,18 : 1
20	244,61	0,796	307,30 : 1
50	229,20	0,796	287,94 : 1
100	242,40	0,796	304,52 : 1
200	243,96	0,796	306,48 : 1
500	237,49	0,796	298,35 : 1

Tabelle 8.6: Full TPC-A: [tx/sec] für PRJVM im resettable Modus

Mit Nachdruck seien an dieser Stelle die in der Tabelle aufgeführten Werte für das Verhältnis zwischen der PRJVM-Technologie und dem Restart JVM Ansatz betont. Auf der Grundlage dieser Beobachtung kann Ergebnis **C** formuliert werden:

Ergebnis C Bei einem TPC-A Benchmark-Szenario erreicht die Persistent Reusable Java Virtual Machines Technologie im Vergleich zu einer regulären Java Virtual Machine eine deutliche Leistungssteigerung (Faktor 327,94).

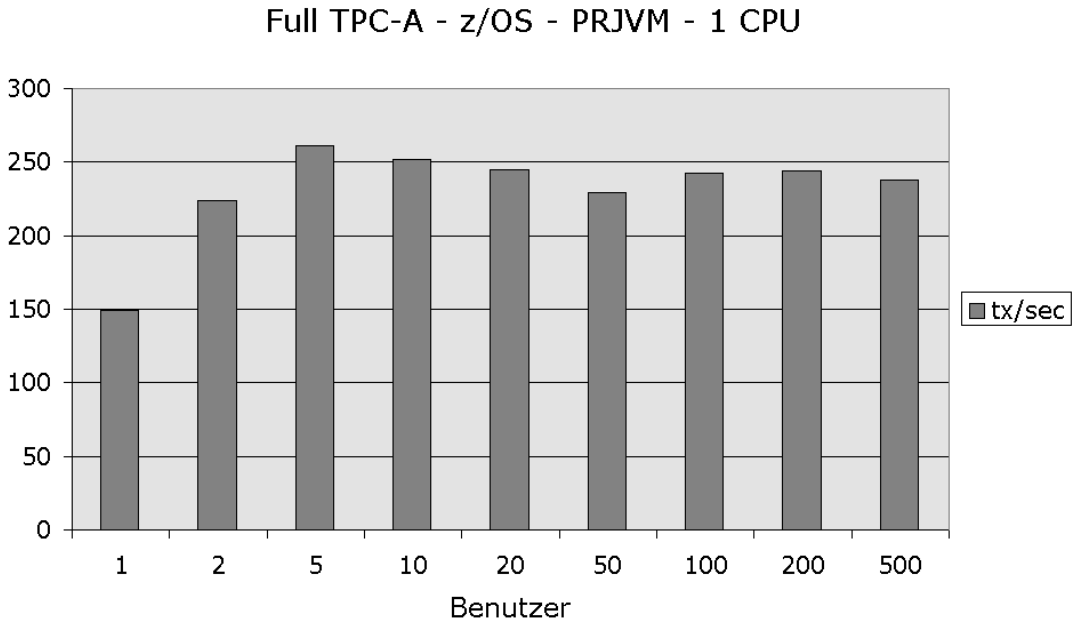

Abbildung 8.8: Full TPC-A: [tx/sec] für PRJVM im resettable Modus

Aufgrund dieses Ergebnisses wurden bei der Benchmark-Konfiguration “PRJVM im resettable Modus” zusätzliche Untersuchungen durchgeführt. Diese brachten im einzelnen folgendes zutage:

- Die Antwortzeit des Online-Banking Systems steigt linear mit der Anzahl der Benutzer.

Unter dem Gesichtspunkt der Skalierung ist es notwendig, nicht nur die Durchsatzrate (Transaktionen pro Sekunde), sondern auch die Antwortzeit zu untersuchen. In der Regel kann dabei festgestellt werden, daß die Antwortzeit ab einer bestimmten Anzahl an Benutzern exponentiell ansteigt und die Durchsatzrate einbricht. Bei diesem Punkt ist das System als überlastet anzusehen. Beim Online-Banking System konnte dieses Verhalten selbst bei 4000 gleichzeitigen Benutzern nicht beobachtet werden.

In der Regel wird bei Online-Banking Systemen eine Antwortzeit von weniger als einer Sekunde gefordert. Da die Antwortzeit bei Messungen mit mehr als 500 Benutzern oberhalb einer Sekunde liegt, wurden die Untersuchungen lediglich bis zu dieser Anzahl an Benutzern durchgeführt. Außerdem brachten die Untersuchungen folgendes zutage:

- Der Transaktionsdurchsatz ist stark abhängig vom verwendeten Transaktionstyp.

So werden mit dem Reduced TPC-A Transaktionstyp wesentlich höhere Benchmark-Ergebnisse erzielt als mit dem Full TPC-A Transaktionstyp (für quantitative Aussagen: siehe Anhang C.1.2). Die Ursache hierfür ist das Verhältnis 5 : 1 (Full TPC-A

: Reduced TPC-A) bezüglich der Anzahl der SQL-Statements, welche in den beiden Transaktionstypen abgearbeitet werden. Weiterhin ist folgendes zu beobachten:

- Die höchsten Werte einer Meßreihe werden ab fünf oder mehr Benutzern erzielt.

Der Grund für diese Beobachtung ist folgender: Der Transaktionsdurchsatz steigt proportional mit der CPU-Auslastung. Die CPU-Auslastung nähert sich mit steigender Benutzerzahl asymptotisch einem Grenzwert. Dieser Grenzwert beträgt bei den meisten Benchmark-Konfigurationen annähernd 100 Prozent und wird bei fünf oder mehr Benutzern erreicht. Dieser Sachverhalt bestätigt, daß im Online-Banking System keine Bottlenecks[9] vorhanden sind.

Das Ansteigen der CPU-Last mit wachsender Benutzerzahl ist intuitiv wie folgt zu erklären: Die zur Verfügung stehende CPU-Zeit kann durch die von einem (einzigen) Benutzer initiierten Transaktionen nicht vollständig ausgenutzt werden. Beispielsweise entstehen durch das Verschicken und Empfangen der Transaktionsanfragen und -antworten Verzögerungszeiten, in denen die CPU keine (transaktionsbezogenen) Instruktionen abarbeiten kann. Erst die parallelen Anfragen mehrerer Benutzer führen dazu, daß diese Verzögerungszeiten von der CPU benutzt werden können. Zusätzlich wurde folgendes beobachtet:

- Die durch das Basic Online-Banking System verursachte CPU-Auslastung wird im wesentlichen durch die PRJVM-bezogenen Komponenten bestimmt.

Diese Beobachtung wurde durch das in Abschnitt 8.3.2 beschriebene Monitoring ermöglicht. Die Ergebnisse dieser Untersuchung sind in Abbildung 8.9 auf Seite 96 festgehalten. Besonders muß dabei auf den geringen Anteil des Datenbankmanagementsystems *DB2 for z/OS and OS/390* hingewiesen werden. In der Benchmark-Konfiguration mit dem Reduced TPC-A Transaktionstyp wird dieser Anteil durch die reduzierte Anzahl an SQL-Statements nahezu vollständig eliminiert (siehe Anhang C.1.2).

Des weiteren muß betont werden, daß die Ergebnisse in Tabelle 8.6 unter Verwendung einer (einzigen) CPU eines zSeries-Rechners ermittelt wurden. Experimente mit dem Online-Banking System haben gezeigt, daß dieses (in Abhängigkeit vom verwendeten Transaktionstyp) annähernd linear mit der Anzahl der Prozessoren skaliert.

Eine Ausnahme hierfür wurde bei der Kombination “Full TPC-A Transaktionstyp und zwei CPUs” beobachtet. Bei dieser Konfiguration wurden die beiden CPUs (in Abhängigkeit von der Anzahl der Benutzer) maximal zu 175 Prozent ausgelastet (theoretisches Maximum: 200 Prozent). Das Online-Banking System erzielte aufgrund dieser Tatsache in der “Full TPC-A Transaktionstyp und zwei CPUs” Benchmark-Konfiguration nicht die doppelte Durchsatzrate der entsprechenden Konfiguration mit einer CPU.

Die Ursache für dieses Verhalten wurde mit Hilfe eines RMF Delay Reports ermittelt. Dieser ist in Abbildung 8.10 auf Seite 96 dargestellt. Eine Erklärung für den

[9] Siehe hierzu Abschnitt 7.1.2, “Konfiguration des Online-Banking Systems”.

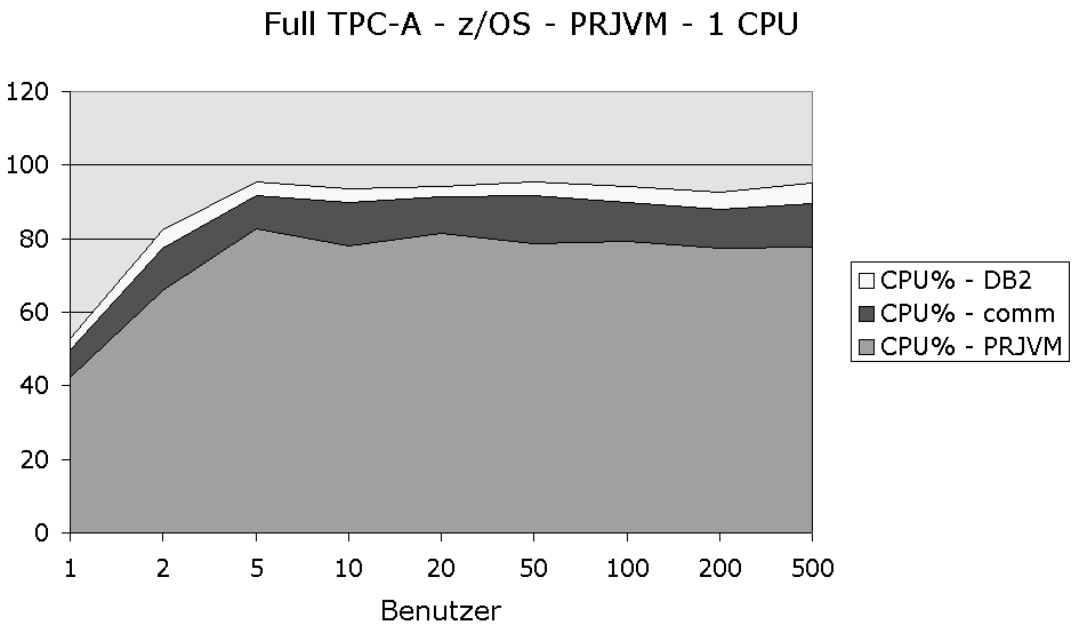

Abbildung 8.9: Full TPC-A: Verteilung der CPU-Zeit für PRJVM im resettable Modus

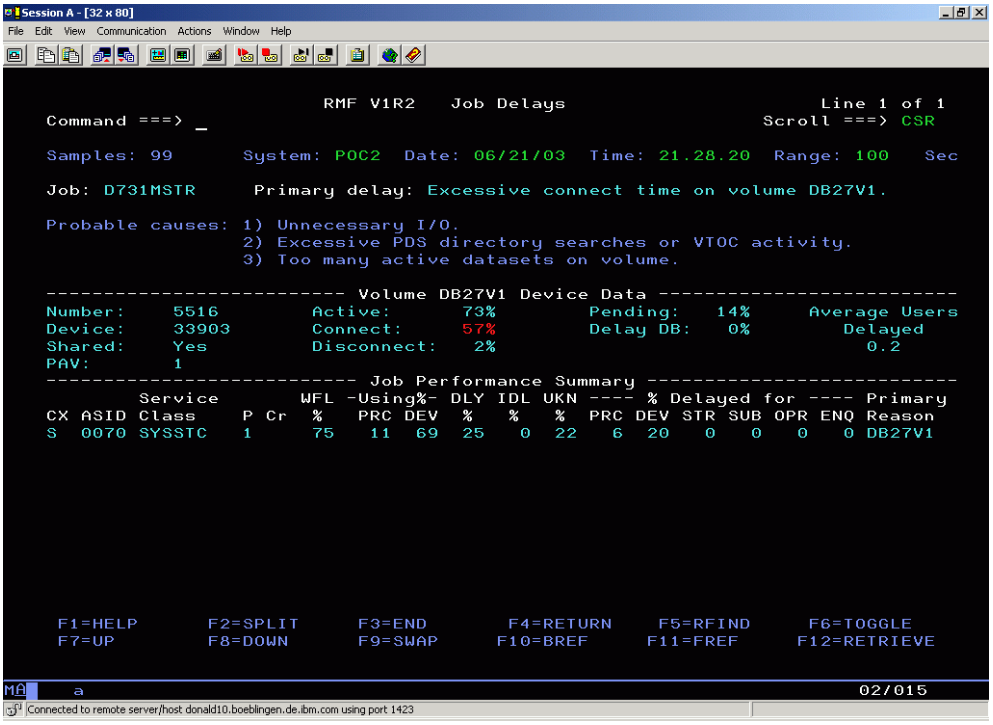

Abbildung 8.10: Excessive connect time on volume DB27V1

sogenannten *Job Delay* ist im Screenshot aufgeführt: “Excessive connect time on volume DB27V1.”

Zur Erklärung: Das *Logging* des DB2 Datenbankmanagementsystems verursachte so viele I/O-Operationen, daß die *Volume*, auf der sich die *Data Sets* für das Log befanden (DB27V1), zu stark belastet wurde. Dadurch wurde DB2 zu Wartezyklen - sogenannten *Delays* - gezwungen. Dieser Mißstand konnte in der gegebenen Zeit nicht behoben werden. Es existiert jedoch eine Lösung für diese Problematik: Die Data Sets für das DB2-Logging können auf verschiedenen Volumes angelegt werden. Dadurch wird die aufkommende I/O-Last auf mehrere Volumes verteilt.

Bei der Kombination “Reduced TPC-A Transaktionstyp und zwei CPUs” erreichte die CPU-Auslastung annähernd 200 Prozent. Im Vergleich mit der Konfiguration “Reduced TPC-A Transaktionstyp und eine CPU” wurde der Transaktionsdurchsatz nahezu verdoppelt. Dies zeigt die sehr gute Skalierungs-Fähigkeit des Online-Banking Systems. Für quantitative Aussagen muß wiederum auf den Anhang verwiesen werden (siehe Anhang C.1.2).

8.3.6 Pure Java unter z/OS (Ergebnis D)

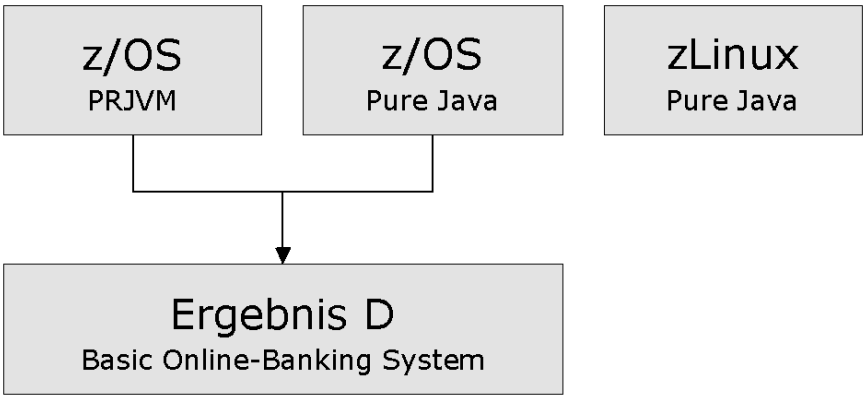

Abbildung 8.11: Pure Java unter z/OS: Einordnung

Das zSeries-System für die Performance-Tests wurde vom *Technical Marketing Competence Center* (TMCC) der IBM Deutschland Entwicklung GmbH zur Verfügung gestellt. Dieses System wird vom TMCC für Kundenprojekte eingesetzt. Dadurch konnten die Performance-Messungen nicht ununterbrochen durchgeführt werden. Zusätzlich wurde das System während der vorliegenden Diplomarbeit für eine weitere Diplomarbeit eingesetzt. Daraus entstand eine organisatorische Situation, die das Durchführen einiger Benchmark-Konfigurationen nicht zuließ. Die wichtigste Benchmark-Konfiguration stellt die Konfiguration “PRJVM im resettable Modus” (siehe letzter Abschnitt) dar. So wurden die Performance-Messungen mit zwei CPUs lediglich für diese Benchmark-Konfiguration durchgeführt. Die Performance-Messungen mit einer CPU wurden vollständig vorgenommen.

Das Verhalten des Online-Banking Systems bezüglich der Antwortzeit (siehe Abschnitt 8.3.5: linearer Anstieg mit der Anzahl der Benutzer), sowie die starke Abhängigkeit des Transaktionsdurchsatzes vom verwendeten Transaktionstyp wurde ebenfalls in der Benchmark-Konfiguration “Pure Java unter z/OS” beobachtet. Die höchsten Werte für den Transaktionsdurchsatz wurden bei Messungen mit zehn oder mehr Benutzern ermittelt (für quantitative Ergebnisse: siehe Anhang C.2). Die I/O-Problematik wurde nicht beobachtet (die CPU wurde nahezu vollständig ausgelastet).

Mit Hilfe dieses Benchmarks wird der Overhead des PRJVM-Programmiermodells unter z/OS bestimmt (Ergebnis **D**). Dieser Overhead setzt sich wie folgt zusammen:

- Vor jeder Transaktion: Lokalisieren der Java-Klassen, deren Instanzen im Transient Heap positioniert werden (im Launcher Subsystem: `FindClass()` JNI-Funktion). Ein Cachen dieser Klassen ist nicht möglich (die Java VM erzeugt ein *Unresettable Event*).
- Nach jeder Transaktion: Reset-Vorgang der PRJVM.
- Allgemein: Synchronisation zwischen den Worker JVMs und dem *communicator*, sowie zwischen den Worker JVMs untereinander (bedingt durch Shared Memory und Semaphore-Techniken: Kopieren der Transaktionsanfragen und -antworten in das bzw. aus dem Shared Memory Segment).

Oben aufgeführte Punkte sind charakteristisch für ein Launcher Subsystem und somit unumgänglich (siehe [PRJVM01]). In Tabelle 8.7 werden die Ergebnisse (Transaktionsdurchsatz) der Benchmark-Konfigurationen “Pure Java unter z/OS” und “PRJVM im resettable Modus” für den Full TPC-A Transaktionstyp verglichen (siehe auch Anhang C.2.1).

Benutzer	Verhältnis Pure Java (z/OS) : PRJVM (z/OS)
1	1,24 : 1
2	1,32 : 1
5	1,47 : 1
10	1,75 : 1
20	1,70 : 1
50	1,81 : 1
100	1,71 : 1
200	1,63 : 1
500	1,66 : 1

Tabelle 8.7: Full TPC-A: [tx/sec] bei PRJVM und Pure Java (z/OS, relativ)

Wie aus der Tabelle ersichtlich ist, verursacht das PRJVM-Programmiermodell unter z/OS einen deutlichen Overhead. Dieser beträgt für den Full TPC-A Transaktionstyp im arithmetischen Mittel 1,59. Mit anderen Worten läßt sich für den Full TPC-A

Transaktionstyp folgendes festhalten: Der hohe Grad an Isolation, welcher durch die PRJVM-Technologie realisiert wird, resultiert im Vergleich zum ungeschützten Abarbeiten der Transaktionen nebeneinander[10] in einer 1,59-fach niedrigeren Ausführungszeit.

Benutzer	Verhältnis Pure Java (z/OS) : PRJVM (z/OS)
1	1,72 : 1
2	1,86 : 1
5	2,14 : 1
10	2,32 : 1
20	2,38 : 1
50	2,35 : 1
100	2,32 : 1
200	2,28 : 1
500	2,22 : 1

Tabelle 8.8: Reduced TPC-A: [tx/sec] bei PRJVM und Pure Java (z/OS, relativ)

Noch deutlicher ist der Overhead zu beobachten, wenn man den Reduced TPC-A Transaktionstyp verwendet. Tabelle 8.8 enthält die Ergebnisse (Transaktionsdurchsatz) für diese Benchmark-Konfiguration (siehe auch Anhang C.2.2). Im arithmetischen Mittel beträgt der Overhead hierbei 2,18. Der im Vergleich zur Konfiguration mit dem Full TPC-A Transaktionstyp höhere Overhead läßt sich wie folgt erklären: Beim Reduced TPC-A Transaktionstyp werden lediglich ein Fünftel der Datenbankoperationen des Full TPC-A Typs durchgeführt. Dadurch verringert sich der Datenbank-Anteil an der gesamten Ausführungszeit einer Transaktion erheblich. Der CPU stehen beim Reduced TPC-A Transaktionstyp für die restlichen Teiloperationen der Transaktion mehr Zyklen zur Verfügung als beim Full TPC-A Transaktionstyp. Zu diesen restlichen Teiloperationen zählen auf der einen Seite die Ausführung der Java-Methoden (Auswertung der Transaktionsparameter usw.), auf der anderen Seite die für ein Launcher Subsystem spezifischen C und Java Native Interface Funktionen. Dadurch tritt der Overhead, welcher durch die PRJVM-spezifischen Teiloperationen verursacht wird, deutlicher zutage als beim Full TPC-A Transaktionstyp. Zusammenfassend läßt sich für die Pure Java Version des Basic Online-Banking System unter z/OS folgendes festhalten:

Ergebnis D Das Programmier- und Ausführungsmodell der Persistent Reusable Java Virtual Machines Technologie verursacht einen deutlich meßbaren Overhead. Dieser ist abhängig vom verwendeten Transaktionstyp.

[10] Dies ist die Vorgehensweise in der *Pure Java* Version des Basic Online-Banking Systems.

8.3.7 Pure Java unter zLinux (Ergebnis E)

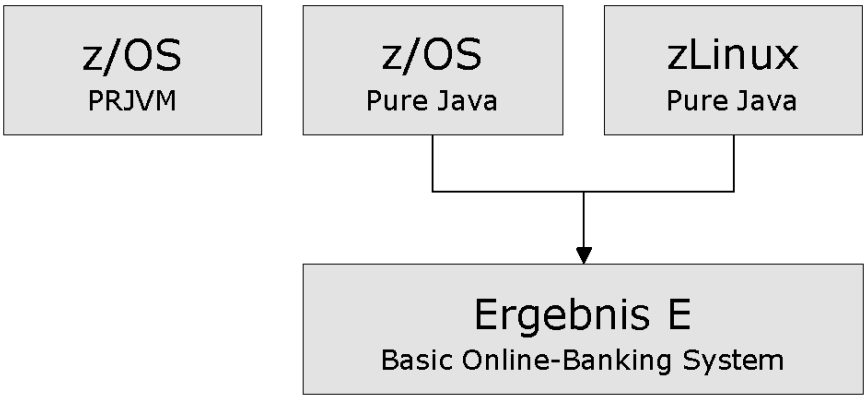

Abbildung 8.12: Pure Java unter zLinux: Einordnung

In der vorliegenden Benchmark-Konfiguration “Pure Java unter zLinux” wurden die Performance-Messungen lediglich mit einer CPU durchgeführt (siehe Bemerkung in Abschnitt 8.3.6). Das Verhalten des Online-Banking Systems bezüglich der Antwortzeit (siehe Abschnitt 8.3.5: linearer Anstieg mit der Anzahl der Benutzer), sowie die starke Abhängigkeit des Transaktionsdurchsatzes vom verwendeten Transaktionstyp wurde hier ebenfalls beobachtet. Die höchsten Werte für den Transaktionsdurchsatz wurden bei Messungen mit zehn (Full TPC-A Transaktionstyp) bzw. zwei (Reduced TPC-A Transaktionstyp) Benutzern ermittelt (für quantitative Ergebnisse: siehe Anhang C.3). Die Aufteilung der CPU-Zeit für die unterschiedlichen Komponenten des Basic Online-Banking Systems ist in Anhang C.3.1 (Full TPC-A Transaktionstyp) und Anhang C.3.2 (Reduced TPC-A Transaktionstyp) aufgeführt.

Um die Performance der Java VMs unter z/OS und zLinux mit einer praxisnahen Java-Applikation zu vergleichen (Ergebnis **E**), wurde die Leistung der Pure Java Version des Basic Online-Banking System unter beiden Plattformen gemessen. Die Ergebnisse dieses Vergleichs werden im folgenden vorgestellt.

Beim Full TPC-A Transaktionstyp erreichte die CPU-Auslastung unter zLinux maximal 55,46 Prozent. Diese Beobachtung deutet auf einen Bottleneck des Basic Online-Banking Systems in dieser Benchmark-Konfiguration hin. Trotz intensiver Analyse und Unterhaltung mit Experten konnte die Ursache hierfür nicht geklärt werden. In diesem Bereich sind weitere Untersuchungen notwendig. Um dennoch diese Benchmark-Konfiguration mit der entsprechenden unter z/OS zu vergleichen (Pure Java, Full TPC-A Transaktionstyp), wurden in Tabelle 8.9 auf Seite 101 beide Ergebnisse auf eine CPU-Auslastung von 100 Prozent hochskaliert und miteinander verglichen. Diese (theoretischen) Werte beruhen auf der Annahme, daß sich die geschilderte Problematik vollständig eliminieren läßt.

Um für die zLinux Java VM reale (gemessene) Werte zur Verfügung zu stellen, wurden Performance-Messungen mit dem Reduced TPC-A Transaktionstyp durchge-

Benutzer	Verhältnis zLinux : z/OS
1	1,44 : 1
2	1,41 : 1
5	1,33 : 1
10	1,22 : 1
20	1,22 : 1
50	1,21 : 1
100	1,21 : 1
200	1,25 : 1
500	1,28 : 1

Tabelle 8.9: Full TPC-A: Skalierte [tx/sec] bei Pure Java (relativ)

führt. Bei diesen wurde eine CPU-Auslastung von nahezu 100 Prozent erreicht. Dadurch kann diese Benchmark-Konfiguration mit der entsprechenden unter z/OS direkt (ohne Skalierung) verglichen werden. In Tabelle 8.10 sind die Ergebnisse dieser Untersuchung festgehalten.

Benutzer	Verhältnis zLinux : z/OS
1	1,43 : 1
2	2,21 : 1
5	1,69 : 1
10	1,48 : 1
20	1,44 : 1
50	1,44 : 1
100	1,42 : 1
200	1,47 : 1
500	1,58 : 1

Tabelle 8.10: Reduced TPC-A: [tx/sec] bei Pure Java (relativ)

Wie aus der Tabelle ersichtlich ist, erreicht die Pure Java Version des Basic Online-Banking Systems unter zLinux eine höhere Performance als unter z/OS. Im arithmetischen Mittel beträgt die Performance im Vergleich 1,57 : 1 (zLinux : z/OS). Die Ursache für diesen Performance-Unterschied konnte auch nach Unterhaltung mit Experten nicht vollständig geklärt werden. Folgendes wurde im Laufe der Untersuchungen beobachtet: Vergleiche mit dem HPROF Profiler Agent[11] brachten zutage, daß der SQLJ/JDBC-Treiber unter z/OS einen wesentlichen Anteil seiner gesamten Ausführungszeit für die `doPrivileged()` Methode der `AccessController` Klasse aus dem `java.security` Package aufwendet. Diese Methode wird dazu verwendet, einen Teil des Programmcodes mit erweiterten Rechten auszuführen, während der

[11] Man erinnere sich an Abschnitt 7.2, “Umsetzung in reines Java (Pure Java)”.

restliche Teil mit den Rechten der aktuellen sogenannten *Protection Domain* ausgeführt wird. Ausführliche Informationen zu dieser Thematik sind beispielsweise in der Dokumentation des Java Development Kit enthalten (Abschnitt “Guide to Features - Security”, “API for Privileged Blocks“). Diese Beobachtung deutet darauf hin, daß unter z/OS einige zusätzliche Sicherheitsmaßnahmen ergriffen werden, die in dieser Form unter zLinux nicht beobachtet werden können. Zusammenfassend läßt sich folgendes Ergebnis formulieren:

Ergebnis E Die Performance des Basic Online-Banking Systems in der Pure Java Version ist unter zLinux höher als unter z/OS. Das Performance-Verhältnis ist dabei abhängig vom verwendeten Transaktionstyp.

Kapitel 9

Zusammenfassung und Ausblick

In Kapitel 3 wurde eine Architekturanalyse der Java Virtual Machines unter den Betriebssystemen z/OS und Linux durchgeführt. Darauf aufbauend wurde unter Verwendung der PRJVM-Technologie (Kapitel 4) das Basic Online-Banking System (Kapitel 5) implementiert. Dieses Online-Banking System wurde für den Leistungsvergleich der Java VMs unter z/OS und *Linux for zSeries* (zLinux) benutzt (Abschnitt 8.3). In Abschnitt 8.2 sind die Ergebnisse der Microbenchmarks (Kapitel 6) festgehalten.

Bei den Benchmarks mit dem Basic Online-Banking System wurde eine Vielzahl an Konfigurationen untersucht (siehe Tabelle 8.4 auf Seite 89). Die Ergebnisse der wichtigsten Benchmark-Konfigurationen sind in folgender Tabelle festgehalten (Transaktionsdurchsatz, Angaben in Transaktionen pro Sekunde):

Transaktionstyp	zLinux	z/OS, bisher	z/OS, PRJVM	z/OS, Verhältnis
Full TPC-A	1,223	0,796	261,04	1 : 327,94
Reduced TPC-A	1,315	0,805	511,77	1 : 635,74

Zwei überraschende Ergebnisse sind besonders hervorzuheben:

1. Unter Gesichtspunkten des Leistungsverhaltens ist der bisherige Ansatz (siehe Abschnitt 2.2.6), Transaktionsverarbeitung mit der Programmiersprache Java zu betreiben, unbrauchbar.

2. Bei der Transaktionsverarbeitung erbringt der Einsatz der Persistent Reusable Java Virtual Machines Technologie im Vergleich zu einer regulären Java Virtual Machine einen Leistungsgewinn um einen Faktor 327,94.

Traditionelle Transaktionsverarbeitungssysteme wie CICS (in Verbindung mit COBOL-Transaktionen) weisen höhere Durchsatzraten auf. Aufgrund der Leistungsmerkmale der PRJVM-Technologie existieren jedoch bereits Aussagen der Art “[...] how close do you have to be?” (siehe [Horo02]). So scheint es zum jetzigen Zeitpunkt nur eine

Frage der Zeit, bis die PRJVM-Technologie zum Standard für Transaktionsverarbeitungssysteme erklärt wird. Eine Umsetzung der Technologie auf andere Plattformen ist (noch) nicht bekannt.

Als Abschluß dieser Diplomarbeit sei bemerkt, daß durch die Einführung der Persistent Reusable Java Virtual Machines Technologie ein Sachverhalt bestätigt wurde, der ergänzend zu den übrigen Ergebnissen dieser Arbeit einzuordnen ist (siehe [ACT03]): “Kein anderes Betriebssystem hat eine ähnlich lange Vergangenheit wie OS/390. Wenige Plattformen erfahren so konstant Neuerungen.”

Anhang A

Inhalt der beigefügten CD

Die beigefügte CD weist folgende Verzeichnisstruktur auf:

- `BOBS`: Dieses Verzeichnis enthält die verschiedenen Versionen des Basic Online-Banking Systems. Voraussetzung für BOBS ist eine DB2-Installation. Für Linux kann eine Testversion des DB2 Datenbankmanagementsystems unter [DB2PF] bezogen werden. Die BOBS-Versionen enthalten Skript-Dateien für die Übersetzung und Ausführung.
- `Literaturreferenzen`: Dieses Verzeichnis enthält die in elektronischer Form verfügbaren Literaturreferenzen (einschließlich Internet-Artikel).
- `Meßergebnisse`: Dieses Verzeichnis enthält die Screenshots der RMF und FCONX Monitoring-Programme (siehe Abschnitt 8.3.2), die zur Datenauswertung benutzt wurden. Zusätzlich sind die Screenshots der Konsolenausgaben enthalten. Weiterhin befindet sich in diesem Verzeichnis eine Tabelle mit den Meßergebnissen im Microsoft Excel-Format.
- `Microbenchmarks`: Dieses Verzeichnis enthält die Microbenchmarks.
- `SQL`: Dieses Verzeichnis enthält die SQL-Anweisungen zum Anlegen der Datenbanktabellen des Basic Online-Banking Systems für die Betriebssysteme z/OS und Linux. Zusätzlich befinden sich hier die *Job Control Language* (JCL) Statements für das Binden des DB2 SQLJ-Plans unter z/OS, sowie Jobs für die `REORG` und `RUNSTATS` Utility-Programme.

Anhang B

Ergebnisse der Microbenchmarks

Der vorliegende Anhang enthält die quantitativen Ergebnisse der Microbenchmarks. Die Diskussion dieser Ergebnisse befindet sich in Abschnitt 8.2.

B.1 Methodenaufrufe

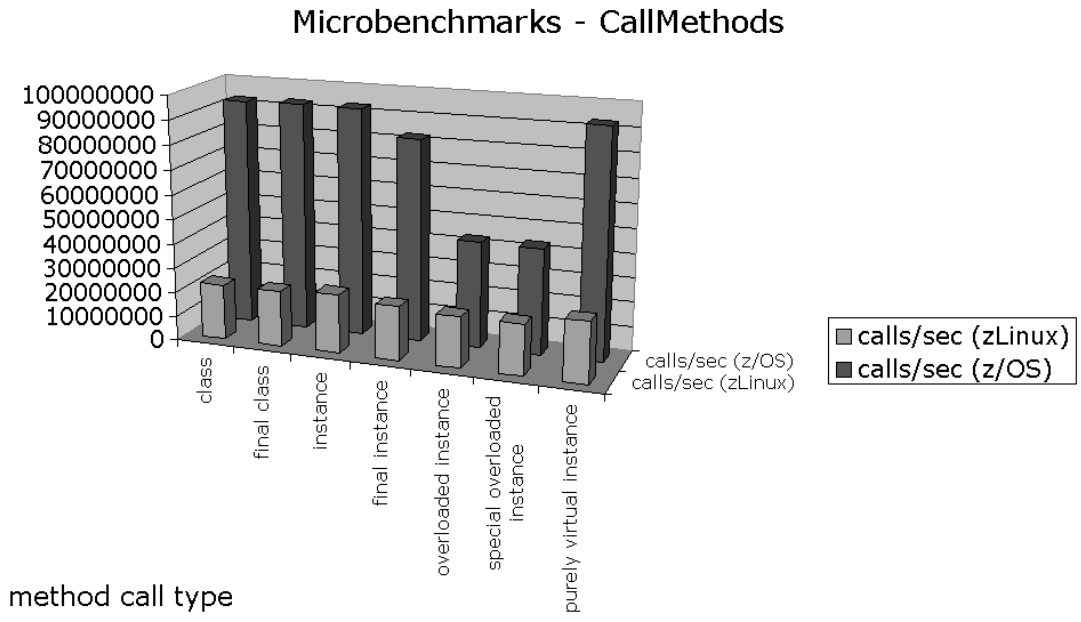

Abbildung B.1: `CallMethods` Vergleichsdiagramm

Deutlich ist in Abbildung B.1 sowie in Tabelle B.1 auf Seite 108 der Performance-Unterschied zwischen den Java VMs unter z/OS und zLinux zu erkennen. Das Verhältnis der gemessenen Methodenaufrufe pro Sekunde (in der Abbildung: *calls/sec*)

Methodentyp	z/OS	zLinux	Verhältnis
class	92850510,68	22598870,06	4,11 : 1
final class	93144560,36	22598870,06	4,12 : 1
instance	93040565,69	24009603,84	3,88 : 1
final instance	82603667,60	21957914,00	3,76 : 1
overloaded instance	43301290,38	20721809,70	2,09 : 1
special overloaded instance	43275056,26	20761245,67	2,08 : 1
purely virtual instance	93092533,98	24783147,46	3,76 : 1

Tabelle B.1: `CallMethods` Ergebnistabelle

beträgt im arithmetischen Mittel 3,4 : 1 (z/OS : zLinux).

B.2 Objekterzeugung

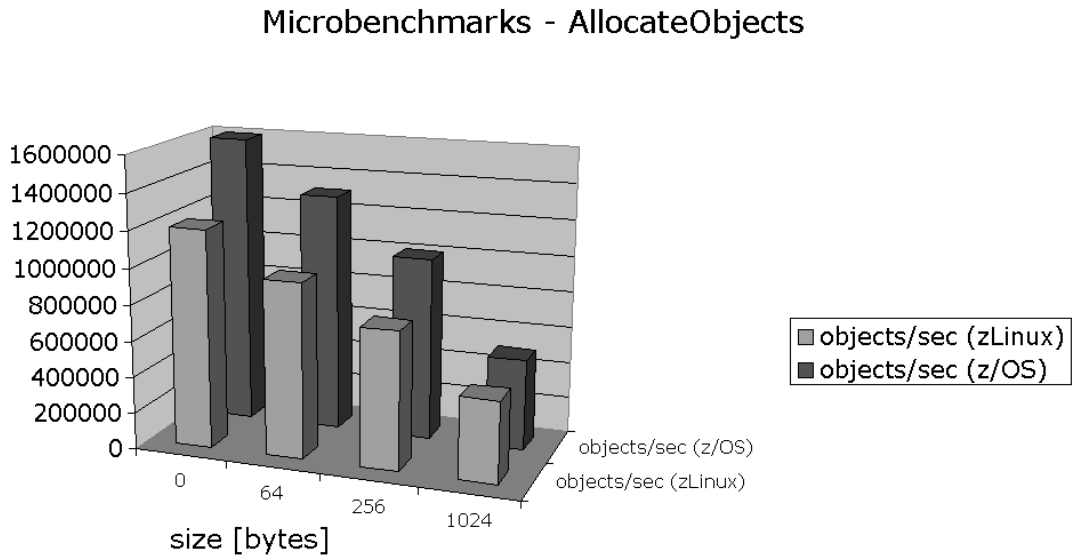

Abbildung B.2: `AllocateObjects` Vergleichsdiagramm

Ähnlich wie beim letzten Microbenchmark ist auch in Abbildung B.2 und Tabelle B.2 auf Seite 109 ein Leistungsunterschied zwischen den beiden Java Virtual Machines zu erkennen. Das Verhältnis der gemessenen Objekte pro Sekunde (in der Abbildung: *objects/sec*) beträgt im arithmetischen Mittel 1,28 : 1 (z/OS : zLinux).

Größe des Objekts in Byte	z/OS	zLinux	Verhältnis
0	1596487,73	1202886,93	1,33 : 1
64	1313074,47	963391,14	1,36 : 1
256	1007625,27	759878,42	1,33 : 1
1024	501756,15	448833,03	1,12 : 1

Tabelle B.2: `AllocateObjects` Ergebnistabelle

B.3 Threads und Locking

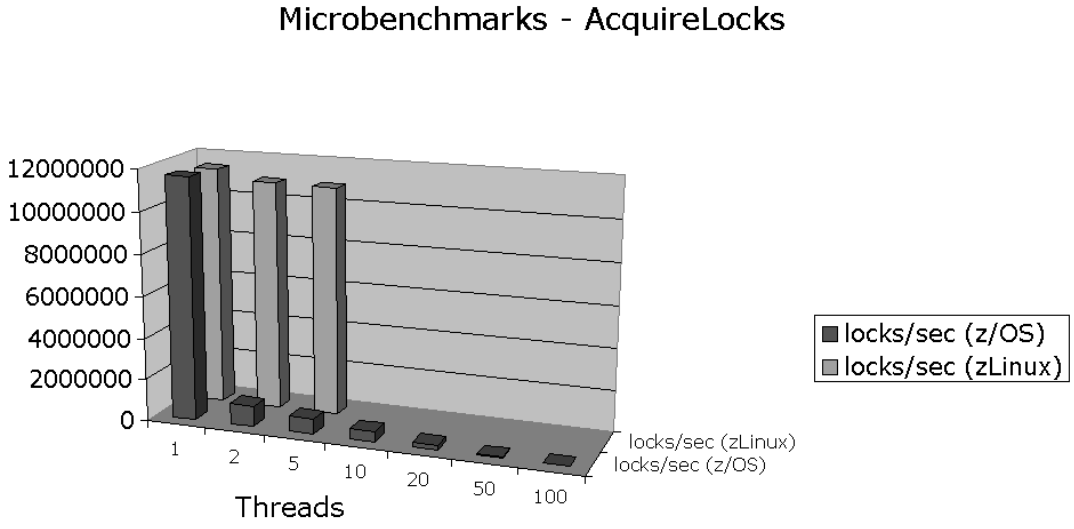

Abbildung B.3: `AcquireLocks` Vergleichsdiagramm

Auffällig ist in Abbildung B.3 der große Abfall der Performance unter z/OS beim Übergang von einem auf zwei konkurrierende Threads. Anders als bei den beiden letzten Microbenchmarks kann hier kein arithmetisches Mittel für die gemessenen Locks pro Sekunde (in der Abbildung: *locks/sec*) angegeben werden, da die Meßreihe für zLinux nach fünf Benutzern abbricht. Eine Diskussion dieser Ergebnisse befindet sich in Abschnitt 8.2.3 auf Seite 85.

Anzahl der Threads	z/OS	zLinux	Verhältnis
1	11603171,53	11446615,35	1,01 : 1
2	1008471,16	10974456,01	0,09 : 1
5	728013,98	10928961,75	0,07 : 1
10	505254,65	-	-
20	220848,06	-	-
50	35536,60	-	-
100	5707,60	-	-

Tabelle B.3: `AcquireLocks` Ergebnistabelle

Anhang C

Basic Online-Banking System: Ergebnisse der Benchmarks

Der vorliegende Anhang enthält die quantitativen Ergebnisse der Benchmarks mit dem Basic Online-Banking System. Die Diskussion dieser Ergebnisse befindet sich in Abschnitt 8.3.

C.1 PRJVM im resettable Modus

Um dem Sachverhalt Rechnung zu tragen, daß bei den Benchmarks mit dem Online-Banking System beide Transaktionstypen (Full TPC-A und Reduced TPC-A) verwendet wurden, ist dieser Abschnitt (und die folgenden) in zwei Unterabschnitte aufgeteilt.

C.1.1 Full TPC-A: Ergebnisse

Die Tabellen und Diagramme für diese Benchmark-Konfiguration sind:

- Für eine CPU: Der Transaktionsdurchsatz (siehe Abschnitt 8.3.5).
- Für eine CPU: Die CPU-Auslastung (Abbildung C.1 auf Seite 112, sowie Tabelle C.1 auf Seite 112).
- Für eine CPU: Die Antwortzeit (Abbildung C.2 auf Seite 113, sowie Tabelle C.2 auf Seite 113).
- Für zwei CPUs: Der Transaktionsdurchsatz (Abbildung C.3 auf Seite 114, sowie Tabelle C.3 auf Seite 114).
- Für zwei CPUs: Die CPU-Auslastung (Abbildung C.4 auf Seite 115, sowie Tabelle C.4 auf Seite 115).
- Für zwei CPUs: Die Antwortzeit (Abbildung C.5 auf Seite 116, sowie Tabelle C.5 auf Seite 116).

CPU-Auslastung

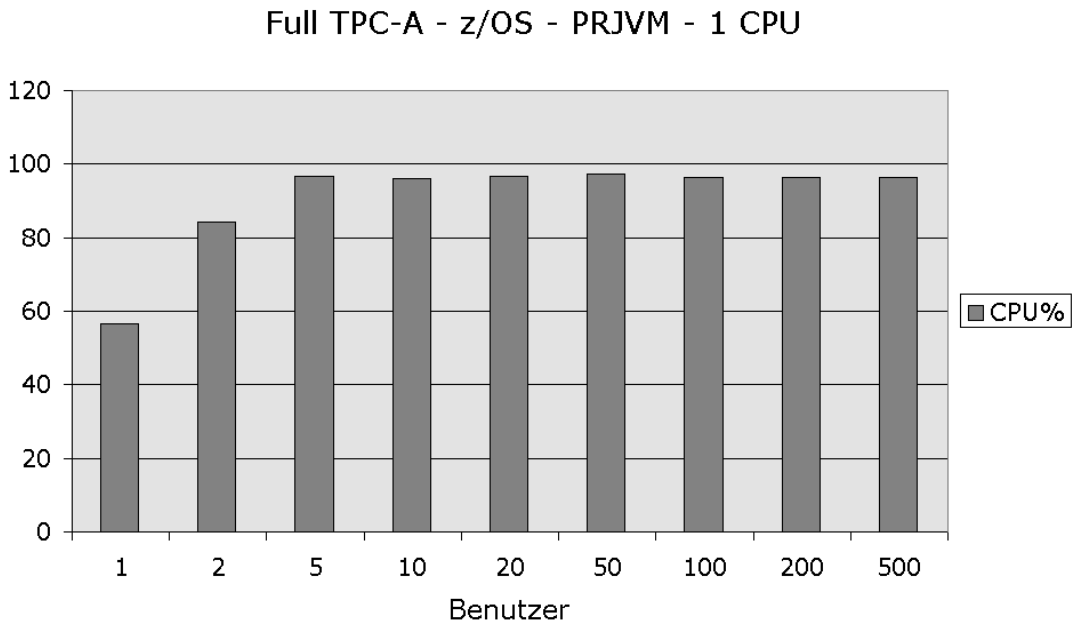

Abbildung C.1: Full TPC-A: CPU-Auslastung

Benutzer	[CPU%]
1	56,47%
2	84,28%
5	96,74%
10	95,99%
20	96,61%
50	97,44%
100	96,24%
200	96,34%
500	96,31%

Tabelle C.1: Full TPC-A: CPU-Auslastung

Deutlich ist die in Abschnitt 8.3.5 erwähnte asymptotische Annäherung an den Maximalwert der CPU-Auslastung zu erkennen. Dieser ist bei fünf Benutzern erreicht.

Antwortzeit

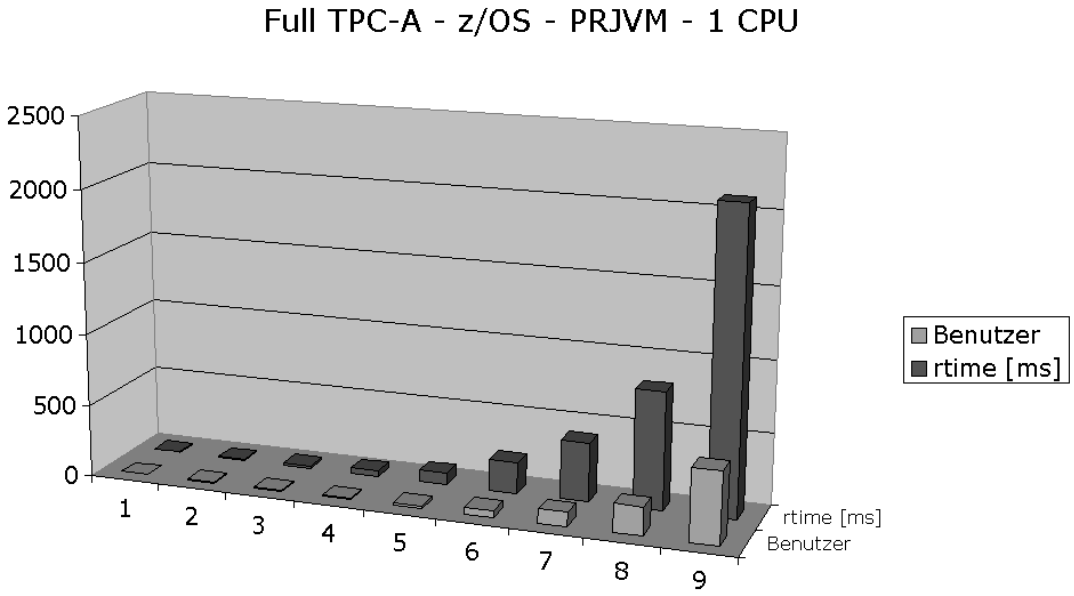

Abbildung C.2: Full TPC-A: Antwortzeit

Meßpunkt	Benutzer	rtime [ms]	rtime [ms] / Benutzer
1	1	6,71	6,71
2	2	8,94	4,47
3	5	19,15	3,83
4	10	39,73	3,97
5	20	81,76	4,09
6	50	218,15	4,36
7	100	412,54	4,13
8	200	819,79	4,10
9	500	2105,38	4,21

Tabelle C.2: Full TPC-A: Antwortzeit versus Anzahl der Benutzer

Abbildung C.2 erweckt den Eindruck eines exponentiellen Wachstums der Antwortzeit. Dieser Eindruck entsteht dadurch, daß die Anzahl der Benutzer bei den Messungen (in der Abbildung: Punkte eins bis neun) exponentiell erhöht wird. Um jedoch das *lineare* Wachstum der Antwortzeit hervorzuheben, wurde in Tabelle C.2 der Quotient (rtime [ms] / Benutzer) aus Antwortzeit (in der Tabelle: *rtime [ms]*) und Anzahl der Benutzer (in der Tabelle: *Benutzer*) aufgetragen.

Transaktionsdurchsatz

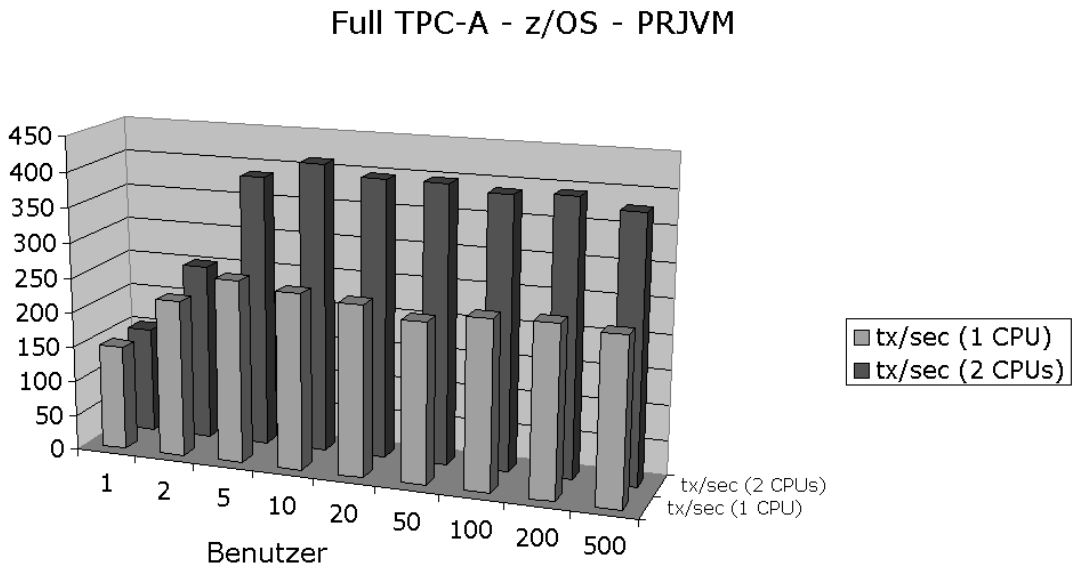

Abbildung C.3: Full TPC-A: [tx/sec] bei einer und zwei CPUs

Benutzer	[tx/sec] 1 CPU	[tx/sec] 2 CPUs	Verhältnis
1	149,08	151,68	1 : 1,02
2	223,65	252,36	1 : 1,13
5	261,04	388,10	1 : 1,49
10	251,68	412,76	1 : 1,64
20	244,61	397,73	1 : 1,63
50	229,20	398,05	1 : 1,74
100	242,40	389,54	1 : 1,61
200	243,96	393,53	1 : 1,61
500	237,49	379,02	1 : 1,60

Tabelle C.3: Full TPC-A: [tx/sec] bei einer und zwei CPUs

Wie in Abschnitt 8.3.5 beschrieben skaliert das Basic Online-Banking System in dieser Konfiguration nicht linear mit der Anzahl der CPUs. Das Verhältnis des Transaktionsdurchsatzes beträgt im arithmetischen Mittel 1 : 1,49 (1 CPU : 2 CPUs).

CPU-Auslastung

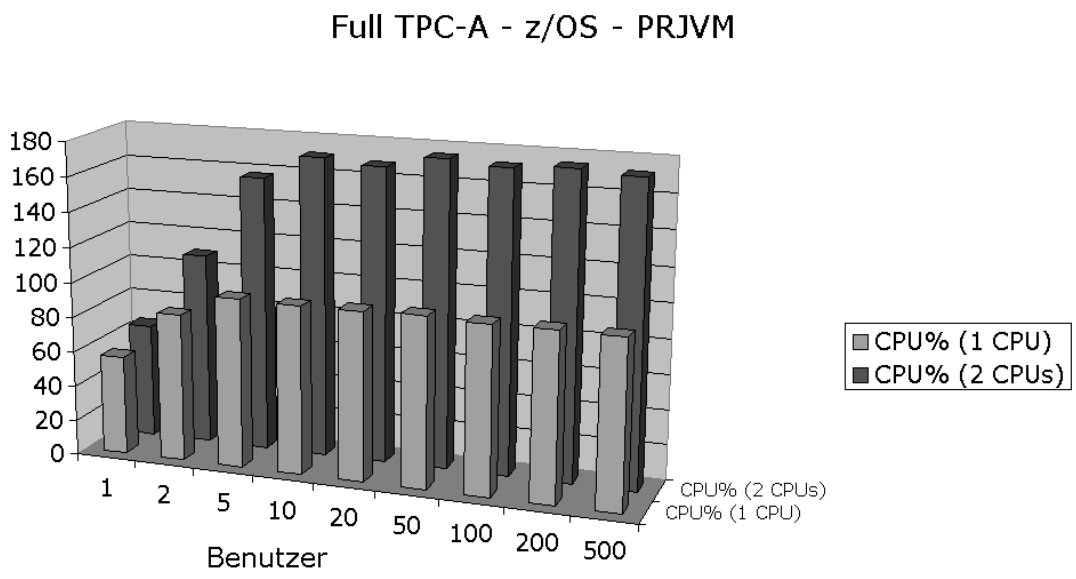

Abbildung C.4: Full TPC-A: [CPU%] bei einer und zwei CPUs

Benutzer	[CPU%] 1 CPU	[CPU%] 2 CPUs	Verhältnis
1	56,47	65,37	1 : 1,16
2	84,28	110,00	1 : 1,31
5	96,74	157,40	1 : 1,63
10	95,99	171,00	1 : 1,78
20	96,61	168,40	1 : 1,74
50	97,44	175,00	1 : 1,80
100	96,24	172,40	1 : 1,79
200	96,34	174,60	1 : 1,81
500	96,31	172,30	1 : 1,79

Tabelle C.4: Full TPC-A: [CPU%] bei einer und zwei CPUs

Die CPU-Auslastung erreicht in dieser Konfiguration nicht das theoretische Maximum von 200 Prozent. Die CPU-Auslastung beträgt im arithmetischen Mittel 151,83 Prozent.

Antwortzeit

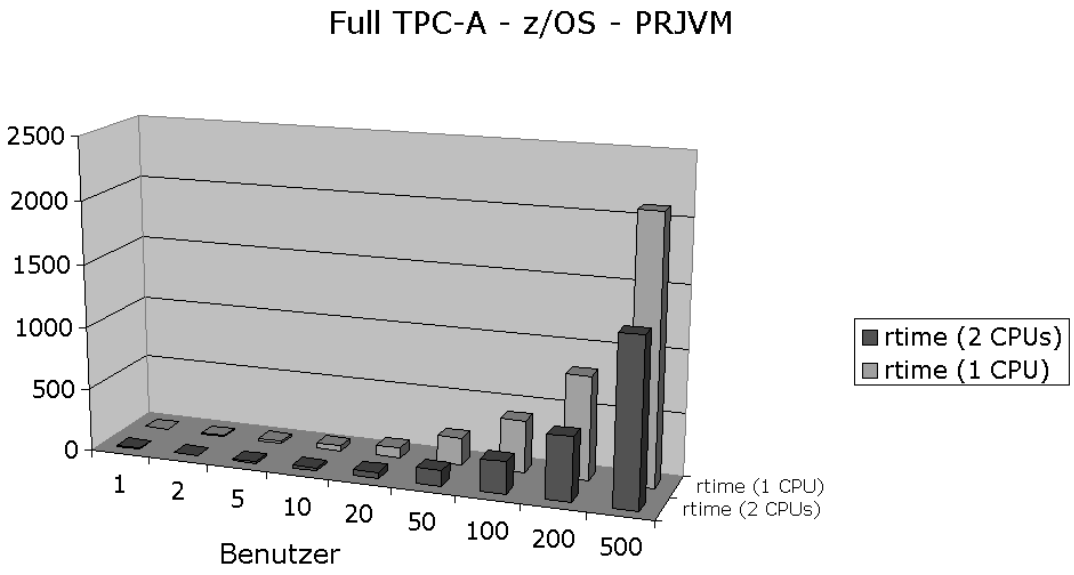

Abbildung C.5: Full TPC-A: rtime [ms] bei einer und zwei CPUs

Benutzer	rtime [ms] 2 CPUs	rtime [ms] 1 CPU	Verhältnis
1	6,59	6,71	1 : 1,02
2	7,93	8,94	1 : 1,13
5	12,88	19,15	1 : 1,49
10	24,23	39,73	1 : 1,64
20	50,29	81,76	1 : 1,63
50	125,61	218,15	1 : 1,74
100	256,71	412,54	1 : 1,61
200	508,22	819,79	1 : 1,61
500	1319,20	2105,38	1 : 1,60

Tabelle C.5: Full TPC-A: rtime [ms] bei einer und zwei CPUs

Das Verhältnis der Antwortzeit beträgt im arithmetischen Mittel 1 : 1,49 (2 CPUs :1 CPU). Für eine Erklärung des exponentiellen Anstiegs der Antwortzeit sei auf die Bemerkung zu Tabelle C.2 verwiesen.

C.1.2 Reduced TPC-A: Ergebnisse

Der Reduced TPC-A Transaktionstyp wurde eingeführt, um den Datenbank-Anteil an einer Transaktion erheblich zu verringern. Wie in Abbildung C.6 ersichtlich ist, wurde dies erreicht. Die von DB2 verbrauchte CPU-Zeit läßt sich lediglich an einer breiten schwarzen Linie bei der Messung mit zehn Benutzern erkennen.

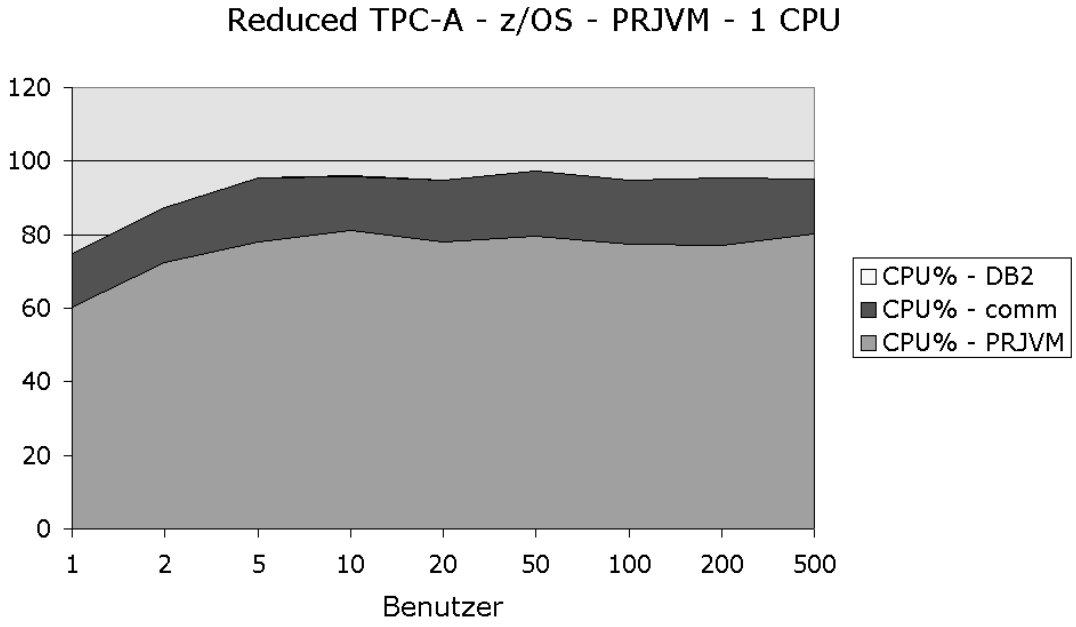

Abbildung C.6: Reduced TPC-A: Verteilung der CPU-Zeit

Aus Platzgründen enthalten die Tabellen und Diagramme für diese Benchmark-Konfiguration jeweils die Werte für eine und zwei CPUs. Diese sind im einzelnen:

- Der Transaktionsdurchsatz (Abbildung C.7 auf Seite 118, sowie Tabelle C.6 auf Seite 118).
- Die CPU-Auslastung (Abbildung C.8 auf Seite 119, sowie Tabelle C.7 auf Seite 119).
- Die Antwortzeit (Abbildung C.9 auf Seite 120, sowie Tabelle C.8 auf Seite 120).

Transaktionsdurchsatz

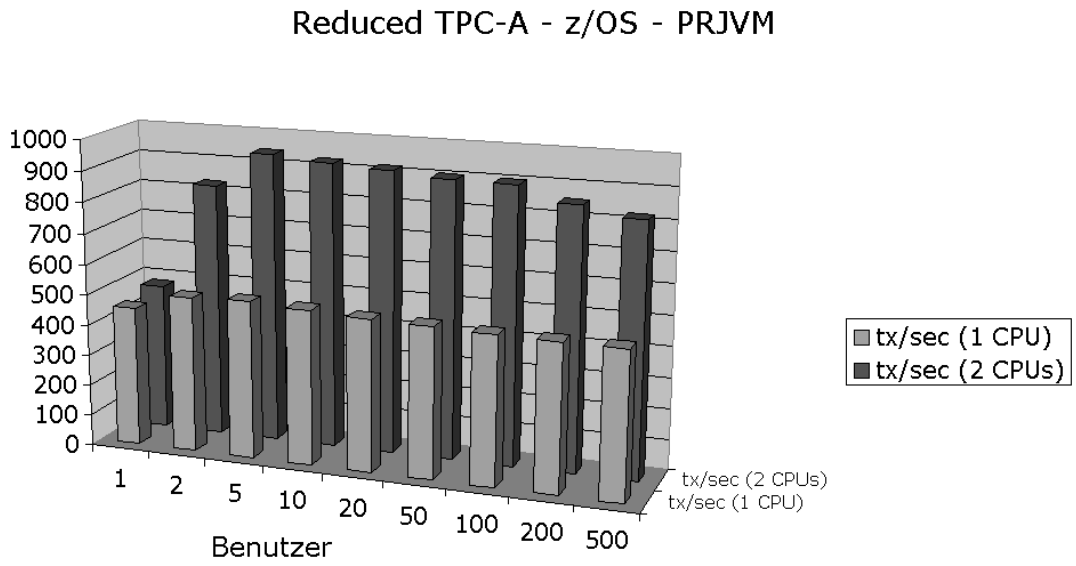

Abbildung C.7: Reduced TPC-A: [tx/sec] bei einer und zwei CPUs

Benutzer	[tx/sec] 1 CPU	[tx/sec] 2 CPUs	Verhältnis
1	453,75	478,73	1 : 1,06
2	504,80	826,60	1 : 1,64
5	511,77	942,20	1 : 1,84
10	502,57	926,89	1 : 1,84
20	493,81	916,63	1 : 1,86
50	488,77	902,43	1 : 1,85
100	483,56	900,63	1 : 1,86
200	480,77	852,69	1 : 1,77
500	479,21	822,68	1 : 1,72

Tabelle C.6: Reduced TPC-A: [tx/sec] bei einer und zwei CPUs

Wie aus der Tabelle ersichtlich, skaliert das Basic Online-Banking System bei dieser Benchmark-Konfiguration nahezu linear mit der Anzahl der CPUs.

CPU-Auslastung

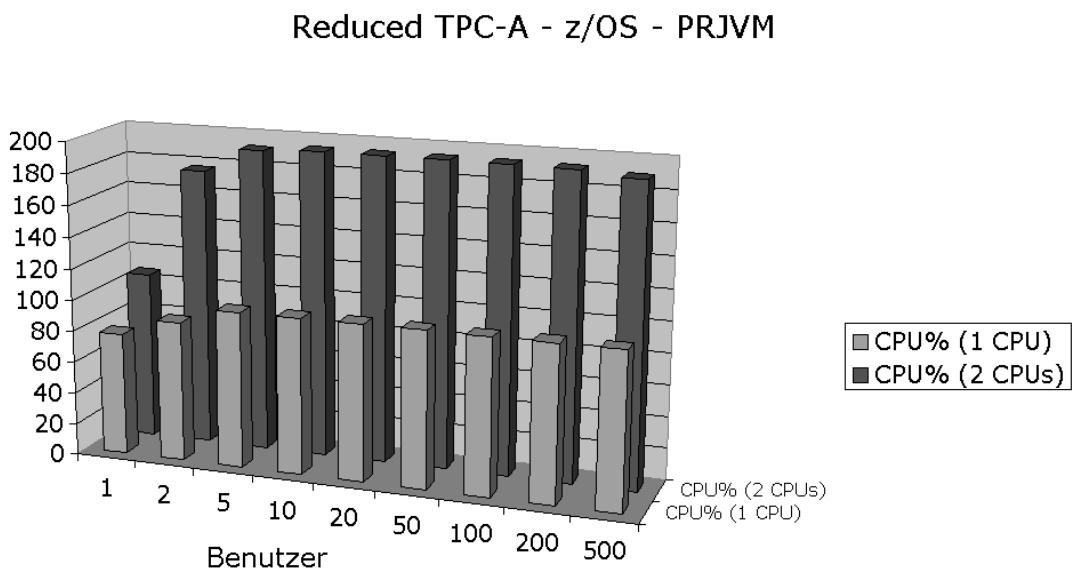

Abbildung C.8: Reduced TPC-A: [CPU%] bei einer und zwei CPUs

Benutzer	[CPU%] 1 CPU	[CPU%] 2 CPUs
1	77,70	106,70
2	88,94	176,60
5	99,09	191,80
10	99,27	193,60
20	99,33	193,70
50	99,30	193,90
100	99,35	193,60
200	99,34	193,10
500	99,34	190,10

Tabelle C.7: Reduced TPC-A: [CPU%] bei einer und zwei CPUs

Deutlich ist in der Abbildung und in der Tabelle zu erkennen, daß das Basic Online-Banking System in dieser Konfiguration sowohl mit einer als auch mit zwei CPUs nahezu die maximale CPU-Auslastung erreicht.

Antwortzeit

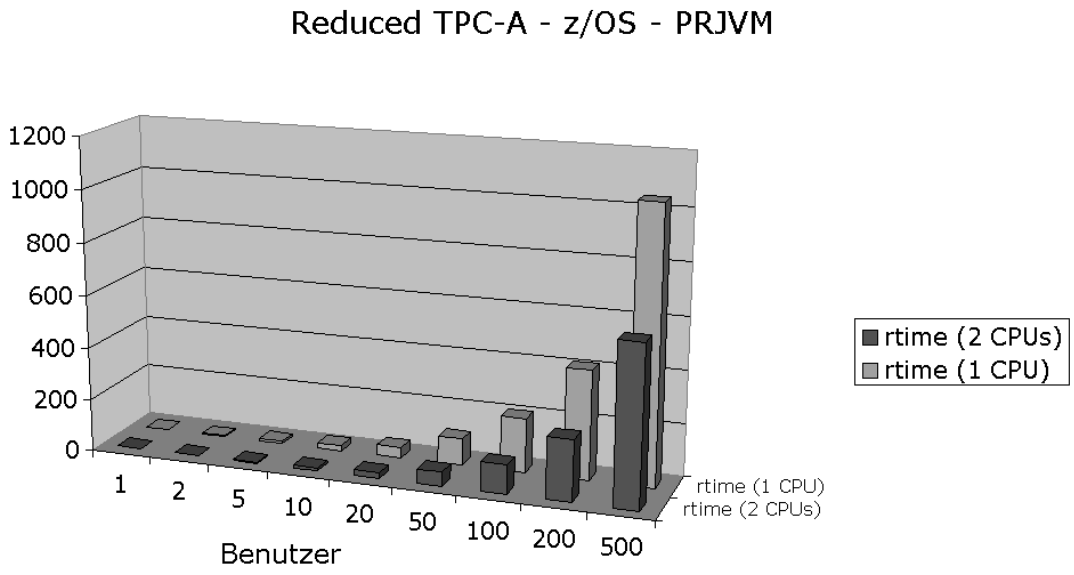

Abbildung C.9: Reduced TPC-A: rtime [ms] bei einer und zwei CPUs

Benutzer	rtime [ms] 2 CPUs	rtime [ms] 1 CPU	Verhältnis
1	2,09	2,20	1 : 1,05
2	2,42	3,96	1 : 1,64
5	5,31	9,77	1 : 1,84
10	10,79	19,90	1 : 1,84
20	21,82	40,50	1 : 1,86
50	55,41	102,30	1 : 1,85
100	111,03	206,80	1 : 1,86
200	234,55	416,00	1 : 1,77
500	607,77	1043,38	1 : 1,72

Tabelle C.8: Reduced TPC-A: rtime [ms] bei einer und zwei CPUs

Für eine Erklärung des exponentiellen Anstiegs der Antwortzeit sei auf die Bemerkung zu Tabelle C.2 verwiesen.

C.2 Pure Java unter z/OS

Auch in der vorliegenden Konfiguration wurden bei den Benchmarks mit dem Online-Banking System beide Transaktionstypen (Full TPC-A und Reduced TPC-A) verwendet. Da diese Messungen die Grundlage für das Ergebnis **D** bildeten (siehe Abschnitt 8.3.6 auf Seite 97), werden im folgenden ausschließlich Abbildungen und Diagramme vorgestellt, welche die Benchmark-Konfigurationen “PRJVM im resettable Modus” und “Pure Java unter z/OS” miteinander vergleichen. Messungen mit zwei CPUs wurden bei der vorliegenden Benchmark-Konfiguration nicht durchgeführt (siehe Abschnitt 8.3.6).

C.2.1 Full TPC-A: Ergebnisse

Die Tabellen und Diagramme für diese Benchmark-Konfiguration sind im einzelnen:

- Der Transaktionsdurchsatz (Abbildung C.10 auf Seite 122, sowie Tabelle C.9 auf Seite 122).
- Die CPU-Auslastung (Abbildung C.11 auf Seite 123, sowie Tabelle C.10 auf Seite 123).
- Die Antwortzeit (Abbildung C.12 auf Seite 124, sowie Tabelle C.11 auf Seite 124).

Transaktionsdurchsatz

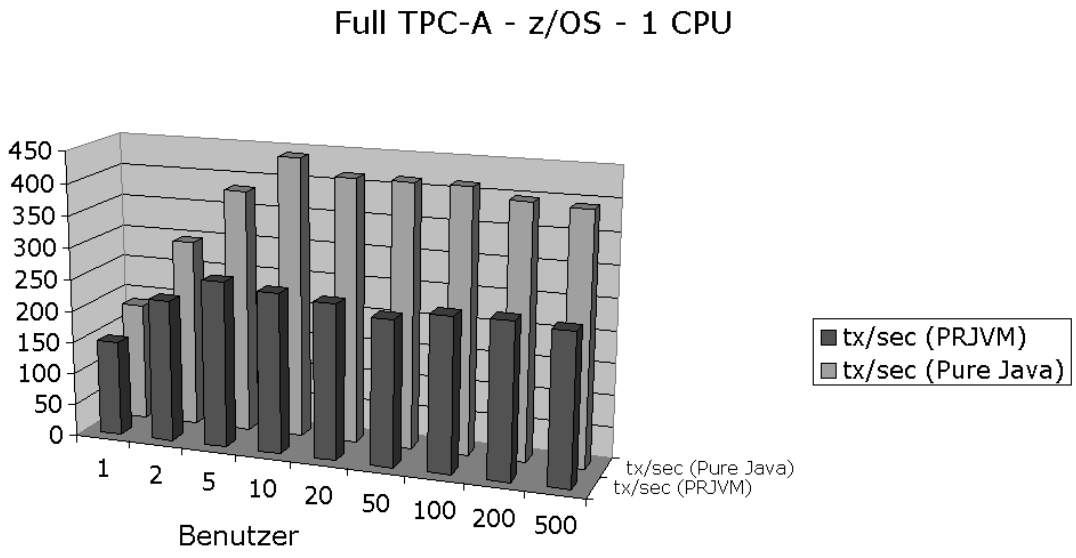

Abbildung C.10: Full TPC-A: [tx/sec] bei PRJVM und Pure Java

Benutzer	[tx/sec] Pure Java	[tx/sec] PRJVM	Verhältnis
1	185,33	149,08	1,24 : 1
2	295,86	223,65	1,32 : 1
5	382,94	261,04	1,47 : 1
10	440,38	251,68	1,75 : 1
20	415,30	244,61	1,70 : 1
50	415,23	229,20	1,81 : 1
100	414,66	242,40	1,71 : 1
200	398,50	243,96	1,63 : 1
500	393,63	237,49	1,66 : 1

Tabelle C.9: Full TPC-A: [tx/sec] bei PRJVM und Pure Java

Deutlich ist der höhere Transaktionsdurchsatz für die Benchmark-Konfiguration “Pure Java unter z/OS” zu erkennen. Das Verhältnis des Transaktionsdurchsatzes beträgt im arithmetischen Mittel 1,59 : 1 (Pure Java : PRJVM).

CPU-Auslastung

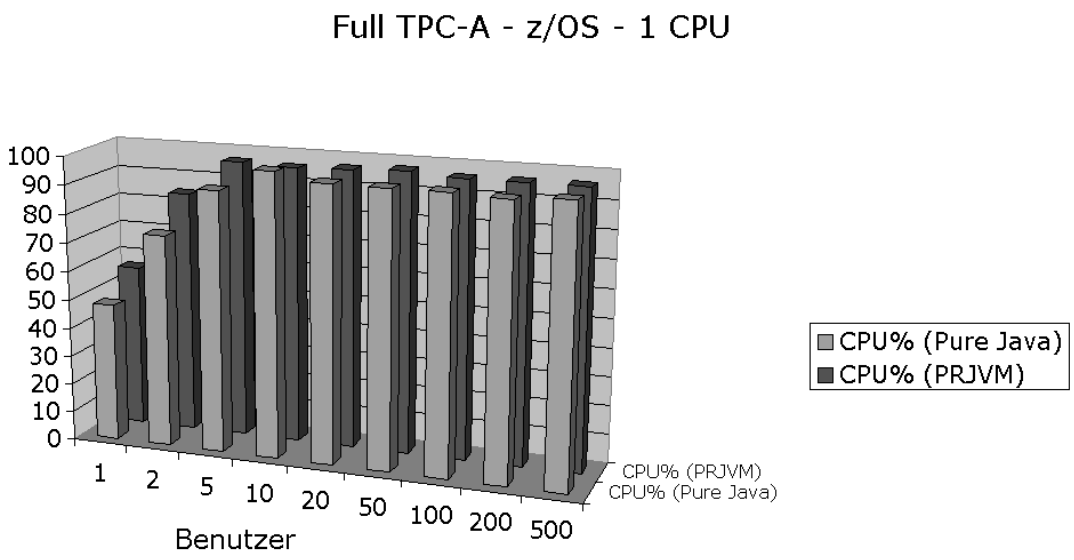

Abbildung C.11: Full TPC-A: [CPU%] bei PRJVM und Pure Java

Benutzer	[CPU%] Pure Java	[CPU%] PRJVM
1	48,02	56,47
2	73,91	84,28
5	90,62	96,74
10	98,37	95,99
20	95,52	96,61
50	95,42	97,44
100	95,37	96,24
200	94,60	96,34
500	95,81	96,31

Tabelle C.10: Full TPC-A: [CPU%] bei PRJVM und Pure Java

Sowohl aus der Abbildung als auch aus der Tabelle ist zu erkennen, daß das Basic Online-Banking System in beiden Benchmark-Konfigurationen nahezu die maximale CPU-Auslastung erreicht.

Antwortzeit

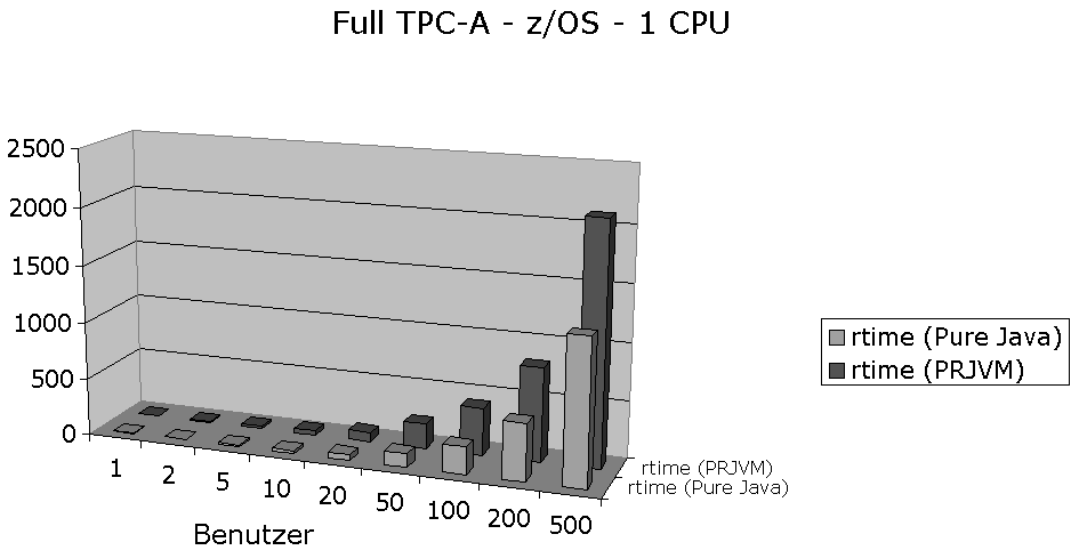

Abbildung C.12: Full TPC-A: rtime [ms] bei PRJVM und Pure Java

Benutzer	rtime [ms] Pure Java	rtime [ms] PRJVM	Verhältnis
1	5,40	6,71	1 : 1,24
2	6,76	8,94	1 : 1,32
5	13,06	19,15	1 : 1,47
10	22,71	39,73	1 : 1,75
20	48,16	81,76	1 : 1,70
50	120,42	218,15	1 : 1,81
100	241,16	412,54	1 : 1,71
200	501,88	819,79	1 : 1,63
500	1270,21	2105,38	1 : 1,66

Tabelle C.11: Full TPC-A: rtime [ms] bei PRJVM und Pure Java

Das Verhältnis der Antwortzeit beträgt im arithmetischen Mittel 1 : 1,59 (Pure Java : PRJVM). Für eine Erklärung des exponentiellen Anstiegs der Antwortzeit sei auf die Bemerkung zu Tabelle C.2 verwiesen.

C.2.2 Reduced TPC-A: Ergebnisse

Eine Erklärung zu diesem Transaktionstyp befindet sich in Abschnitt 8.3.3 auf Seite 91. Die Tabellen und Diagramme für diese Benchmark-Konfiguration sind im einzelnen:

- Der Transaktionsdurchsatz (Abbildung C.13 auf Seite 126, sowie Tabelle C.12 auf Seite 126).
- Die CPU-Auslastung (Abbildung C.14 auf Seite 127, sowie Tabelle C.13 auf Seite 127).
- Die Antwortzeit (Abbildung C.15 auf Seite 128, sowie Tabelle C.14 auf Seite 128).

Transaktionsdurchsatz

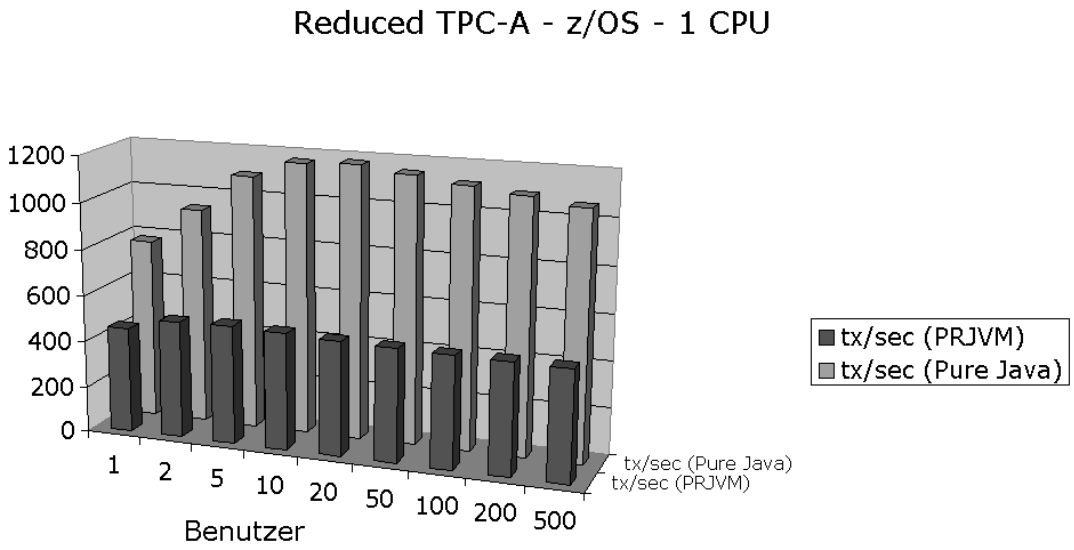

Abbildung C.13: Reduced TPC-A: [tx/sec] bei PRJVM und Pure Java

Benutzer	[tx/sec] Pure Java	[tx/sec] PRJVM	Verhältnis
1	780,15	453,75	1,72 : 1
2	936,97	504,80	1,86 : 1
5	1096,90	511,77	2,14 : 1
10	1168,29	502,57	2,32 : 1
20	1177,72	493,81	2,38 : 1
50	1150,07	488,77	2,35 : 1
100	1121,32	483,56	2,32 : 1
200	1095,68	480,77	2,28 : 1
500	1064,25	479,21	2,22 : 1

Tabelle C.12: Reduced TPC-A: [tx/sec] bei PRJVM und Pure Java

Deutlich ist der höhere Transaktionsdurchsatz für die Benchmark-Konfiguration “Pure Java unter z/OS” zu erkennen. Das Verhältnis des Transaktionsdurchsatzes beträgt im arithmetischen Mittel 2,18 : 1 (Pure Java : PRJVM).

CPU-Auslastung

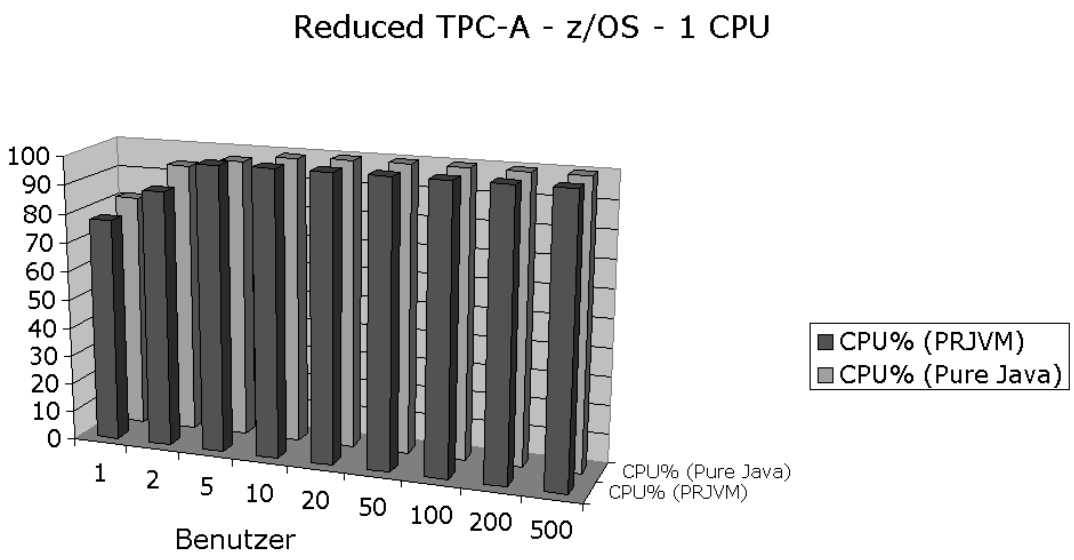

Abbildung C.14: Reduced TPC-A: [CPU%] bei PRJVM und Pure Java

Benutzer	[CPU%] Pure Java	[CPU%] PRJVM
1	81,60	77,70
2	94,17	88,94
5	96,97	99,09
10	99,15	99,27
20	99,78	99,33
50	99,88	99,30
100	99,87	99,35
200	99,80	99,34
500	99,87	99,34

Tabelle C.13: Reduced TPC-A: [CPU%] bei PRJVM und Pure Java

Sowohl aus der Abbildung als auch aus der Tabelle ist zu erkennen, daß das Basic Online-Banking System in beiden Benchmark-Konfigurationen nahezu die maximale CPU-Auslastung erreicht.

Antwortzeit

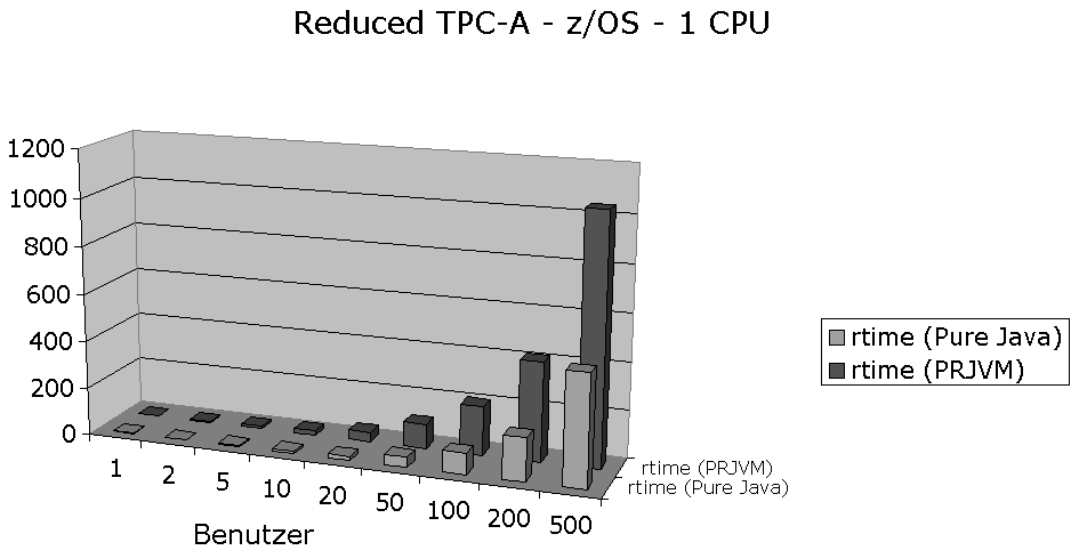

Abbildung C.15: Reduced TPC-A: rtime [ms] bei PRJVM und Pure Java

Benutzer	rtime [ms] Pure Java	rtime [ms] PRJVM	Verhältnis
1	1,28	2,20	1 : 1,72
2	2,13	3,96	1 : 1,86
5	4,56	9,77	1 : 2,14
10	8,56	19,90	1 : 2,32
20	16,98	40,50	1 : 2,39
50	43,48	102,30	1 : 2,35
100	89,18	206,80	1 : 2,32
200	182,53	416,00	1 : 2,28
500	469,82	1043,38	1 : 2,22

Tabelle C.14: Reduced TPC-A: rtime [ms] bei PRJVM und Pure Java

Das Verhältnis der Antwortzeit beträgt im arithmetischen Mittel 1 : 2,18 (Pure Java : PRJVM). Für eine Erklärung des exponentiellen Anstiegs der Antwortzeit sei auf die Bemerkung zu Tabelle C.2 verwiesen.

C.3 Pure Java unter zLinux

Die Benchmark-Konfiguration im vorliegenden Abschnitt bildete die Grundlage für Ergebnis **E** (siehe Abschnitt 8.3.7 auf Seite 100). Daher sind in allen Diagrammen für diese Benchmark-Konfiguration auch die Ergebnisse für die Konfiguration “Pure Java unter z/OS” abgebildet. Zusätzlich wurden die Ergebnisse für “PRJVM im resettable Modus” hinzugefügt. Mit Hilfe des `top` Utility-Programms (siehe hierzu Abschnitt 8.3.2) wurde für den Full TPC-A Transaktionstyp unter zLinux folgende Verteilung der CPU-Zeit ermittelt (deutlich ist dabei der große Anteil des DB2 Datenbanksystems zu erkennen):

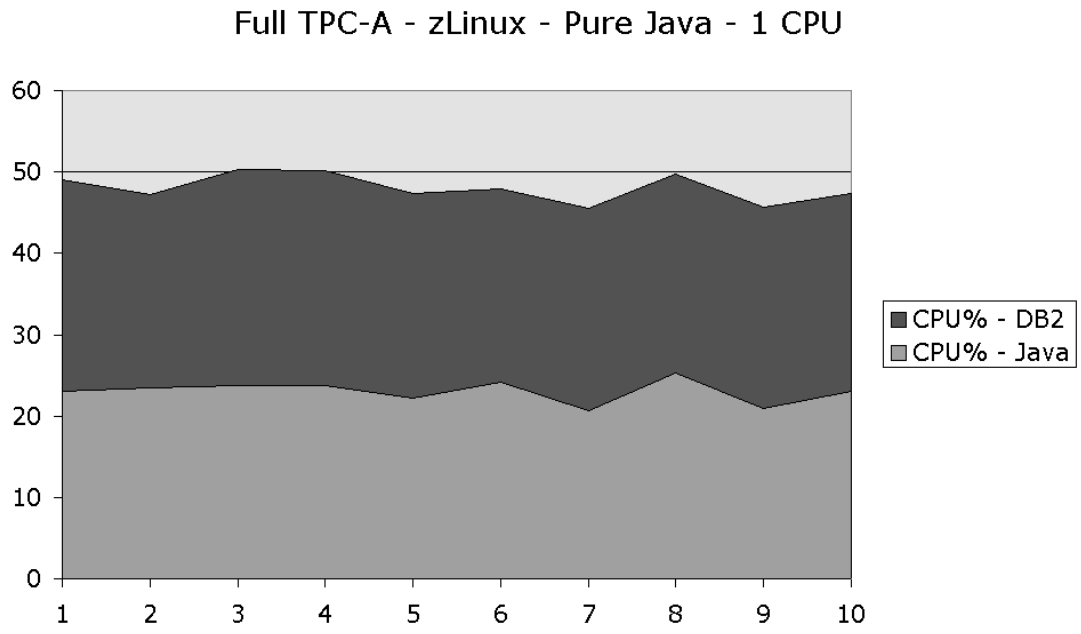

Abbildung C.16: Full TPC-A: Verteilung der CPU-Zeit für Pure Java unter zLinux

C.3.1 Full TPC-A: Ergebnisse

Die Diagramme und Tabellen für den Full TPC-A Transaktionstyp unter zLinux sind:

- Der Transaktionsdurchsatz (Abbildung C.17 auf Seite 130, sowie Tabelle C.15 auf Seite 130).
- Die CPU-Auslastung (Abbildung C.18 auf Seite 131, sowie Tabelle C.16 auf Seite 131).
- Die Antwortzeit (Abbildung C.19 auf Seite 132, sowie Tabelle C.17 auf Seite 132).

Transaktionsdurchsatz

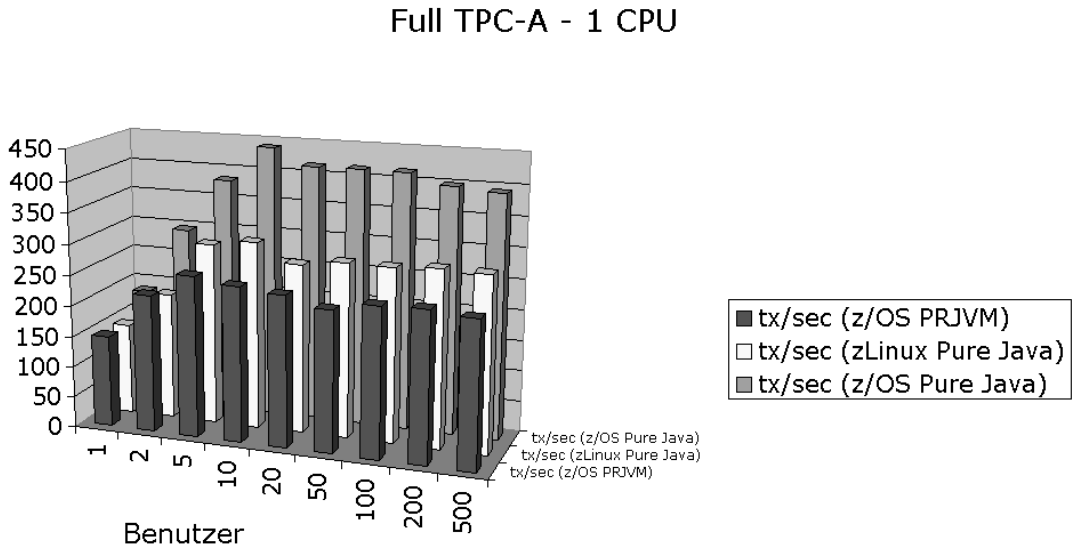

Abbildung C.17: Full TPC-A: [tx/sec] für alle Konfigurationen

Benutzer	[tx/sec] zLinux	[tx/sec] z/OS	Verhältnis
1	146,81	185,33	1 : 1,26
2	205,78	295,86	1 : 1,44
5	294,35	382,94	1 : 1,30
10	303,79	440,38	1 : 1,45
20	274,64	415,30	1 : 1,51
50	283,76	415,23	1 : 1,46
100	283,18	414,66	1 : 1,46
200	287,21	398,50	1 : 1,39
500	285,57	393,63	1 : 1,38

Tabelle C.15: Full TPC-A: [tx/sec] bei Pure Java

Das hier vorgestellte Verhältnis des Transaktionsdurchsatzes muß unter Berücksichtigung der Hinweise in Abschnitt 8.3.7 betrachtet werden (Bottleneck).

CPU-Auslastung

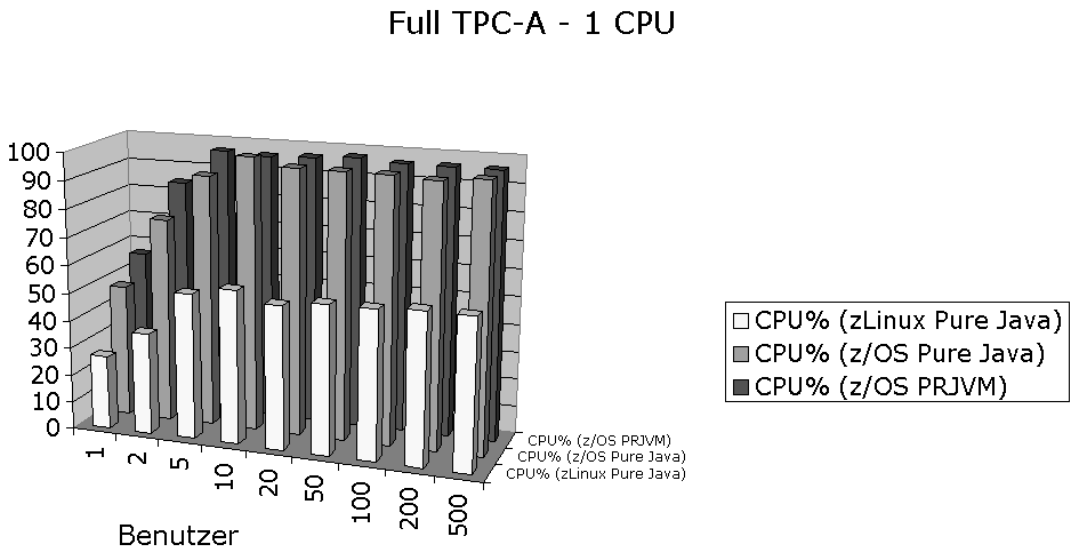

Abbildung C.18: Full TPC-A: [CPU%] für alle Konfigurationen

Benutzer	[CPU%] zLinux	[CPU%] z/OS
1	26,48	48,02
2	36,52	73,91
5	52,57	90,62
10	55,46	98,37
20	51,70	95,52
50	53,69	95,42
100	53,66	95,37
200	54,73	94,60
500	54,47	95,81

Tabelle C.16: Full TPC-A: [CPU%] bei Pure Java

Die in der Tabelle aufgeführten Werte für die CPU-Auslastung unter zLinux bilden die Grundlage für die in Abschnitt 8.3.7 vorgestellte Vermutung (Bottleneck im Basic Online-Banking System in der Benchmark-Konfiguration “Pure Java unter zLinux” mit dem Full TPC-A Transaktionstyp). Die CPU ist in dieser Benchmark-Konfiguration lediglich zur Hälfte (maximal 55,46 Prozent) ausgelastet.

Antwortzeit

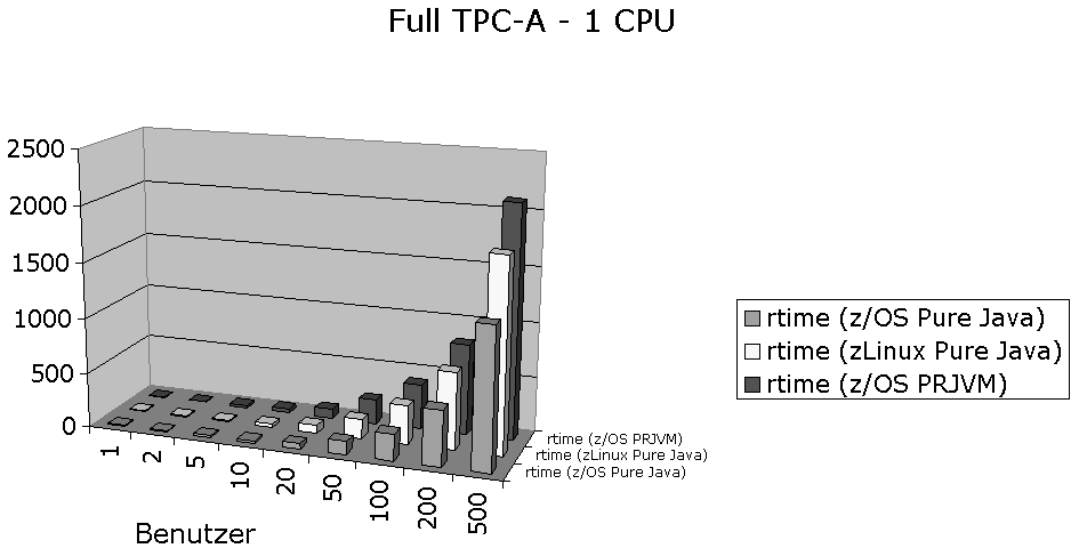

Abbildung C.19: Full TPC-A: rtime [ms] für alle Konfigurationen

Benutzer	rtime [ms] zLinux	rtime [ms] z/OS	Verhältnis
1	6,81	5,40	1,26 : 1
2	9,72	6,76	1,44 : 1
5	16,99	13,06	1,30 : 1
10	32,92	22,71	1,45 : 1
20	72,82	48,16	1,51 : 1
50	176,21	120,42	1,46 : 1
100	353,13	241,16	1,46 : 1
200	696,36	501,88	1,39 : 1
500	1750,88	1270,21	1,38 : 1

Tabelle C.17: Full TPC-A: rtime [ms] bei Pure Java

Das hier vorgestellte Verhältnis der Antwortzeit muß ebenfalls unter Berücksichtigung der Hinweise in Abschnitt 8.3.7 betrachtet werden (Bottleneck). Für eine Erklärung des exponentiellen Anstiegs der Antwortzeit sei auf die Bemerkung zu Tabelle C.2 verwiesen.

C.3.2 Reduced TPC-A: Ergebnisse

Für den Reduced TPC-A Transaktionstyp unter zLinux wurde ebenfalls eine Verteilung der CPU-Zeit ermittelt. Das Ergebnis dieser Untersuchung ist in Abbildung C.20 festgehalten. Besonders sei hierbei auf den DB2-Anteil der gesamten CPU-Zeit hingewiesen. Dieser ist im Vergleich zur entsprechenden Abbildung C.6 unter z/OS (Benchmark-Konfiguration “PRJVM im resettable Modus”) unverhältnismäßig hoch.

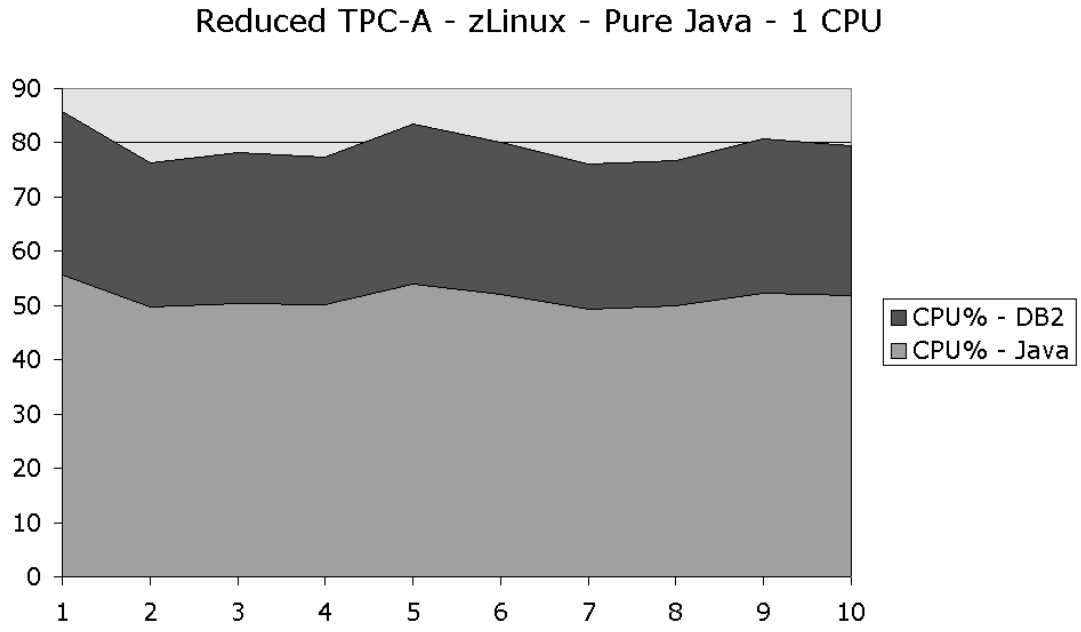

Abbildung C.20: Reduced TPC-A: Verteilung der CPU-Zeit für Pure Java unter zLinux

Die Diagramme und Tabellen für den Reduced TPC-A Transaktionstyp unter zLinux sind im einzelnen:

- Der Transaktionsdurchsatz (Abbildung C.21 auf Seite 134, sowie Tabelle C.18 auf Seite 134).
- Die CPU-Auslastung (Abbildung C.22 auf Seite 135, sowie Tabelle C.19 auf Seite 135).
- Die Antwortzeit (Abbildung C.23 auf Seite 136, sowie Tabelle C.20 auf Seite 136).

Transaktionsdurchsatz

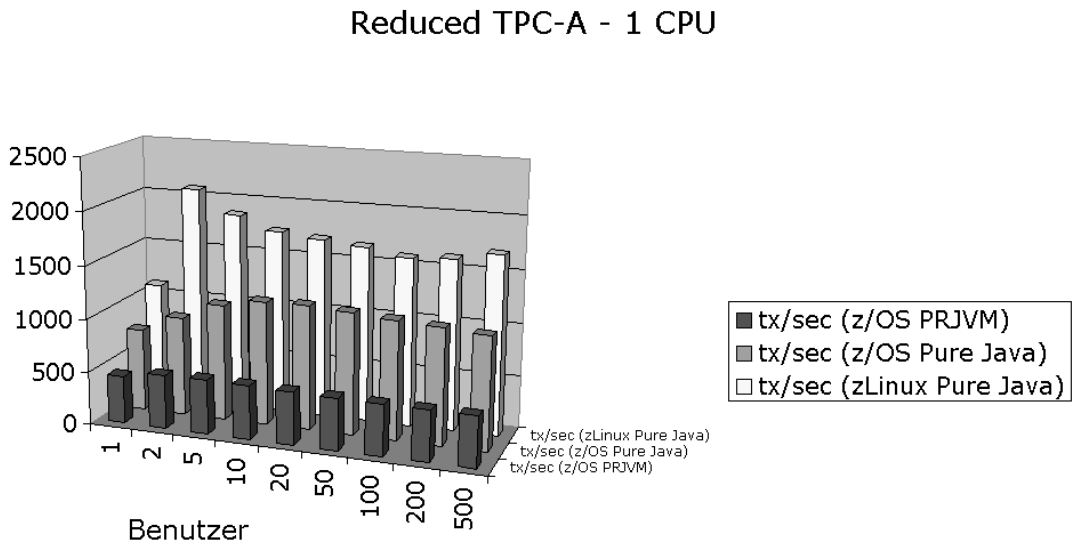

Abbildung C.21: Reduced TPC-A: [tx/sec] für alle Konfigurationen

Benutzer	[tx/sec] zLinux	[tx/sec] z/OS	Verhältnis
1	1112,29	780,15	1,43 : 1
2	2075,15	936,97	2,21 : 1
5	1858,06	1096,90	1,69 : 1
10	1733,13	1168,29	1,48 : 1
20	1692,10	1177,72	1,44 : 1
50	1654,34	1150,07	1,44 : 1
100	1586,92	1121,32	1,42 : 1
200	1613,40	1095,68	1,47 : 1
500	1685,45	1064,25	1,58 : 1

Tabelle C.18: Reduced TPC-A: [tx/sec] bei Pure Java

Deutlich ist der höhere Transaktionsdurchsatz für die Benchmark-Konfiguration “Pure Java unter zLinux” zu erkennen. Das Verhältnis des Transaktionsdurchsatzes beträgt im arithmetischen Mittel 1,57 : 1 (Pure Java unter zLinux : Pure Java unter z/OS).

CPU-Auslastung

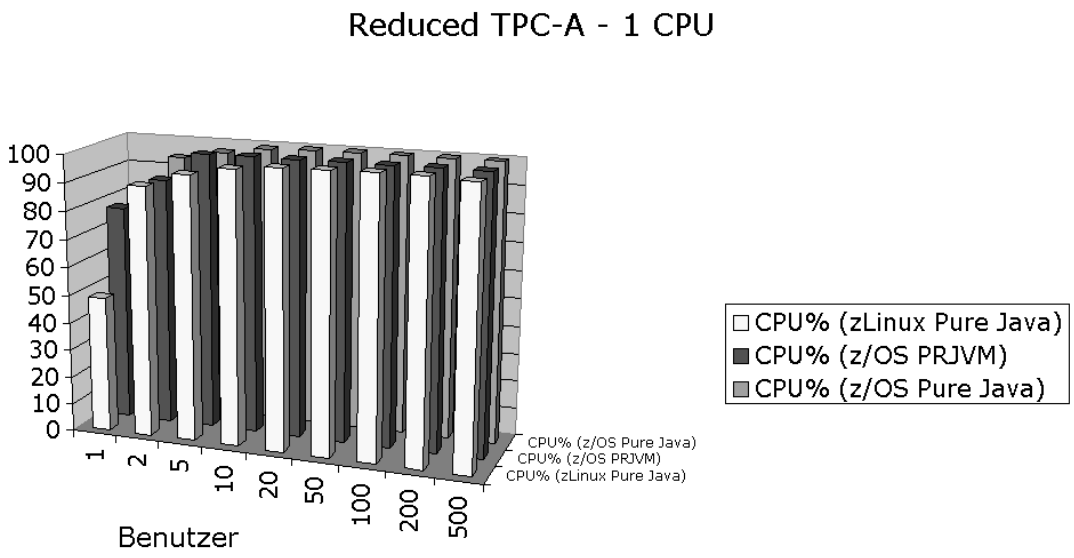

Abbildung C.22: Reduced TPC-A: [CPU%] für alle Konfigurationen

Benutzer	[CPU%] zLinux	[CPU%] z/OS
1	48,67	81,60
2	89,94	94,17
5	94,98	96,97
10	97,82	99,15
20	99,17	99,78
50	99,45	99,88
100	99,79	99,87
200	99,67	99,80
500	98,84	99,87

Tabelle C.19: Reduced TPC-A: [CPU%] bei Pure Java

Sowohl aus der Abbildung als auch aus der Tabelle ist zu erkennen, daß das Basic Online-Banking System in allen Benchmark-Konfigurationen nahezu die maximale CPU-Auslastung erreicht.

Antwortzeit

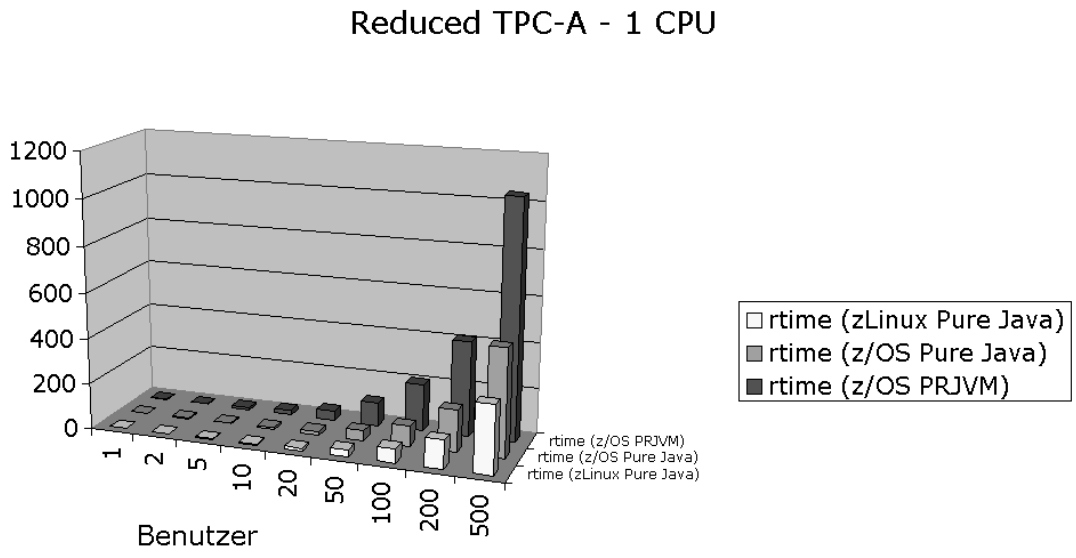

Abbildung C.23: Reduced TPC-A: rtime [ms] für alle Konfigurationen

Benutzer	rtime [ms] zLinux	rtime [ms] z/OS	Verhältnis
1	0,90	1,28	1 : 1,42
2	0,96	2,13	1 : 2,21
5	2,69	4,56	1 : 1,70
10	5,77	8,56	1 : 1,48
20	11,82	16,98	1 : 1,44
50	30,22	43,48	1 : 1,44
100	63,01	89,18	1 : 1,42
200	123,96	182,53	1 : 1,47
500	296,66	469,82	1 : 1,58

Tabelle C.20: Reduced TPC-A: rtime [ms] bei Pure Java

Das Verhältnis der Antwortzeit beträgt im arithmetischen Mittel 1 : 1,57 (Pure Java unter zLinux : Pure Java unter z/OS). Für eine Erklärung des exponentiellen Anstiegs der Antwortzeit sei auf die Bemerkung zu Tabelle C.2 verwiesen.

Abbildungsverzeichnis

Tabellenverzeichnis

Index

Literaturverzeichnis

[Litt03] Alan Little: *Meet the Experts: Alan Little on WebSphere Application Server for z/OS*. http://www7b.boulder.ibm.com/wsdd/library/techarticles/0303_little/little.html, 2003.

[CICS02] *Enterprise JavaBeans for z/OS and OS/390: CICS Transaction Server V2.2*. IBM Document No. SG24-6284-01, 2002.

[IMS02] *IMS Version 7: Java Update*. IBM Document No. SG24-6536-00, 2002.

[Herr02] Paul Herrmann, Udo Kebschull, Wilhelm G. Spruth: *Einführung in z/OS und OS/390*. Oldenbourg, 2002.

[Java00] *Java: A Strategy for Enterprise Success*. IBM Document No. G326-4000-00, 2000.

[Borm01] Sam Borman, Susan Paice, Matthew Webster, Martin Trotter, Rick McGuire, Alan Stevens, Beth Hutchison, Robert Berry: *A Serially Reusable Java Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing*. IBM Document No. TR 29.3406, 2001.

[PRJVM01] *New IBM Technology featuring Persistent Reusable Java Virtual Machines*. IBM Document No. SC34-6034-01, 2001.

[TPC94] Transaction Processing Performance Council (TPC): *TPC Benchmark A, Standard Specification*. Revision 2.0, 1994.

[J2EETut] *The J2EE Tutorial, A beginner's guide to developing enterprise applications on the Java 2 Platform, Enterprise Edition SDK version 1.3*, http://java.sun.com/j2ee/tutorial/

[Gray93] Jim Gray, Andreas Reuter: *Transaction Processing: Concepts and Techniques (Morgan Kaufmann Series in Data Management Systems)*. Morgan Kaufmann, 1st Edition, 1993.

[Dill00] D. Dillenberger, R. Bordawekar, C. W. Clark, D. Durand, D. Emmes, O. Gohda, S. Howard, M. F. Oliver, F. Samuel, R. W. St. John: *Building a Java virtual machine for server applications: The Jvm on OS/390*. IBM Systems Journal, Volume 39, Number 1, 2000.

[Harr02] H. Harrer, H. Pross, T.-M. Winkel, W. D. Becker, H. I. Stoller, M. Yamamoto, S. Abe, B. J. Chamberlin, G. A. Katopis: *First- and second-level packaging for the IBM eServer z900*. IBM Journal of Research and Development, Volume 46, Numbers 4/5, 2002.

[Amre00] Erich Amrehn et al.: *Linux for S/390*. IBM Document No. SG24-4987-00, 2000.

[ABCs00a] *ABCs of OS/390 System Programming, Volume 1*. IBM Document No. SG24-5597-00, 2000.

[ACSR02] *z/OS UNIX System Services Programming: Assembler Callable Services Reference*. IBM Document No. SA22-7803-03, 2002.

[PortGuide] *z/OS UNIX System Services Porting Guide*, http://www-1.ibm.com/servers/eserver/zseries/zos/unix/bpxa1por.html

[Coms02] Steven H. Comstock: *z/OS, Language Environment, and UNIX - How They Work Together*. The Trainer s Friend, Inc., http://www.trainersfriend.com/Papers/zos_unix.pdf, 2002.

[ABCs00b] *ABCs of OS/390 System Programming, Volume 4*. IBM Document No. SG24-5654-00, 2000.

[USSCR02] *z/OS UNIX System Services Command Reference*. IBM Document No. SA22-7802-04, 2002.

[NGPT] *Next Generation POSIX Threading*, http://www-124.ibm.com/developerworks/oss/pthreads/index.html

[LinuxThreads] *The LinuxThreads library*, http://pauillac.inria.fr/~xleroy/linuxthreads/index.html

[XPLink00] *XPLink: OS/390 Extra Performance Linkage*. IBM Document No. SG24-5991-00, 2000.

[TPC] *Transaction Processing Performance Council*, http://www.tpc.org

[Brun02] Paolo Bruni, Florence Dubois, Claudio Nacamatsu: *DB2 for z/OS and OS/390 Version 7 - Selected Performance Topics*. IBM Document No. SG24-6894-00, 2002.

[IEEE90] Institute of Electrical and Electronics Engineers, Inc.: *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standard 0610.12-1990, 1990.

[SPEC] *Standard Performance Evaluation Corporation*, http://www.spec.org

[Trip01] Dana Triplett: *Spotlight on Java performance*. http://www-106.ibm.com/developerworks/java/library/j-berry/index.html, 2001.

[jMocha] *The jMocha Microbenchmark Framework and Suite for Java*, http://www-124.ibm.com/developerworks/oss/jmocha/

[Lind99] Tim Lindholm, Frank Yellin: *The Java Virtual Machine Specification*. Addison-Wesley, 2nd Edition, 1999.

[LECus02] *z/OS Language Environment Customization*. IBM Document No. SA22-7564-03, 2002.

[Chow03] Kingsum Chow, Ricardo Morin, Kumar Shiv: *Enterprise Java Performance: Best Practices*. Intel Technology Journal, Volume 07, Issue 01, 2003.

[Vila02] Tamas Vilaghy, Marc Beyerle, Jürgen Lange, Axel Mester, Frank Pani: *e-business Cookbook for z/OS Volume III: Java Development*. IBM Document No. SG24-5980-01, 2002.

[Bull01] J. M. Bull, L. A. Smith, L. Pottage and R. Freeman: *Benchmarking Java against C and Fortran for Scientific Applications*. Edinburgh Parallel Computing Centre, The University of Edinburgh, 2001.

[IOCCC] *The International Obfuscated C Code Contest*, http://www.ioccc.org

[Shir03] Jack Shirazi: *Java Performance Tuning*. O'Reilly & Associates, 2nd Edition, 2003.

[JavaPerf] *Java Performance Tuning*, http://www.javaperformancetuning.com

[DB2UGR02] *DB2 Universal Database for OS/390 and z/OS: Utility Guide and Reference*. IBM Document No. SC26-9945-03, 2002.

[DB2Java02] *DB2 Universal Database for OS/390 and z/OS: Application Programming Guide and Reference for Java*. IBM Document No. SC26-9932-02, 2002.

[Pool02] Kishnakumar Pooloth: *High Performance Inserts on DB2 UDB EEE using Java*. http://www7b.software.ibm.com/dmdd/library/techarticle/0204pooloth/0204pooloth.html, 2002.

[DB2AG01] *IBM DB2 Universal Database: Administration Guide: Performance*. IBM Document No. SC09-2945-01, 2001.

[An01] Yongli An, Peter Shum: *DB2 Tuning Tips for OLTP Applications*. http://www7b.software.ibm.com/dmdd/library/techarticle/anshum/0107anshum.html, 2001.

[Whit02] Bill White, Rama Ayyar, Velibor Uskokovic: *zSeries HiperSockets*. IBM Document No. SG24-6816-00, 2002.

[IrfanView] *IrfanView*, http://www.irfanview.com

[RMFPMG02] *z/OS Resource Measurement Facility: Performance Management Guide*. IBM Document No. SC33-7992-02, 2002.

[RMFUG03] *z/OS Resource Measurement Facility: User's Guide*. IBM Document No. SC33-7990-04, 2003.

[Horo02] Joanne Horowitz, Robert Catterall: *What DB2 for OS/390 People Should Know About Java*. http://www.share.org/proceedings/sh98/data/S1323.PDF, 2002.

[ACT03] ACT IT-Consulting & Services AG: *OS/390 - Vom Dinosaurier zum modernen High-End-Server*. http://www.act-online.de/OS390_2.html, 2003.

[DB2PF] *DB2 Product Family*, http://www.ibm.com/software/data/db2/