Diplomarbeit im Fach Informatik

Modellierung eines Goal orientierten Workload-Managers

Vorgelegt von
Jingkai Chen

1. November 2006

Wilhelm-Schickard-Institut für Informatik
Universität Tübingen

Sand 13, 72076 Tübingen, Germany

Email:
jingkaichen@student.uni-tuebingen.de

Aufgabensteller Prof. Dr.-Ing.Wilhelm G. Spruth
Institut für Informatik, Universität Leipzig
Wilhelm-Schickard-Institut für Informatik, Eberhard-Karls-Universität Tübingen

Betreuer Peter Bäuerle
IBM Deutschland Entwicklung GmbH

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Tübingen, 30.Oktober 2006

Zusammenfassung

Der Goal orientierte Workload-Manager ist ein integraler Bestandteil des z/OS-Großrechner-Betriebssystems von IBM. In dieser Diplomarbeit wird ein Modell für den Workload-Manager beschrieben und implementiert. Mit Hilfe dieses Modells wird versucht, den Workload-Manager zu verstehen. Verschiedene Fragestellungen werden untersucht, darunter:

- Wie werden Zielerfüllungsprobleme automatisch erkannt?
- Wie werden die Ursachen für Zielerfüllungsprobleme ermittelt?
- Kann durch Änderung von Kontrollparametern eine Zielerfüllung erwartet werden?
- Welche Kontrollparameter werden wann und wie angepasst?

Das Modell ist in Java implementiert und exportiert die experimentellen Daten in Excel Sheets. Es besteht aus einem z/OS-Workload-Manager, einem Rechnersystem und einem Workloadgenerator. Mit dem Modell wurde ein besseres Verständnis der komplexen Zusammenhänge des Workload-Managers erreicht.

Eine Zusammenfassung der Ergebnisse ist in den Abschnitten 7.3 und 7.4 dargestellt.

Für meine Eltern,

die mir meinen Weg bis hierhin erst ermöglicht haben.

Danksagung

An dieser Stelle möchte ich allen danken, die mich während der Entstehung dieser Diplomarbeit auf vielfältige Art und Weise unterstützt haben.

Mein besonderer Dank gilt meine Frau Yuhui Wang, die mich während meines gesamten Studiums unterstützt hat.

Für die Ermöglichung dieser Diplomarbeit möchte ich mich bei Herrn Prof. Spruth und bei Herrn Prof. Rosenstiel (Lehrstuhl Technische Informatik) bedanken.

Der Firma IBM Deutschland Entwicklung GmbH, besonders Herrn Peter Bäuerle, der mit mir diskutiert und mir neue Blickwinkel auf die Problematiken eröffnet hat, danke ich für die gute Zusammenarbeit.

Für ihre Geduld und rasche Unterstützung bei dem Korrigieren von Rechtschreibfehlern und Sprache danke ich ganz herzlich Frau Sarah Kleeberg, Frau Yidi Zhang und besonders Herrn Jan Petranek.

Dankbar bin auch Herrn Clemens Gebhard für das angenehme Mitfahren zu IBM.

Ich bedanke mich für die freundliche und fachliche Unterstützung durch die IBM Deutschland Entwicklung GmbH.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung und Motivation

1.1 Motivation

Workload-Management ist heute die ausgefeilteste Methode, um Betriebsmittel und Workloads in einem Rechnersystem zu steuern. Es wird für alle Aspekte, die mit der Verteilung von Workload und der Zuweisung von Betriebsmitteln in Rechnern zu tun haben, angewendet. Workload-Manager als Betriebssystemkomponenten finden sich in vielen heutigen Betriebssystemen, vor allem in nahezu allen größeren kommerziellen UNIX-Systemen. Dies bedeutet aber nicht, dass sich deren Funktionalität ähnelt.

Als Workload-Manager (WLM) wird häufig ein Router eingesetzt, der die Anwendungen mehr oder weniger gleichmäßig auf die einzelnen Systeme verteilt, z.B. mit einem Round-Robin-Algorithmus. Bekannte Scheduling-Algorithmen sind u. a. Round-Robin, Prioritäts-Scheduling, Mehrere Warteschlangen, Shortest-Job-First und Garantierendes Scheduling (siehe [Gla06]). Keiner der bisher betrachteten Scheduler kann Eingabedaten von Benutzern (oder Benutzerprozessen) annehmen, um Scheduling-Entscheidungen zu treffen. Der Scheduler kann nicht die beste Entscheidung treffen. Die Bewältigung dieser Probleme erfordert einen automatischen, selbstoptimierenden Ansatz zur Performance-Kontrolle des Systems. Dabei wird die Verarbeitung systemseitig ständig überwacht und analysiert, so dass Problemsituationen unmittelbar erkannt werden können. Kontrollparameter des Systems sind automatisch einzustellen und in Abhängigkeit des aktuellen Systemzustands anzupassen, z.B. zur Behandlung von Leistungsproblemen. Eine gute, praktikable Lösung mit dem Namen *Goal orientierter Workload-Manager (WLM)* wurde von IBM entwickelt.

Der Goal orientierte Workload-Manager ist ein integraler Bestandteil des z/OS-

Großrechner-Betriebssystems von IBM. Die Idee des Goal orientierten WLMs ist es, einen Kontrakt zwischen Endbenutzer und Rechnersysten zu schließen und das System entsprechend den im Kontrakt vereinbarten Bedingungen zu steuern ([TV04]). Dieser Workload-Manager unterstützt die Automatisierung eines Workload-Managementprozesses in Übereinstimmung mit seinen Zielvorgaben. Die Realisierung des Goal orientierten Workload-Managers ist komplex. Es müssen Fragestellungen betrachtet werden wie z.B.:

- Wie werden Zielerfüllungsprobleme automatisch erkannt?
- Wie werden die Ursachen für Zielerfüllungsprobleme ermittelt?
- Kann durch Änderung von Kontrollparametern eine Zielerfüllung erwartet werden?
- Welche Kontrollparameter werden wann und wie angepasst?

Daneben muss der Workload-Manager gewährleisten, dass der automatische Kontrollansatz auch in Überlastsituationen stabil arbeitet und nur vergleichsweise geringen Overhead verursacht.

In dieser Arbeit wird die Funktionalität des Workload-Managers modelliert und eine Simulationsumgebung entwickelt. Das Modell stellt eine Grobarchitektur dar und simuliert die zentrale Funktionalität des Workload-Managers. Das Ziel dieser Arbeit ist, anhand dieses einfacheren Modells den Workload-Manager selbst zu erklären, sowie zu analysieren, welche wichtigen Verfahren für die Steuerung des Systems verwendet werden und wie er arbeitet.

1.2 Aufbau der Arbeit

Diese Arbeit ist in sieben Kapitel untergliedert.
Kapitel 1 ist eine Einleitung und beschreibt das Ziel der Diplomarbeit.
Kapitel 2 erläutert die Grundlagen des z/OS Workload-Managers.
Kapitel 3 beschäftigt sich mit dem gefundenen Lösungskonzept zur Modellierung des WLMs.
Kapitel 4 stellt die Modellierung von Workload und Service-Klassen vor.
Kapitel 5 dokumentiert den Aufbau des Rechnersystems zur Modellierung der Betriebsmittel.
Kapitel 6 beschreibt den zentralen Algorithmus des WLMs.

Kapitel 7 schließt mit den Experimentergebnissen ab.
Kapitel 8 enthält eine Zusammenfassung und einen Ausblick.

Kapitel 2

Grundlagen

Workload-Management befasst sich mit Workloads und deren Zugang zu Betriebsmitteln. Es regelt das Zusammenspiel unterschiedlichster Aufgaben im System.

Einfache Workload-Manager (WLM) existieren für die Windows-, Linux- und Solaris-Plattformen. Eine vergleichbare Funktionalität wie der z/OS Goal orientierte Workload-Manager ist auf anderen Plattformen nicht verfügbar. Zum besseren Verständnis ist es wichtig, die Grundkonzepte des z/OS Workload-Managers zu diskutieren.

Der Goal orientierte Workload-Manager wurde 1995 mit Multiple Virtual Storage (MVS) 5.1 (vgl. [bib05a]) eingeführt. Er ist darüber hinaus die einzige Komponente, die umfangreiche adaptive Algorithmen einsetzt, um Rechner zu steuern, so dass Zielvorgaben des Anwenders erfüllt werden. Dabei wird vollständig auf den Einsatz von Daumenregeln und fest einzustellenden Parametern verzichtet.

2.1 Der *System Resource Manager*

Der Goal orientierte Workload-Manager ist nicht im leeren Raum entstanden. Vor der Goal orientierten Steuerung der Systeme gab es bereits seit den 70er Jahren den System Resource Manager (SRM) als festen Bestandteil des Betriebssystems. Seine Funktionen wurden nicht unmittelbar durch den Workload-Manager ersetzt, sondern existieren in allen Versionen bis z/OS 1.2 weiter und haben den Namen *Compatibility Mode*. Die Goal orientierte Steuerung der Systeme wird im Gegensatz dazu als *Goal Mode* bezeichnet. Viele Workload-Manager-Konzepte und -Funktionen basieren auf SRM, der auch im *Goal Mode* ein Teil des Workload-Managers ist.

Die Hauptaufgabe des SRMs liegt in der Verteilung von Betriebsmitteln auf

die anfallende Arbeit in einem MVS System. Als Grundlage dazu dienen vom Anwender erstellte Definitionen, die detailliert festlegen, wie der Zugang und der Verbrauch der Betriebsmittel zwischen den Workloads aufgeteilt. Es soll auf Schwierigkeiten hingewiesen werden, die bei der Entwicklung von Rechnersystemen über einen langen Zeitraum entstehen. Um den Anforderungen in unterschiedlichsten Situationen gerecht zu werden, werden zum einen die Algorithmen immer weiter verfeinert. Zum anderen werden dem Benutzer aber auch immer umfangreichere Kontrollmöglichkeit an die Hand gegeben. Betrachtet man alle Variationen, die im *Compatibility Mode* existieren, so kann die Installation dem System die Freiheit lassen, sich weitestgehend selbst zu verwalten, oder sie kann bis in das unterste Detail jeden Entscheidungsschritt des Systems festlegen. Insbesondere die letzte Art der Begrenzung des Systems bedingt, dass eine ständige Überwachung und Anpassung der System-Parameter notwendig ist.

Ein großer Nachteil des *Compatibility Mode* war und ist es, dass dies eine sehr anspruchsvolle Aufgabe ist und dass dafür sehr detaillierte Kenntnisse des Systems benötigt werden. Der zweite Nachteil ist, dass die Funktionalität des SRMs unflexibel war, da sie sich nur bedingt Änderungen im Systemverhalten automatisch anpassen konnte.

Um diesen Schwächen, der wachsenden Komplexität, dem begrenzten Erkennen von Abläufen der Anwendungen und dem Wartungssaufwand für die Einstellungen entgegen zu wirken, wurde mit MVS 5.1 Mitte der 90er Jahre der *Goal Mode* bzw. Goal orientierte Workload-Manager eingeführt ([TV04]).

2.2 Der *Goal Mode*

Die Idee des *Goal Modes* ist es, einen Kontrakt zwischen Endbenutzer und Rechnersystem zu schließen und das System entsprechend den im Kontrakt vereinbarten Bedingungen zu steuern. Der Benutzer definiert mittels Zielvorgaben, welche Leistungen unter welchen Umständen von dem Rechnersystem zu erbringen sind. Das System überwacht während der Abarbeitung die Workloads, vergleicht ihre Laufzeitergebnisse mit den Zielvorgaben und passt den Zugang zu und Verbrauch an Betriebsmitteln dynamisch auf der Basis der Zielerfüllung an.

In den folgenden Unterkapiteln wollen wir die Konzepte des *Goal Mode* detaillierter untersuchen und uns einen Überblick über die Mechanismen verschaffen, die es einem Rechnersystem erlauben, dynamisch Workloads und Betriebsmittel zu verwalten.

2.2.1 Service-Klassen und Klassifizierung

Um die Workload mit Zielen zu versehen, müssen diese vorher in Service-Klassen eingeteilt werden. Die Service-Klassen sind Einheiten von Workload mit sehr ähnlichen Charakteristiken, für die die Ziele definiert werden. Die Service-Klasse stellt das Grundkonstrukt im *Goal Mode* dar. Sie ist das Ergebnis der Klassifizierung der Workloads mit unterschiedlichen Leistungsmerkmalen in Gruppen, die mit Zielvorgaben versehen werden können.

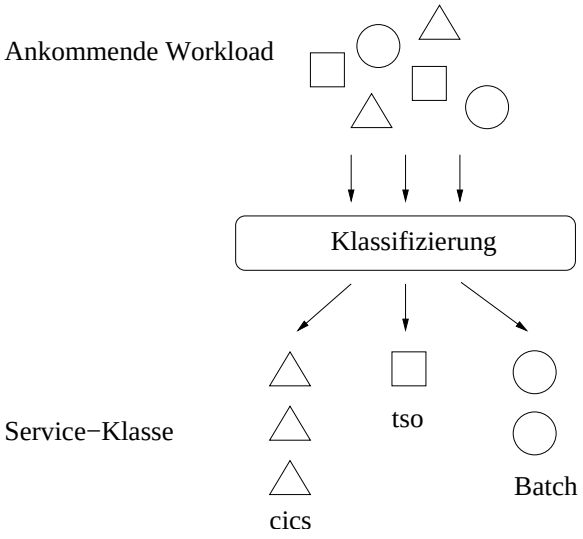

Abbildung 2.1: Klassifizierung von Arbeit

Die Klassifikation basiert auf den Attributen einer individuellen Arbeitsanforderung. Letztere kann die UserID, das aufgerufene Transaktions-Programm, die Arbeits-Umgebung oder das Subsystem, an das die Anforderung gerichtet ist usw. einschließen (siehe [HKS04]).

Der Workload-Manager unterstützt beispielsweise folgende Subsysteme :

- IMS
- JES2/JES3
- TSO/E
- CICS
- OMVS

- DB2
- DDF

Weiterführende Informationen zu Subsystemen findet man in [EOO05].

Eine Workload-Manager Service Definition wird mittels einer speziellen administrativen Anwendung, der *WLM Administrative Application*, erstellt und abgeändert. Sie kann unter *Interactive System Productivity Facility* (ISPF siehe [EOO04]) von TSO-Benutzern gestartet werden.

Der wichtigste Punkt bei der Definition einer Service-Klasse ist es, festzulegen, wie wichtig sie ist, um die übergeordneten Geschäftsziele zu erreichen. Die Wichtigkeit wird dargestellt, indem eine *Business Importance* für die Service-Klasse definiert wird. Diese *Business Importance* ist später für das System der wesentliche Entscheidungsfaktor, wenn der Zugang zu den Betriebsmitteln geregelt werden muss.

Eine Service-Klasse enthält eine Folge von Perioden mit einem installationsabhängigen Wert, der angibt, wie lange eine Arbeitsanforderung zu einer Periode gehört. Jede Arbeitsanforderung beginnt in der Periode 1 und wird solange entsprechend dem Ziel in der Periode 1 behandelt, bis genügend Service in Anspruch genommen und die Periodendauer überschritten wurde. Danach beginnt für die Arbeitsanforderung die Periode 2, in der sie entsprechend dem Ziel der Periode 2 abgearbeitet wird usw. Jede Periode hat ein mit ihr verbundenes Ziel und eine spezielle Bedeutung. Die Ziele für verschiedene Service-Klassen können verschieden sein [HKS04].

2.3 Arten von Zielen

Für die Definition von Zielvorgaben bietet der z/OS-Workload-Manager drei Arten von Zielen [TV04]:

- *Response Time Goals*
- *Execution Velocity Goals*
- *Discretionary*

2.3.1 *Response Time Goals*

Unter einem *Response Time Goal* versteht man die Zeitspanne zwischen Start und Fertigstellung einer Workload, die in Sekunden pro Programm gemessen wird. Als

Zeit wird die vollständige Verweildauer inklusive der Zeit, in der die Workload nicht arbeitet, betrachtet, dadurch kann man die Wartezeit eines Benutzers erkennen.

Es gibt zwei Möglichkeiten, *Response Time Goals* zu definieren: *Average Response Time Goals* und *Percentile Response Time Goals*.

Average Response Time Goals

Ein *Average Response Time Goal* beschreibt die durchschnittliche Ausführungsdauer aller Workloads einer Service-Klasse im System. Die Zieldefinition besteht aus einer einfachen Zeitangabe. Zum Beispiel kann man für eine Service-Klasse eine durchschnittliche Ausführungsdauer aller Workloads von kleiner als eine Sekunde definieren.

Percentile Response Time Goals

Man definiert den Prozentsatz aller Workloads, die innerhalb eines vorgegebenen Zeitwertes enden sollen. Die Zieldefinition besteht aus zwei Teilen, dem Zeitwert und dem Prozentwert der Workloads, die innerhalb des Zeitwertes enden sollen, zum Beispiel: 95% aller Workloads sollen innerhalb von 0,6 Sekunden enden oder 95% < 0,6 Sekunden.

2.3.2 *Execution Velocity Goals*

Wenn zu wenige Workloads in einer Service-Klasse enden, kann ein *Response Time Goal* nicht verwendet werden. Um für derartige Workloads Ziele definieren zu können, sollte ein *Execution Velocity Goal* verwendet werden. Bei einem *Execution Velocity Goal* werden nur die Zustände betrachtet, bei denen eine Workload Betriebsmittel benutzt (*using*) oder darauf wartet (*delay*).

Zuvor sollte erläutert werden, wie der Workload-Manager feststellt, welche Betriebsmittel eine Workload benutzt und auf welche sie wartet. Der Workload-Manager sammelt Daten über den aktuellen Zustand aller verwalteten Betriebsmittel. Dies sind Information über die Prozessoren, den Speicher und die Benutzung der Platteneinheit. Zeiträume, in denen die Workload nicht gearbeitet hat und auch keine Anforderungen an das System gestellt hat, werden nicht berücksichtigt. Dasselbe gilt für Zustände, die vom Workload-Manager nicht erfasst werden können, wie zum Beispiel das Warten auf die Freigabe eines Locks, das durch eine Anwendung verwaltet wird. Aus den gesammelten Daten wird für jede Workload fest-

gestellt, ob sie Betriebsmittel benutzt oder darauf wartet. Der Workload-Manager verwendet diese Daten, um den Zugang der Service-Klassen zu den Betriebsmitteln zu regeln.

Execution Velocity ist wie folgt definiert [bib05b]:

$$Execution\ Velocity \quad = \quad \frac{Total\ Usings}{Total\ Usings + Total\ Delays} \times 100 \qquad (2.1)$$

Wie man aus der Formel (2.1) erkennt, legt die Zieldefinition den Anteil der akzeptablen Wartezeit bei der Ausführung von Programmen fest. *Execution Velocity* lässt sich einfach durch ein Beispiel veranschaulichen. Man nehme an, dass eine Workload fünf Mal auf Betriebsmittel wartet und fünf Mal Betriebsmittel benutzt, dann erhält man eine *Execution Velocity* von 50.

$$\begin{aligned} Execution\ Velocity \quad &= \quad \frac{5\ Usings}{5\ Usings + 5\ Delays} \times 100 \\ &= \quad 50 \end{aligned}$$

Der Wert der Execution Velocity liegt immer zwischen 1 und 100.

2.3.3 *Discretionary*

In z/OS-Systemen gibt es eine Gruppe von Workloads, für die keine bestimmte Zielvorgabe existiert. Diese Workloads erhalten nur dann Betriebsmittel, wenn diese ausreichend vorhanden sind, und sie sind die Ersten, die keinen Zugang mehr erhalten, wenn es eng im System wird. Derartige Gruppen von Workload werden als *Discretionary* bezeichnet.

Um sicherzustellen, dass eine mit einem *Discretionary*-Ziel versehene Workload Zugang zu den Betriebsmitteln erhält, wenn Betriebsmittel zur Verfügung stehen, tritt der Workload Management Regelmechanismus in Kraft, der den Betriebsmittelzugang für Service-Klassen mit Zielvorgaben begrenzt, wenn diese ihre Ziele deutlich übererfüllen.

2.4 *Performance Index*

Es ist eine zentrale Frage, wie Service-Klasse ihre Ziele erfüllen und wie sie sich im Vergleich zu anderen Service-Klassen verhalten. Durch eine Definition eines *Performance Indexes* lösen die Workload-Manager-Algorithmen dieses Problem. Der *Performance Index* (PI) ist wie folgt definiert:

Execution Velocity Goal:

$$PI = \frac{Execution\,Velocity\,Goal}{Aktuell\,erreichte\,Exec\,Velocity} \tag{2.2}$$

Response Time Goal:

$$PI = \frac{Aktuelle\,Response\,Time}{Response\,Time\,Goal} \tag{2.3}$$

Wie man aus den Formeln erkennt, kann der *Performance Index* für *Average Response Time Goals* und *Execution Velocity Goals* aus den Zieldefinitionen und den aktuell gemessenen Werten ermittelt werden. Für *Percentile Response Time Goals* wird als aktuelle Zeit der Wert des Abschnittes verwendet, bis zu dem die vorgegebene Menge an Workloads endet, um die Zielvorgabe zu erfüllen.

Durch den *Performance Index* kann man leicht darstellen, wie Service-Klassen ihre Ziele erfüllen. Ein Wert von größer als 1 bedeutet, dass das Ziel nicht erfüllt wird und dass die Workload-Manager-Algorithmen aktiv werden müssen, um den Engpass zu beheben. Ein Wert von 1 signalisiert Zielerfüllung und ein Wert kleiner als 1, dass das Ziel übererfüllt ist.

Kapitel 3

Lösungskonzept für die Modellierung des WLMs

3.1 Problematik der Modellierung

Die Zielsetzung, den z/OS Goal orientierten Workload-Manager zu modellieren, soll durch ein Programm erreicht werden, das es zu entwickeln und implementieren gilt. Dieses Modell soll nicht alle Funktionalitäten des originalen z/OS-WLMs erfassen, sondern nur diejenigen, die für ein besseres Verständnis der komplexen Zusammenhänge des WLMs relevant erscheinen.

Die erste Frage, die in diesem Zusammenhang gestellt werden muss, ist,

- Welche Funktionalität des z/OS WLMs modelliert werden soll?

In diesem Modell werden die grundlegenden Algorithmen des z/OS WLMs erfasst. Der WLM im Sysplex ist nicht berücksichtigt, d.h. die Entscheidung des WLMs muss unabhängig davon getroffen werden, ob das z/OS-System Teil einer Sysplex-Umgebung ist.

Der WLM kann nicht allein arbeiten, da er eine Betriebssystemkomponente von z/OS ist. Um den z/OS WLM zu simulieren, muss eine Laufzeitumgebung bzw. ein Rechnersystem für den WLM modelliert werden.

Die zweite Frage ist,

- Welche Hardwarekomponenten und Betriebsmittel im Rechnersystem modelliert werden sollen?

Der Modellrechner soll die typischen Hardware-Komponenten enthalten, die in Großrechnern auftreten. Die klassischen Hardware-Komponenten eines z900-Rechners

sind die CPUs, der Speicher und die Platten des Ein-/Ausgabe-Subsystems. Der Modellrechner soll teilweise Betriebssystemfunktionen unterstützen, z.B. Prozessverwaltung und Speicherverwaltung.

Neben dem grundlegenden WLM-Algorithmus und dem Modellrechner müssen die Workloadverteilung und die Definitionen der Service-Klassen modelliert werden. Im nächsten Unterkapitel beschreiben wir eine Grobstruktur für das Modell des z/OS Goal orientierten Workload-Managers. Auf die Implementierung wird im nächsten Kapitel eingegangen.

3.2 Grobstruktur des WLM-Modells

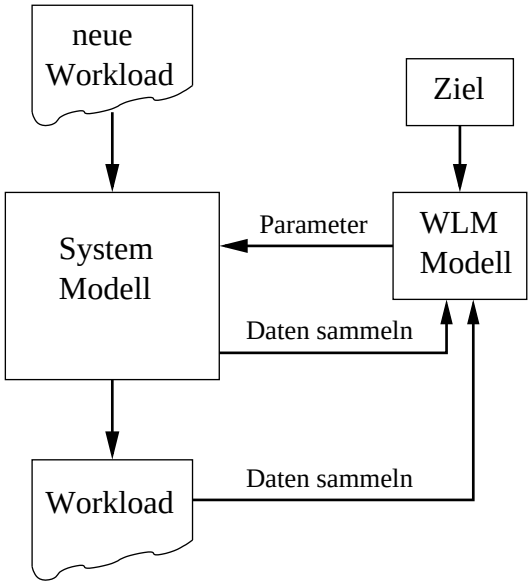

Abbildung 3.1: Grobstruktur des WLM-Modells

Die Abbildung 3.1 zeigt der Grobstruktur des Modells. Das Modell besteht aus vier Komponenten,

- *Workloads*, die zu verschiedenen Service-Klasse gehören, werden vom Workloadgenerator erzeugt und laufen im System ab. Sie benutzen Betriebsmittel.
- *Ziele*, die vom Benutzer definiert sind.
- Das *Rechnersystem* enthält die wichtigsten Betriebsmittel und simuliert einen primitiven Rechner, auf dem Workloads ablaufen können. Es verwaltet Be-

triebsmittel wie CPU, Hauptspeicher etc. durch entsprechende Resource-Manager, z.B. CPU-Dispatcher und Speicherverwaltung.

- Der *WLM* überwacht das System und sammelt zu bestimmten Zeiten die aktuellen Daten im System. Wenn sich dabei herausstellt, dass Ziele nicht erfüllt werden, steuert der WLM durch Parameter das System entsprechend den Zielen. Die wichtigsten Parameter sind Dispatching-Priorität und Multi Programming Level (MPL).

3.3 Die Komponenten des Modell-Rechnersystems

In diesem Abschnitt gehen wir etwas genauer auf das Rechnersystem ein. Wie in der Abbildung 3.2 gezeigt, haben wir zum Simulieren der Umgebung für den WLM ein Rechnersystem modelliert. Im Rechnersystem gibt es vor allem Prozessoren (CPUs). Diese CPUs können als Uniprozessoren konfiguriert werden oder eine beliebige Anzahl an Prozessoren aufweisen, da die S/390-Rechner Multiprozessor-Systeme sind. Der Zugang zu den CPUs wird durch die Verwendung einer Dispatch-

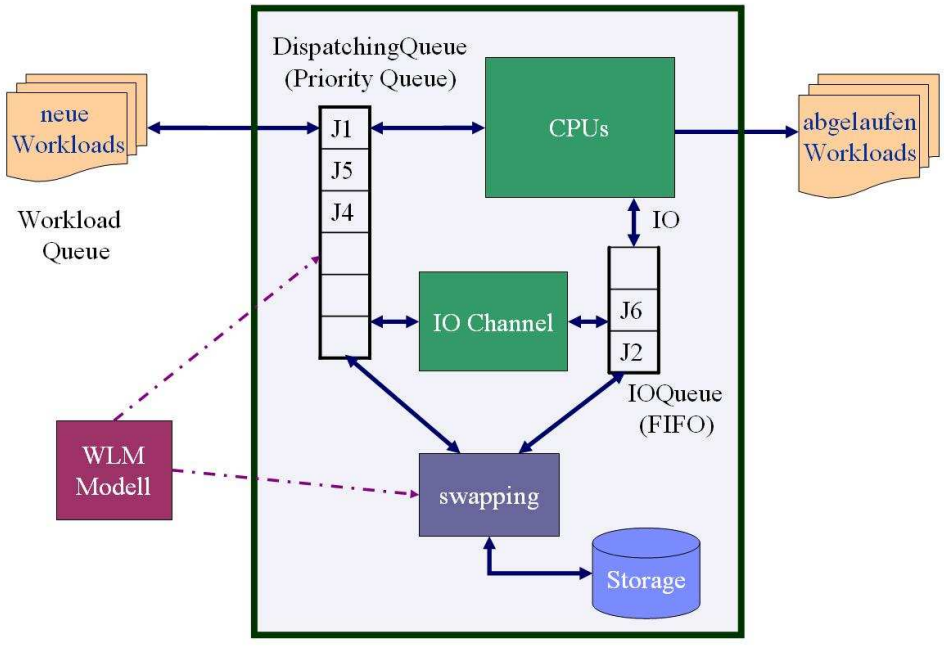

Abbildung 3.2: Die Komponenten des Rechnersystems

Queue gesteuert. Jeder Prozess, d.h. jede im Hauptspeicher befindliche Workload, erhält eine Dispatch-Priorität und ist innerhalb dieser Dispatch-Queue nach seiner

Priorität geordnet. Der Prozessscheduler ordnet immer den ersten Prozess in der Dispatch-Queue mit der höchsten Priorität einer verfügbaren CPU zu.

In der Abbildung 3.2 gibt es neben Prozessoren noch einen Ein-/Ausgabe-Kanal (*I/O-Channel*). In z/OS ist dieser Betriebsmittelzugang tatsächlich durch die Festlegung von Prioritäten für den Ein-/Ausgabe-Kanal geregelt. Um das in der Simulation verwendete Szenario möglichst einfach und anschaulich zu halten, wurde eine FIFO-Queue (*First In First Out*) als Ein/Ausgabe-Queue (*I/O-Queue*) verwendet. Hierbei werden alle Prozesse in der Reihenfolge ihres Eingangs bearbeitet.

Für die Speicherverwaltung haben wir ein Swapping modelliert. Wir nehmen an, dass sich der Prozess, der in der Dispatching-Queue oder Ein/Ausgabe-Queue liegt, im Hauptspeicher befindet. Wenn die Workload ungesteuert abläuft, kann es zu Engpässen bei den Betriebsmitteln kommen. Um das zu vermeiden, wird für jede Service-Klasse festgelegt, wie viele Adressräume gleichzeitig im System sein dürfen und wie viele auf jeden Fall im System bleiben sollen. Die Zahl der Adressräume, die gleichzeitige im System sind, wird in der Literatur als *Multiprogramming Level* (MPL) bezeichnet. Der MPL ist Teil eines wesentlichen Steuerungsmechanismus des Systems. Die Unter- und Obergrenze für den MPL darf nur durch den WLM dynamisch geändert werden. Wenn das System feststellt, dass Engpässe auftreten, kann der Workload-Manager den MPL der Service-Klasse reduzieren, damit Adressräume auslagert werden und die verbleibenden Adressräume effizienter arbeiten können.

3.4 Prozesszustände und Übergänge

Bisher haben wir die Struktur des WLM-Modells und der Betriebsmittel besprochen. Die folgende Liste enthält die 7 Zustände der im WLM-Modell ablaufenden Workloads (Prozesszustände):

1. **inaktiv**: Der Prozess befindet sich in einem Übergangszustand. Er ist zwar vorhanden, ist aber weder lauffähig noch im Schlafzustand.

2. **Ablauf**: Der Prozess ist dem Prozessor zugeteilt. Der Prozess kann solange ablaufen, bis eine Unterbrechung eintritt. Dabei geht man davon aus, dass die Prozesse nicht dauernd Rechenleistung benötigen, da sie zwischendurch auf Ereignisse irgendwelcher Art warten müssen.

3. **Schlaf im Speicher**: Ein Prozess befindet sich in der Ein/Ausgabe-Queue, er kann aber nicht ausgeführt werden kann, solange nicht ein bestimmtes Er-

eignis eintritt. Das Ereignis ist in diesem WLM-Modell nur die Beendigung einer Ein/Ausgabe-Operation.

4. **Schlaf, ausgelagert**: Der Prozess schläft und der Swapper hat ihn auf den Sekundärspeicher ausgelagert, um sicherzustellen, dass genügend Betriebsmittel für andere Prozesse zur Verfügung stehen.

5. **Ausführbar im Speicher**: Der Prozess befindet sich in der Dispatching-Queue. Er ist nicht in Bearbeitung, ist aber lauffähig, und wartet daruf, dass ein Prozessor ihm zugeteilt wird.

6. **Ausführbar, ausgelagert**: Der Prozess ist nicht im Hauptspeicher. Er ist aber lauffähig, doch muss er erst vom Swapper in die Dispatching-Queue geladen werden, bevor ihm ein Prozessor zum Ablauf zugeteilt werden kann.

7. **komplett**: Der Prozess ist beendet.

In Abbildung 3.3 werden die Ereignisarten aufgeführt, die für einen Prozess zu einer Zustandsänderung führen. Nicht jeder Prozess muss alle Zustände durchlaufen. Die folgenden Zustandsänderungen in diesem WLM-Modell sind möglich:

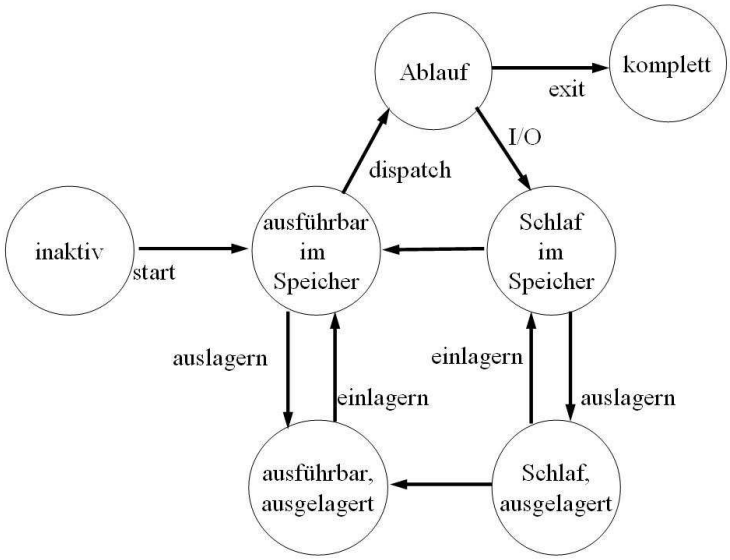

Abbildung 3.3: Diagramm der Prozesszustände und möglicher Übergänge

- **inaktiv $\longrightarrow$ ausführbar im Speicher**
  Wenn ein Prozess gestartet wird, beginnt er seinen Weg. Er wird sofort in

die Dispatch-Queue eingeordnet d.h. er wechselt von Status “inaktiv” in den Zustand “ausführbar im Speicher”.

- **ausführbar im Speicher $\longrightarrow$ Ablauf**
  Wenn der Prozess die höchste Priorität hat, wird er einer verfügbaren CPU zugeordnet, bzw. der Prozess wechselt in den Zustand “Ablauf”. Es dürfen mehrere Prozesse in dem Zustand “Ablauf” sein. Da das Rechnermodell ein Multiprozessorsystem ist, erlaubt es, mehrere Prozesse gleichzeitig auf dem Rechnersystem auszuführen.

- **Ablauf $\longrightarrow$ Schlaf im Speicher**
  Der Prozess wechselt von dem Zustand “Ablauf” in “Schlaf im Speicher”, wenn der Prozess anzeigt, dass er eine Ein/Ausgabe-Unterbrechung hat. Er wird hinten an die Ein/Ausgabe-Queue angefügt und versetzt sich selbst bis zur Beendigung der Ein/Ausgabe-Operation in Schlaf.

- **Schlaf im Speicher $\longrightarrow$ Schlaf, ausgelagert**
  Möglicherweise wählt der Swapper den Prozess zum Auslagern aus. Der Prozess wechselt in den Zustand “Schlaf, ausgelagert” und er befindet sich dann im Sekundärspeicher. Der Swapper kann das Auslagern des Prozesses verursachen, falls der WLM die Obergrenze für den MPL reduziert.

- **Schlaf, ausgelagert $\longrightarrow$ Schlaf im Speicher**
  Der Prozess, der im Sekundärspeicher liegt, kann wieder in die Ein/Ausgabe-Queue eingelagert werden, wenn die gesamte aktuelle Anzahl an Adressräumen der Prozesse unter der Obergrenze des MPL liegt. Die Erhöhung der Obergrenze des MPL oder die Beendigung eines anderen Prozesses kann zum Einlagern des ausgelagerten Prozesses führen.

- **Schlaf im Speicher $\longrightarrow$ Ausführbar im Speicher**
  Falls der Prozess in dem Zustand “Schlaf im Speicher” ist, könnte der Prozess nach Beendigung der Ein/Ausgabe-Operation in den Zustand “Ausführbar im Speicher” wechseln. Er befindet sich in der Dispatch-Queue und wird nach seiner Priorität eingeordnet.

- **Schlaf, ausgelagert $\longrightarrow$ Ausführbar, ausgelagert**
  Der Prozess, der sich im Zustand “Schlaf, ausgelagert” befindet, wechselt in den Zustand “Ausführbar, ausgelagert”, wenn die Ein/Ausgabe-Operation beendet ist.

- **Ausführbar im Speicher $\longrightarrow$ Ausführbar, ausgelagert**
  Wenn der Prozess von der Dispatch-Queue in den Sekundärspeicher ausgelagert wird, wechselt er in den Zustand “Ausführbar, ausgelagert”. Er ist aber lauffähig. Der Prozess kann wieder eingelagert werden, wenn die Anzahl der Adressräume der Prozesse kleiner als der maximale MPL ist.

- **Ausführbar, ausgelagert $\longrightarrow$ Ausführbar im Speicher**
  Der Prozess wird wieder in die Dispatch-Queue eingefügt, wenn der MPL sich verringert. Er wechselt in den Zustand “Ausführbar im Speicher”.

- **Ausführbar im Speicher $\longrightarrow$ Ablauf**
  Wenn der Prozess aus der Dispatch-Queue zur Abarbeitung ausgewählt wird, erhält er den Zustand “Ablauf” und wird weiter bearbeitet.

- **Ablauf $\longrightarrow$ komplett**
  Wenn der Prozess seine Aufgabe beendet hat, erreicht den Zustand “komplett”.

In diesem Abschnitt haben wir die Übergänge zwischen den Prozesszuständen auf logischer Ebene beschrieben. Die detaillierten Eigenschaften eines Zustandes werden nicht berücksichtigt, dazu gehören insbesonderes virtuelle Adressräume bzw. Paging.

3.5 Implementierungssprache

Das Modell des Workload-Managers wird mit der Programmiersprache Java implementiert. Java ist sowohl eine objektorientierte Programmiersprache in der Tradition von Smalltalk als auch eine klassische imperative Programmiersprache nach dem Vorbild von C.

Bei einer Java-Implementation sind Zielmaschine und Betriebssystem unabhängig voneinander. Die Java-Klassenbibliothek bietet mit einer ganzen Reihe nützlicher Klassen und Interfaces die Möglichkeit, sehr problemnah zu programmieren.

Die Java Implementierung des WLM-Modells besteht aus 20 Java-Klassen. Alle in dieser Arbeit verwendeten Java-Klassen befinden sich im Paket *wlm*. Eine Einführung in Java findet man unter [BF05].

Kapitel 4

Modellierung von Workloads und Service-Klassen

Kapitel 3 stellt die Struktur des WLM-Modells auf höherer Ebene dar. Dieses Kapitel behandelt die Implementierung des vorliegenden WLM-Modells und zeigt, wie dessen Komponenten modelliert werden können. Im ersten Abschnitt wird die Modellierung der Workloads vorgestellt; der nächste Abschnitt beschreibt die Implementierung der Service-Klassen. Eine für Service-Klassen wichtige Listenstruktur wird im darauf folgenden Abschnitt vorgestellt. Der letzte Abschnitt beschreibt die Erzeugung der Workloads der Service-Klassen durch einen Workloadgenerator.

4.1 Modellierung der Workloads

In S/390-Rechnern läuft eine größere Anzahl von verschiedenen Workloads parallel ab, z.B. *APPC* für *Advanced Programm-To-Programm Communication*, *TSO* für *Time Sharing Option*, *JES2*- und *JES3*-Batch-Jobs. Wir bezeichen als Arbeitseinheit die Komponenten, aus den die Workload für eine speiziale einzelne Service-Klasse besteht. Z.B. ist einzelne Transaktion die Arbeitseinheit der Service-Klasse CICS. Die Arbeitseinheiten sind sehr unterschiedlich. Eine Frage ist, festzustellen, wie Arbeitseinheiten modelliert werden können. Eine Modellierung kann von Eigenschaften der Arbeitseinheit ausgehen.

Vom Betriebssystemaspekt her gesehen, ist eine Arbeitseinheit das durch den Prozessor auszuführende Programm. Das Programm besteht aus einer Folge von Befehlen. Der Prozessor liest einen einzelnen Befehl aus dem Speicher (Befehlsabruf) und führt ihn danach aus. Die Programmausführung besteht aus dem wieder-

holten Einlesen und Ausführen von Befehlen. Ein Programm in der Ausführung wird als ein Prozess bezeichnet. Ein Prozess benötigt nicht dauernd Rechenleistung, da er zwischendurch auf Ereignisse irgendwelcher Art warten muss. [Tan94]. Dies können Ereignisse, wie z.B. Signale oder Meldungen von anderen Prozessen, oder auch Peripheriezugriffe mit Wartezeiten (Ein-/Ausgabe auf Platten, Netzwerkschnittstellen usw.) sein. Deshalb geht die Modellierung der Service-Klassen von zwei fundamentalen Parametern aus. Dies sind “CPU-Verbrauch” und “Unterbrechungsrate”. In der Abbildung 4.1 werden wichtige Folgerungen für die Parallel-

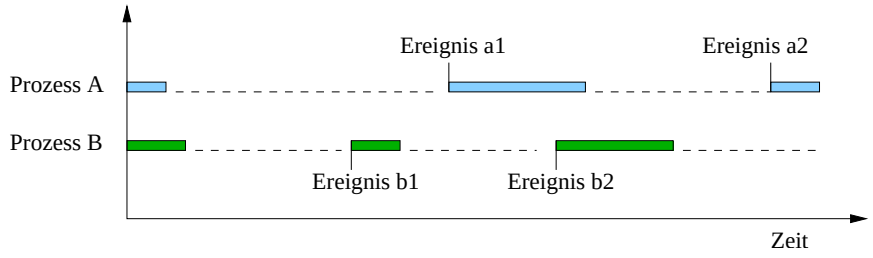

Abbildung 4.1: Zeitlicher Ablauf in einem Multiprozessorsystem

verarbeitung in einem Multiprozessorsystem dargestellt.

4.1.1 Implementierung von Java-Klassen

Der erste Schritt für die Modellierung der Workload ist, dass man zuerst die Arbeitseinheiten als logisches Konstrukt festlegt. Um die Arbeitseinheit mit Java zu implementieren, wird eine Java-Klasse *Job.java* definiert. Jede Instanz dieser Java-Klasse repräsentiert eine Arbeitseinheit im System. Die Java-Klasse *Job* enthält die folgenden wichtigen Membervariablen:

- *jobID* ist ein Schlüssel, welcher der eindeutigen Identifikation der Arbeitseinheit dient. Eine *jobID* wird hauptsächlich für das Scheduling benötigt, sie wird gesetzt, wenn die Arbeitseinheit erzeugt wird.
- *serviceClassID* ist ein Schlüssel, um die Arbeitseinheit einer Service-Klasse zuzuordnen.
- *incomingTime* enthält den Zeitpunkt, zu dem die Arbeitseinheit gestartet wird.
- *cpuDemand* gibt den gesamten CPU-Verbrauch der Arbeitseinheit an. Der CPU-Verbrauch wird in *Service Units* gemessen. Der Begriff *Service Units* wird in Abschnitt 5.1 diskutiert.

- *ioTimes* macht eine Annahme darüber, wie viel Wartezeit auf Ereignisse irgendwelcher Art während der Abarbeitung der Arbeitseinheit auftreten wird. Diese Daten werden für das Workloadscheduling benötigt.
- *ioInterval* enthält den CPU-Verbrauch zwischen zwei Unterbrechungen.
- *pCPUTimer* zeigt den restlichen CPU-Verbrauch an.
- *pIOIntvTimer* zeigt die restliche Zeit bis zur nächste Unterbrechung an.
- In *usingSum* und *delaySum* wird aufsummiert, wie oft die Arbeitseinheit Betriebsmittel benutzt und wie oft sie auf Betriebsmittel wartet. Beide werden nur für Service-Klassen mit einem Velocity-Ziel benötigt.
- *state* enthält den Arbeitseinheitausführungszustand. Auf den Ausführungszustand wurde in Abschnitt 3.4 eingegangen.
- *beginTime* zeigt den Startzeitpunkt an.
- *endTime* zeigt die Endzeit des Zustands an.
- *cpuDelay* enthält die Wartezeit auf CPUs.
- *swapInDelay* enthält die Zeit, die die Arbeitseinheit insgesamt auf das Einlagern in den Hauptspeicher wartet.

Deklarierung des *Job*-Objekts

Ein *Job*-Objekt kann wie folgt erzeugt werden:

```
Public Job( ServiceClass sc , int incTime, int cpuDemand, int ioTime)
```

Der Konstruktor *Job* erwartet vier Parameter. Der erste ist ein *ServiceClass*-Objekt (Abschnitt 4.2), zu der die Arbeitseinheit gehört. Der zweite Parameter gibt den Zeitpunkt, zu dem die Arbeitseinheit gestartet werden soll, an. Der dritte und vierte Parameter sind der CPU-Verbrauch und die Ein- /Ausgabe-Anforderung der Arbeitseinheit.

Vergleichen und Sortieren

Jede Arbeitseinheit hat eine eigene Startzeit. Die Java-Klasse *Job* bietet die Möglichkeit, die *Job*-Objekte in der Reihenfolge ihrer Startzeit zu sortieren oder zwei

Job-Objekte miteinander zu vergleichen. Zum Vergleich zweier *Job*-Objekten implementiert *Job* die besondere Methode *equals*. Jede Java-Klasse besitzt die Methode *equals*, da die universelle Oberklasse *Object* sie vererbt.

```
public boolean equals(Object o)
```

Die *Job*-Instanzen unterscheiden sich anhand der Startzeit, und *equals* liefert *true*, wenn die Startzeiten von beiden *Job*-Objektvariablen übereinstimmen.

Die Java-Klasse *Job* implementiert das Interface *comparable* und bietet daher die Methode *compareTo*, die aufgerufen wird, um das aktuelle Objekt mit anderen zu vergleichen.

```
public int compareTo(Object o)
```

- *compareTo* muss einen Wert kleiner 0 zurückgeben, wenn die Startzeit des aktuellen Objekts **früher** als die des zu vergleichenden Objekts ist.
- *compareTo* muss einen Wert größer 0 zurückgeben, wenn die Startzeit des aktuellen Objekts **später** als die des zu vergleichenden Objekts ist.
- *compareTo* muss 0 zurückgeben, wenn die Startzeit des aktuelle Objekts und die des zu vergleichenden Objekts **gleich** ist.

Eine Einführung zum Thema *equals* und *compareTo* bietet [Ull06].

4.2 Modellierung einer Service-Klasse

Arbeitseinheiten in S/390-Rechnern sind unterschiedlich, trotzdem haben die Arbeitseinheiten, die zu denselben Service-Klassen gehören, ein relativ gleichmäßiges Profil. Sie benötigen ähnliche Ressourcen bezüglich CPU-Verbrauch, Ein-/Ausgabe-Anforderungen und Hauptspeicherplatzbedarf. Aus diesen Grund können wir diese Parameter für Arbeitseinheiten in der Service-Klasse definieren. Mit diesen Parametern werden Arbeitseinheiten automatisch mit einer bestimmten Verteilung erzeugt. Die Workloadverteilung wird in dem Abschnitt 4.5 beschrieben.

4.2.1 Die Struktur einer Service-Klasse

Jede Service-Klasse repräsentiert eine Arbeitseinheit mit identischen Business Performance-Zielen. Mit der Java-Klasse *ServiceClass.java* werden die wichtigsten Konstrukte

der Service-Klasse modelliert. Die *ServiceClass.java* enthält folgende Membervariablen:

- *serviceClassName* gibt die Bezeichnung der Service-Klasse an,
- *serviceClassID* enthält einen eindeutigen Schlüssel der Service-Klasse. Die *serviceClassID* wird bei der Definition der Service-Klasse gesetzt.
- *importance* ist eine positive ganze Zahl und repräsentiert die Wichtigkeit, mit der die Service-Klasse bevorzugt werden soll, wenn die Betriebsmittel im System nicht mehr ausreichend zur Verfügung stehen. Je kleiner der Wert der *importance* ist, desto wichtiger ist die Service-Klasse. Der Wert "1" ist wichtiger als alle anderen Werte.
- *goalType* zeigt den Typ des Ziels an. Wie in Kapitel 2 beschrieben, ist der Zieltyp entweder *Response Time* oder *Execution Velocity*. Mögliche Werte sind *GOAL_response* und *GOAL_velocity*. Sie sind als Konstanten in der Java-Klasse *Cst.java* deklariert.
- *goal* enthält einen Gleitkomma-Zahlenwert. Der Wert bedeutet bei einer *Response Time*-Zieldefinition die Ausführungsdauer der Arbeitseinheiten der Service-Klasse, und bei einer *Execution Velocity*-Zieldefinition einen Velocitywert zwischen 1 und 100.
- *cpuDemand* zeigt den durchschnittlichen CPU-Verbrauch an, die die Arbeitseinheiten für die Bearbeitung benötigen.
- *ioTimes* enthält die durchschnittliche Anzahl der Unterbrechungen der Workloads.
- *frequency* beschreibt die Zahl von ankommenden Arbeitseinheiten pro Sekunde.
- *mpl* steht für *Multi Programming Level*. Es ist die Zahl von Adressräumen, die sich gleichzeitig im Hauptspeicher aufhalten.
- *extraFreq* enthält die Zahl von sporadisch ankommenden Arbeitseinheiten pro Sekunde.
- *extraBegin* und *extraEnd* beschreiben den Zeitraum von sporadisch ankommenden Arbeitseinheiten.

4.2.2 Service-Klassen Deklaration

Jede Instanz der Java-Klasse *ServiceClass* repräsentiert eine Definition der Service-Klasse. Das folgende Programm definiert die Service-Klasse mit dem Name *scCICS1*:

Listing 4.1: Deklaration der Service-Klasse “scCICS1”

```
1 ServiceClass scTSO1 = new ServiceClass("scCICS1");
2
3 scCICS1.setGoal(Cst.GOAL_response,0.6);
4 scCICS1.serviceClassID  = 0;
5 scCICS1.importance      = 1;
6 scCICS1.cpuDemand       = 80;
7 scCICS1.ioTimes         = 20;
8 scCICS1.frequency       = 16;
9 }
```

Mit *ServiceClass* wurde eine Instanz der Klasse erzeugt. Der Konstruktor dieser Java-Klasse böntigt als Parameter den Namen der Service-Klasse. Danach wurde das Ziel mit der Methode *setGoal()* definiert. Der erste Parameter ist eine Zielart und der zweite der Zielwert. Das Ziel ist hier eine *Response Time* und bedeutet, dass die durchschnittliche Ausführungsdauer aller Arbeitseinheiten innerhalb 0,6 Sekunden liegen soll. Die nächste Zeile gibt einen eindeutigen Schlüssel der Service-Klasse an. Der Membervariablen *scCICS1.importance* wird der Wert “1” zugewiesen und bedeutet, dass die Service-Klasse *scCICS1* am wichtigsten ist und daher bevorzugt werden muss, wenn nicht mehr ausreichend Betriebsmittel im System zur Verfügung stehen. In ähnlicher Weise wird in der 6. und 7. Zeile deklariert, dass der durchschnittliche CPU-Verbrauch 80 Service-Units beträgt und die Anzahl der durchschnittlichen Unterbrechungen 20 ist. Die letzte Zeile definiert die Zahl der ankommenden Arbeitseinheiten pro Sekunde.

Zum Erzeugen von zusätzlichen Arbeitseinheiten gibt es eine Methode in der Java-Klasse *ServiceClass*, mit die der sporadisch ankommenden Arbeitseinheiten erzeugt werden können:

```
public void setExtraJob (int beginTime,int endTime,double frequency)
```

Die Methode erwartet drei Parameter, die ersten beiden Parameter sind die Anfangszeit und Endzeit von sporadischen Arbeitseinheiten, der letzte Parameter ist

die Zahl von Arbeitseinheiten pro Sekunde.

4.3 Bemerkungen zur Zieldefinition

4.3.1 Voraussetzung der *Response Time*-Zieldefinition

Die richtige Verwendung der *Response Time*-Zieldefinition ist an zwei grundlegende Bedingungen geknüpft. Zuerst müssen die Subsysteme und Anwendungen den Beginn und das Ende einer Arbeitseinheit dem WLM mitteilen, so dass der WLM die Ausführungszeit der Arbeitseinheit erfassen kann. Des Weiteren müssen ausreichend viele Arbeitseinheiten innerhalb eines bestimmten Beobachtungszeitraumes im System ablaufen.

Wenn der WLM überhaupt die Anfangs- und Endzeiten nicht erkennt oder zu wenige Arbeitseinheiten in einer Service-Klasse enden, kann eine *Response Time*-Zieldefinition nicht verwendet werden. In diesem Fall muss die *Execution Velocity*-Zieldefinition verwendet werden, da nur die Zustände betrachtet werden, bei denen eine Arbeitseinheit Betriebsmittel benutzt oder darauf wartet. In der Tat werden die *Execution Velocity*-Ziele immer für Batch-Workloads definiert, da diese relative lange ablaufen. Für beispielweise CICS-, IMS- und TSO-Arbeitseinheit wird eine *Response Time*-Zieldefinition definiert.

4.3.2 Nachteile der *Response Time*-Zieldefinition

Wie in Kapitel 2 beschrieben, gibt es zwei Möglichkeiten, das *Response Time* Ziel zu definieren, nämlich die durchschnittliche und prozentuale *Response Time*-Zieldefinition.

Bei der Verwendung der durchschnittlichen *Response Time*-Zieldefinition muss man ihre Aussagekraft berücksichtigen. Die durchschnittliche *Response Time*-Zieldefinition ist sehr anfällig gegenüber einigen wenigen, lange laufenden Arbeitseinheiten, d.h. die durchschnittliche Ausführungszeit kann lang sein, obwohl nur wenige Arbeitseinheiten wegen ihrer Komplexität eine lange Bearbeitungszeit im System benötigen. Aus diesen Grund folgt, dass eine durchschnittliche *Response Time*-Zieldefinition nur bei einer sehr gleichförmigen Verteilung der Antwortzeiten sinnvoll ist. Da das in der Praxis oft nicht der Fall ist, sollte ein prozentuales *Response Time*-Ziel als realistischere Zielvorgabe verwendet werden.

4.4 Liste von Service-Klassen

Eine Instanz aus der Java-Klasse *ServiceClass* stellt eine Definition der Service-Klasse dar. Es werden insgesamt 5 Benutzer-Service-Klassen modelliert:

- *scCICS1* für die CICS-Arbeitseinheiten.
- *scTSO1* für die TSO-Arbeitseinheiten.
- *scDB21* für die Datenbank-Anwendungen.
- *scJES1* für die Batch-Workloads mit höherer Priorität.
- *scJES2* für die Batch-Workloads mit niedrigen Priorität.

Diese fünf *ServiceClass*-Objekte werden in einer Listenstruktur gespeichert. Die Listenstruktur ist in einer Java-Klasse *ServiceClassList.java* implementiert. *ServiceClassList* dient dazu, die Deklaration aller Service-Klassen zu speichern und ihre Priorität zu verwalten. Sie stellt einen parameterlosen Konstruktor zur Verfügung, mit dem die *ServiceClass* instanziert werden kann und besitzt eine Methode *initPriority*, mit der die Priorität für jede Instanz der Service-Klasse initialisiert werden kann:

```
public ServiceClassList ()

private void initPriority ( ServiceClass sc)
```

Die Dispatching-Priorität kann zwischen 0 bis 255 liegen. Die Abbildung 4.2 stammt aus [BCD+05] und zeigt Anwendungen der Priorität im System mit unterschiedlicher Priorität. Je größer der Wert ist, desto höher ist die Priorität. Im *Goal-Mode* kann der Administrator die Priorität der Service-Klassen nicht einstellen, dies regelt der WLM. Da die vom WLM angepasste Priorität ein numerischer Wert im Bereich von 208 bis 252 ist, werden die beiden Werte im Modell als Ober- und die Untergrenze für die Dispatching-Priorität betrachtet. Weiterführende Informationen zu dieser Thematik findet man unter [BCD+05].

Neben der Methode zur Initialisierung der Priorität bietet die *ServiceClassList* auch die Möglichkeit der Verwaltung der Priorität. Für das Erhöhen und Reduzieren der Priorität gibt es folgende Methoden:

```
public int priorityDown(int scID)
public int priorityUp (int scID)
```

255	SYSTEM
254	SYSSTC
253	kleine Anwendungen
252 208	Prioritäten für dynamische Anpassung
207 202	nicht verwendet
201 192	Prioritäten für Discretionanry−Anwendungen

Abbildung 4.2: Anwendungen der Priorität

```
public void allPriorityDown ()
public void resetPriority ()
```

Mit *priorityUp* und *priorityDown* kann die Priorität der Service-Klasse immer um 1 erhöht und erniedrigt werden, wenn der Wert der Priorität nicht auf der Ober- bzw. Untergrenze (Priorität ist 252 bzw. 208) liegt. Die *allPriorityDown*-Methode dient dazu, Prioritäten aller Service-Klassen um eins zu reduzieren (siehe Abschnitt 6.2.3). Mit der Methode *resetPriority* können die Prioritäten aller Service-Klassen auf ihre Initialisierungswerte zurückgesetzt werden.

```
public int getPriority ( int scID)
```

Schließlich gibt es noch die Methode *getPriority*. Sie erwartet den Schlüssel einer Service-Klasse und gibt ihre Priorität zurück.

4.5 Arbeitseinheit-Erzeugung und -Verteilung

4.5.1 Die Java-Klasse *JobQueue*

Die Java-Klasse *JobQueue* dient dazu, Arbeitseinheiten zu generieren und zu speichern. Wie in Abschnitt 4.1 besprochen, haben Arbeitseinheiten, die zu denselben Service-Klassen gehören, ein relativ gleichmäßiges Profil. Aus diesem Grund basiert die Erzeugung von Arbeitseinheiten auf Service-Klassen, d.h. zuerst werden die *ServiceClassList*-Objekte vor der Erzeugung von Arbeitseinheiten erstellt, danach werden die Arbeitseinheiten nach den in der Service-Klasse vordefinierteten Parametern (z.B. CPU-Verbrauch und Ein-/Ausgabe-Anforderungen) generiert.

Membervariable zum Speichern von Arbeitseinheiten

Zum Speichern von erzeugten Arbeitseinheiten bietet *JobQueue* eine Membervariable mit dem Namen *jobQueue*:

```
LinkedList jobQueue = new LinkedList();
```

LinkedList ist eine Liste aus dem Java-Paket *java.util*, deren Elemente als doppelt verkettete lineare Liste gehalten werden. Die Membervariable *jobQueue* ist vom Typ *LinkedList* definiert, da ihre Einfüge- und Löschoperationen im Prinzip (viele Elemente vorausgesetzt) performanter als die der *ArrayList* sind.

Konstruktor

Das folgende Listing zeigt den Konstruktor der *jobQueue* (verkürzt):

Listing 4.2: Konstruktor der *jobQueue*

```
1  public JobQueue(ServiceClassList scList){
2
3     Iterator scItr = scList.itr();
4     ServiceClass sc;
5
6     LinkedList jobs;
7
8     while(scItr.hasNext()){
9        sc = (ServiceClass) scItr.next();
10       jobs = jobGenerator(sc);
11       jobQueue.addAll(jobs);
```

```
12    }
13
14    Collections . sort (jobQueue);
15    setJobID(jobQueue);
16  }
```

Der Konstruktor der *JobQueue* erwartet ein *ServiceClassList*-Objekt als Argument. Mit Hilfe des Iteratoren *scItr* (siehe [Krü05]) kann ein einzelnes *ServiceClass*-Objekt aus der *ServiceClassList* ausgelesen und in einer *ServiceClass*-Variable *sc* in der Zeile 9 für kurze Zeit gespeichert werden.

Die Erzeugung von Arbeitseinheiten für die *ServiceClass*-Variable erfolgt durch Aufruf der Methode *jobGenerator* (Zeile 10). Sie liefert die dazu passenden Arbeitseinheiten, die in der lokalen Variable *jobs* liegen. Da *jobGenerator* die zentrale Methode in dieser Java-Klasse ist, werden wir im nächsten Unterabschnitt diese detailliert vorstellen.

Arbeitseinheiten, die für jedes *ServiceClass*-Objekt erzeugt werden, sind in der Membervariable *jobQueue* gespeichert, sie sind aber unsortiert. Mit Hilfe der Methode *sort* aus der Java-Klasse *Collection* können Arbeitseinheiten, die in *jobQueue* gespeichert sind, in der Reihenfolge ihrer Startzeit sortiert werden, da die Java-Klasse *job* die Methode *compareTo* implementiert.

Nach dem Sortieren steht die jeweils frühere vor der späteren Arbeitseinheit. Mit Hilfe der Methode *setJobID* wird den Arbeitseinheiten ein Service-Klassen-Schlüssel gesetzt.

Der Workloadgenerator *jobGenerator*

Arbeitseinheiten, die zu derselben Service-Klasse gehören, haben ein relativ gleichmäßiges Profil, trotzdem sind sie unterschiedlich, d.h. bei der Erzeugung von Arbeitseinheiten muss garantiert werden, dass Arbeitseinheiten ähnliche Ressourcen bezüglich CPU-Verbrauch, Ein-/Ausgabe-Anforderungen und Hauptspeicherplatzbedarf benötigen, aber dennoch nicht identisch sind. Um dies zu implementieren, werden Zufallszahlen gebraucht. Java stellt zu diesem Zweck eine Klasse *Random* zur Verfügung, mit der Zufallszahlen erzeugt werden können.

```
public Random()
public Random(long seed)
```

Die Java-Klasse *Random* basiert auf dem **Linear-Kongruenz**-Algorithmus. Sie er-

laubt das Instanzieren eines Zufallszahlengenerators mit oder ohne manuelles Setzen des *seed*-Wertes. Um die experimentellen Ergebnisse später reproduzierbar zu machen, wenn dasselbe Modell erneut ausgeführt wird, wird der *seed*-Parameter übergeben und der Zufallszahlengenerator initialisiert seinen internen Zähler mit diesem Wert.

Mit Hilfe der Java-Klasse *Random* kann der Workloadgenerator *jobGenerator* verschiedene Arbeitseinheiten erzeugen. Das folgende Programm zeigt die Implementierung der Methode *jobGenerator* (verkürzt):

Listing 4.3: Workloadgenerator *jobGenerator*

```
1  private LinkedList jobGenerator(ServiceClass sc){
2
3      int jobSum;
4      jobSum = (int)(SysInfo.priodInSecond * sc.frequency);
5      if(jobSum<1)
6          jobSum=1;
7
8      Random incomingRandom = new Random(10);
9      Random cpuRandom = new Random(20);
10     Random ioRandom = new Random(30);
11
12     LinkedList jobs = new LinkedList();
13     Job aJob;
14
15     int incTime;
16     int cpuDemand;
17     int ioTimes;
18
19     for(int i=0;i<jobSum;i=i+1){
20         incTime = incomingGenerator(incomingRandom);
21         cpuDemand = cpuGenerator(sc.cpuDemand,cpuRandom);
22         ioTimes = ioGenerator(sc.ioTimes,ioRandom);
23
24         aJob = new Job(sc,incTime,cpuDemand,ioTimes,pageNum);
25         jobs.addLast(aJob);
26     }
27
28     if(sc.extraFreq!=Cst.EXTRA_NO_JOB)
29         jobs.addAll(extraJobGenerator(sc));
30
31     return jobs;
32 }
```

Die Methode *jobGenerator* erwartet als Argument ein *ServiceClass*-Objekt. Bei der Generierung der Arbeitseinheiten dieser Service-Klasse wird zuerst die Zahl *jobSum* (in der 4. Zeile) berechnet. *jobSum* ist wie folgt definiert:

$$jobSum \quad = \quad Frequenz \times Dauer\ in\ Sekunden \tag{4.1}$$

Wie man aus der Formel 4.1 erkennt, ist die Anzahl der Arbeitseinheiten gleich dem Produkt ihrer Frequenz, d.h. wieviele Arbeitseinheiten pro Sekunde gestartet werden, und dem Zeitraum des Experiments.

private int incomingGenerator(Random r)

Durch Aufruf der Methode *incomingGenerator* wird die Startzeit der Arbeitseinheit ermittelt und an den Aufrufer zurückgegeben. Die Methode erwartet eine *Random*-Instanz als Zufallszahlengenerator. Zurückgegeben werden gleichverteilte Startzeiten innerhalb der Zeiträume des Experiments.

private int cpuGenerator(**int** cpuD, Random r)

Mit Hilfe von *cpuGenerator* wird der CPU-Verbrauch der Arbeitseinheit generiert. Die Methode erwartet neben dem *Random*-Argument einen durchschnittlichen Wert des CPU-Verbrauchs dieser Service-Klasse. Dieser Wert wird im *ServiceClass*-Objekt definiert (siehe Abschnitt 4.2.1). *cpuGenerator* ermittelt den CPU-Verbrauch für die Arbeitseinheit nach der folgenden Formel und gibt das ganzzahlige Ergebnis in der Variable *cpuDemand* zurück.

$$f(cpuD) = \begin{cases} > 1\ und < cpuD, & \text{wenn } cpuD \leq 5 \\ > \lceil cpuD \times 0.85 \rceil\ und < \lceil cpuD \times 1.15 \rceil, & \text{wenn } cpuD > 5 \end{cases} \tag{4.2}$$

In der Formel 4.2 ist *cpuD* der durchschnittliche CPU-Verbrauch der Service-Klasse. Falls *cpuD* nicht größerer als 5 ist, wird durch Aufruf von *cpuGenerator* ein zufälliger CPU-Verbrauch im Wertebereich zwischen 1 bis 5 erzeugt. Falls *cpuD* größer als 5 ist, wird der zurückgegebene CPU-Verbrauch auf den Wertebereich zwischen $cpuD \times 0.85$ und $cpuD \times 1.15$ beschränkt. Diese angenommene,

zufällige Streuung des Wertebereichs des CPU-Verbrauchs kann verkleinert oder vergrößert werden, indem eine entsprechende Variable im Modell neu eingestellt wird.

Zur Generierung der Ein- /Ausgabe-Anforderung wird die Methode *ioGenerator* verwendet.

```
private int ioGenerator(int io, Random r)
```

ioGenerator erwartet das *Random*-Objekt und die durchschnittliche Ein- /Ausgabe-Anforderungen der Service-Klasse als Argument. *ioGenerator* erzeugt zufällige Ein- /Ausgabe-Anforderungen in folgenden Wertebereich:

$$f(io) = \begin{cases} > 1 \; und < io, & \text{wenn } io \leq 4 \\ > \lceil io \times 0.9 \rceil \; und < \lceil io \times 1.1 \rceil, & \text{wenn } io > 4 \end{cases} \tag{4.3}$$

Das zurückgegebene Ergebniss des *ioGenerator*s ist ein ganzzahliger Wert.

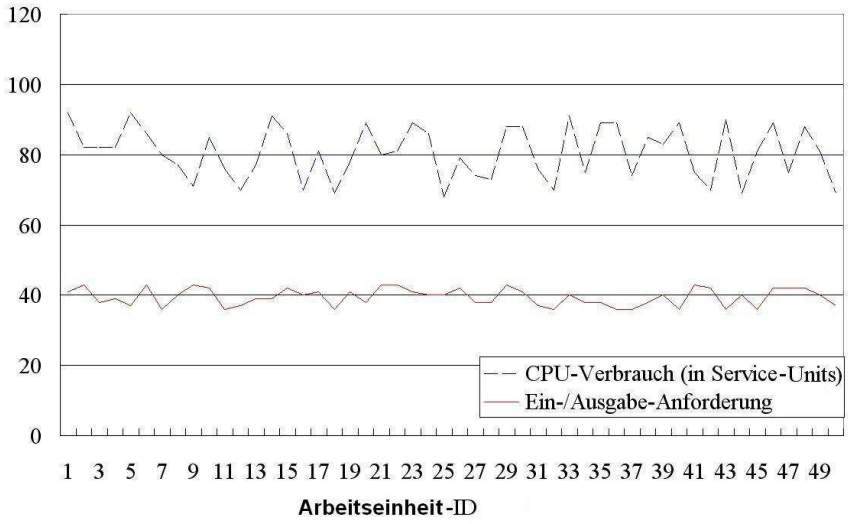

Abbildung 4.3: CPU-Verbrauch und Ein-/Ausgabe-Anforderung von Arbeitseinheiten der Service-Klasse *scDB21*

In Abbildung 4.3 sind bespielweise der CPU-Verbrauch und Ein-/Ausgabe-Anforderung von 50 Arbeitseinheiten dargestellt. Diese Arbeitseinheiten gehören zu der Service-Klasse *scDB21*. Sie sind verschieden, aber der durchschnittliche CPU-Verbrauch beträgt 80 Service-Units und die durchschnittliche Ein-/Ausgabe-Anforderung ist ungefähr 40.

Der sporadische Workloadgenerator *extraJobGenerator*

Um sporadische Arbeitseinheiten zu erzeugen, sind zwei Schritte notwendig:

- Aufruf des *jobGenerators* zur Erzeugung von gleichverteilten Arbeitseinheiten,
- Aufruf des *extraJobGenerators* zur Erzeugung von zusätzlichen sporadischen Arbeitseinheiten.

Die Methode *extraJobGenerator* basiert in ähnlicher Weise auf *Random* zur Generierung von Arbeitseinheit.

```
private LinkedList extraJobGenerator ( ServiceClass sc )
```

extraJobGenerator erwartet ein *ServiceClass*-Objekt *sc* als Argument. Falls die Membervariablen *extraFreq*, *extraBegin* und *extraEnd* in *sc* definiert sind, generiert *extraJobGenerator* die sporadischen Arbeitseinheiten der Service-Klasse *sc*. Fall sie nicht definiert sind, dann hat *extraJobGenerator* nichts zu tun.

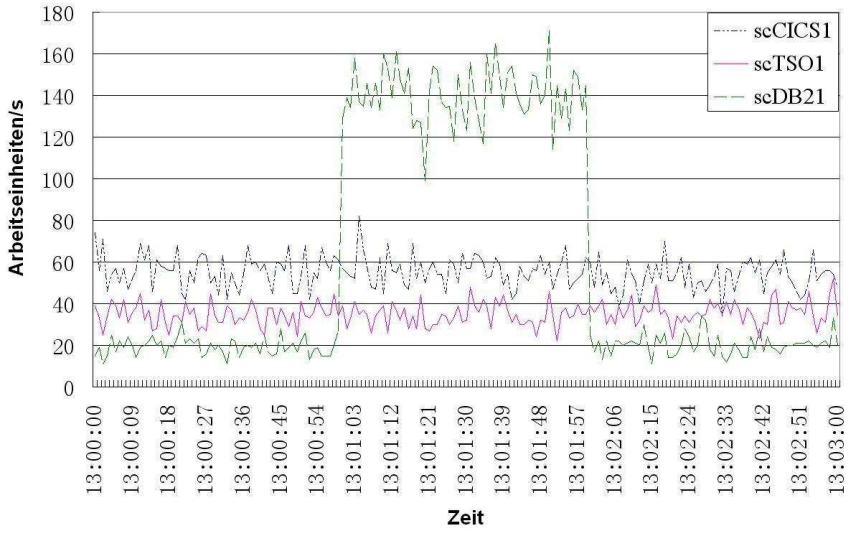

Abbildung 4.4: Workload-Verteilung

Abbildung 4.4 zeigt beispielhaft eine Workload-Verteilung erzeugt durch Verwendung der Methoden *jobGenerator* und *extraJobGenerator*. Das Simulationsszenario fängt um 13:00 Uhr an und endet etwa um 13:03 Uhr. Es gibt insgesamt 3 Service-Klassen. Arbeitseinheiten von *scCICS1* und *scTSO1* laufen von 13:00 Uhr

bis 13:03 Uhr und sind gleichmäßig verteilt, da keine zusätzliche Arbeitseinheit erzeugt wird. Arbeitseinheiten der Service-Klasse *scDB21* werden nach 13:01 Uhr sporadisch gestartet und enden ca. um 13:02 Uhr.

Kapitel 5

Modellierung des Rechners bzw. der Betriebsmittel

In Kapitel 3 haben wir das Rechnersystem des WLM-Modells vorgestellt. Es enthält die typischen Komponenten, die in Großrechnern auftreten. Die klassischen Hardware-Komponenten eines z900-Rechners sind die CPUs, der Speicher und die Platten des Ein-/Ausgabe-Subsystems. In diesem Kapitel werden diese Betriebsmittel mit Java-Klassen implementiert.

5.1 Prozessoren

5.1.1 Prozessor-Technologie der S/390-Architektur

Keine Computersysteme ohne Prozessoren - hier werden die Daten verarbeitet, die uns in der Datenverarbeitung so wichtig sind. Ein wesentlicher Unterschied zwischen S/390- und anderen Rechnern besteht in der Art der Verpackungs-Technologie. Die CPU-Chips der S/390-Rechner sind auf einem Multi-Chip Module (MCM) untergebracht. Ein MCM umfasst alle Prozessoren, Level-2-Cache Module und Anschlüsse für den Bus auf einem gemeinsamen keramischen Träger. Das MCM besteht aus einem 127 x 127 mm großen mehrschichtigen Glas-Keramik-Träger mit mehr als 2,5 Milliarden Transitoren. Es werden für die z900-Rechner insgesamt zwei unterschiedliche MCMs hergestellt, einer mit 20 Prozessor-Chips und einer mit 12 Prozessor-Chips. Aus diesen Chips können alle Modelle konfiguriert werden, vom Uniprozessor bis zu einem Rechner mit 16 Prozessoren (CPUs).

Als Beispiel soll hier das z900 MCM genommen werden. Die Prozessor-Chips teilen sich auf in 16 CPUs, 3 als *System Assist Processor* (SAP) für Ein-/Ausgabe-

Verarbeitung und eine Reserve-CPU, welche im Fehlerfall automatisch eine CPU ersetzen kann.

5.1.2 CPU-Service-Units und die Zeit im WLM-Modell

Da sich die Geschwindigkeit der Rechnermodelle sehr stark unterscheidet, ist es notwendig, ein standardisiertes Maß für den CPU-Verbrauch einzuführen. Dieses Maß ist die CPU-Service-Unit. Für die Berechnung einer Service-Unit wird die Ausführungszeit der Workload ermittelt und mit einer Konstanten multipliziert, die modellabhängig ist und Effekte wie die Anzahl der Prozessoren des Systems beinhaltet. Damit wird die Service-Unit übertragbar zwischen den verschiedenen Rechnern [TV04]. Mit dem Begriff Service-Unit kann man unterschiedliche Rechner einfach miteinander vergleichen. Zum Beispiel hatte der ersten Rechner mit nur einem Prozessor (9672-R11) 696 CPU-Service-Units pro Sekunde. Auf einem zSeries-System mit dem 16-Wege-Prozessor z900 2003-116 liegt die Leistung bei 8.117,7 Service-Units pro Sekunde und pro Prozessor. Weitere Informationen zum Thema “Service-Unit” findet man unter [bib03].

Der Taktzyklus

Neben der CPU-Service-Unit ist der Begriff Taktzyklus von besonderer Bedeutung. Ein Taktzyklus ist die kleinstmögliche verarbeitbare Einheit. Das Zeitintervall zwischen zwei Taktzyklen wird als Taktzykluszeit bezeichnet. Der Reziprokwert der Taktzykluszeit ist die geläufigere Taktfrequenz. Um den Taktzyklus im WLM-Modell einfach zu simulieren, nehmen wir an, dass in jedem Taktzyklus eine Service-Unit bearbeitet wird. Es ergibt sich folgende Gleichung:

$$Taktzykluszeit = \frac{1\ Sekunde}{Service\ Unit\ pro\ Sekunde\ und\ pro\ CPU} \quad (5.1)$$

oder

$$Taktfrequenz = Service\ Unit\ pro\ Sekunde\ und\ pro\ CPU \quad (5.2)$$

Nehmen wir in den beiden obigen Formeln beispielsweise eine Taktzykluszeit von 1 Mikrosekunde an, dann erhalten wir eine Taktfrequenz von 10^6 und eine Service-Unit pro CPU von 1,000,000/s.

```
public static int timeInterval = 1000;
```

Die Taktfrequenz wird als die Variable *timeInterval* in der Java-Klasse *SysInfo.java* definiert.

Die Modellzeit

Zeit wird im WLM-Modell als Basis für Meß- und Entscheidungsintervalle verwendet. Die WLM-Modellzeit zählt die vergangenen Taktzyklen seit dem Zeitpunkt des Starts des Experiments. Demzufolge muss man den Taktzyklus in Sekunden umrechnen und zu dieser Uhrzeit addieren. Die Umrechnung in eine menschenlesbare Form wird mit folgender Formel definiert:

$$Aktuelle\ Modellzeit \quad = \quad Startzeit + \frac{Anzahl\ der\ Taktzyklen}{Taktfrequenz} \tag{5.3}$$

Es wird eine Systemvariable definiert, um die Modellzeit darzustellen.

```
public static int startTime = 13

public static String toTime(int clock)
```

startTime ist mit dem Wert 13 in *SysInfo.java* definiert, d.h. die Startzeit der Simulation ist 13 Uhr. Mittels der Systemmethode *toTime* kann man die Anzahl der Taktzyklen in Sekunden umrechnen. *toTime* erwartet die Zahl der Taktzyklen als Argument und liefert die Sekunden zurück. Wenn die Taktfrequenz den Wert 1000 hat, kann man mit Hilfe von *toTime* und der Startzeit die aktuelle Modellzeit nach der Formel 5.3 berechnen, zum Beispiel,

$$1000.\ Taktzyklus\ bedeutet\ 13:00:01$$
$$60000.\ Taktzyklus\ bedeutet\ 13:01:00$$
$$100000.\ Taktzyklus\ bedeutet\ 13:01:40$$

5.1.3 Modellierung des Prozessors

Dispatching-Queue

Die Java-Klasse *Dispatching-Queue* stellt eine CPU-Dispatching-Queue dar. Prozesse, d.h. die ablaufenden Workloads, werden nach ihre Priorität sortiert. Die Implementierung stellt verschiedene Methoden zum Zugriff auf die Dispatching-Queue zur Verfügung:

```
public void insert (Job job)

public Job getFirst ()

public Job remove(int i)

public Job removeSCJob(int scID)
```

Die Aufgabe der Methode *insert* besteht darin, die Workload in der Dispatching-Queue nach ihrer Priorität einzuordnen. Die erste Workload in der Dispatching-Queue hat die höchste Priorität. Der Zugriff auf die Workload erfolgt mithilfe der parameterlosen Methode *getFirst*, sie gibt immer die erste Workload zurück.

Das Austragen der Workload aus der Dispatching-Queue erfolgt mit den Methoden *remove* und *removeSCJob*. *remove* erwartet den Index der zu löschenden Workload und liefert den dazu passenden Wert. Mit *removeSCJob* wird die letzte Workload von der gegebenen Service-Klasse gelöscht. Sie erwartet einen Service-Klassen-Schlüssel *scID* als Argument und liefert diese Workload zurück. Die Suche beginnt bei der Workload mit der geringsten Priorität und fährt mit ansteigender Priorität fort. Falls keine Workload der gegebenen Service-Klasse enthalten war, ist der Rückgabewert *null*. Diese Methode *removeSCJob* ist für die Auslagerung der Workload am wichtigsten.

```
public int size ()
```

Zusätzlich zu den bisher erwähnten Methoden gibt es noch eine weitere Methode mit den Namen *size*. Sie liefert die Anzahl der Workloads in der Dispatching-Queue.

Implementierung des Prozessors

Mit Java definieren wir eine Java-Klasse *PUs*, in der wir die abstrakten Eigenschaften von Prozessoren spezifizieren. Die wichtigste Membervariable in dieser Klasse ist *cpus*.

```
private ArrayList cpus;
```

Sie ist mit den Typ *ArrayList* definiert. Jedes Element der *cpus* repräsentiert einen Prozessor. Die Länge der Liste *cpus* ist gleich der Anzahl der CPUs, sie ist durch den Wert *cpuNum* in der Java-Klasse *SysInfo* definiert. Wenn ein Element eine Workload enthält, bedeutet das, dass die CPU dieser Workload zugeteilt ist, sie ist besetzt. Eine CPU ist frei, falls das entsprechende Element *null* ist. Wenn eine Workload ein Element besetzt, bedeutet das, dass sie von der CPU verarbeitet wird, sie ist in dem Zustand “Ablauf”.

Zur Instanzierung steht ein Konstruktor zur Verfügung:

```
PUs(DispatchQueue dQ, IOQueue ioQ, ServiceClassList scl)
```

Der Konstruktor erwartet drei Parameter. Indem das *DispatchQueue*-Objekt und *IOQueue*-Objekt übergegeben werden, kann die Instanz der *PUs* sich mit der Dispatching- und Ein-/Ausgabe-Queue verbinden. Der dritte Parameter ist ein *ServiceClassList*-Objekt, das alle Informationen der Service-Klasse enthält.

```
public void run(int aTime)
```

Die Methode *run* dient zum Wechseln der unterbrochenen Workloads. Als Argument wird die aktuelle Zeit angegeben. Sie überprüft, ob die Workload wegen einer Ein-/Ausgabe-Anforderung unterbrochen ist, dann fügt *run* die unterbrochene Workload der Ein-/Ausgabe-Queue hinzu und entfernt das entsprechende Element aus der Membervariable *CPUs*. Die Workload wechselt in den Zustand “Schlaf im Speicher”. Wenn es keine unterbrochene Workload gibt, tut *run* nichts.

```
private void loadJobs(int aTime)
```

Mit *loadJobs* wird die Workload auf den CPUs zum Laufen gebracht. Sie erwartet

die aktuelle Zeit. Die Methode *loadJobs* kann automatisch erkennen, ob es freie CPUs gibt, wenn ja, wird die Workload von der CPU-Dispatchingqueue einer CPU zugeteilt, bis keine weiteren CPUs frei sind. Gibt es keine Workload in der CPU-Dispatchingqueue, kehrt *loadJobs* sofort zurück.

5.2 IO-Channel

Im Gegensatz zu den meisten anderen Rechnerarchitekturen verfügt die S/390 über eine eindeutig definierte Ein-/Ausgabe-Architektur. Sie gilt grundsätzlich für den Anschluss beliebiger Arten von Ein-/Ausgabe-Einheiten. Eine Ein-/Ausgabe-Einheit ist in der Regel ein physikalisch identifizierbares Ein-/Ausgabe-Gerät. Prinzipiell können mehrere Einheiten einem Ein-/Ausgabe-Geräte zugeordnet werden [HKS04]. Der gesamte S/390-Ein-/Ausgabe-Verkehr wird durch das *Channel Subsystem* gesteuert. Es können bis zu 65536 Ein-/Ausgabe-Einheiten angeschlossen werden. Das Channel Subsystem kann 65536 logische Verbindungen unterhalten, die durch eine 16-Bit-Adresse identifiziert werden und als *Subchannels* bezeichnet werden. Typische System-Installationen haben einige hundert bis zu mehreren tausend Ein-/Ausgabe-Einheiten.

5.2.1 Implementierung

Verzögerungsdefinition

Wenn die Workload auf eine Ein-/Ausgabe-Operation wartet, kann sie ihre Ausführung nicht fortsetzen, sie wird blockiert. Wenn die Ein-/Ausgabe-Operation beendet ist, wird die Workload, die darauf gewartet hat, entblockiert und kann weiter ausgeführt werden, falls eine CPU frei ist. Die Zeit für das Warten auf das Beenden der Ein-/Ausgabe-Operation wird als Verzögerung bezeichnet. Um die Verzögerung der Ein-/Ausgabe-Operation zu simulieren, wird ein konstante Variable *ioDelay* in der Java-Klasse *SysInfo* definiert.

```
public static int ioDelay = 10;
```

ioDelay wird der Wert 10 zugewiesen, dies bedeutet, dass die Verzögerung der Ein-/Ausgabe-Operation 10 Taktzyklen ist, d.h. die Workload muss 10 Taktzyklen warten, bis die Ein-/Ausgabe-Operation endet, danach kann die Workload weiter ausgeführt werden.

Java-Klasse *Channel*

Die Java-Klasse *Channel* realisiert das S/390-*Channel Subsystem*. Zur Instanzierung steht ein Konstruktor zur Verfügung:

```
Channel(DispatchQueue dq,IOQueue ioQ)
```

Als Parameter werden das *DispatchQueue*- und *IOQueue*-Objekt angegeben. Durch diese beiden Parameter wird das *Channel Subsystem* an die CPU-Dispatchingqueue und Ein-/Ausgabequeue angeschlossen.

Channel bietet eine Methode zur Verwaltung der auf Ein-/Ausgabe-Operationen wartenden Workloads:

```
public void run(int aTime)
```

run nimmt die aktuelle Zeit als Argument. Sie berechnet, ob die Workload schon eine bestimmte Zeit (*ioDelay*) wartet. Nach Ablauf der Ein-/Ausgabe-Verzögerung wird die Workload in die Dispatchingqueue hinzufügt, sie wechselt in den Zustand “Ausführbar im Speicher”.

5.3 Speicherverwaltung

Die z/OS-Speicherverwaltung teilt sich in drei Bereiche auf ([TV04]):

- Die Verwaltung des realen Hauptspeichers durch den *Real Storage Manager (RSM)*. Sie setzt virtuelle in reale Adressen um, und verwaltet die Verfügbarkeit und Zugehörigkeit von Hauptspeicher-Pages.
- Die Verwaltung des Paging Storage durch den *Auxiliary Storage Manager (ASM)*, dabei geht es um das Ein- und Auslagern von Seiten des Hauptspeichers auf den externen Speicher. Der *ASM* verwaltet außerdem die Page-Datasets und stellt die Ein-/Ausgabe-Routinen für das Paging zur Verfügung.
- Die Verwaltung des virtuellen Speichers durch den *Virtual Storage Manager (VSM)*.

In z/OS wird die Anzahl der Adressräume der Workloads durch Verwendung des *Multi Programming Level* eingeschränkt. Der MPL legt fest, wie viele Adreßräume gleichzeitig im System sein dürfen und viele auf jeden Fall im System blei-

ben sollen. Die Unter- und Obergrenze für den MPL darf nur durch den WLM dynamisch geändert werden.

5.3.1 Implementierung

Die Hauptspeicherverwaltung ist durch die Java-Klasse *StorageManager* implementiert. *StorageManager* bietet eine Methode *run* zum Ein-/Auslagern der Workload.

```
public void run(int actTime)
```

run erwartet die aktuelle Systemzeit und überprüft, ob eine Workload gestartet werden soll. Sie ruft eine private Methode *insertNewJob* auf, um die Workload der Dispatch-Queue hinzuzufügen:

```
private void insertNewJob
```

Neben *insertNewJob* verwendet *run* noch zwei private Methoden *swapIn* und *swapOut*.

```
private void swapIn(int scID, int num)
private void swapOut(int scID, int num)
```

Durch Aufruf von *swapIn* kann die Workload vom Sekundärspeicher in den Hauptspeicher eingelagert werden. Abhängig von ihrem Zustand wird sie entweder in der Dispatching-Queue oder in der Ein-/Ausgabe-Queue hinzugefügt. *swapIn* erwartet zwei Parameter. Der erste ist der Schlüssel der Service-Klasse, der zweite legt fest, wie vieles Workloads von dieser Service-Klasse eingelagert werden sollen. Ebenso erwartet *swapOut* dieselben Parameter, sie dient dazu, Workloads von der Dispatching-Queue oder von der Ein-/Ausgabe-Queue auszulagern.

Kapitel 6

Modellierung der WLM-Algorithmen

In diesem Kapitel wollen wir die Modellierung des Workload Managers besprechen. Abschnitt 6.1 beschreibt die Datensammlung im WLM-Modell, da die Daten die Basis für die WLM-Algorithmen sind. In dem Abschnitt 6.2 wird die Grundlage der WLM-Algorithmen dargestellt. Sie bestehen aus drei Schritten: Auswahl des Receivers, Feststellen des Engpasses und Beheben des Engpasses. Vor der Darstellung der Algorithmen müssen wir zwei Zeitabstände unterscheiden:

- Das Sammlungintervall ist die Zeiteinheit, die bestimmt, in welchen Abständen der WLM Daten über den aktuellen Zustand aller verwalteten Betriebsmittel erhebt. Im Original-z/OS ist das Sammlungintervall 250 ms.

- Das Adjustmentintervall bestimmt, wann der Anpassungsalgorithmus des WLMs abläuft, er läuft alle 10 Sekunde im z/OS-System.

6.1 Datensammlung und Speichern

Alle 250 ms ermittelt der z/OS-WLM den aktuellen Zustand der Betriebsmittel. Dies sind Informationen über die Prozessoren, den Speicher und die Benutzung des Ein-/Ausgabe-Systems. Der WLM stellt das Verhältnis von Warte- zu Benutzungszuständen fest. Die Delay- und Using-Werte bilden die Grundlage für den Regelmechanismus, der feststellt, welche Engpässe für die Service-Klasse existieren und wie sie behoben werden können. Der WLM verwendet diese Daten, um den Zugang der Service-Klassen zu den Betriebsmitteln zu regeln.

Die Datensammlung wird durch die Java-Klasse *Performance* implementiert. *Performance* besitzt eine Konstruktor mit einem Parameter:

```
Performance(ServiceClass sc)
```

Jede Service-Klasse ist mit eine *Performance*-Objekt verbunden, deswegen erwartet der Konstruktor ein *ServiceClass*-Objekt *sc* als Argument. Es darf für eine Service-Klasse ein einzelnes *Performance*-Objekt deklariert werden. *Performance* führt zwei Funktionen aus. Die erste besteht darin, den aktuellen Performance-Index der Service-Klasse *sc* aus den gesammelten Daten zu berechnen. Der PI wird als Entscheidungsgrundlage für den WLM-Algorithmus benutzt. Die zweite Funktion besteht im Speichern des PI der Service-Klasse *sc*, damit der WLM-Algorithmus sie verwenden kann.

Nach der Formel 2.3 kann das *Performance*-Objekt den aktuellen PI für Service-Klassen mit *Response Time*-Zielvorgaben berechnen. Um die durchschnittliche *Response Time*-Zielvorgaben zu ermitteln, wird eine FIFO-Struktur *piHistory* verwendet, in der die kompletten Workloads gespeichert werden, da die Ausführungsdauer für die noch laufenden Workloads nicht berechnet werden kann. Der Größe der *piHistory* wird mit dem Wert *historyDataSize* in der Java-Klasse *SysInfo* festgelegt. Bei der Berechnung des PI für die Service-Klassen mit einer *Execution Velocity*-Zielvorgabe werden nur die noch nicht beendeten Workloads berücksichtigt, da vorwiegend Batch-Workloads *Execution Velocity*-Zielvorgaben benutzen. Diese laufen meist sehr lange ab und sind für *Response Time*-Zielvorgaben nicht geeignet. Die Methode *calcuPI* dient zur Berechnung der PI.

```
public void calcuPI()
```

calcuPI kann automatisch den Typ der Service-Klasse erkennen und verwendet die entsprechende Formel zur Berechnung der PI.

Zur gleichen Zeit ermittelt *Performance* noch die Wartezeit auf die einzelnen Betriebsmittel. Wartezeit ist die Zeit, die eine Workload zum Ablauf auf ein bestimmtes Betriebsmittel warten muss, sie ist in Taktzyklen dargestellt. CPU- und Swap-In-Wartezeit werden von *Performance* berechnet. CPU-Wartezeit ist die Zeit, die die Workload auf eine freie CPU wartet, d.h. die Workload ist im Zustand “ausführbar im Speicher”. Swap-In-Wartezeit bedeutet, dass die Workload im Zustand “Schlaf und ausgelagert” ist, sie befindet sich in dem Sekundärspeicher und wartet auf das Einlagern in den Hauptspeicher. Die Wartezeit ist später für das Feststellen

des Betriebsmittelengpasses der wesentliche Entscheidungsfaktor.

Dieses WLM-Modell kann ein graphisches Ergebnis liefern, es zeigt die Änderung am PI nach Anpassung durch den WLM. Um das zu implementieren, speichert *Performance* alle historische PIs in einer PI-Tabelle. Die PI-Tabelle besteht aus mehreren Java-Collection *ArrayList*, jede *ArrayList*-Instanz wird als ein Feld betrachtet. Die *Performance* enthält die folgenden Felder:

- *sampleTime* enthält den Zeitpunkt, zu dem die PI berechnet wurde, er ist in Taktzyklen dargestellt.
- *piHistory* enthält den historischen Performance-Index.
- *cpuDelayHistory* enthält die Wartezeit für die CPU.
- *swapDelayHistory* enthält die Wartezeit für das Einlagern in den Hauptspeicher.

Performance bietet mit den Methoden *getActPI*, *getCPUDelay* und *getSwapDelay* Zugriff auf diese Daten, die drei sind parameterlos. Mithilfe der Methode *getActPI* kann der letzte PI der Service-Klasse gelesen werden. Beim Aufrufen von *getCPUDelay* und *getSwapDelay* werden die letzte Wartezeit für die CPU bzw. für das Einlagern in den Hauptspeicher zurückgeliefert werden.

```
public double getActPI()

public double getCPUDelay()

public double getSwapDelay()
```

Es werden insgesamt 5 *Performance*-Objekte im Programm deklariert, da es 5 Service-Klassen gibt. Alle *Performance*-Objekte werden in der Datenstruktur *PerformanceList* gespeichert.

6.2 Der WLM-Algorithmus

6.2.1 Der Anpassungsalgorithmus des WLMs

Die Anpassungsalgorithmus spielt im z/OS-WLM eine zentrale Rolle. Grundlage für die Entscheidung sind die Zieldefinitionen und die Zielerfüllung. Der Algo-

rithmus läuft alle 10 Sekunden im System ab. Um die Performance der Service-Klassen automatisch und selbstoptimierend verwalten zu können, bedienen sich die Original-Anpassungsalgorithmen im z/OS umfassender Strategien zur Engpassbestimmung und -behebung. Sie können zum Beispiel mehrere Engpässe gleichzeitig erkennen und für die Service-Klasse den Zugang zu Betriebsmitteln genau anpassen. Eine detaillierte Besprechung der Algorithmen des z/OS-WLMs findet man unter [AEE+97]. Um die Anpassungsalgorithmen einfach darzustellen, nehmen wir an, dass die Anpassungsalgorithmen nur einen Engpass erkennen und beheben müssen. Abbildung 6.1 zeigt den im WLM-Modell verwendeten Anpassungsalgorithmus. Dieser Algorithmus ist einfach, trotzdem enthält er die wichtigsten Schrit-

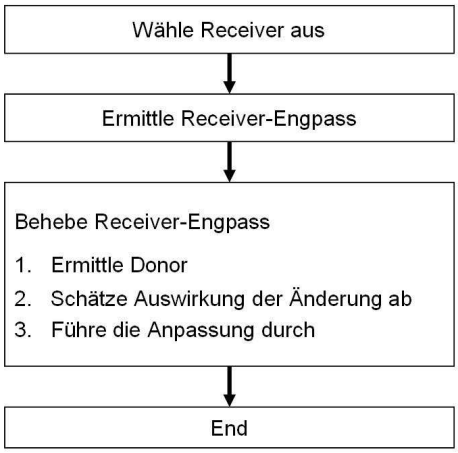

Abbildung 6.1: Anpassungsalgorithmus des WLM-Modells

te des z/OS-WLMs. Zur Vorbereitung werden die in der Zwischenzeit gesammelten Daten aufbereitet und für diesen Algorithmus abgelegt. Der Anpassungsalgorithmus besteht aus drei Schritten: Auswahl des Receivers, Feststellen des Engpasses und Beheben des Engpasses. In den Unterabschnitten wollen wir jeden einzelnen Schritt besprechen.

6.2.2 Auswahl des Receivers und Feststellen des Engpasses

Unter dem Receiver versteht man diejenige Service-Klasse, welche einen Performance-Index von größer als 1 besitzt und deren Importance höher ist wie die jeder anderen Service-Klasse, die ihre Ziele nicht erfüllt. Wenn mehrere Service-Klassen ihre Ziele nicht erfüllen, wird die Service-Klasse mit der höhsten Importance ausgewählt, da der WLM in diesem Modell bei jedem Ablauf nur einen Receiver aus-

wählt. Wenn es keine Service-Klasse mit PI von größer als 1 gibt, d.h. alle Service-Klassen können ihre Ziele erfüllen, dann wird kein Receiver ausgewählt.

Ein Engpass ist eine Ressourcen-Knappheit, die die größte Auswirkung auf die Zielerfüllung der Service-Klasse hat. Im WLM-Modell werden zwei Engpässe modelliert, CPU-Engpass und Swap-In-Engpass. Der Ein-/Ausgabe-Engpass wird nicht berücksichtigt, da das Verfahren zur Erkennung und Behebung der Ein-/Ausgabe-Engpässe ähnlich wie bei CPU-Engpässen ist. Um den Engpass festzustellen, werden die CPU- und Swap-In-Wartezeit im Modell gesammelt (siehe Abschnitt 6.1). Mit diesen Daten kann der Engpass ermittelt werden.

- CPU-Engpass, falls CPU-Wartezeit größer oder gleich der Swap-In-Wartezeit ist.
- Swap-In-Engpass, falls CPU-Wartezeit kleiner als Swap-In-Wartezeit ist.

Das Feststellen des Engpasses erfolgt nur dann, wenn es einen Receiver gibt. Falls kein Receiver existiert, wird das Feststellen des Engpasses nicht durchgeführt. Der Ergebnis wird für den nächsten Schritt abgelegt.

6.2.3 Beheben des Engpasses

Um den Engpass zu beheben, muss erst der Donor ermittelt werden. Der Donor ist eine Service-Klasse, die ihr Ziel übererfüllt und die niedrigste Importance aus der Menge von Service-Klasse mit PI kleiner als 1 hat. Bei der Anpassung wird immer

Importance	Service class	PI
1	scCICS1	1.5
2	scTSO1	0.5
3	scDB21	1.5
4	scJES1	0.5
5	scJES2	0.5

a)

scCICS1 ist ein Receiver
scJES2 ist ein Donor

Importance	Service class	PI
1	scCICS1	0.5
2	scTSO1	0.5
3	scDB21	0.5
4	scJES1	1.5
5	scJES2	1.5

b)

scJES1 ist ein Receiver
scDB21 ist ein Donor

Abbildung 6.2: Auswahl von Receiver und Donor

nur ein Donor ausgewählt. Falls keine Service-Klasse ihre Ziele übererfüllt, bedeutet das, dass kein Donor existiert. In diesem Fall kann das Beheben des Engpasses nicht mehr weiter durchgeführt werden, der Anpassungsalgorithmus endet, da die Betriebsmittel insgesamt nicht ausreichen.

Beim Abschätzen der Auswirkung der Änderung berechnet der z/OS-WLM, ob eine Umverteilung einen Nutzen für den Receiver darstellt, und wie sich die Zielerfüllung für Donor und Receiver verändern. Der Donor kann sein Ziel nach der Änderung nur dann noch erreichen, wenn er sein Ziel deutlich übererfüllt, deswegen hat im WLM-Modell der Donor einen PI kleiner als 0.9. Die Abbildung 6.2 zeigt zwei Beispiele für die Auswahl von Receiver und Donor. Je kleiner der Wert der Importance ist, desto höher die *Business Importance*. Die Service-Klasse *scCICS1* hat die wichtigste *Business Importance* mit dem Wert von 1. Demgegenüber ist die Importance der Service-Klasse *scJES2* am niedrigsten. Im Beispiel *a)* ist *scCICS1* ein Receiver, sie besitzt einen PI größer als 1 und deren Importance ist höher, als die Importance der Service-Klasse *scDB21*. *scJES2* ist ein Donor, da sie ihr Ziel übererfüllt und die niedrigste Importance im Vergleich zu den Service-Klassen *scTSO1* und *scJES1* hat. Im Beispiel *b)* ist der Receiver die Service-Klasse *scJES1* und der Donor *scDB21*.

Wenn sowohl der Receiver als auch der Donor existieren, führt der WLM die Behebung des Engpasses weiter durch. Abhängig vom Delay-Type wird als nächstes ein Algorithmus ausgewählt, der den Engpass beheben soll. Ziel ist es, den Zugang zu den Betriebsmitteln so zu verändern, dass die Zahl der gemessenen Delays verkleinert werden kann.

Beheben des CPU-Engpasses

Der Zugang zu den CPUs ist hauptsächlich durch die Festlegung von der Priorität der Workload geregelt. In der unten dargestellten Tabelle 6.3 ist aufgelistet, wie sich der Receiver (R) und der Donor (D) verändern, um den CPU-Engpass zu beheben. Um das Verhältnis der PI von Receiver und Donor in Abbildung 6.3 übsichtlich darzustellen, nehmen wir zur Vereinfachung nur 5 Prioritätsstufen an, in der Tat kann die Priorität Werte von 208 bis zu 252 annehmen (siehe Abbildung 4.2). Ein Maßnahme zum Beheben des CPU-Engpasses ist abhängig davon, ob die Priorität des Receivers niedriger ist als die des Donors, oder ob die Prioritäten von Receiver und Donor auf der Obergrenze (Priorität=252) und Untergrenze (Priorität=208) liegen. Es gibt insgesamt vier Maßnahmen:

- Falls die Priorität des Receivers niedriger oder gleich der Priorität des Do-

		Donor				
Receiver	Priorität der Service-Klasse	252	249	241	227	208
	252	Alle P ↓ (R) P ↑	(D) M ↓	(D) M ↓	(D) M ↓	(D) M ↓
	249	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(D) M ↓	(D) M ↓	(D) M ↓
	241	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(D) M ↓	(D) M ↓
	227	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(D) M ↓
	208	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(R) P ↑ (D) P ↓	(R) P ↑

(R): receiver P: priority ↑: erhöhen
(D): donor M: MPL ↓: reduzieren

Abbildung 6.3: Matrix zum Beheben des CPU-Engpasses

nors ist und sie nicht auf der Ober- oder Untergrenze liegt, erhöht der WLM die Priorität des Receivers um 1 und reduziert die Priorität des Donors um 1, dadurch bekommt der Receiver besseren Zugang zu der CPU. Dieser Fall wird in der Tabelle 6.3 mit den Farbe **Rosa** gezeigt. Der z/OS-WLM kann berechnen, um wie viel die Priorität erhöht oder reduziert werden muss, um sicherzustellen, dass eine Veränderung für die Gesamtheit aller Workloads positiv ist. Dieser Wert wird als Nettowert (englisch “net value”) bezeichnet. Der Nettowert für die Priorität wird im WLM-Modell immer mit 1 angenommen, d.h. die Priorität wird immer um 1 erhöht oder reduziert.

- In den **blauen** Feldern der Tabelle 6.3 liegt die Priorität des Receivers über der des Donors. In diesen Fall würde sich nichts an der Beziehung zwischen Receiver und Donor ändern, wenn die Priorität des Receivers erhöht und die Priorität des Donors reduziert würde. Die wichtigste Maßnahme in diesem Fall ist die Reduzierung des MPL-Werts des Donors. Der Nettowert für den MPL wird durch folgende Formel festgelegt.

$$f_{reduzieren}(mpl) \; = \; \lceil mpl \times 0.2 \rceil \tag{6.1}$$

$$f_{erhoehen}(mpl) \; = \; \lceil mpl \times 0.5 \rceil \tag{6.2}$$

Durch die MPL-Anpassung erfolgt eine Reduzierung des CPU-Delays des Receivers sowie eine Erhöhung des MPL-Delays des Donors. Da die zur Verfügung stehende Prozessorkapazität endlich ist, muss die CPU-Ressource, die einer Service-Klasse mehr gegeben wird, bei den restlichen Service-Klasse abgezogen werden. Je weniger ausführbare Prozesse es gibt, desto größer werden die Chancen für den CPU-Zugang für den einzelnen Prozess.

- Sind die Prioritäten von Receiver und Donor gleich der Obergrenze (Priorität=252), d.h. die Priorität des Receivers kann nicht mehr steigen, reduziert der WLM die Prioritäten aller Service-Klassen um 1 und erhöht anschliessend die Priorität des Receivers um 1, wie im **gelben** Feld gezeigt.

- Wenn die Prioritäten von Receiver und Donor gleich der Untergrenze (Priorität=208) sind, erhöht der WLM die Priorität des Receivers um 1, d.h. die Priorität des Donors kann nicht mehr gesenkt werden.

Es kann sein, dass eine Anpassung allein nicht ausreicht, um einen besseren Zugang zur CPU zu bekommen und mehrmalige WLM-Anpassungen nötigt werden.

Beheben des Swap-In-Engpasses

Die wichtigste Maßnahme zur Behebung des Swap-In-Engpasses ist die Einstellung des MPL-Wertes. Eine MPL-Anpassung kann auf zwei Weisen verursacht werden, zum einem durch eine *Policy adjustment routine* und zum anderen durch eine *Resource adjustment routine*. Die *Policy adjustment routine* führt eine MPL-Anpassung nur aus, wenn eine Service-Klasse ihr Ziel nicht erfüllt. Die *Resource adjustment routine* gehört zum Betriebssystem und berücksichtigt nicht direkt das Ziel der Service-Klasse, sondern die Auslastung (produktive Zeit) des Systems. Man erkennt leicht, dass die beiden Ziele sich manchmal widersprechen. Damit z. B. Ziele von allen Service-Klassen erreicht werden können, sollte der gesamte Durchsatz bzw. der gesamte MPL-Wert erhöht werden; dies führt dazu, dass die unproduktive Zeit ansteigt, und das System uneffizient wird. Diese Arbeit befasst sich nur mit dem Goal orientierten WLM, deshalb wird die *Resource adjustment routine* nicht mehr modelliert.

		Donor	
		MPL > mpl_in_target	MPL = mpl_in_target
Receiver	MPL > mpl_in_target	(R) M ↑ (D) M ↓	(R) M ↑
	MPL = mpl_in_target	(R) M ↑ (D) M ↓	(R) M ↑

(R): receiver M: MPL ↑: erhöhen
(D): donor ↓: reduzieren

Abbildung 6.4: Matrix zum Beheben des Swap-In-Engpasses

In der Tabelle wird gezeigt, dass ein Swap-In-Engpass durch Erhöhen des MPLs des Receivers und Reduzieren des MPLs des Donors behandelt werden kann. Bei der WLM-Anpassung kann für den MPL-Wert ein *mpl_in_target* angegeben werden, welcher die Anzahl von Prozessen bezeichnet, die im System auf jeden Fall verbleiben dürfen. Ist die Untergrenze unterschritten, werden Adressräume der Workload in den Hauptspeicher eingelagert. Es gibt insgesamt zwei Maßnahmen:

- Ist der MPL des Donors größer als *mpl_in_target*, erhöht der WLM die MPL des Receivers um den Nettowert, der nach der Formel 6.2 ermittelt wird, und reduziert den MPL des Donors um den aus den Formel 6.1 berechneten Nettowert.
- Ist der MPL des Donors gleich *mpl_in_target*, erhöht der WLM nur den MPL des Receivers, da der MPL des Donors nicht reduziert werden darf.

Bei der MPL-Anpassung besteht die Gefahr, dass abwechselnd eine Reduzierung und Erhöhung von MPL-Werten erfolgt, so dass der primäre Engpass nur verlagert, jedoch nicht behoben wird. Um dies zu verhindern, erfolgt eine MPL-Erhöhung generell nur, wenn das Ziel des Donors deutlich übertroffen wird, d.h. der PI des Donors einen kritischen Wert unterschreitet.

6.2.4 Implementierung

Der WLM-Algorithmus wird durch die Java-Klasse *WLM* implementiert. Zur Instanzierung steht ein Konstruktor zur Verfügung:

```
WLM(ServiceClassList sclist , PerformanceList pfmlist )
```

Der erste Parameter ist ein *ServiceClassList*-Objekt. Es enthält alle *ServiceClass*-Objekte, die vom WLM gesteuert werden. Der zweite sind die gesammelten Performancedaten der Service-Klasse. *WLM* implementiert den Algorithmus durch Aufrufen der zentralen Methode *DRPolicy*. Das folgende Programm von *DRPolicy* zeigt die Realisierung des WLM-Algorithmus.

Listing 6.1: Code der zentralen Methode *DRPolicy*

```
1  public void DRPolicy( ){
2
3  int rcvIndex = getReceiver ();
4  int donorIndex = Cst.NO_DONOR;
5  int delayType = Cst.NO_DELAYTYPE;
6
7    if (rcvIndex!=Cst.NO_RECEIVER){
8        delayType = getBottleneck (pfmList.get(rcvIndex));
9        donorIndex = getDonor();
10
11       if (donorIndex!=Cst.NO_DONOR)
12       switch(delayType){
13           case Cst.DELAY_CPU: adjustPriority(rcvIndex,donorIndex);
14               break;
15           case Cst.DELAY_SWAP: adjustMPL(rcvIndex,donorIndex);
16       }
17    }
18    saveData(rcvIndex, donorIndex,delayType);
19 }
```

In *DRPolicy* werden die folgenden Methoden zur Auswahl des Receivers, zum Ermitteln und Beheben des Engpasses aufgerufen:

```
private int getReceiver ()
```

Die parameterlose Methode *getReceiver* dient zur Auswahl des Receivers. Sie liefert den Schlüssel des Receivers, wenn es einen Receiver gibt. Wenn es keinen Receiver gibt, ist der Rückgabewert eine Konstante *Cst.NO_RECEIVER*, sie ist in

der Java-Klasse *Cst* als Konstante definiert und bedeutet, dass es keinen Receiver gibt. Wenn keine Receiver ausgewählt sind, endet die Methode sofort.

private int getBottleneck (Performance perfOfServiceclass)

Mithilfe der Methode *getBottleneck* kann die Art des Engpasses beim Receiver festgestellt werden. *getBottleneck* erwartet die Performance des Receivers als Argument. Der Rückgabewert ist *Cst.DELAY_CPU*, falls ein CPU-Engpass ermittelt wird. Bei einem Swap-In-Engpass ist er *Cst.DELAY_SWAP*.

private int getDonor()

Mit *getDonor* kann der Donor ermittelt werden. Wenn es einen passenden Donor gibt, liefert diese Methode den Schlüssel des Donors, andernfalls gibt sie den Wert *Cst.NO_DONOR* zurück.

private void adjustPriority (**int** receiverID , **int** donorID)

private void adjustMPL(**int** receiverID , **int** donorID)

adjustPriority implementiert die in der Tabelle 6.3 gezeigte Massnahme zum Beheben des CPU-Engpasses, sie erwartet die Schlüssel von Receiver und Donor als Parameter. Analog zu *adjustPriority* implementiert *adjustMPL* die Massnahme zum Beheben des Swap-In-Engpasses in der Tabelle 6.4.

6.3 Hauptfunktion des gesamten Programms

Die zentrale Java-Klasse *Start* enthält die *main* Methode, die die Startmethode für jede Java-Applikation ist. Die folgende Abbildung 6.5 zeigt den Algorithmus der Hauptfunktion des gesamten Programms. Er basiert auf einer Schleife, die von Anfangszeit bis Endzeit durchlaufen wird. Zur Vorbreitung werden Workloads durch den Generator erzeugt. In der Schleife wird zuerst die Hauptspeicherverwaltung ausgeführt, die zum Ein-/Auslagern der Workload dient. Danach läuft die Ein-/Ausgabe-Verwaltung ab, dann arbeitet die CPU eine Zeiteinheit lang. Falls seit

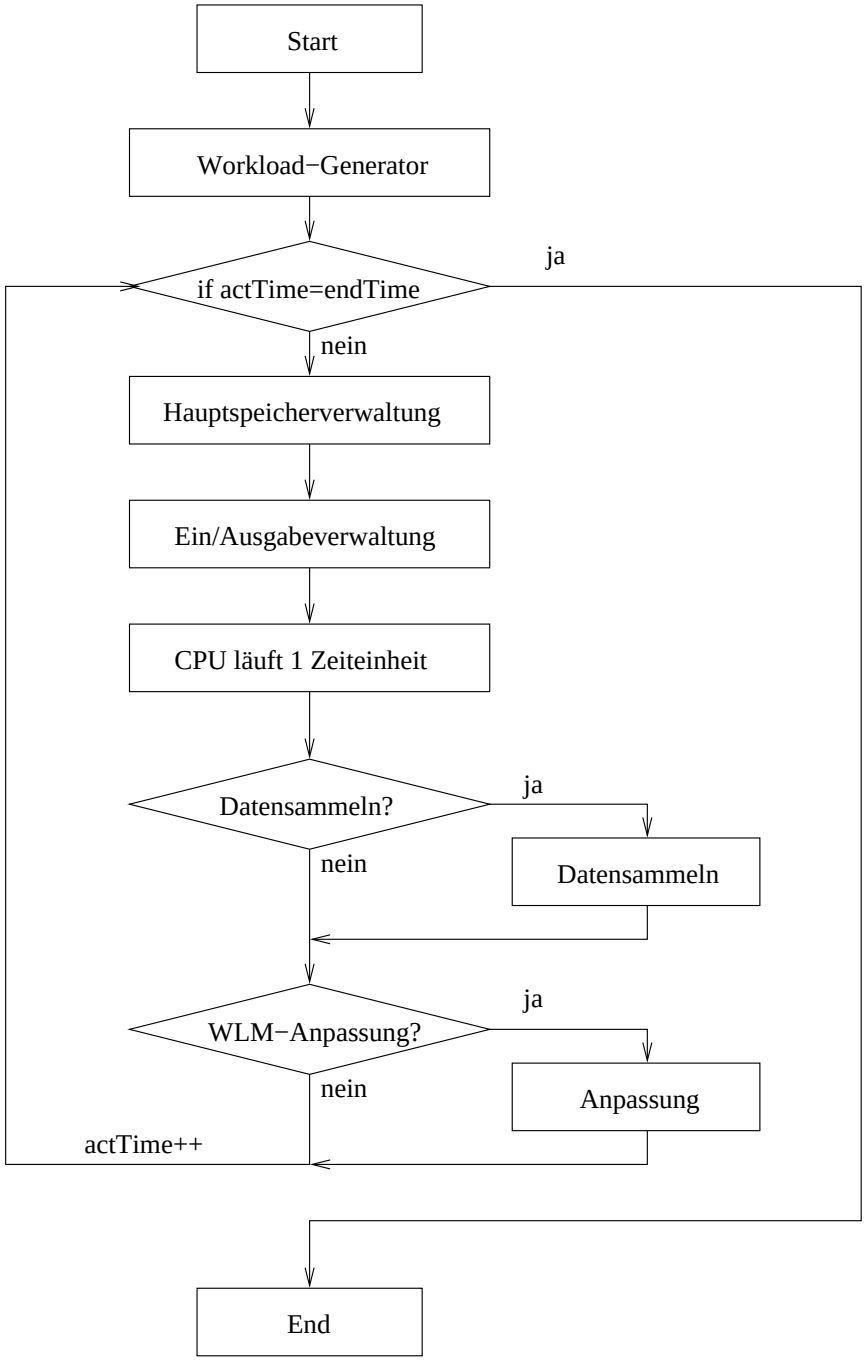

Abbildung 6.5: Der Algorithmus der Hauptfunktion *main*

dem letzten Sammeln von Daten 250ms vergangen sind, sammelt die WLM die aktuelle Daten im System. Alle 2s wird der WLM-Anpassungsalgorithmus durchlaufen.

Kapitel 7

Experimentelle Ergebnisse

Anschließend wollen wir zur Betrachtung der grundlegend WLM-Algorithmen ein Beispiel diskutieren. Es wird ein System betrachtet, auf dem Workloads von fünf Service-Klassen gemischt laufen. Im Abschnitt 7.1 wird das Simulationsszenario dargestellt. Die Definitionen der Service-Klassen werden im Abschnitt 7.2 beschrieben. Zum Schluß werden die Ergebnisse graphisch dargestellt.

7.1 Simulationsszenario

Das Simulationsszenario wird durch Systemparameter in der Java-Klasse *SysInfo* definiert. Die wichtigsten Parameter werden beim Experiment mit den Werten im folgenden Programm definiert, und ihre Bedeutung wird im folgenden aufgelistet.

Listing 7.1: Deklaration der Systemparameter

```
public static int startTime = 13;
public static int cpuNum = 16;
public static int periodInSecond = 181;
public static int timeInterval = 1000;
public static int sampleFrequency = 5;
public static int wlmFrequency = 2;
public static int wlmInterval = wlmFrequency * timeInterval ;
public static int periodInMS = periodInSecond * timeInterval ;
public static int ioDelay = 10;
public static int historyDataSize = 130;
```

- *startTime* definiert die Startzeit des Experiments.

- *cpuNum* ist die Anzahl der CPUs.
- *periodInSecond* ist die Dauer des Experiments in Sekunden.
- *periodInMS* ist die Dauer des Experiments in Taktzyklen.
- *timeInterval* ist die Zahl der Taktzyklen pro Sekunde.
- *sampleFrequency* ist die Zahl der Datensammlungen pro Sekunde.
- *wlmFrequency* definiert, wie oft in Sekunde der WLM-Anpassungsalogrithmus abläuft.
- *wlmInterval* stellt *wlmFrequency* in Taktzyklen dar.
- *ioDelay* definiert die Verzögerung der Ein-/Ausgabe-Operation.
- *historyDataSize* ist die Zahl der in der FIFO-Struktur gespeicherten, letzten PIs.

7.1.1 Ausgaben der Ergebnisse

Das WLM-Modell kann die Ergebnisse als Exceldateien ausgeben. Die Dateien enthalten verschiedene Informationen über die Vorgänge im Experiment und werden im folgenden aufgelistet.

- *Job.xls* enthält alle Arbeitseinheiten in der Reihenfolge ihrer Startzeit.
- *JobClassify.xls* enthält alle Arbeitseinheiten, sortiert nach Service-Klassen.
- *JobDistr.xls* enthält Informationen über die Zahl der neue gestarten Arbeitseinheiten pro Sekunde.
- *WLM.xls* enthält bei jeder WLM-Anpassung Informationen über PI, Engpasstyp, Receiver und Donor.
- *Performance.xls* enthält die Daten über den PI bei der Datensammlung.
- *PerformanceDetail.xls* enthält neben dem PI noch die Daten über die CPU- und Swap-In-Wartezeit der Service-Klassen bei der Datensammlung.
- *MPL.xls* enthält Informationen über Änderungen des MPLs und die Zahl der Arbeitseinheiten im Hauptspeicher und im Sekundärspeicher.
- *Readme.xls* enthält die Systemparameter.

7.2 Service-Klassen Definition für das Experiment

Die Definition der Service-Klassen wird in der Java-Klasse *ServiceClassList* implementiert. Die folgende Tabelle 7.1 zeigt, welche Service-Klassen und Ziele in dem Experiment definiert wurden.

Service−Klasse	ID	Zieltyp	Zielwert	Importance
scCICS1	0	Response Time	0.25 s	1
scTSO1	1	Response Time	0.4 s	2
scDB21	2	Response Time	0.3 s	3
scJES1	3	Execution Velocity	68	4
scJES2	4	Execution Velocity	28	5

Abbildung 7.1: Definition von Service-Klassen im Experiment

Es werden insgesamt 5 Service-Klassen definiert, davon drei Service-Klassen, die Antwortzeitziele haben und zwei haben ein *Execution Velocity* Ziel. Für die Batch-Job wurden 2 Service-Klassen definiert, einmal mit einem schnellen, einmal mit einem langsamen *Execution Velocity* Ziel. Für *scCICS, scTSO1* und *scDB21* sind Antwortzeitziele definiert. Die Service-Klasse *scCICS1* soll in dem Experiment-Szenario die höchste Importance erhalten, gefolgt von *scTSO1*, dann *scDB21* und der Batch-Job von *scJES1*. Die Batch-Job *scJES2* hat die niedrigste Importance. In diesem Modell wird keine System-Service-Klasse dargestellt, sie sind für das Experiment nicht relevant, da sie mit höherer Priorität laufen.

7.3 Workloadverteilung

Wir bezeichen als Arbeitseinheit die Komponenten, aus den die Workload für eine speiziale einzelne Service-Klasse besteht. Z.B. sind einzelne Transaktion die Einheit der Service-Klasse CICS.

In der folgenden Grafik 7.2 ist die Workloadverteilung pro Sekunde für die Arbeitseinheiten *scCICS1*, *scTSO1*, *scDB21*, *scJES1* und *scJES2* während des Experiment dargestellt.

Es werden ca. 50 Arbeitseinheiten *scCICS1* pro Sekunde gestartet. Durchschnittlich 30 neue Arbeitseinheiten *scTSO1* kommen pro Sekunde hinzu. Die Batch-Arbeitseinheiten *scJES1* und *scJES2* haben ein Verteilung mit dem Wert 1

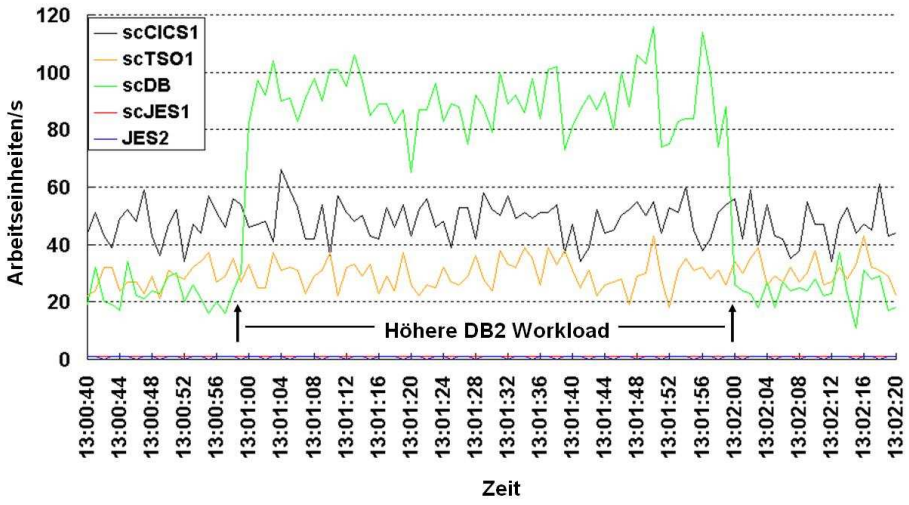

Abbildung 7.2: Workloadverteilung

pro Sekunde. Von 13:01:00 Uhr bis 13:02:00 Uhr ist eine große Anzahl (ca. 95) von Arbeitseinheiten *scDB21* angekommen und wird im System bearbeitet.

7.4 Ergebnisse

In der Grafik 7.3 sind die *Performance Indices* dargestellt. Auf der Abszisse ist die Zeit aufgetragen, während die Ordinate die zugehörigen PIs darstellt Ein Wert von 1 signalisiert Zielerfüllung. Ein Wert von größer als 1 bedeutet, dass das Ziel nicht erfüllt wird und ein Wert kleiner als 1, dass das Ziel übererfüllt ist.

Die Grafik 7.5 zeigt die entsprechenden Dispatch-Prioritätsänderungen der Service-Klassen. In der Grafik 7.6 sind die entsprechenden MPL-Änderungen der Service-Klassen dargestellt.

Wir sehen in der Grafik 7.3, dass die PIs von *scCICS1*, *scTSO1*, *scDB21*, *scJES1* und *scJES2* Schwankungen unterworfen ist. Diese werden durch WLM-Anpassungen hervorgerufen. Dabei werden die Parameter der Service-Klassen so verändert, dass nach Möglichkeit alle Service-Klassen ihre Ziele erfüllen können. Der WLM-Anpassungsalgorithmus läuft alle 2 Sekunden im Modell ab, d.h. er wird bei jeder geraden Sekunde durchgeführt, wie 13:00:56, 13:00:58, 13:01:00. . . .

Vor 13:01:00 Uhr sind die PIs der Service-Klassen stabil (1a. Pfeil in Grafik 7.3). Die Service-Klassen *scCICS1* und *scDB21* haben einen PI von ungefähr 0.5, während *scTSO1* und *scJES2* PIs von 0.4 aufweisen. Die Ziele für *scCICS1*, *scT-*

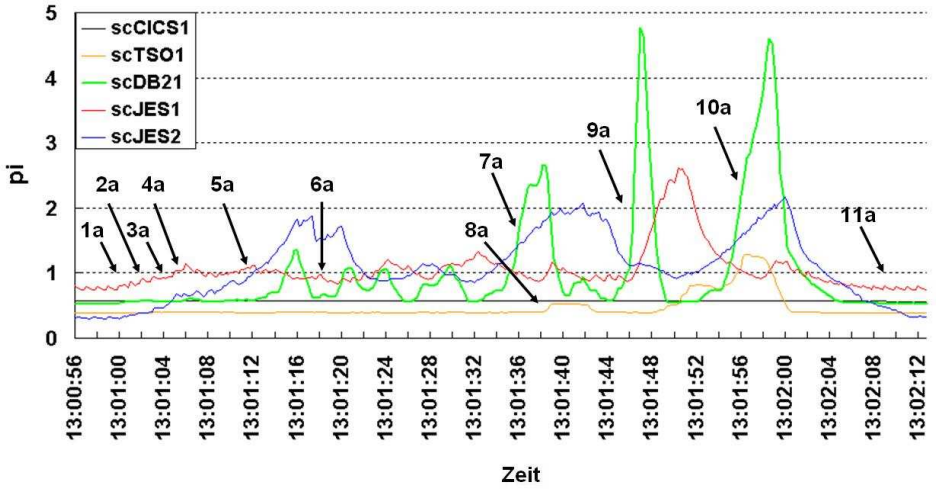

Abbildung 7.3: Performance Index

SO1, *scDB21* und *scJES2* sind nicht so eng gefasst, deshalb ist es auch möglich, die Ziele deutlich überzuerfüllen. Die Service-Klasse *scJES1* hat einen PI von ca. 0.8.

Betrachten wir noch einmal die Zielvorgaben für die *scJES1*-Service-Klasse. Mit einem Zielwert von 68 ist diese eng, da das maximale *Execution Velocity* Ziel 100 ist. Die Vorgabe kann zwar erreicht, aber nicht mehr wie bei den anderen Service-Klassen so deutlich übertroffen werden.

Nach 13:01:00 Uhr ändert sich die Situation, da die Arbeitseinheiten der Service-Klasse *scDB21* plötzlich sprunghaft angsteigen. Die PIs von *scJES1* und *scJES2* sind gestiegen, weil ihre Dispatch-Prioritäten niedriger als die von *scDB21* sind.

Um 13:01:02 Uhr und 13:01:04 Uhr wird er WLM führt die Behebung des Engpasses nicht durch (2a.und 3a. Pfeil), da keine Service-Klasse ihre Ziele übererfüllt, bedeutet das, dass kein Engpasses existiert.

Der 4a. Pfeil zeigt, dass der PI der Service-Klasse *scJES1* um 13:01:05 Uhr den Wert 1 übersteigt, d.h. *scJES1* kann ihr Ziel nicht mehr erfüllen. Der WLM beginnt jetzt gegenzusteuern. Um 13:01:06 Uhr läuft der WLM-Anpassungsalgorithmus ab (siehe Grafik 7.6). *scJES1* wird als Receiver ausgewählt und *scJES2* als Donor. Der 4c. Pfeil in Grafik 7.6 zeigt, dass der WLM den MPL-Wert der Service-Klasse *scJES2* von 30 auf 74 erhöht und gleichzeitig den MPL-Wert der Service-Klasse *scJES1* von 30 auf 29 reduziert. Nach der Anpassung hat *scJES1* einen PI von ca. 1. Wir betrachten folgend nur wichtigen Zeitpunkt.

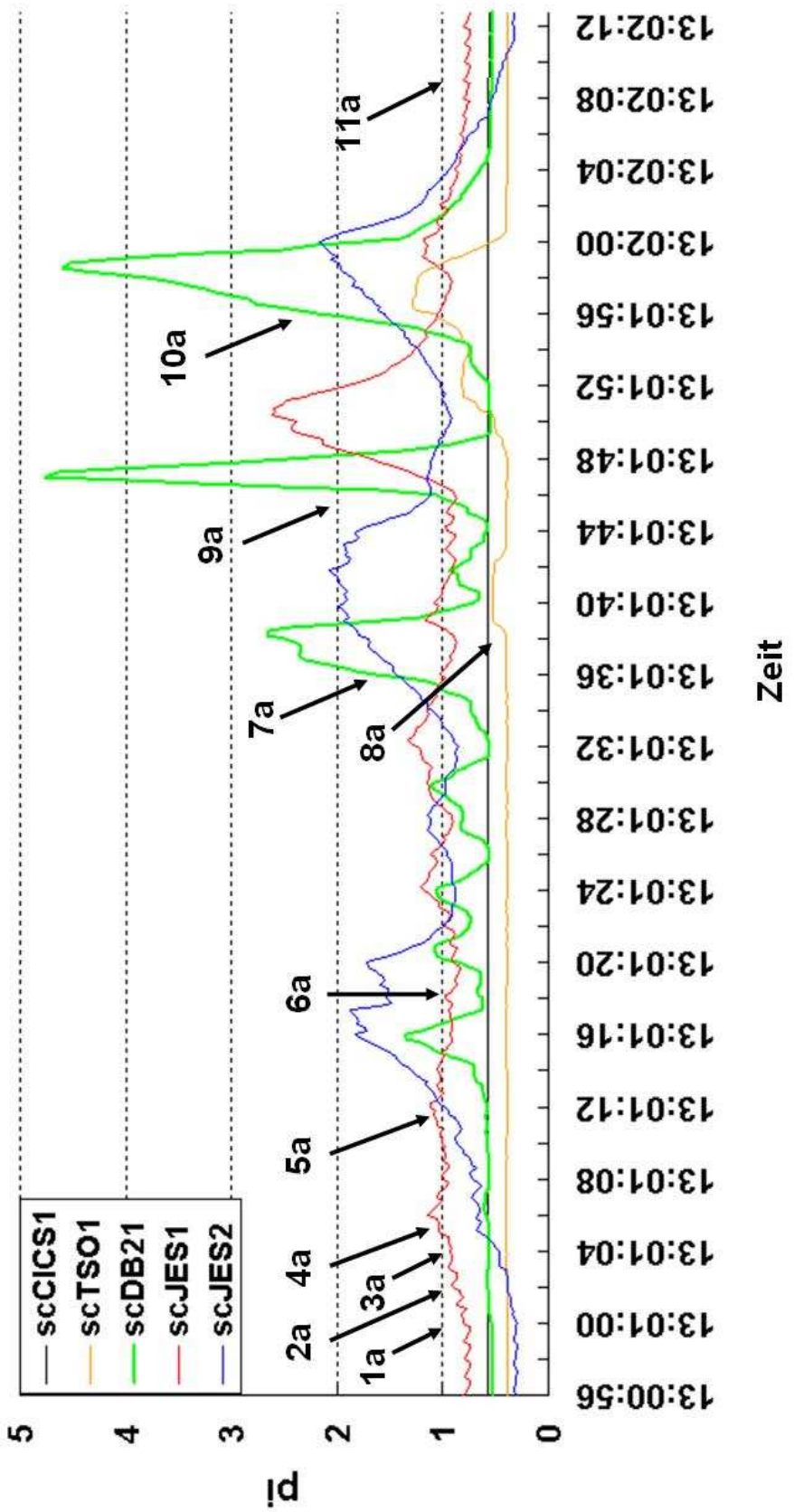

Abbildung 7.4: Performance Index

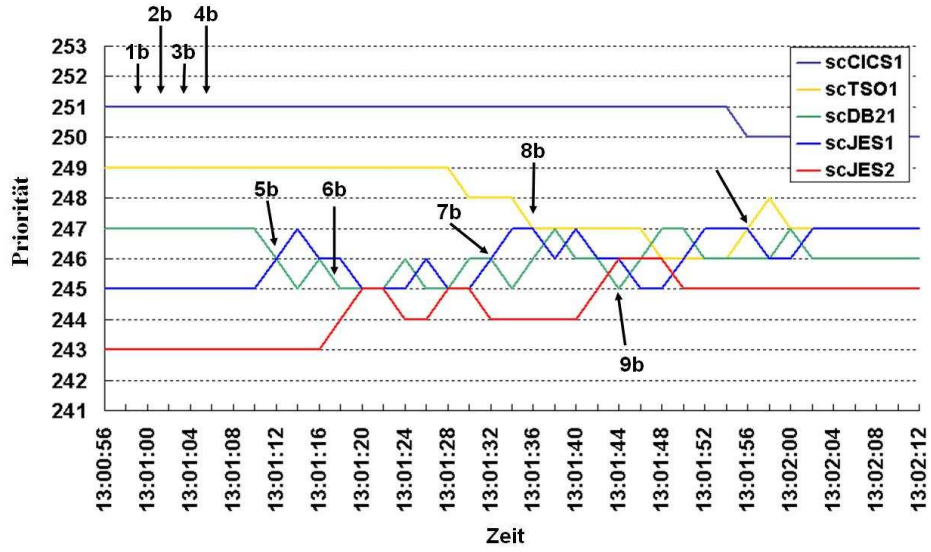

Abbildung 7.5: Prioritätsänderung der Service-Klassen

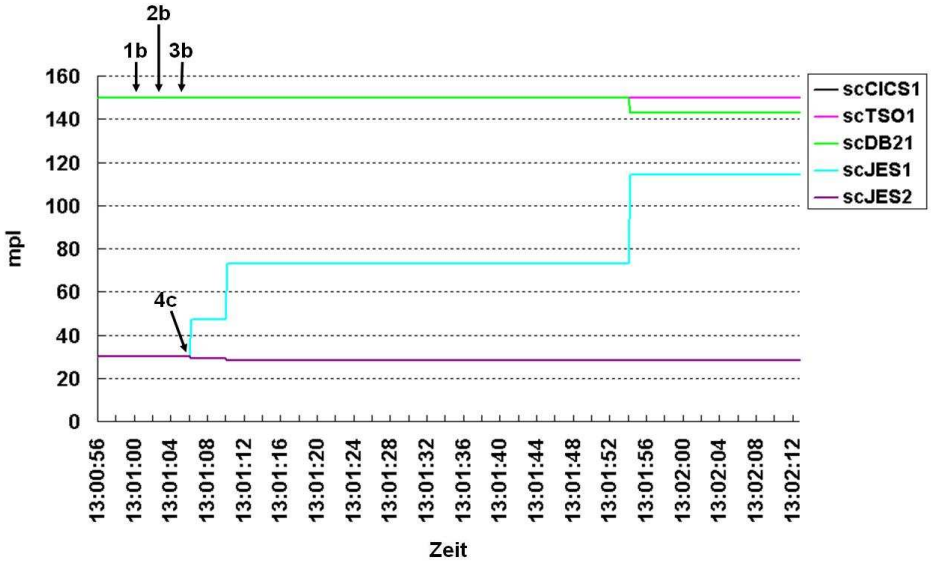

Abbildung 7.6: MPL-Änderung der Service-Klasse

Vor 13:01:12 Uhr gibt es keinen Einfluss auf *scCICS1*, *scTSO1*, *scDB21*, wie der 2. Pfeil zeigt, da die Anpassungen sich nur auf *scJES1* und *scJES2* beziehen. Um 13:01:12 Uhr gibt es zwei Service-Klassen *scJES1* und *scJES2* mit einem PI größer als 1. WLM hat *scJES1* als den Receiver und *scDB21* als den Donor ausgewählt. Die Priorität der *scDB21* wird von 247 auf 246 verringert und die Priorität der *scJES1* wird gleichzeitig von 245 auf 246 erhöht (5b. Pfeil in Grafik 7.5), daraufhin ist der PI der *scDB21* schnell schlecht geworden. *scJES1* hat den besser Zugang zu Betriebsmitteln und ihr PI sinkt unter 1. Der PI der *scJES2* steigt weiter, da ihr Zugang zu Betriebsmitteln von dieser Anpassung nicht beeinflusst wird.

Beachten wir den Zeitpunkt um 13:01:18 Uhr (6a. Pfeil in Grafik 7.3), so erkennen wir, dass *scJES1* zwar ihr Ziel erreicht hat, aber sie nicht als Donor ausgewählt wird, da ihr PI nicht den kritischen Wert 0.9 unterschreitet. Bei der Anpassung ist *scJES2* der Receiver und *scDB21* der Donor (6b. Pfeil). Die Priorität der *scDB21* wird von 246 auf 245 verringert und die Priorität der *scJES2* wird gleichzeitig von 243 auf 244 erhöht.

Von 13:01:34 Uhr bis 13:01:38 Uhr steigt der PI der *scDB21* nahezu ungebremst (7a. Pfeil). Die Ursache ist aus Grafik 7.5 ersichtlich. Die Anpassung um 13:01:34 Uhr konnte keine Beziehung unter den Service-Klassen verändern (7b. Pfeil), d.h. eine Anpassung allein reicht nicht aus, um einen besseren Zugang zu Betriebsmitteln zu bekommen und mehrmalige WLM-Anpassungen werden nötig. Nach zwei Anpassungen wird der Engpass der *scDB21* behoben. Der Vorgang dauert 4 Sekunden. Angesichts der Zielvorgabe von 0.3 Sekunden ist dies ein langer Zeitraum, deshalb steigt der PI sehr schnell.

Die WLM-Anpassung hat um 13:01:38 auch den PI der *scTSO1* beeingeflusst (8a. Pfeil). *scTSO1* wird als der Donor ausgewählt, da sie ihr Ziel übererfüllt und die niedrigste Importance aus der Menge von Service-Klassen mit einem PI kleiner als 1 hat. Der Receiver ist *scDB21*. Die Priorität der *scTSO1* wird von 248 auf 247 verringert und die Priorität der *scDB21* wird gleichzeitig von 245 auf 246 erhöht (8b. Pfeil).

Um 13:01:44 Uhr (Pfeil 9a) ist der PI der *scDB21* nochmal deutlich gestiegen, da ihre Priorität von 246 auf 245 reduziert wird (9b. Pfeil). Es ist zwar nur eine Reduktion um 1, aber die Beziehung von *scDB21* zu allen anderen Service-Klassen hat sich stark verändert. Nach der Anpassung ist die Priorität der *scDB21* die Niedrigste von allen fünf Service-Klassen. Arbeitseinheiten der *scDB21* müssen sehr lange warten, bis sie einen Prozessor zugeteilt bekommen.

Obwohl der PI der *scDB21* ab 13:01:54 Uhr weiter angestiegen ist (10a. Pfeil),

haben die WLM-Anpassungen um 13:01:56 Uhr und um 13:01:58 Uhr *scDB21* nicht berücksichtigt. Weil die Service-Klasse *scTSO1* eine höhere Importance als *scDB21* aufweist und ihr Ziel nicht erfüllt hat, wird *scTSO1* als Receiver in beiden Anpassungen ausgewählt (10b. Pfeil). Im Original-z/OS-WLM können mehrere Receiver gleichzeitig ausgewählt werden.

Um etwa 13:02:12 Uhr endet das Experiment (11a. Pfeil).

Offensichtlich ist die Reaktion des WLM unbefriedigend, da extreme Oszillationen auftreten. Im Rahmen der Arbeit wurde kein Versuch gemacht, einen optimaleren algorithmus zu entwickeln.

Informationen auf Standard-Ausgabe

Das WLM-Modell kann die Informationen über die WLM-Anpassungen auf der Standard-Ausgabe während dem Durchführen anzeigen. Der Anhang B zeigt die Informationen der Standard-Ausgabe während des Experiements.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Modellierung eines Goal orientierten Workload-Managers vorgestellt. Dieses Modell erfasst nicht alle Funktionalitäten des Original-z/OS-WLMs, sondern nur diejenigen, die für ein besseres Verständnis der komplexen Zusammenhänge des WLMs relevant erscheinen.

Um den z/OS WLM zu simulieren, wurde im ersten Schritt eine Laufzeitumgebung bzw. ein Rechnersystem für den WLM modelliert. Der Modellrechner enthält die typischen Hardware-Komponenten eines Großrechners. Dies sind die CPUs, der Speicher und die Platten des Ein-/Ausgabe-Subsystems. Der WLM verwaltet den Betriebsmittelzugang durch die Festlegung von Prioritäten für die CPU. Der Zugang zum Speicher wird durch die Definition eines *Multi Programming Levels*, d.h. der Anzahl der im Speicher gehaltenen Adressräumen einer Service-Klasse, gesteuert. Deshalb werden zwei zentrale Parameter, Dispatching-Prioritäten und der MPL bzw. die zwei damit korrespondierenden Engpässe, der CPU-Delay und der Swap-In-Delay in dieser Arbeit modelliert.

Neben dem Modellrechner werden die Workloadverteilung und die Definitionen der Service-Klassen modelliert. Workloads werden vom Workloadgenerator erzeugt und laufen im System ab. Sie benutzen Betriebsmittel. Die Ziele der Service-Klassen werden vom Benutzer definiert.

Im zweiten Schritt wurde der WLM modelliert. Das Modell erfasst die grundlegenden Verfahren des z/OS-WLMs für die Steuerung des Systems. Der WLM im Sysplex ist nicht berücksichtigt, d.h. die Entscheidung des WLMs muss unabhängig davon getroffen werden, ob das z/OS-System Teil einer Sysplex-Umgebung ist. Um Zielerfüllungsprobleme automatisch zu erkennen, wird ein *Performance Index* verwendet. Um die Ursachen für Zielerfüllungsprobleme zu ermitteln, berechnet der WLM die Art des Engpasses. Abhängig vom Engpass-Typ wird als nächstes

ein Algorithmus ausgewählt, der den Engpass beheben soll.

Das WLM-Modell kann die Ergebnisse als Exceldateien ausgeben. Die Dateien enthalten verschiedene Informationen über die Vorgänge im Experiment. Mit der graphischen Darstellung der Ergebnisse wurde ein besseres Verständnis der komplexen Zusammenhänge des Workload-Managers erreicht.

Das Modell ist sinnvoll, um die komplexen Zusammenhänge besser zu verstehen. Eine Erweiterung der Arbeit könnte entlang von zwei unterschiedlicher Zielrichtungen verfolgen. 1. Die in Abbildung 7.3 sichtbaren Oszillierenden PI könnten mit Sicherheit durch einen anderen WLM-Algorithmus verbessert werden. Auch würden Variationen in dem Algorithmus hilfreich sein, um das Verhalten des WLMs besser zu verstehen. 2. Das entwickte Modell ist relativ primtiv. Zur Erweiterung des Modell können folgende Punkte berücksichtigt werden:

- Ein/Ausgabe Prioritäten,
- Paging,
- Sysplex.

Literaturverzeichnis

[AEE+97] J. Aman, C. K. Eilert, D. Emmes, P. Yocom und D. Dillenberger. *adaptive algorithms for managing a distributed data processing workload*. IBM Systems journal, No 2, 1997. Vol. 36, Seiten 242-283.

[BCD+05] P. Bari, P. Cassier, A. Defendi, J. Hutchinson, A. Maneville, G. Membrini und C. Ong. *System Programmer's Guide to: Workload Manager*. IBM Redbooks. International Technical Support Organization, 2. Auflage, September 2005. ISBN 0738493511. SG24-6472-01.

[BF05] U. Böttcher und D. Frischalowski. *Java 5 Programmierhandbuch, Einstieg und professioneller Einsatz*. Software & Support Verlag, 02 2005. ISBN 3-935042-63-9.

[bib03] *MVS Initialization and Tuning Guide*. IBM, 3. Auflage, 2003. ISBN Publication No. SA22-7591-01.

[bib05a] *MVS Programming: Workload Management Services*. z/OS. IBM, 2005. Publication No. SA22-7619-09.

[bib05b] *z/OS MVS Planning: Workload Managerment*. IBM Deutschland Entwicklung GmbH, 11. Auflage, September 2005. Publication No. SA22-7602-10.

[EOO04] M. Ebbers, W. O'Brien und B. Ogden. *Interactive System Productivity Facility (ISPF) User's Guide Volume I*. z/OS. 4. Auflage, September 2004. Publication No. SA34-4822-03.

[EOO05] M. Ebbers, W. O'Brien und B. Ogden. *Introduction to the New Mainframe: z/OS Basics*. International Technical Support Organization, October 2005. Publication No. SA24-6366-00.

[Gla06] Eduard Glatz. *Betriebssysteme: Grundlagen, Konzepte, Systemprogrammierung*. dpunkt.verlag, 2006. ISBN 3-89864-355-7.

[HKS04] P. Herrmann, U. Kebschull und W. G. Spruth. *Einführung in z/OS und OS/390*. Oldenbourg Wissenschaftsverlag GmbH, 2. Auflage, 2004. ISBN 3-486-27393-0.

[Krü05] Guido Krüger. *Handbuch der Java-Programmierung*. Addison-wesley Verlag, 4. Auflage, 2005. ISBN 3-8273-2201-4.

[Tan94] Andrew S. Tanenbaum. *Moderne Betriebssysteme*. Carl Hanser und Prentice-Hall International, 1994. ISBN 3-446-17472-9.

[TV04] M. Teuffel und R. Vaupel. *Das Betriebssystem z/OS und die zSeries*. Oldenbourg Wissenschaftsverlag GmbH, 2004. ISBN 3-486-27528-3.

[Ull06] Christian Ullenboom. *Java ist auch eine Insel Programmieren mit der Java Standard Edition Version 5*. Galileo Computing, 5. Auflage, 2006. ISBN 3-89842-747-1.

Anhang A

Glossar

APPC	Advanced Program to Program Communication
ARM	Automatic Restart Manager
ASM	Auxiliary Storage Manager
CICS	Customer Information Control System
CPU	Central Processing Unit
DASD	Direct Access Storage Device
DDF	Distributed Data Facility
FICON	Fiber Connectivity
ICF	Integrated Coupling Facility
IMS	Information Management System
IOCP	I/O Configuration Program
ISPF	Interactive System Productivity Facility
JCL	Job Control Language
JES	Job Entry Subsystem
LPAR	Logical Partition

MCM	Multi-Chip Module
MPL	Multi Programming Level
MVS	Multiple Virtual Storage
OMVS	Open MVS (siehe MVS)
OS	Operating System
RSM	Real Storage Manager
SRM	System Resource Manager
TSO	Time Sharing Option
VSM	Virtual Storage Manager
WLM	Workload Manager

Anhang B

Informationen auf Standard-Ausgabe

Der folgende Code zeigt die Informationen der Standard-Ausgabe während des Experiements.

```
−−− JobQueue print
*** scheduling
−−−−−−−−−−−−−−− time 13:0:2 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:0:4 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:0:6 −−−−−−−−−−−−−−−
                ~ ~
−−−−−−−−−−−−−−− time 13:1:0 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:1:2 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:1:4 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:1:6 −−−−−−−−−−−−−−−
   [scJES1] mpl:30   up:17    new mpl:47  [up]
   [scJES2] mpl:30   down:1  new mpl:29  [down]

−−−−−−−−−−−−−−− time 13:1:8 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:1:10 −−−−−−−−−−−−−−−
   [scJES1] mpl:47   up:26    new mpl:73  [up]
   [scJES2] mpl:29   down:1  new mpl:28  [down]

−−−−−−−−−−−−−−− time 13:1:12 −−−−−−−−−−−−−−−
   (scJES1)  prio:245      new prio:246      (up)
   (scDB21) prio:247      new prio:246      (Down)

−−−−−−−−−−−−−−− time 13:1:14 −−−−−−−−−−−−−−−
   (scJES1)  prio:246      new prio:247      (up)
   (scDB21) prio:246      new prio:245      (Down)

−−−−−−−−−−−−−−− time 13:1:16 −−−−−−−−−−−−−−−
   (scDB21) prio:245      new prio:246      (up)
   (scJES1)  prio:247      new prio:246      (Down)
```

−−−−−−−−−−−−−−− time 13:1:18 −−−−−−−−−−−−−−−
(scJES2) prio:243 **new** prio:244 (up)
(scDB21) prio:246 **new** prio:245 (Down)

−−−−−−−−−−−−−−− time 13:1:20 −−−−−−−−−−−−−−−
(scJES2) prio:244 **new** prio:245 (up)
(scJES1) prio:246 **new** prio:245 (Down)

−−−−−−−−−−−−−−− time 13:1:22 −−−−−−−−−−−−−−−
−−−−−−−−−−−−−−− time 13:1:24 −−−−−−−−−−−−−−−
(scDB21) prio:245 **new** prio:246 (up)
(scJES2) prio:245 **new** prio:244 (Down)

−−−−−−−−−−−−−−− time 13:1:26 −−−−−−−−−−−−−−−
(scJES1) prio:245 **new** prio:246 (up)
(scDB21) prio:246 **new** prio:245 (Down)

−−−−−−−−−−−−−−− time 13:1:28 −−−−−−−−−−−−−−−
(scJES2) prio:244 **new** prio:245 (up)
(scJES1) prio:246 **new** prio:245 (Down)

−−−−−−−−−−−−−−− time 13:1:30 −−−−−−−−−−−−−−−
(scDB21) prio:245 **new** prio:246 (up)
(scTSO1) prio:249 **new** prio:248 (Down)

−−−−−−−−−−−−−−− time 13:1:32 −−−−−−−−−−−−−−−
(scJES1) prio:245 **new** prio:246 (up)
(scJES2) prio:245 **new** prio:244 (Down)

−−−−−−−−−−−−−−− time 13:1:34 −−−−−−−−−−−−−−−
(scJES1) prio:246 **new** prio:247 (up)
(scDB21) prio:246 **new** prio:245 (Down)

−−−−−−−−−−−−−−− time 13:1:36 −−−−−−−−−−−−−−−
(scDB21) prio:245 **new** prio:246 (up)
(scTSO1) prio:248 **new** prio:247 (Down)

−−−−−−−−−−−−−−− time 13:1:38 −−−−−−−−−−−−−−−
(scDB21) prio:246 **new** prio:247 (up)
(scJES1) prio:247 **new** prio:246 (Down)

−−−−−−−−−−−−−−− time 13:1:40 −−−−−−−−−−−−−−−
(scJES1) prio:246 **new** prio:247 (up)
(scDB21) prio:247 **new** prio:246 (Down)

−−−−−−−−−−−−−−− time 13:1:42 −−−−−−−−−−−−−−−
(scJES2) prio:244 **new** prio:245 (up)
(scJES1) prio:247 **new** prio:246 (Down)

−−−−−−−−−−−−−−− time 13:1:44 −−−−−−−−−−−−−−−
(scJES2) prio:245 **new** prio:246 (up)
(scDB21) prio:246 **new** prio:245 (Down)

−−−−−−−−−−−−−−− time 13:1:46 −−−−−−−−−−−−−−−

```
   (scDB21) prio:245      new prio:246      (up)
   (scJES1)  prio:246     new prio:245      (Down)

 ---------- time 13:1:48 ----------
   (scDB21) prio:246      new prio:247      (up)
   (scTSO1) prio:247      new prio:246      (Down)

 ---------- time 13:1:50 ----------
   (scJES1)  prio:245     new prio:246      (up)
   (scJES2)  prio:246     new prio:245      (Down)

 ---------- time 13:1:52 ----------
   (scJES1)  prio:246     new prio:247      (up)
   (scDB21) prio:247      new prio:246      (Down)

 ---------- time 13:1:54 ----------
   [scJES1]  mpl:73   up:41     new mpl:114 [up]
   [scDB21] mpl:150 down:7  new mpl:143 [down]

 ---------- time 13:1:56 ----------
   (scTSO1) prio:246      new prio:247      (up)
   (scCICS1) prio:251     new prio:250      (Down)

 ---------- time 13:1:58 ----------
   (scTSO1) prio:247      new prio:248      (up)
   (scJES1)  prio:247     new prio:246      (Down)

 ---------- time 13:2:0 ----------
   (scDB21) prio:246      new prio:247      (up)
   (scTSO1) prio:248      new prio:247      (Down)

 ---------- time 13:2:2 ----------
   (scJES1)  prio:246     new prio:247      (up)
   (scDB21) prio:247      new prio:246      (Down)

 ---------- time 13:2:4 ----------
 ---------- time 13:2:6 ----------
 ---------- time 13:2:8 ----------
          ~ ~
 ---------- time 13:2:58 ----------
 ---------- time 13:3:00 ----------
  >>>print  perfomance.xls
  >>>print  performanceDetail . xls
  >>>print  mpl.xls
  >>>print  readme.xls
>>> Program End <<<
```

Anhang C

Inhalt der Beigefügten DVD

Die beigefügte DVD weist folgende Verzeichnisstruktur auf:

- Verzeichnis \Diplomarbeit:
  Enthält die Diplomarbeit als PDF-Dokument.
- Verzeichnis \Software: Enthält die verwendete Software Eclipse für Windows.
- Verzeichnis \Source: Enthält den Quellcode des im Rahmen der Diplomarbeit geschriebenen Porgramms.