11. April 2005

Persistent Reusable Java Virtual Machine unter z/OS und Linux

Marc Beyerle,
IBM Entwicklung und Forschung,
Böblingen.

Joachim Franz,
IBM Business Consulting Services,
Application Innovation Services.

Wilhelm G. Spruth,
Wilhelm-Schickard-Institut für Informatik, Universität Tübingen,
Institut für Informatik, Universität Leipzig.

Zusammenfassung

Die Nutzung von Java zur Entwicklung von Anwendungen für Hochleistungstransaktionssysteme stellt besondere Anforderungen an die Isolation und das Leistungsverhalten in der transaktionalen Ausführung.

In dem vorliegenden Beitrag wird eine neuartige Technologie beschrieben, die verbesserte Isolationseigenschaften und eine erhebliche Leistungssteigerung ermöglicht. Die *Persistent Reusable Java Virtual Machine* erweitert eine normale Java Virtual Machine um gemeinsam genutzte Klassen und um eine Reset-Funktionalität, die eine serielle Wiederverwendbarkeit ermöglicht.

In einer Benchmark-Untersuchung unter z/OS wurde hiermit ein Performance-Gewinn um einen Faktor 325 im Vergleich mit einer normalen JVM erreicht. Für das Erstellen von Transaktionsanwendungen unter dem Betriebssystem z/OS sind Enterprise Java Beans damit eine ernstzunehmende Alternative zu bisherigen Programmiermodellen.

Abstract

The use of Java for the development of new applications in high-end transaction systems generates special isolation and performance requirements.

A new technology offering improved isolation and a significant performance improvement is discussed in this article. The *Persistent Reusable Java Virtual Machine* extends a regular Java Virtual Machine by adding the facility to share classes and by providing a reset functionality, which permits serial reuse of the JVM.

A z/OS benchmark demonstrated a performance improvement by a factor of 325, compared to a regular JVM. Thus the use of Enterprise Java Beans has to be considered as a serious alternative when developing new transactional applications under the z/OS operating system.

Schlüsselwörter

ACID, CICS, DB2, EJB, IMS, J2EE, JVM, Persistent Reusable Java Virtual Machine, PRJVM, OS/390, Serial Reusability, Transaction, WebSphere, zLinux, z/OS, zSeries.

1. Java im transaktionalen Umfeld

Java leidet unter dem Ruf, sehr langsam für die Verwendung in Transaktionsverarbeitungssystemen zu sein. Für einen Einsatz in unternehmenskritischen Umgebungen ist es erforderlich, ein hohes

Transaktionsvolumen zu gewährleisten und gleichzeitig die harten ACID-Bedingungen der Transaktionsverarbeitung strikt einzuhalten.

Die ACID-Bedingungen der Transaktionsverarbeitung beinhalten:

- Atomizität (Atomicity): Eine Transaktion wird entweder vollständig ausgeführt oder überhaupt nicht. Der Übergang vom Ursprungszustand zum Ergebniszustand erfolgt ohne erkennbare Zwischenzustände, unabhängig von Fehlern oder Abbrüchen. Änderungen betreffen Datenbanken, Nachrichten, Statusinformationen und andere.

- Konsistenzerhaltung (Consistency): Eine Transaktion überführt die transaktionsgeschützten Daten des Systems von einem konsistenten Zustand in einen anderen konsistenten Zustand. Diese Eigenschaft garantiert, dass die Daten der Datenbank nach Abschluss einer Transaktion schemakonsistent sind, d. h. alle im Datenbankschema spezifizierten Integritätsbedingungen erfüllen.

- Isolation: Die Auswirkungen einer Transaktion werden erst nach ihrer erfolgreichen Beendigung für andere Transaktionen sichtbar. Auch wenn zwei oder mehr Transaktionen gleichzeitig ausgeführt werden, wird der Anschein einer seriellen Verarbeitung gewahrt.

- Dauerhaftigkeit (Durability): Die Auswirkungen einer erfolgreich beendeten Transaktion gehen nicht verloren. Das Ergebnis einer Transaktion ist real, mit allen Konsequenzen. Es kann nur mit einer neuen Transaktion rückgängig gemacht werden.

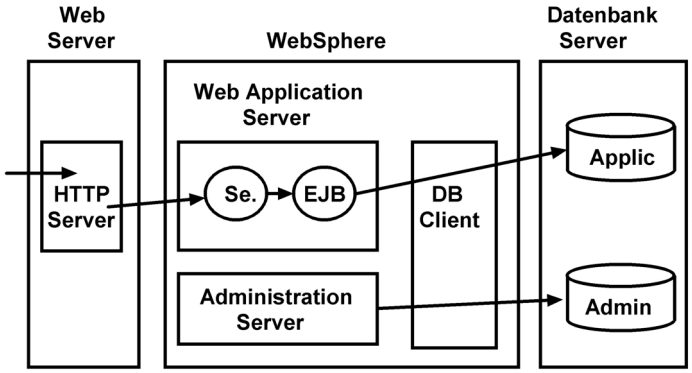

Abb. 1 : Basis-WebSphere-Konfiguration

Maßnahmen, die ein hohes Transaktionsvolumen beim Einsatz von Java gewährleisten, sind für den WebSphere Application Server in (13) beschrieben. Die Basis-Konfiguration ist in Abb. 1 wiedergegeben. Angenommen ist, dass der Zugriff auf den Server durch HTTP-Klienten erfolgt, die durch einen normalen HTTP-Server, z.B. Apache, bedient werden.

WebSphere enthält als wichtigste Komponente einen Java Web Application Server, der eine Servlet Engine (SE) und einen EJB-Container enthält. Beide laufen unter einer gemeinsamen Java Virtual Machine (JVM). Die gemeinsame JVM hat den Vorteil, dass EJB-Methoden-Aufrufe durch Servlet-Klassen nicht über RMI oder RMI/IIOP stattfinden, sondern direkt erfolgen können. Es ist möglich, dass auch Komponenten wie Web Server, ORB und Administration in der gleichen JVM laufen.

Ein zusätzlicher Administrationsserver und ein Datenbank-Klient (DB Client) vervollständigen die Konfiguration. Die Datenbank läuft entweder auf einem getrennten Rechner oder als separater Prozess in einem eigenen virtuellen Adressenraum auf dem gleichen Rechner wie WebSphere.

Für ein gutes Leistungsverhalten ist die parallele Ausführung von Transaktionen erforderlich. Die Sun HotSpot JVM unterstützt die Ausführung paralleler Java Threads und wurde spezifisch In Hinblick auf Leistungssteigerungen entwickelt. Dabei ist ein erheblicher Leistungsgewinn bei der Thread Synchronisation möglich (30). Die Nutzung von Java Threads in der Transaktionsverarbeitung ist wegen der ACID-Forderung jedoch problematisch, siehe z.B. (8):

> *The existing application isolation mechanisms, such as class loaders, do not guarantee that two arbitrary applications executing in the same instance of the JVM will not interfere with one another. Such interference can occur in many places. For instance, mutable parts of classes can leak object references and can allow one application to prevent the others from invoking certain methods. The internalized strings introduce shared, easy to capture monitors. Sharing event and finalization queues and their associated handling threads can block or hinder the execution of some application. Monopolizing of computational resources, such as heap memory, by one application can starve the others.*

Es existieren eine Reihe ähnlicher Aussagen, siehe z.B. (28):

> *Java gives the virtuoso thread programmer considerable freedom, but it also presents many pitfalls for less experienced programmers, who can create complex programs that fail in baffling ways.*

Die aktuellen EJB-Regeln schränken ein derartiges Verhalten ein, können es aber nicht eliminieren. Wenn z.B. eine fehlerhafte Transaktion eine JVM zum Absturz bringt, möglicherweise wegen eines Fehlers in einer Library Routine oder auf Grund eines übermäßigen Speicherverbrauchs, werden alle anderen Transaktionen in der gleichen JVM dadurch in Mitleidenschaft gezogen.

Bei komplexen Transaktionsumgebungen existieren weitergehende Schwächen des EJB-Modells. Einzelheiten hierzu sind in (26) wiedergegeben.

Um diese Problematik abzuschwächen, existieren eine Reihe von Lösungsansätzen (27). Es kann versucht werden, die Isolation mittels Class-Loader-Hierarchien und Restriktionen in der EJB-Spezifikation zu verbessern. Echidna ist eine Class Library, die es mehreren Prozessen ermöglicht, innerhalb einer Virtual Machine zu laufen (10). Der J-Kernel ist ein portierbares Java basiertes System, welches die zugrunde liegende Java Virtual Machine (JVM) erweitert. Es stellt mehrfache Protection Domains sowie Kommunikationskanäle zwischen den Domains zur Verfügung (20), (32).

Ein anderer Ansatz ist die Isolation mittels einer modifizierten JVM. Hierfür wird auch der Begriff *Java Operating System* verwendet: *An execution environment for Java byte code that attempts to provide isolation* (2). Beispiele sind das Alta Operating System (31) und Janos (23).

Die JSR-121 Java-Application *Isolation API* geht auf die von Sun entwickelte *Java Multitasking Virtual Machine* zurück (8). Sie spezifiziert die gleichzeitige isolierte Ausführung mehrerer Anwendungen innerhalb einer JVM (19). Diese Anwendungen werden als *Isolates* bezeichnet. Die neue Java 2 Version 1.5 J2SE Plattform (Tiger) hatte ursprünglich eine Unterstützung der Isolation API vorgesehen; aus unbekannten Gründen hat sich dies verzögert. Von der Fa. Sun existiert eine Referenz-Implementierung, welche die Isolation mit Hilfe getrennter Betriebssystemprozesse verwirklicht. Eine weitere prototypische Implementierung ist in Janos vorhanden (18).

Die Java-2 Version 1.5 J2SE ist für Arbeitsplatzrechner vorgesehen. Eine Referenz-Implementierung für J2EE Version 1.3.1 ist in (21) beschrieben.

Alle genannten Ansätze haben einen experimentellen Status und werden von den etablierten Web Application Servers wie Netweaver, Weblogic und WebSphere bisher nicht eingesetzt. In der Praxis werden Transaktionsanwendungen häufig als Threads mit sorgfältiger Vermeidung der genannten Probleme und entsprechendem Testaufwand implementiert. Es ist allerdings auch festzustellen, dass bisher nur sehr wenige unternehmenskritische und in Java geschriebene Transaktionsanwendungen Produktionsstatus haben.

WebSphere setzt für die parallele Verarbeitung von unternehmenskritischen Transaktions-anwendungen ein als *Cloning* bezeichnetes Verfahren ein. Hierbei werden ansonst identische Instanzen des Java Web Application Server als getrennte Prozesse in getrennten virtuellen Adressenräumen gefahren. Jeder der Prozesse hat eine eigene JVM und verarbeitet in jedem Augenblick eine einzige Transaktion, siehe Abb. 2. Der HTTP-Server speichert eintreffende Anforderungen in einer Warteschlange (Work Queue). Im einfachsten Fall verteilt er sie anschließend mit Hilfe eines Round-Robin-Verfahrens auf die einzelnen Prozesse. Unter z/OS kann die Lastverteilung durch den z/OS Workload Manager (WLM) im *Queueing Manager Services* (QMS) Mode erfolgen, siehe (12), S. 8. Nach Abschluss einer Transaktion (Commit oder Rollback) steht der

Prozess für eine weitere Transaktion zur Verfügung. Für ein evtl. erforderliches Session Management ist der Administrationsserver zuständig.

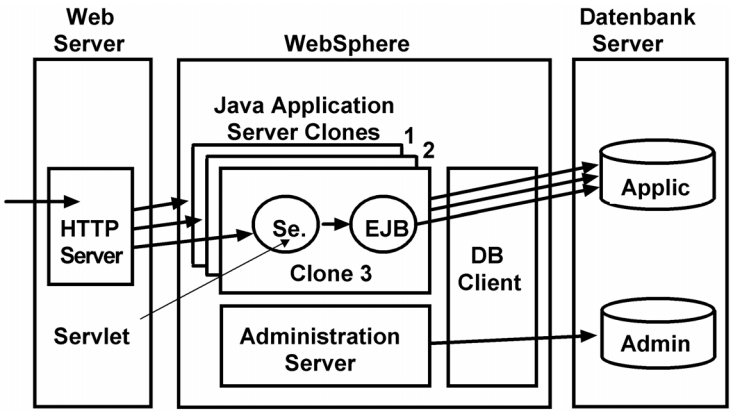

Abb. 2 : Mehrfache Instanzen des Web Application Servers

Referenzen (3) und (4) enthalten eine detaillierte Beschreibung der internen WebSphere-Struktur unter z/OS. Es existieren wesentliche Erweiterungen gegenüber den WebSphere-Implementierungen auf Unix-, Linux- und Windows-Plattformen.

2. Wiederverwendung einer JVM für aufeinanderfolgende Transaktionen

Sun's Java Virtual Machine wird innerhalb eines Prozesses erstellt, um eine Anwendung zu laden und auszuführen. Der Prozess wird durch den Scheduler des Betriebssystem-Kernels aktiviert, läuft in der Regel in einem eigenen virtuellen Adressenraum und gestattet die Ausführung von Threads. Die JVM wird gestartet, wenn die Anwendung startet, und sie wird terminiert, wenn die Anwendung abgeschlossen ist. Wir bezeichnen diese Art der JVM als eine *normale JVM*.

Häufig ist ein Java-Programm ein länger laufender Prozess mit einer Dauer von Sekunden, Minuten oder sogar Stunden. Im Vergleich hierzu ist der Zeitaufwand für den Start und die Initialisierung einer JVM vergleichsweise gering.

Anders sieht es bei Enterprise Java Beans aus, besonders in einem CICS-Umfeld. Häufig verarbeitet CICS Hunderte oder Tausende von Transaktionen pro Sekunde mit einer Verarbeitungsdauer für die Geschäftslogik von weniger als einer Sekunde pro Transaktion. Im Vergleich hierzu ist der Aufwand für das Starten und die Initialisierung einer JVM nicht vernachlässigbar.

Das Vorgehensmodell, eine Transaktion isoliert in einer eigenen Java-VM auszuführen, kann auf verschiedene Arten implementiert werden, siehe Abb. 3. Im einfachsten Fall wird für jede Transaktion eine neue JVM gestartet und nach Abschluss der Transaktion wieder beendet. Dies ist der klassische Ansatz, für den eine für die Bedürfnisse von z/OS optimierte JVM-Implementierung geschaffen wurde. Diese Implementierung ist voll kompatibel mit dem Java-Standard und ist Bestandteil des z/OS JDK.

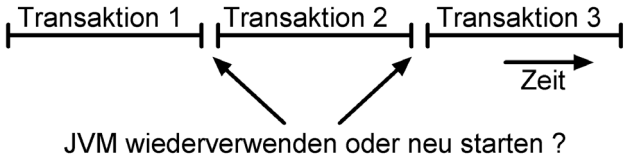

Abb. 3 : Was geschieht mit der JVM nach Abschluss einer Transaktion ?

Die Ausführung einer Java-Anwendung erfordert die Erstellung und Initialisierung einer eigenen JVM Instanz mit Hilfe eines Bootstrap Class Loader, welcher als Teil des Startvorgangs einer JVM die elementaren Java-Klassen lädt. Dieser wird durch einen externen Prozess aufgerufen (z.B. durch einen JNI-Call) und initialisiert sich anschließend, wobei unter anderem Systemklassen geladen werden, um den Kontext der JVM aufzusetzen. Das Hoch- und Herunterfahren einer JVM hat jedoch

einen erheblichen Zeitaufwand zur Folge. Eine Studie der Java-Anwendungen unter OS/390 Unix System Services hat ergeben, dass bei jeder Initialisierung einer JVM etwa 60 Systemklassen geladen sowie 700 Array-Objekte und über 1000 non-Array-Objekte allokiert und angelegt werden müssen (9). Nach (7) beträgt die Pfadlänge "*between 20 and 100 million instructions, dependent on the underlying platform*". Daher erreicht man mit dieser Vorgehensweise nur sehr geringe Transaktionsraten.

Um den Aufwand für eine volle Initialisierung zu vermeiden, ist es naheliegend, eine existierende JVM für mehrere aufeinanderfolgende Transaktionen wiederzuverwenden. Bei einem *Serial-Reuse*-Transaktionsmodell werden mehrere Transaktionen nacheinander in der gleichen JVM durchgeführt. System- und Middleware-Klassen werden nur einmal geladen und verifiziert. Jeder Prozess führt in jedem Augenblick eine einzige Transaktion aus.

Wenn dieselbe JVM für mehrere in sich abgeschlossene Transaktionen hintereinander benutzt wird, müssen die ACID-Eigenschaften gewährleistet sein. Hierzu ist es erforderlich, dass die JVM bei jeder neuen Benutzung in einen sauberen (clean) Zustand versetzt wird, unabhängig von der vorherigen Nutzung.

Es ist grundsätzlich möglich, dass eine Transaktion die Ausführung der Folgetransaktion beeinflusst, indem sie den Zustand (State) der JVM ändert. Beispiele für eventuelle sicherheitskritische Reste der vorhergehenden Transaktion sind überschriebene statische Variablen, das Starten von Threads oder geladene native Bibliotheken.

3. Konzept der PRJVM

Unter z/OS steht die P*ersistent Reusable JVM* (PRJVM) zur Verfügung. Hier wird die gewünschte Zuverlässigkeit und Transaktionssicherheit erreicht, indem die JVM um eine Reset-Funktionalität erweitert wird. Nach einem Reset-Aufruf kann die Folgetransaktion nicht herausfinden, ob sie in einer neuen oder einer vorher bereits benutzten JVM läuft. Die PRJVM ist eine Erweiterung der normalen JVM und damit eine *fully compliant* Java-Implementierung. Das bedeutet, dass die PRJVM auch ohne Nutzung der Reset-Funktionalität eingesetzt werden kann, wobei sie sich wie eine normale JVM verhält.

Die in (7) und (14) vorgestellte Implementierung der PRJVM basiert auf einem Prototypen, der von Dillenberger *et al.* entworfen wurde (9).

Eine JVM wird normalerweise durch ein als *Launche*r *Subsystem* bezeichnetes Programm erstellt (17). Für eine PRJVM ist der dabei eingesetzte JNI_CreateJavaVM Call um eine neue Option **-Xresettable** erweitert. Diese Option spezifiziert, dass die JVM eine zusätzliche Reset-Funktionalität erhält. Eine Transaktion wird durch einen Middleware-Aufruf einer getWork-Methode von einer Work Queue gestartet. Nach Beendigung erfolgt ein Reset der JVM mit einem Aufruf *ResetJavaVM*. Die nächste Transaktion erhält eine saubere JVM.

Die meisten Transaktionen verhalten sich gutartig; für sie ist ein Reset der JVM unproblematisch. Aus einer Reihe von Gründen ist es jedoch möglich, dass die Ausführung einer Transaktion die JVM in einen Zustand *unresettable* versetzt. Dies kann auf einem „asozialen", jedoch legalen Verhalten der Transaktion beruhen. Damit ist die Wiederverwendbarkeit der JVM für folgende Transaktionen nicht mehr gegeben, da eine Garantie für ACID-Eigenschaften nicht erfüllt werden kann. Eine derartige JVM wird als *unsauber* (dirty) bezeichnet.

In diesem Fall vernichtet der Launcher die JVM und initialisiert eine neue, saubere JVM. Um den Initialisierungs-Aufwand gering zu halten, existieren zwei Ausprägungen der PRJVM als Teil der PRJVM-Technologie, die gemeinsam mit der normalen JVM Teile des z/OS JDK sind. Eine *Master JVM* ist ausschließlich für das Laden und Linken der erforderlichen Systemklassen zuständig (*resource-owning JVM*). Dies gilt besonders für die etwa 60 Systemklassen, die für die Initialisierung einer JVM erforderlich sind.

Eine *Worker JVM* ist für die Ausführung von Java-Anwendungen gedacht (realisiert durch Middleware- und Anwendungs-Klassen). Mehrere Worker JVMs können nun gemeinsam Systemdaten nutzen (z.B. classes, method tables, constant pools, field blocks). Diese stehen in einem gemeinsam genutzten

Hauptspeicherbereich. Zusätzlich unterhalten die einzelnen Worker JVMs private lokale Speicherbereiche.

Ein Launcher-Programm ist für den Lebenszyklus (life cycle) und die Wiederverwendbarkeit (reuse) der JVMs zuständig. Der Launcher führt eine äußere Schleife aus, in der die Worker JVM geladen und initialisiert wird. In einer inneren Schleife führt die Worker JVM eine Java-Transaktion aus und bestimmt nach ihrem Abschluss, ob eine Wiederverwendung (Reset) möglich ist. Wenn ja, wird die Worker JVM zurückgesetzt und steht dann für die nächste Transaktion wiederverwendbar bereit (siehe Abb. 4).

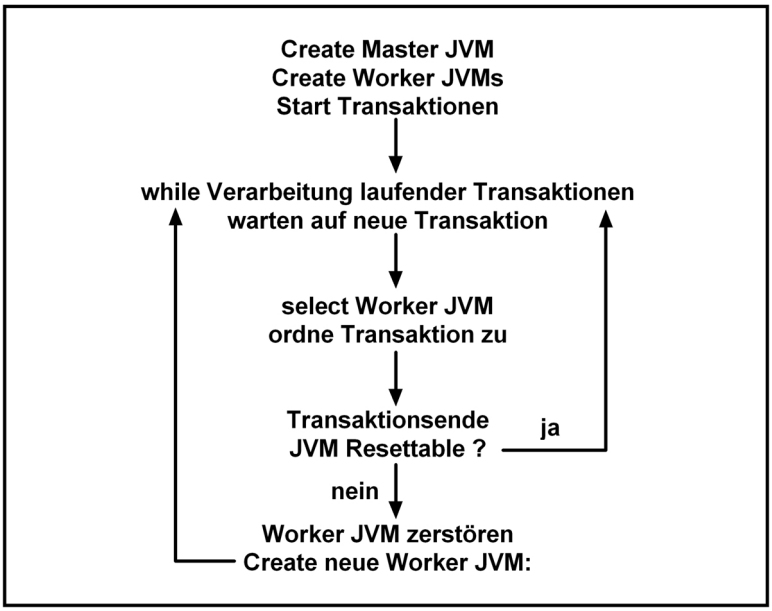

Abb. 4 : Launcher Program

Nach einer Transaktion muss sichergestellt werden, dass keine Java Exceptions ausstehen. Dieses wird von dem *Launcher Subsystem* erledigt. Danach werden in den Middleware-Klassen spezielle Methoden aufgerufen, in denen die Aufräumarbeiten für die Middleware stattfinden.

Eine Worker JVM kann ggf. bei Transaktionsende als unsauber (*dirty)* markiert werden. Es führen all diejenigen Vorgänge zu 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, übrig gebliebene geöffnete Dateien oder noch existierende Java Threads. Eine weitere Bedingung für einen erfolgreichen Reset ist, dass keine Referenzen von persistenten Objekten in den Transient Heap zu Anwendungsobjekten 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 Anwendungsobjekten zu entfernen.

Eine ausführliche Diskussion ist in (5) und in (14), Kapitel 5, “Developing Applications” zu finden. Wenn eine dieser Bedingungen auftritt, erzeugt der Aufruf *ResetJavaVM* den Rückgabewert *JNI_ERR (Reset failed)*.

Verallgemeinert kann man sagen, dass hierbei ungefähr die gleichen Restriktionen zur Geltung kommen, wie sie für Enterprise Java Beans festgelegt wurden. So dürfen zum Beispiel von den Anwendungsklassen keine Java Threads gestartet werden.

Grundsätzlich hat die PRJVM zwei Operations-Modi. Das hier besprochene Verfahren wird als *Reset-Modus* bezeichnet. Die Alternative ist der *Running-Modus*. Er kann eingesetzt werden, wenn die Anwendungsarchitektur mit reduzierten Isolationseigenschaften zwischen Transaktionen auskommt, welche die gleiche JVM benutzen.

4. Virtueller Adressenraum mit mehrfachen JVMs

Für z/OS und OS/390 Sprachen wie Cobol, PL/1, C/C++ und Fortran existiert eine als *LE* (*Language Environment*) bezeichnete Einrichtung. Dies ist zunächst eine Laufzeitumgebung von einheitlichen Bibliotheken, die von den genannten Sprachcompilern gemeinsam benutzt werden kann.

Ein Teil von LE sind die *Run Units* oder *Enclaves*. Enclaves sind Thread-ähnliche Einheiten, die jeweils unter einem eigenen Prozessleitblock (Process Control Block PCB, unter z/OS als Task Control Block, TCB bezeichnet) gemeinsam in einem virtuellen Adressenraum laufen. Sie benutzen hierfür einen eigenen Speicherschutzschlüssel (storage protection key), normalerweise Schlüssel Nr. 8. Der zSeries-Hardware-Speicherschutz arbeitet unabhängig und zusätzlich zu dem üblichen Speicherschutz über Segment- und Seitentafeln und ist eine Einrichtung der z/Series-Hardware-Architektur, die auf anderen Plattformen nicht verfügbar ist (siehe (11), S. 14).

LE Enclaves können eingesetzt werden, um unterschiedliche Anwendungen innerhalb des gleichen virtuellen Adressenraums voneinander zu isolieren. Sie werden u.a. in transaktionalen Subsystemen wie CICS, IMS und DB2 benutzt und wurden ursprünglich entwickelt, um Threads für Anwendungsprogramme zu ermöglichen, die in prozeduralen Programmiersprachen wie C, Cobol oder PL/1 geschrieben sind. Bei der Programmierung in Java sind Enclaves geeignet, mehrere JVMs innerhalb eines Adressenraums laufen zu lassen, wobei jede Enclave eine JVM aufnimmt. In dieser als *Open Transaction Environment* (OTE) bezeichneten Umgebung läuft jede JVM als ein selbständiger Thread.

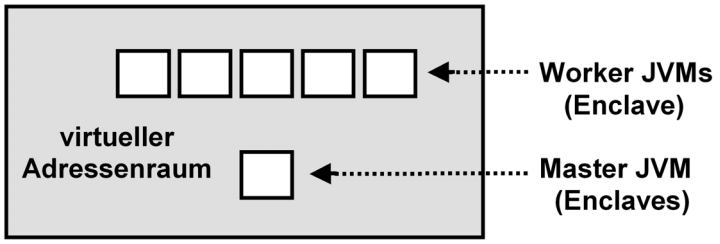

Abb. 5 : JVM Set

Hierzu kann man in einem virtuellen Adressenraum des z/OS einen *JVM Set* oder *JVM Pool* anlegen, siehe Abb. 5. Die Anzahl der JVMs innerhalb eines virtuellen Adressenraums und eines JVM Sets wird durch einen Systemparameter festgelegt. Alle JVMs teilen sich dabei einen *System Heap.* Dadurch wird die Startzeit einer einzelnen JVM erheblich verkürzt, da später gestartete JVMs den Großteil der Java-Systemklassen bereits im System Heap vorfinden. Außerdem reduziert diese Anordnung den Hauptspeicherplatzbedarf, da der Speicher für die Systemklassen nur einmal zugeordnet werden muss. Die erste JVM übernimmt dabei die Rolle der *Master JVM* und kontrolliert den JVM Set:

- 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 Systemklassen und *Shareable* Anwendungsklassen benötigt wird.

Nachdem die Master JVM den System Heap und diejenigen JVM-Optionen, welche für das JVM Set relevant sind (zum Beispiel die verschiedenen Class-Loader-Einstellungen), initialisiert hat, wird die eigentliche (transaktionsbezogene) Arbeit den Worker JVMs übertragen.

Wenn eine neue Transaktion verarbeitet werden soll, wird ihr eine Worker JVM aus dem JVM Set zugeordnet. Die Worker JVMs innerhalb des JVM Sets sind nicht notwendigerweise identisch; beispielsweise können sie mit unterschiedlichen Parametern oder Heap-Größen initialisiert worden sein. Die Eigenschaften einer Worker JVM werden durch einen Parameter JVMPROFILE festgelegt; alle Worker JVMs mit dem gleichen JVMPROFILE sind identisch. Ein JVM Set kann Worker JVMs mit unterschiedlichen Profilen enthalten. Einer neu zu verarbeitenden Transaktion wird eine Worker JVM mit einem passenden Profile zugeordnet (siehe Abb. 6).

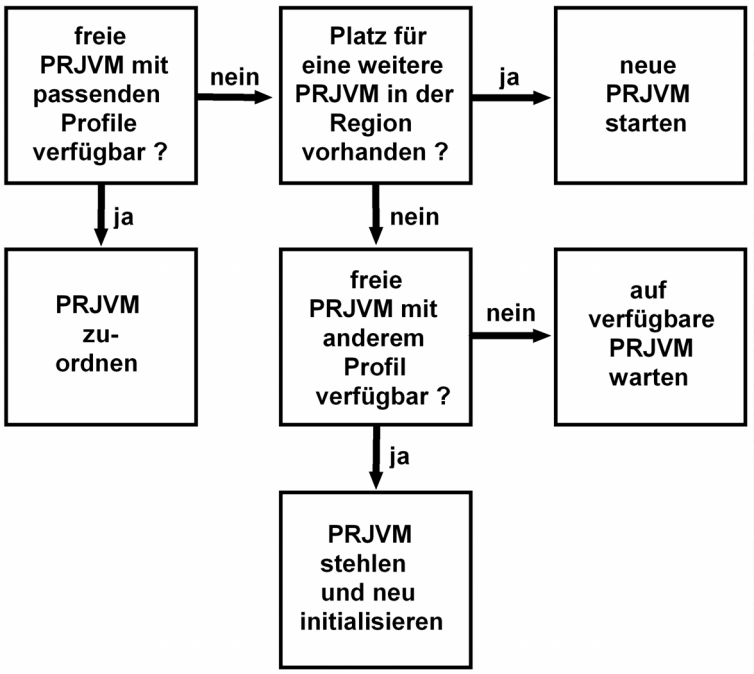

Abb.6 : Auswahl einer Worker JVM

Eine Enclave speichert drei Arten von Klassen:

- *JVM Code (auch als primordial Code bezeichnet)*, bestehend aus Systemklassen und standardmäßige Java-Erweiterungsklassen, die Basis-Dienste für die JVM bereit stellen.
- M*iddleware* stellt Funktionen zur Verfügung, die auf Ressourcen zugreifen. Dazu gehören JCICS-Schnittstellenklassen, Klassen für JCA und CICS Transaction Gateway, SQLJ/JDBC-Klassen und JNDI-Klassen. Middleware-Klassen können sich selbst in einen sauberen Zustand zurücksetzen. Sie laden Anwendungen und führen sie aus.
- *Gemeinsam nutzbarer (shareable) Anwendungscode.* Dieser besteht aus den Enterprise Beans und weiterem gemeinsam nutzbaren Anwendungscode.

Middleware-Klassen sind Teil der Infrastruktur eines Transaktionsverarbeitungssystems und dürfen ihren inneren Zustand über die Grenzen einer Transaktion hinaus aufrechterhalten. Middleware-Klassen dürfen auch solche Aktionen durchführen, welche eine Worker JVM in einen unsauberen (*dirty)* Zustand versetzen, wenn sie von *Anwendungsklassen (Application Classes)* initiiert werden.

Anwendungsklassen sind “normale” Java-Klassen, die in einem Transaktionsverarbeitungssystem die eigentlichen Transaktionsanwendungen darstellen. Hier wird derjenige Teil der *Geschäftslogik* 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 Anwendungsklassen platziert werden. Typischerweise benutzen Anwendungsklassen Teile der Middleware, um Daten in Enterprise Information Systems zu aktualisieren. Dies hat unter anderem Performance-Gründe, 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.

Shareable bedeutet dabei, dass diese Klassen von allen JVMs in einem JVM Set gemeinsam benutzt werden können und deshalb nur einmal in den entsprechenden Teil des Heaps geladen werden müssen. *Nonshareable* Anwendungsklassen sind ebenfalls denkbar, werden aber im praktischen Einsatz weniger häufig vorkommen.

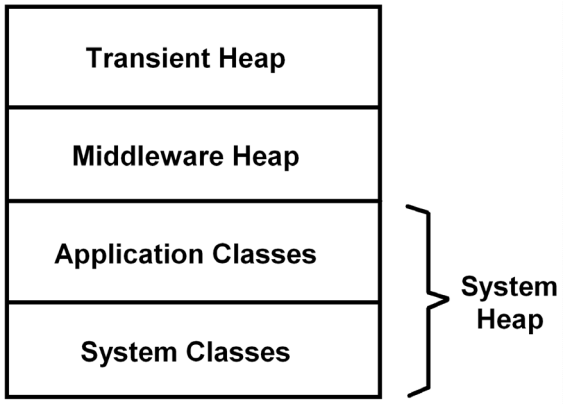

Abb. 7 : Aufteilung des Heaps

Die PRJVM verwaltet ihren Laufzeit-Speicher in mehreren getrennten Heaps mit unterschiedlichen Eigenschaften (siehe Abb. 7):

- Der *Transient Heap* enthält Objekte, die von den Anwendungsklassen erzeugt werden.
- Der *Middleware Heap* enthält Objekte, die von den Middleware-Klassen erzeugt werden. Middleware-Klassen haben höhere Privilegien als die Anwendungsklassen. Beispielsweise können Klassen und Libraries in einen Cache geladen werden, die von mehreren Anwendungen benutzt werden.
- Der *System Heap* besteht aus dem Main System Heap, der die Systemklassen enthält, und dem Shareable Application Class Heap. Einzelheiten sind in (33) zu finden.

Die Aufteilung in mehrere Heaps ermöglicht eine Aufteilung in transaktionsbezogene (und somit kurzlebige) und in langlebigere Daten. Dies ist 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 (15).

Für den System Heap existiert keine Garbage Collection. Die Systemklassen bleiben unverändert; die Anwendungsklassen werden bei Bedarf zurückgesetzt. Middleware-Referenzen auf den Transient Heap müssen bei Transaktionsende zurückgesetzt werden. Weiterhin müssen auch solche Ressourcen freigegeben werden, welche von einer Middleware-Klasse angefordert wurden und für die nächste Transaktion nicht mehr zur Verfügung stehen dürfen.

Die Garbage Collection für den Middleware Heap wird nicht automatisch nach jedem Reset durchgeführt, sondern muss vom Launcher Subsystem in regelmäßigen Abständen initiiert werden, siehe (14), Kapitel 2, Abschnitt “Required garbage collection”. Hiermit ist eine wesentliche Leistungssteigerung möglich. Tabelle 7.2 in (6) enthält hierzu ein Beispiel: Wenn man die Garbage Collection bei diesem Transaktionstyp z.B. nach jedem 175. Reset durchführt, ist die Gesamtleistung des Systems 11,72 Mal höher als bei einer Einstellung, bei welcher die Garbage Collection nach jedem Reset durchgeführt wird.

Im Transient Heap befinden sich diejenigen Objekte, die im Anwendungskontext einer Transaktion erzeugt werden, und deren Gültigkeit mit dem Ende einer Transaktion erlischt. Beispiele für solche Objekte sind nonshareable Anwendungsklassen und solche Objekte, die von Anwendungsklassen erzeugt und nicht im Middleware Heap platziert wurden. Eine Worker JVM benutzt den Transient Heap um temporäre Hash-Table-Einträge, temporäre Strings sowie Daten abzuspeichern, die lokalen Objekten zugeordnet sind. Am Ende einer Transaktion sollten alle diese Objekte von den persistenten 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. Abb. 8 stellt dar, wie der Transient Heap nach jedem Reset der Worker JVM gelöscht wird, während dies für den Middleware Heap lediglich nach n Resets erfolgt.

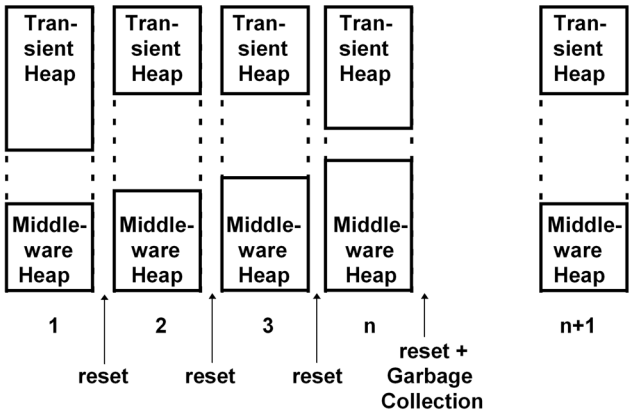

Abb. 8 : Leeren des Middleware Heap nach mehreren Transaktionen

Damit dies gelingt, ist es unbedingt notwendig, dass keine Referenzen in den transienten Teil des Heaps mehr existieren. Sollten am Ende einer Transaktion doch Referenzen übrig sein, muss die PRJVM in einer aufwendigen Operation (`TraceForDirty()`) feststellen, ob eine dieser Referenzen noch aktiv ist. Wenn der Algorithmus eine aktive Referenz findet, wird die Java-VM als *dirty* markiert.

Die hier vorgestellte Split-Heap-Struktur ist dem in einigen Java Virtual Machines realisierten Ansatz *Generational Garbage Collection* ähnlich (1). Bei diesem wird der Heap ebenfalls in verschiedene Bereiche unterteilt, die als *Generations* bezeichnet werden. Der in der Regel *Nursery* 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 hierzu hat die Split Heap Struktur des PRJVM Ansatzes den Vorteil, dass man die erwartete Lebensdauer der Objekte *a priori* bestimmen kann. Somit wird ein Verschieben der Objekte im Speicher unnötig. Auch fehlt in der Technik Generational Garbage Collection der hohe Grad an Isolation, den die PRJVM bietet.

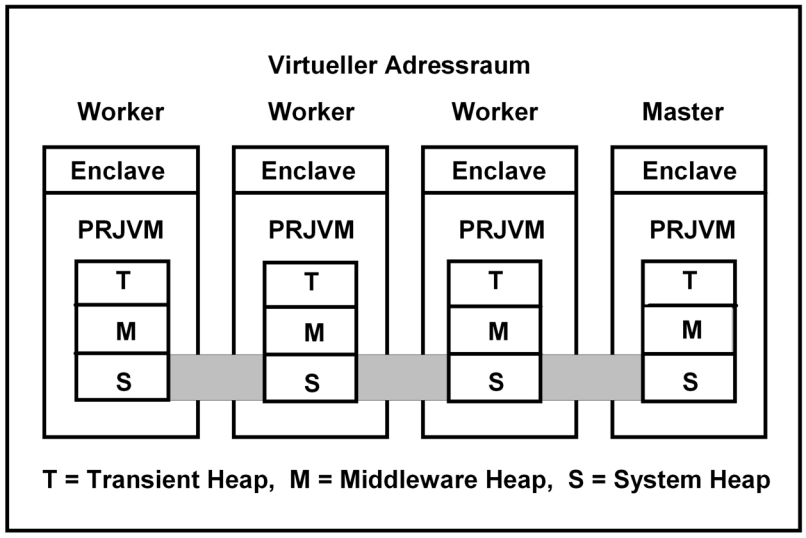

Abb. 9 : Gemeinsame Nutzung von Klassen

Wie in Abb. 9 dargestellt können jetzt alle JVMs eines JVM Sets innerhalb des gleichen Adressenraums den System Heap gemeinsam nutzen. Das Gleiche gilt für solche Klassen, die durch den Just-in-Time Compiler übersetzt wurden. Dies verringert den erforderlichen Aufwand, wenn eine Worker JVM zerstört und eine neue, saubere Worker JVM initialisiert werden muss.

Das hier geschilderte Verfahren wird u.a. von CICS, IMS und von den DB2 Stored Procedures eingesetzt. Ein Beispiel für die Nutzung der PRJVM unter z/OS CICS innerhalb einer Linux-Grid-Umgebung ist in (24) beschrieben. In (25) wird die Nutzung der PRJVM in einer z/OS DB2 Stored

Procedure erläutert. Die Workload-Manager-Komponente des z/OS-Betriebssystems (siehe (11), S. 121) wird für die Lastverteilung eingesetzt. Unter IMS erlauben die Java Message Processing Region und die Java Batch Processing Region den Einsatz der PRJVM (22).

5. Leistungsgewinn

5.1 Benchmark

Die folgende Leistungsuntersuchung erfolgte mit einer konkreten Anwendung, dem Basic Online Banking System. Dieses implementiert mit kleinen Abweichungen eine Referenzimplementierung für Online-Banking-Systeme, die *TPC Benchmark A, Standard Specification* des *Transaction Processing Performance Council.* Der Durchsatz wird in Transaktionen pro Sekunde gemessen. Die Transaktionen werden von einem getrennten Terminal-Emulator initiiert, welcher Online-Terminals simuliert.

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

Der TPC-A-Benchmark legt fest, wie die einzelnen Schritte der Transaktion zu verrichten sind. Auch der Aufbau der Datenbank ist vorgeschrieben. Weiterhin definiert die TPC-A-Spezifikation, wie die Antwortzeit *(Response Time*) und der Durchsatz *(Throughput*) zu messen sind. Auf diese Vorgaben wurde bei der Entwicklung des Online-Banking-Systems geachtet.

Verglichen wird das Leistungsverhalten auf einer zSeries-Plattform unter den beiden Betriebssystemen z/OS Version 1 Release 4 und Linux for zSeries (SuSE Linux Enterprise Server SLES-7, Kernel 2.4.7). Unter Linux for zSeries (zLinux) ist lediglich die normale JVM verfügbar. Unter z/OS wurde alternativ eine normale JVM und die neue PRJVM eingesetzt. Ein Unterschied zwischen den z/OS und zLinux Java Virtual Machines sind die unterschiedlichen *Default Encoding*s. Während die JVM unter zLinux ASCII benutzt (ISO-8859-1), verwendet die JVM unter z/OS standardmäßig EBCDIC (CP1047), um zum Beispiel Zeichenketten in eine Datei zu schreiben. In diesem Zusammenhang ist erwähnenswert, dass ca. 60 - 70 Prozent aller geschäftsrelevanten Daten weltweit im EBCDIC- und nicht im ASCII-Format vorliegen.

Die Online-Banking-Daten wurden unter z/OS in einer DB2-Datenbank abgelegt (DB2 for z/OS and OS/390). Dabei lief DB2 unter derselben Betriebssystem-Instanz (z/OS) wie das Basic Online Banking System. Unter zLinux wurden die Daten ebenfalls in einer DB2-Datenbank abgelegt (DB2 Universal Database for Linux). Dabei lief DB2 ebenfalls unter derselben Betriebssystem-Instanz (zLinux) wie das Basic Online Banking System. Dies wurde so festgelegt, um sowohl unter z/OS als auch unter zLinux dasselbe Setup zu verwenden.

5.2 Ergebnisse

Die Untersuchungen wurden mit drei unterschiedlichen Testumgebungen durchgeführt. In allen drei Fällen hatte die Umgebung die in Abb. 10 wiedergegebene Konfiguration.

Eine Java-Klienten-Anwendung sendet eine Nachricht über TCP/IP an die Kommunikator-Komponente des Servers. Diese speichert sie in einer Warteschlange. Nachrichten in der Warteschlange werden von der Java-Komponente ausgelesen und verarbeitet. Die Kommunikation mit der DB2-Komponente erfolgt über SQLJ.

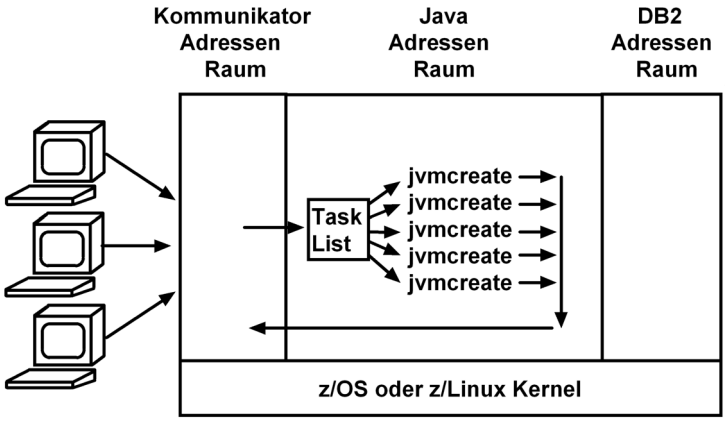

Abb. 10 : Testumgebung: Architektur des Basic Online Banking System

Der Java-Adressenraum enthält zunächst eine JVM, eine Middleware-Klasse und eine Anwendungsklasse, welche die Testanwendung Basic Online Banking System implementieren. Es werden auf der Klienten- und auf der Server-Seite normale Java-Klassen (keine Servlets oder EJBs) eingesetzt. Das Basic Online Banking System ist ein „hybrides“ Programm, bestehend aus in C und in Java geschriebenen Teilen. Die C-Komponenten werden für die Launcher-Funktionen benötigt (siehe Abb. 4).

Untersucht wurde der Durchsatz an Transaktionen pro Sekunde mit drei unterschiedliche Test Konfigurationen, die in Abb. 11 wiedergegeben sind:

- Benutzung einer normalen JVM unter dem Betriebssystem zLinux
- Benutzung einer normalen JVM unter dem Betriebssystem z/OS
- Benutzung einer Persistent Reusable JVM unter dem Betriebssystem z/OS

Die PRJVM ist derzeitig außer z/OS bzw. OS/390 auf keiner anderen Plattform verfügbar, und damit auch nicht unter Linux.

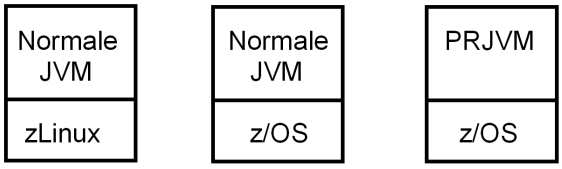

Abb. 11 : Drei alternative Testkonfigurationen

Die in Abb. 12 dargestellte Tabelle enthält die Ergebnisse. Die Werte sind dabei in Transaktionen pro Sekunde angegeben.

Normale JVM zLinux Tx/s	normale JVM z/OS Tx/s	Verhältnis zLinux zu z/OS	PRJVM z/OS Tx/s	Verhältnis PRJVM zu normale JVM
1,223	0,796	1,54	261,04	327,94

Abb. 12 : Gemessene Werte in Transaktionen/s

Eine normale JVM ist unter z/OS in der Lage, 0,796 Transaktionen/s auszuführen. Bei Benutzung der PRJVM steigt dies auf 261,04 Transaktionen/s. Dies ist eine Verbesserung um einen Faktor 327,94.

Unter zLinux werden 1,223 Transaktionen/s mit der normalen JVM erreicht, im Gegensatz zu den 0,796 Transaktionen/s unter z/OS. Der Unterschied ist auf Grund der zusätzlichen Fähigkeiten des

z/OS Kernel nachvollziehbar, die hier nicht zum Tragen kommen. In beiden Fällen lag die Auslastung der CPU bei nahezu 100 %.

Einzelheiten sind in einer am Wilhelm-Schickard-Institut für Informatik der Universität Tübingen entstandenen Diplomarbeit enthalten (6).

6. Zusammenfassung und Ausblick

Die multithreaded Ausführung von Java-Transaktionen leidet unter fehlenden Isolationseigenschaften des derzeitigen JVM-Standards. Die Wiederverwendung der JVM für aufeinanderfolgende Transaktionen ist ebenfalls problematisch, weil eine vorhergehende Transaktion den Zustand der JVM möglicherweise verändert hat. Der Anwendungsprogrammierer muss darauf achten, die Isolation mit Hilfe programmtechnischer Maßnahmen sicherzustellen. Bei wachsender Anwendungskomplexität ist dies ein zunehmend problematischer Faktor.

Eine radikale Lösung besteht darin, lediglich eine Transaktion pro JVM zuzulassen, und die JVM nach jedem Gebrauch zu zerstören und neu zu starten. Dies führt zu einem wenig akzeptablen Leistungsverhalten (Abb. 12, Tabellenwerte „Normale JVM“). Der PRJVM-Ansatz löst dieses Problem, indem eine JVM zu jedem Zeitpunkt nur eine Transaktion ausführt. Multithreading ist dennoch möglich, indem mehrere JVMs innerhalb eines virtuellen Adressenraums gleichzeitig ausgeführt werden. Hierzu wird der existierende z/OS-Enclave-Mechanismus benutzt, dessen Transaktionssicherheit langjährig erprobt ist. Gleichzeitig erlaubt eine zusätzliche Reset-Eigenschaft eine Wiederverwendung der JVM für aufeinanderfolgende Transaktionen (Abb. 12, Tabellenwerte „PRJVM“).

Die Leistungsverbesserung wurde im Zusammenhang mit einem leicht abgeänderten TPC-A-Benchmark untersucht. Im Vergleich zu der erwähnten radikalen Lösung erhöhte sich der Transaktionsdurchsatz beim Einsatz der PRJVM um den Faktor 327. Bei komplexerer Geschäftslogik wird die Ausführungszeit der eigentlichen Anwendung an Bedeutung gewinnen und der Unterschied im Durchsatz entsprechend kleiner werden.

Die hier wiedergegebenen Ergebnisse sind viel versprechend für den zukünftigen Einsatz von Java und der PRJVM in Hochleistungstransaktionssystemen.. Auch andere Untersuchungen bestätigen eine signifikante Leistungsverbesserung (siehe (24), S. 14). Derzeitig ist die PRJVM unter dem z/OS-Betriebssystem als Bestandteil der CICS und IMS Transaction Server und der DB2 Stored Procedures verfügbar, siehe (16).

In Anbetracht des in dieser Arbeit aufgezeigten Leistungsgewinns ist anzunehmen, dass eine Verbesserung der Isolationseigenschaften der JVM, vergleichbar zu der PRJVM-Technologie, früher oder später auch auf anderen Plattformen verfügbar sein wird. Eine entsprechende Entwicklung wird derzeitig mit der Multi-Tasking Virtual Machine von der Fa. Sun verfolgt (21). Es bestehen Ähnlichkeiten, aber auch Unterschiede zur PRJVM. Dies ist eine viel versprechende, aber bisher noch nicht abgeschlossene Entwicklung, über deren Tragfähigkeit noch keine Erkenntnisse vorliegen.

Für lang laufende Transaktionen, besonders bei der Abbildung von Geschäftsprozessen, wird häufig eine großzügigere Interpretation der ACID-Regeln gefordert, um damit einen höheren Durchsatz an Transaktionen zu ermöglichen. Eine Möglichkeit hierbei ist es einen Geschäftsprozess in eine Grob- und Feinsteuerung aufzuteilen. Hierbei kann die Verwendung der PRJVM-Technologie für die Feinsteuerung ("short running" transaktionale Subworkflows) einen entscheidenden Vorteil bringen. Ein solcher Ansatz ist in (29) beschrieben.

7. Danksagung

Die Autoren danken Prof. Dr. Rosenstiel, Wilhelm-Schickard-Institut für Informatik der Universität Tübingen, sowie Dr. Bernhard Dierberger, Erich Amrehn, Dr. Nikolaus Breuer, Uwe Denneler, Axel Lonzer, Hans Dieter Mertiens und Elisabeth Puritscher, IBM Entwicklung und Forschung, Böblingen, für die Unterstützung der Arbeit. Weiterhin danken Wir Herrn Prof. Dr. Theo Härder, TU Kaiserslautern, für zahlreiche wertvolle Hinweise.

8. Abkürzungen

ACID	Atomicity, Concurrency, Isolation, Durability
CICS	Customer Information Control System
DB2	Data Base 2 (IBM relationales Datenbanksystem)
EBCDIC	Extended Binary Coded Decimal Interchange Code
EJB	Enterprise Java-Bean
GC	Garbage Collection
IIOP	Internet Inter-ORB Protocol
IMS	Information Management System
JCA	Java Connection Architecture
JCP	Java Community Process
JDK	Java Development Kit
JDBC	Java Data Base Connectivity
JIT	Just In Time
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JSR	Java Specification Request
JVM	Java Virtual Machine
LE	Language Environment
OTE	Open Transaction Environment
PCB	Prozessleitblock (Process Control Block)
PRJVM	Persistent Reusable JVM
QMS	Queueing Manager Services
RMI	Remote Method Invocation
SMP	Symmetric Multiprocessor
SQLJ	SQL for Java
SVC	Supervisor Call (System Call)
TCB	Task Control Block (IBM Terminologie für den PCB)
WLM	Workload Manager
zLinux	Linux Port auf die zSeries Plattform
z/OS	zSeries Betriebssystem, 64 Bit Nachfolger von OS/390

9. Literatur

(1) A. W. Appel: *Simple generational garbage collection and fast allocation.*
Source Software - Practice & Experience, Volume 19 , Issue 2 (February 1989), p. 171 – 183.

(2) G. Back, P. Tullmann, L. Stoller, W. Hsieh, J. Lepreau: *Techniques for the Design of Java Operating Systems*.
USENIX Annual Technical Conference, June 2000.

(3) Don Bagwell: *An Introduction to WebSphere for z/OS Version 5.*
IBM Washington Systems Center, Gaithersburg, MD, USA, Sept. 2003.
http://www-1.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP100339.

(4) Don Bagwell: *WebSphere for z/OS Version 5 - "Gen 5" Wildfire Workshop Presentations*, *Unit 1, WebSphere for z/OS Version 5 Overview.*
IBM Washington Systems Center, Gaithersburg, MD, USA, May 2004.
http://www-1.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/PRS708.

(5) K. Barabash et al. : *Mostly Accurate Stack Scanning*. Proceedings of the Java Virtual Machine Research and Technology Symposium, Monterey, California, April 23–24, 2001.

(6) M. Beyerle : *Architekturanalyse der Java Virtual Machine unter z/OS und Linux.*
Diplomarbeit, Wilhelm-Schickard Institut für Informatik, Universität Tübingen, 2003.

(7) S. Borman, S. Paice, M. Webster, M. Trotter, R. McGuire, A. Stevens, B. Hutchison, R. Berry: *A Serially Reusable Java(tm) Virtual Machine Implementation for High Volume, Highly Reliable, Transaction Processing.*
Technical Report TR 29.3406, IBM Corporation, Hursley, United Kingdom.

(8) Grzegorz Czajkowski, Laurent Daynès: *Multitasking without Compromise: a Virtual Machine Evolution.* ACM OOPSLA'01, Tampa, FL.
http://research.sun.com/projects/barcelona/papers/oopsla01.pdf.

(9) 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/39*0. IBM Systems Journal, Volume 39, Number 1, 2000.

(10) *Echidna - A free multiprocess system in Java*.
http://www.javagroup.org/echidna/.

(11) P. Herrmann, U. Kebschull, W.G. Spruth: *Einführung in z/OS und OS/390*.
Oldenbourg-Verlag, 2. Auflage, 2004, ISBN 3-486-27393-0.

(12) *MVS Programming:Workload Management Services* .
IBM Form No GC28-1773-08, September 2000.

(13) *WebSphere Scalability: WLM and Clustering.*
IBM Form No. SG24-6153-00, September 2000.

(14) *New IBM Technology featuring Persistent Reusable Java Virtual Machines*. IBM Form No. SC34-6034-01, October 2001.

(15) *Enterprise JavaBeans for z/OS and OS/390, CICS Transaction Server V2.2.*
IBM Form No. SG24-6284-01, July 2002.

(16) *CICS Transaction Server Version 2.2 (tutorial).*
http://www-306.ibm.com/software/webservers/learn/cs04/.

(17) *Persistent Reusable Java Virtual Machine User 's Guide*.
IBM Form No. SC34-6201-01, Fifth Edition (September 2003).
www-1.ibm.com/servers/eserver/ zseries/software/java/pdf/prjvm14.pdf

(18) *Janos Virtual Machine.*
http://www.cs.utah.edu/flux/janos/janosvm.html.

(19) Java Community Process : *JSR 121 Application Isolation API Specification*.
http://www.jcp.org/en/jsr/detail?id=121.

(20) *JKernel and JServer Overview*.
http://www.cs.cornell.edu/slk/jk-0.91/doc/Default.html.

(21) M. Jordan, L. Daynès, G. Czajkowski, M. Jarzab, C. Bryce:
Scaling J2EE Application Servers with the Multi-Tasking Virtual Machine.
http://research.sun.com/techrep/2004/abstract-135.html.

(22) B. Klein: *IMS Integration Strategy for On Demand Service Oriented Architecture*.
http://www-306.ibm.com/software/data/ims/shelf/presentations.

(23) J. Lepreau: *The Janos Project*.
http://www.cs.utah.edu/flux/janos/index.html.

(24) H. Min, P. Wanish, K. Muckenhaupt: *A CICS to Linux Grid Implementation*.
IBM Redbooks Paper, 2003.
www-900.ibm.com/developerWorks/cn/grid/gr-redbook/redp3758.pdf

(25) H. Min, P. Wanish: *Linux Grid Implementation from DB2 on z/OS*.
IBM Redbooks Paper, 2004.
www.redbooks.ibm.com/redpapers/pdfs/redp3933.pdf, 2004.

(26) M. Prochazka: *Advanced Transactions in Enterprise JavaBeans*. In W. Emmerich and S. Tai (Eds.): EDO 2000, LNCS 1999, Springer-Verlag Berlin Heidelberg 2001, pp. 215-230.

(27) Oliver Raible: *Transaktionsverarbeitung in J2EE-Systemen.*
Diplomarbeit, Wilhelm-Schickard Institut für Informatik, Universität Tübingen, 2003.

(28) Bo Sandén: *Coping with Java Threads*. IEEE Computer, Vol. 37, Nr. 4, April 2004, p. 20.

(29) W. G. Spruth, J. Franz: *Reengineering von Kernanwendungssystemen auf Großrechnern*.
Informatik Spektrum, Band 26, April 2003, S. 83.

(30) *The Java HotSpot ™ Virtual Machine. v1.4.1*. Sun Technical White Paper, Sept. 2002.
http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html.

(31) P. Tullmann: *The Alta Operating Syystem.*
Master Thesis, Department of Computer Science, University of Utah, 1999.
http://www.cs.utah.edu/flux/papers/tullmann-thesis-base.html.

(32) T. von Eicken: *J-Kernel: a Capability-Based Operating System for Java.*
Department of Computer Science, Cornell University, 1999.
www.cs.cornell.edu/Info/People/chichao/sip99.pdf.

(33) Matthew Webster: *The IBM Persistent Reusable JVM*.
SHARE Technical Conference, Nashville, TN, March 3- 8, 2002.
http://www.share.org/proceedings/sh98/data/S8351.PDF.

Für die in digitaler Form verfügbaren Dokumente existiert ein Mirror unter
http://www-ti.informatik.uni-tuebingen.de/~spruth/publish.html.