

Smarter software for a smarter planet

IBM smarter

# Vorstellungsrunde

- Marc Bauer
- IBM Software Group Service for System z Software
- Aufgaben:
  - WebSphere Application Server Produktfamilie
  - Installation und Konfiguration
  - Java Problem Determination
  - Java Performance Tuning

# Agenda

- Vorstellungsrunde
- Einführung
- Vorstellung von verschiedenen Tools
- Gedanken zu Java und WAS-Konfigurationen
- Performance-Probleme (mit Fallbeispielen)
- Server-Abstürze (mit Fallbeispielen)
- Zusammenfassung

# Einführung

## Betriebsfreundliche Anwendungen

- Anwendungsentwickler vs. Betrieb
- Standardisierter Fehlerkatalog
  - Abgestimmte Fehlercodes
  - Abgestimmte Reaktionen auf auftretende Fehler
- Meldung im Log bedeutet: “Betrieb hilf!”
  - Keine anwendungsrelevanten Ausgaben: z.B. “Produkt 17 wurde ausgewählt”
  - Keine Exceptions, die man ignorieren soll
- Besser: eigenes Log für fachliche Informationen und Exceptions, die die Entwickler später analysieren möchten
- Folgen:
  - Einfacheres Monitoring
  - Eindeutige Zuordnung der Zuständigkeiten
  - Weniger Missverständnisse beim Betrieb, da Fehlermeldungen bekannt sind
  - Erleichterung des Betriebs
  - Standardisierung
- Standardisierungsgrad ist maßgebend für den Automatisierungsgrad!

## Nichtfunktionale Anforderungen

- Funktionale Anforderungen sind wichtig
- Jedoch werden die nicht-funktionalen Anforderungen an Software oft wenig betrachtet
- Folgende Fragen sollten jedoch zwingend geklärt werden:
  - In welcher Zeit soll ein Request bearbeitet werden abhängig vom Requesttyp?
  - Wieviele eingeloggte Benutzer sollten maximal unterstützt werden?
  - Wieviele parallel arbeitende Benutzer sollten unterstützt werden?
  - Wieviele echt parallele Requests sollen vom System bearbeitet werden können?
- Nur so ist ein Abgleich von IST- und SOLL-Zustand möglich

- Java Entwicklung:
  - Ist geprägt von permanentem Kostendruck
  - Anwendungen müssen schnell auf den Markt gebracht werden
  - unzählige Frameworks werden während der Entwicklung eingesetzt
  - „Elegante“ Lösung steht oft vor Performance
- Eingekaufte Software wird unzureichend getestet
  - Nachimplementierungen kosten extra → Betrieb muss dies ausbaden
- Frameworks
  - Sind oft sehr generisch
  - Sind selten auf Enterprise-Betrieb ausgelegt
  - Werden zu selten hinterfragt
- Jedoch: Werkzeuge zur Fehleranalyse und Performanceverbesserung sind vorhanden
  - Java bietet auf Grund seiner JVM sehr angenehme Dumps
  - Java gibt viel über den laufenden Code preis
  - Java ist in der Lage performant zu arbeiten!

## Antwortzeiten

- Gefühlte mittlere Antwortzeit ist die, die schlechter ist, als 90% aller Anfragen
- Es kommt also nicht auf die mittlere Antwortzeit an, sondern auf das Vermeiden der Ausreißer
- Eine Verbesserung der Antwortzeit wird erst wahrgenommen, wenn sie signifikant ist (>20%)
- Daher Verbesserungen sammeln und konsolidiert verteilen
- Wenn sich Verbesserungen finanziell auswirken, ist dies eine andere Geschichte

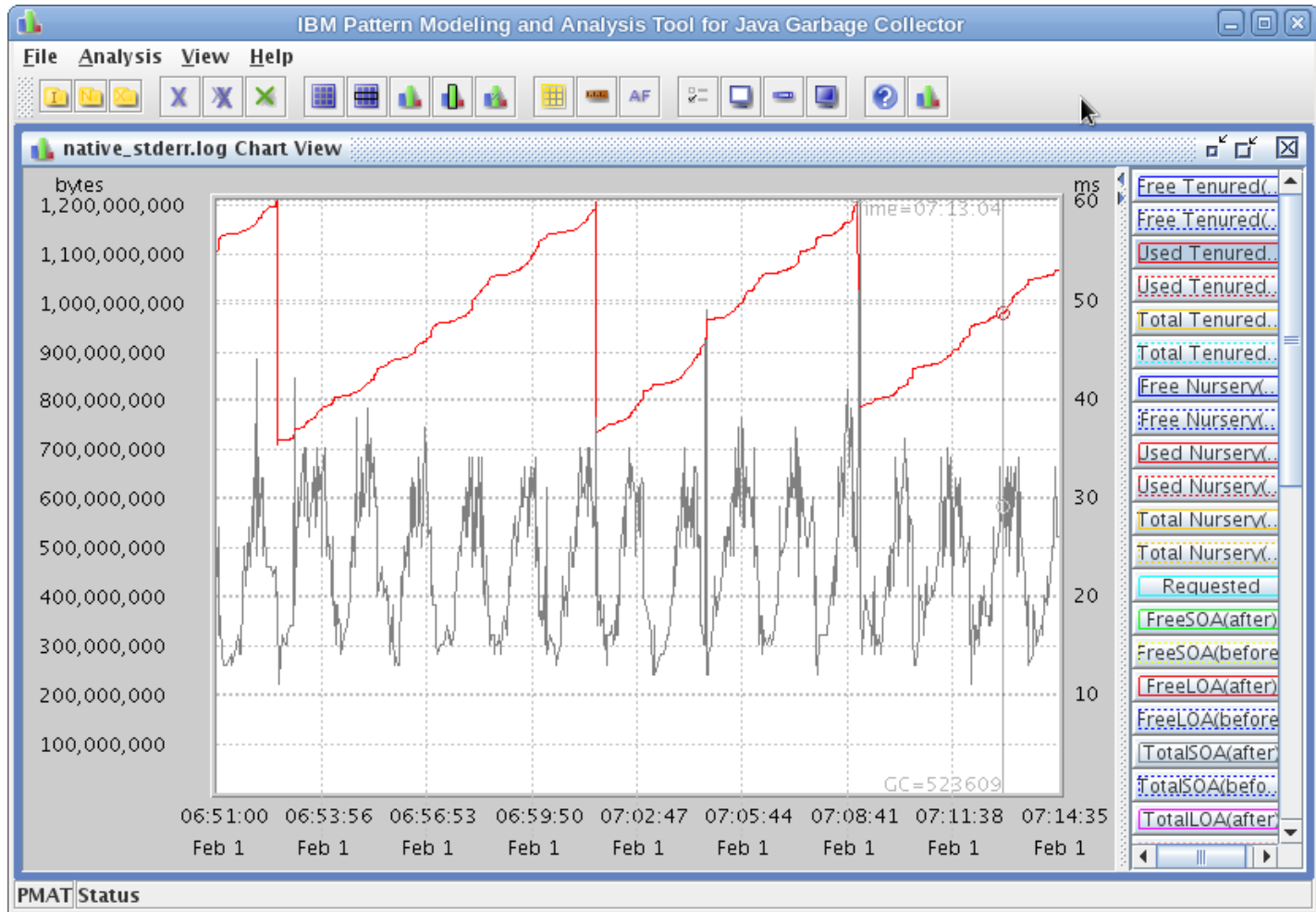


# Tool Time

## PMAT (Garbage Collection Analyzer)

- PMAT dient zur grafischen Aufbereitung von Ausgaben der Garbage Collection
- Bei jeder Garbage Collection wird eine entsprechende Ausgabe in das Log geschrieben, wenn der JVM-Parameter `-verbose:gc` aktiviert ist
- Diese Ausgaben beinhalten unter anderem Informationen über Zeitpunkt und Dauer der Garbage Collection, sowie Informationen über den Zustand des Java Heaps vor und nach der Collection
- Diese Informationen werden als XML geliefert und sind zur Analyse von Laufzeitproblemen und zur Ermittlung des Gesundheitszustands des Systems äußerst wichtig
- Die Garbage Collection Ausgaben sollten vor allem bei interaktiven Systemen standardmäßig aktiviert werden
- PMAT ist mit weiteren Tools im IBM Support Assistant kostenfrei erhältlich

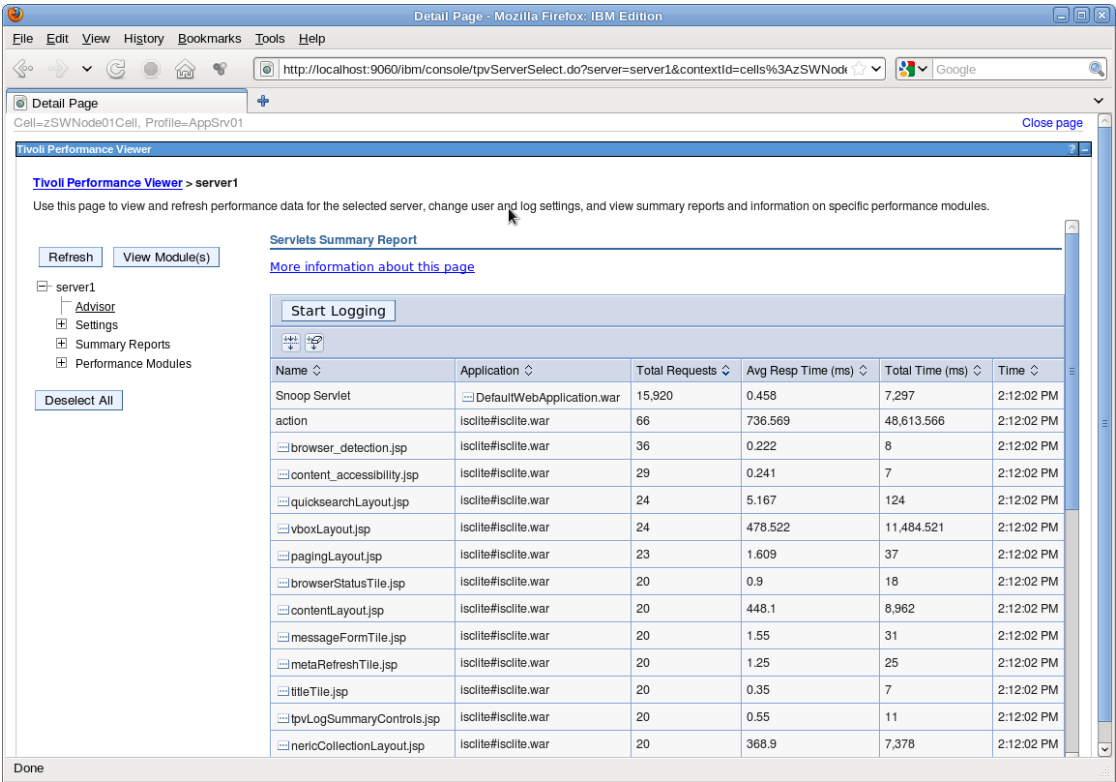
## PMAT (Garbage Collection Analyzer)



## Tivoli Performance Viewer und Request Metrics

- Der Tivoli Performance Viewer und die Request Metrics sind Features des WebSphere Application Servers
- Sie bieten Informationen über die Systemauslastung (Poolsizes, Thread Pools)
- Kaum Overhead
- Leider keine Request bezogenen Daten
- Eher als Monitoring der Laufzeitumgebung (WAS) gedacht, nicht der Anwendung
- Request Metrics bieten Laufzeitanalysen einzelner Requests
- Request Metrics kommen mit nennenswertem Overhead
- Können Laufzeit auf Servlet, EJB, Datenbank-Ebene aufzeigen
- Sind eingebettet in WAS internen Code
- Kein Monitoring von Frameworks wie Spring möglich, da sie nicht durch den WAS internen Code laufen

# Tivoli Performance Viewer und Request Metrics



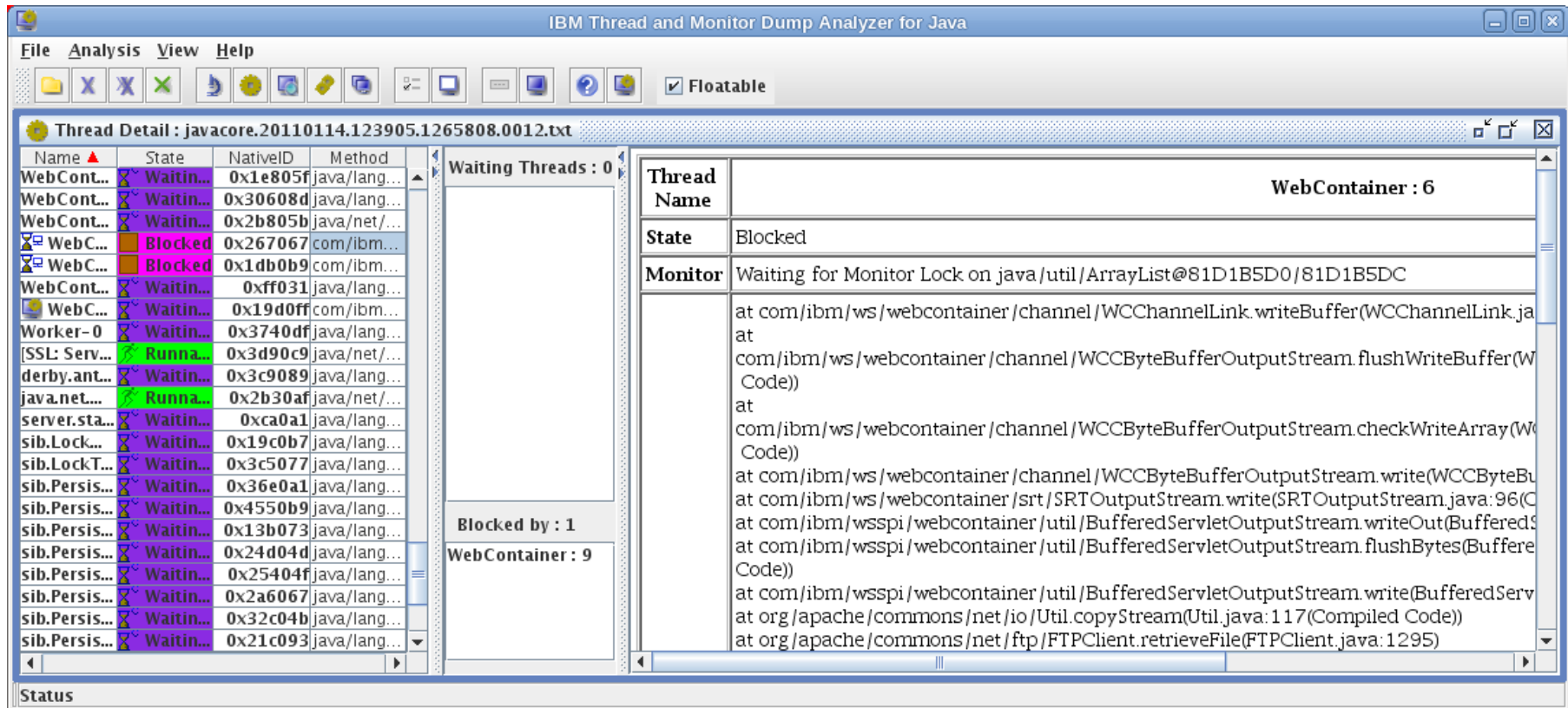
Message: BBOO0222I: PMRM0003I: parent:ver=1,ip=192.168.7.241,time=1299675583070,pid=000001f800000016,reqid=89,event=1 - current:ver=1,ip=192.168.7.241,time=1299675583070,pid=000001f800000016,reqid=117,event=1 type=EJB detail=com.ibm.websphere.samples.trade.ejb.TradeBean.updateQuotePriceVolume elapsed=921

Message: BBOO0222I: PMRM0003I: parent:ver=1,ip=192.168.7.241,time=1299675583070,pid=000001f800000016,reqid=77,event=1 - current:ver=1,ip=192.168.7.241,time=1299675583070,pid=000001f800000016,reqid=89,event=1 type=URI detail=/trade/app elapsed=2062

## TMDA (Thread and Monitor Dump Analyzer)

- TMDA dient zur Analyse von javacore Dumps (auch Thread Dumps genannt)
- Ein javacore Dump ist ein sehr kleiner Dump (<2MB)
- Stellt einen Snapshot dar
- Er enthält folgende Informationen:
  - Welche Java Threads existieren?
  - In welcher Stelle des Programmcodes befinden sich die jeweiligen Threads?
  - Welcher Thread hält Locks auf welche Java Objekte?
  - Informationen über die JVM Parametrisierung
- TMDA ist ebenfalls im IBM Support Assistant enthalten

## TMDA (Thread and Monitor Dump Analyzer)

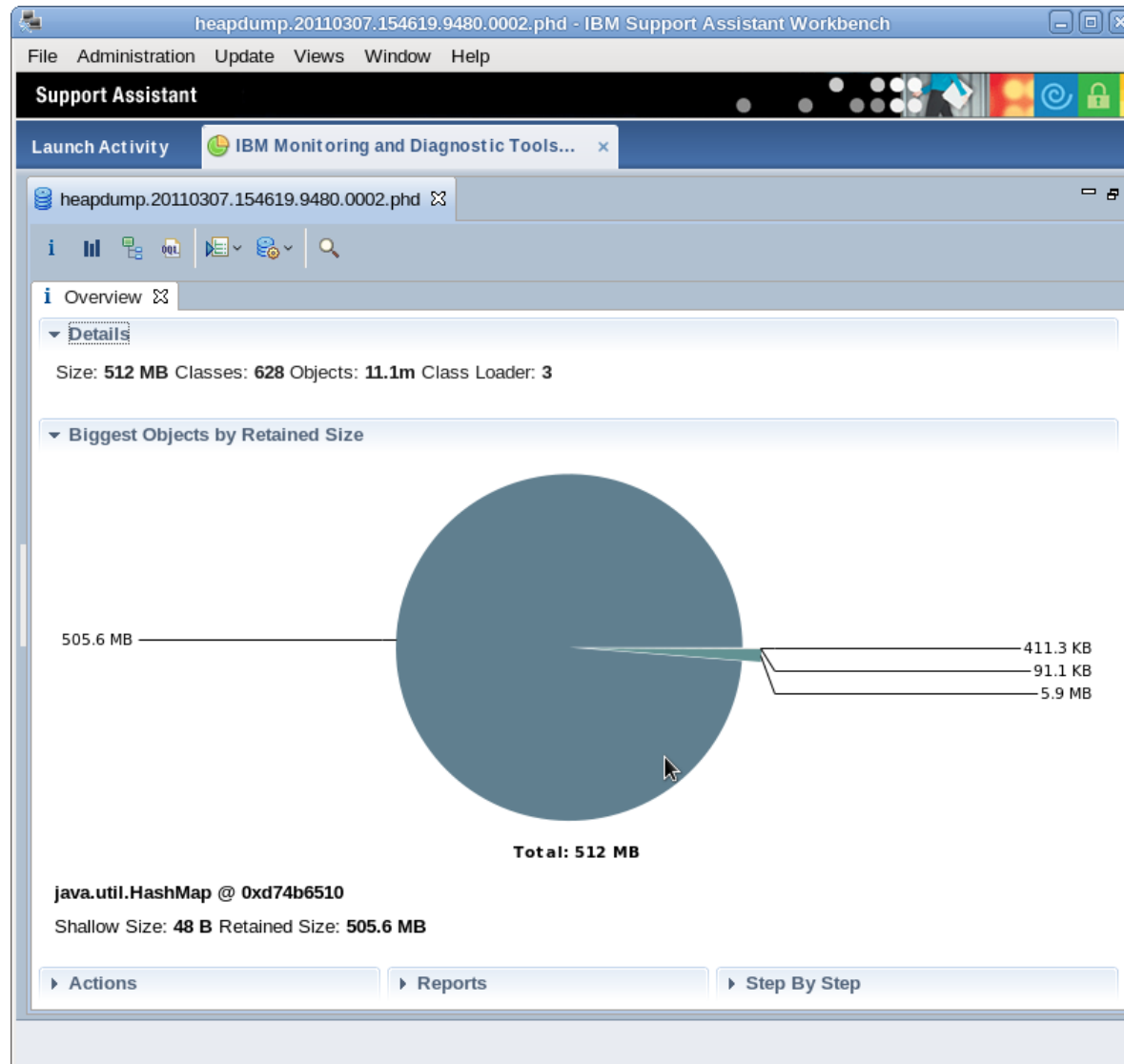


## MAT (Memory Analyzer für Heap- und Core Dumps)

- MAT dient zur visuellen Aufbereitung von Heap- und Core Dumps
- Heap Dumps sind Snapshots des Java Heaps und beinhalten Informationen über Java Objekte und deren Verbindungen untereinander
- Core Dumps sind Snapshots des gesamten Adressraums und beinhalten zusätzlich noch Informationen über Variablen- und Objektnamen und Inhalte
- Für Analysezwecke ist ein Core Dumps (mehrere GB) einem Heap Dump (mehrere hundert MB) vorzuziehen
- Achtung: Core Dump muss erst mit Jextract verarbeitet werden
- MAT ist ebenfalls im IBM Support Assistant enthalten



## MAT (Memory Analyzer für Heap- und CoreDumps)



# Hprof

- Hprof ist ein Profiler, welcher in der JVM direkt eingebaut ist
- Mit Hprof kann man sehr schnell herausbekommen:
  - In welchen Zeilen die meiste Laufzeit hängt
  - Wieviel CPU-Zeit verbraucht wird
  - Wieviele Objekte allokiert werden
- Hprof arbeitet mit Sampling
  - Anwendung wird kurz gestoppt
  - Die aktuellen Stacktraces werden analysiert
- Eingriffsmöglichkeiten hinsichtlich Anzahl der Samples und Tiefe der Analyse

TRACE 300092:

com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile(GCAnalyzer.java:76)

com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInFile(GCAnalyzer.java:53)

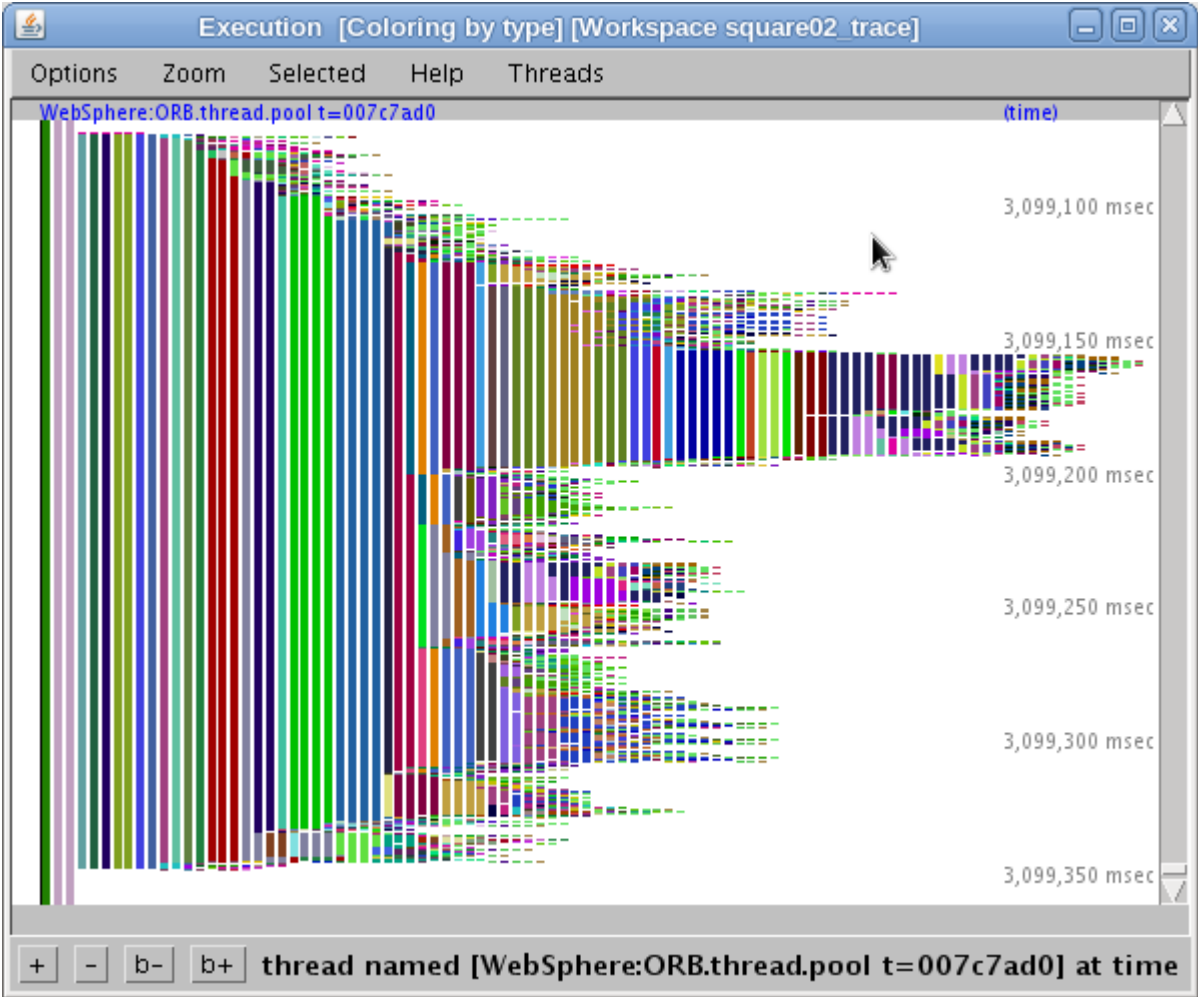
com.ibm.zSW.wasGCAnalyzer.Main.main(Main.java:22)

CPU SAMPLES BEGIN (total = 567) Mon Mar 7 12:05:15 2011

| rank | self   | <u>accum</u> | count | trace  | method   |
|------|--------|--------------|-------|--------|--|
| 1    | 12.70% | 12.70%       | 72    | 300091 | java.util.zip.Inflater.inflateBytes                        |
| 2    | 12.52% | 25.22%       | 71    | 300092 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 3    | 7.94%  | 33.16%       | 45    | 300107 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 4    | 6.88%  | 40.04%       | 39    | 300100 | java.io.BufferedReader.readLine                            |
| 5    | 6.70%  | 46.74%       | 38    | 300088 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 6    | 6.00%  | 52.73%       | 34    | 300089 | sun.nio.cs.UTF8_Decoder.decodeArrayLoop                    |
| 7    | 5.64%  | 58.38%       | 32    | 300095 | java.io.BufferedReader.readLine                            |
| 8    | 5.29%  | 63.67%       | 30    | 300105 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 9    | 3.53%  | 67.20%       | 20    | 300102 | java.io.BufferedReader.readLine                            |
| 10   | 3.35%  | 70.55%       | 19    | 300114 | java.io.BufferedReader.readLine                            |
| 11   | 3.00%  | 73.54%       | 17    | 300090 | sun.nio.cs.UTF8_Decoder.decodeArrayLoop                    |
| 12   | 1.59%  | 75.13%       | 9     | 300106 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 13   | 1.59%  | 76.72%       | 9     | 300116 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 14   | 1.59%  | 78.31%       | 9     | 300120 | com.ibm.zSW.wasGCAnalyzer.GCAnalyzer.searchForGCsInIBMFile |
| 15   | 1.23%  | 79.54%       | 7     | 300121 | java.io.BufferedReader.readLine                            |

- Jinsight ist Java Profiler welcher speziell für z/OS und zLinux eingesetzt werden kann
- Jinsight greift über JVM Schnittstellen (JVM Tool Interface) alle Ereignisse wie Methodenaufrufe und das Verlassen von Methoden ab
- Es wird der exakte Ausführungsplan der Anwendung dargestellt
- Zusammenhänge von Methoden sind deutlich sichtbarer als im Quellcode selbst
- Die Auswirkung von Frameworks werden verdeutlicht
- Sollte nur für sehr gut vorausgewählte Bereiche eingesetzt werden
- Jinsight verlangsamt die Ausführung des Programms signifikant und ist somit nur für konkrete Tuning-Maßnahmen zu empfehlen
- Jinsight ist das stärkste Mittel zur Quellcodeanalyse

# Jinsight



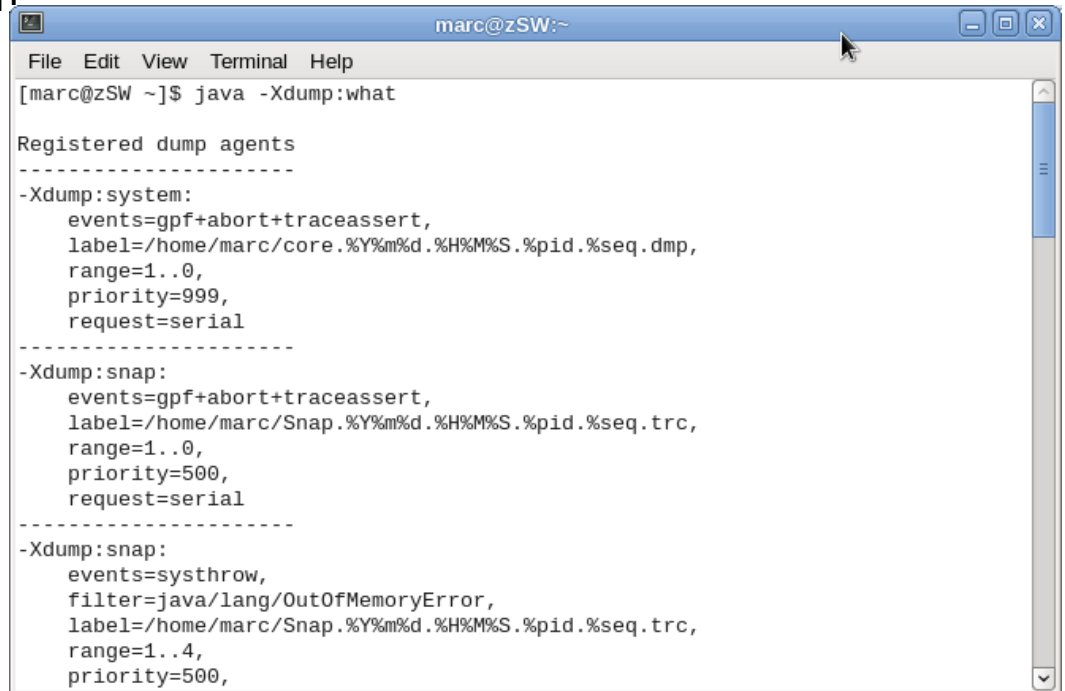
# Konfiguration für Stabilität und Fehleranalyse

# Konfiguration für Fehleranalysen

- Typisches Szenario: Nach Serviceausfall steht die Wiederaufnahme absolut im Vordergrund für andere Aktivitäten bleibt keine Zeit.
- Daten für Fehleranalysen können auf Zeitgründen nicht manuell gesammelt werden
- Ziel ist es, dass für die am häufigsten auftretenden Fehler automatisch die jeweiligen Analysedaten ermittelt werden
- Das bedeutet grundsätzlich:
  - OutOfMemory Exception → Core- oder Heap-Dump
  - Hängender Thread → JavaCore-Dump

## Dump Agents

- Bei unterschiedlichsten Events können unterschiedlichste Dumps und sogar selbstentwickelte Skripte ausgeführt werden
- Dump Agents können per JVM-Parameter konfiguriert werden
- Auch als Enviroment Variable möglich
- Sind ein Feature der IBM-JVM
- Verschiedene default Dump Agents können überschrieben werden
  - z.B. Core statt Heap-Dump
- Beispiel:
  - „Kill -3 <PID der JVM>“
  - Löst event „user“ aus
  - JVM läuft weiter



```

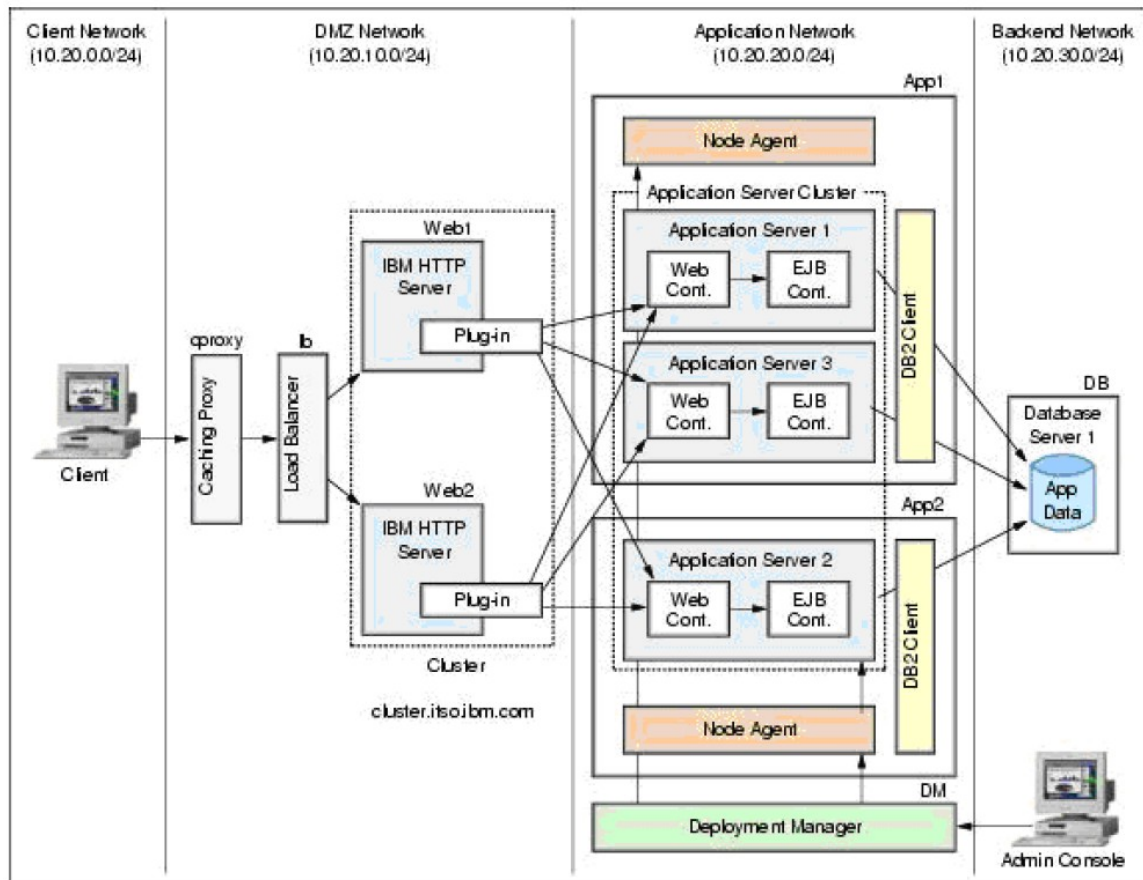
marc@zSW:~$ java -Xdump:what

Registered dump agents
-----
-Xdump:system:
  events=gpf+abort+traceassert,
  label=/home/marc/core.%Y%m%d.%H%M%S.%pid.%seq.dmp,
  range=1..0,
  priority=999,
  request=serial
-----
-Xdump:snap:
  events=gpf+abort+traceassert,
  label=/home/marc/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc,
  range=1..0,
  priority=500,
  request=serial
-----
-Xdump:snap:
  events=systhrow,
  filter=java/lang/OutOfMemoryError,
  label=/home/marc/Snap.%Y%m%d.%H%M%S.%pid.%seq.trc,
  range=1..4,
  priority=500,
  request=serial
-----
  
```



## Konfiguration für Stabilität

- Typische Infrastruktur einer WebSphere Application Server Umgebung



## Konfiguration für Stabilität

- Wieviele Datenbank-Connections verträgt mein System?
  - Abhängig von Menge und Komplexität der Anfragen
- Was ist die Obergrenze an parallelen Requests um die Infrastruktur auszulasten?
  - Abhängig von Anwendungscharakteristik (z.B. viel vs. wenig I/O)
- Wieviele Requests dürfen sich vor dem Application Server aufstauen?
  - Abhängig von Lastcharakteristik (konstant vs. dynamisch)
  - Dauer der Anfragen
  - Bearbeitungszeit der Anfragen
- Wie lange soll auf Ressourcenzugriffe gewartet werden?
  - Wie lange soll auf Datenbankverbindungen gewartet werden?
  - Wie lange darf eine Anfrage auf der Queue zwischen Controller und Servern liegen?

# Performance Probleme in den Griff bekommen

# Problemsuche

- Performanceprobleme treten durch einen Mangel an Ressourcen auf
- Suche nach dem Bottleneck:
- CPU
  - Hohe Auslastung der CPU
  - Back-end Systeme liefern „schnell genug“
- I/O
  - Die CPU Auslastung ist meist eher gering
  - Die Anwendung warten hauptsächlich
- Synchronisation
  - Die CPU Auslastung ist eher gering
  - Die Anwendung wartet auf die Freigabe von Ressourcen (Locks)
- Memory
  - Keine garantierte Aussage über CPU Verhalten möglich

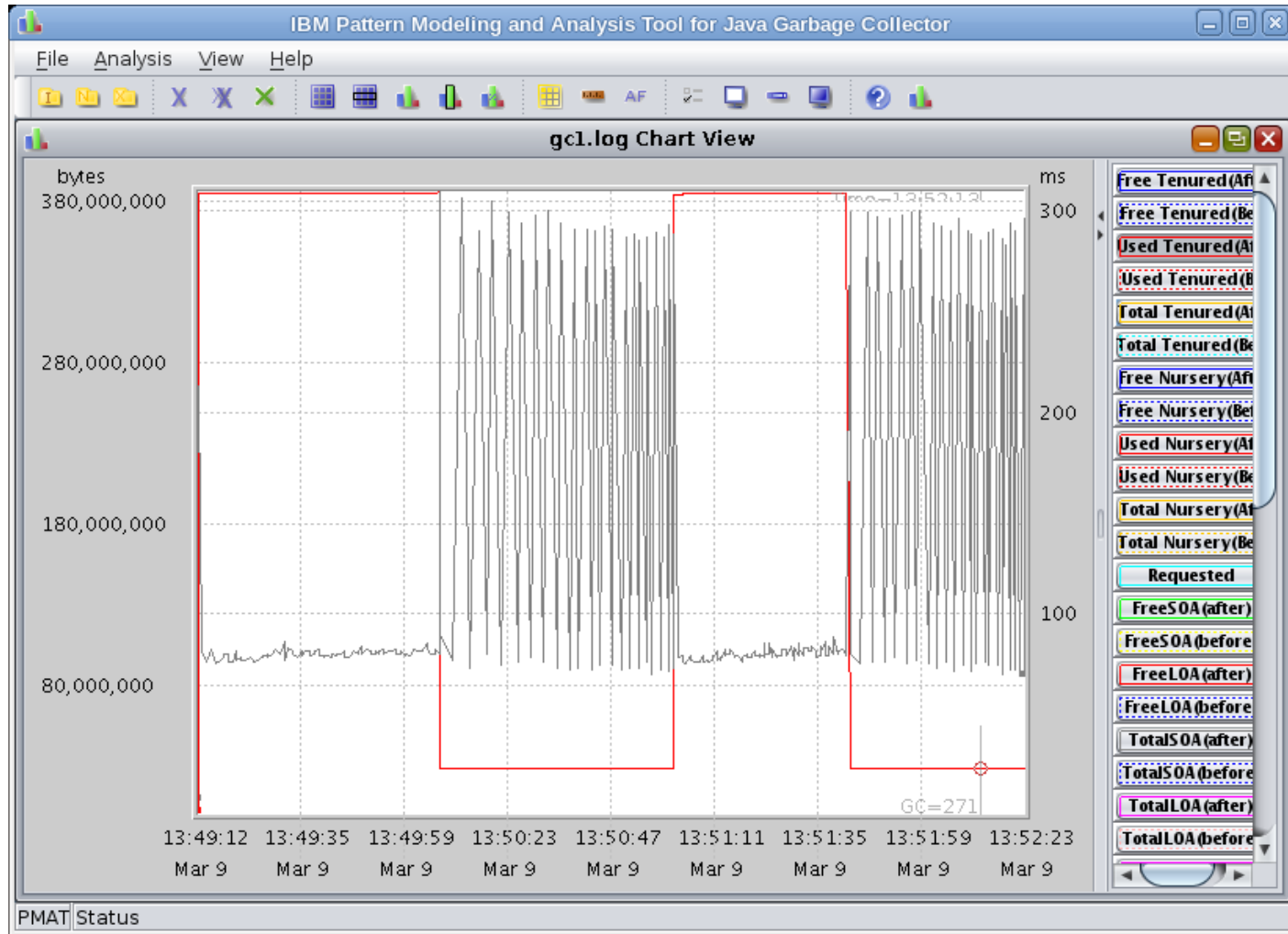
# Memory

- Anwendungsdurchsatz leidet
- CPU wird mit Garbage Collection blockiert → die Anwendung wird langsamer
- Die maximale Anzahl eingeloggter Benutzer wird reduziert → Scale Out benötigt
- Weniger Daten können parallel abgearbeitet werden → Wartezeiten verlängern sich
- Die Gefahr von Abstürzen ist signifikant höher
- Stellschrauben:
  - Benötigter Speicher muss komplett in den RAM passen → kein Paging
  - Der benötigte Speicherbedarf muss möglichst präzise abgeschätzt werden
  - Die Garbage Collection Policy sollte zu den Anwendungscharakteristiken passen
  - Objektallokationen sollten überdacht werden
  - Die Größe von Sessions und Caches sollte hinterfragt werden

## Garbage Collection

- Nicht mehr verwendete (referenzierte) Objekte werden von speziellen Java-Threads entfernt
- Während die Garbage Collection läuft werden die Anwendungsthreads ausgebremst...
- Garbage Collection ist NICHT in der Java Spezifikation enthalten
  - Herstellerabhängige Algorithmen mit Vor- und Nachteilen
  - Auch gar keine Garbage Collection wäre möglich
- Garbage Collection kann starken Einfluss auf die Anwendung haben
- Ist wichtigster “Hebel” bezüglich Performance und Stabilität außerhalb der Anwendung
- Sammlung von Garbage Collection Ausgaben ist für Problem Analysen und Tuning Maßnahmen Pflicht! Sollte stets eingeschaltet sein, da sie wichtige Daten enthält
  - Dauer und Häufigkeit der Garbage Collection
  - Wie lange wurde die Anwendung für die Garbage Collection gestoppt?
  - Wie groß sind die Objekte die erzeugt werden?
  - Wie groß ist der Speicherbedarf der Anwendung?
  - Passt der Speicherbedarf zum Benutzerverhalten?
  - Haben wir Memory Leaks oder Doom-Switches?

## Garbage Collection



- Ein CPU-Engpass kann das Programm ausbremsen, wenn Ressourcen nicht mehr fair verteilt werden können
- Garbage Collection kann die CPU signifikant belasten
- Profiling kann die Haupt-CPU-Konsumenten ermitteln
- Code- und Designoptimierungen können die Hauptkonsumenten gezielt optimieren
- Stellschrauben
  - Garbage Collection optimieren
  - Anwendungsprofiling
  - Code-Optimierung
  - Design-Optimierung
  - Thread-Optimierung



## Profiling mit HProf

- Gezielte Suche nach den Methoden, die meiste Zeit auf dem Stack verbringen
- Ist neben Tivoli Performance Viewer und Request Metrics der erste Schritt nachdem die Garbage Collection überprüft wurde
- Problematisch beim Programmieren mit mehrere Threads, da die Übersicht leidet
- Da Sampling verwendet wird ist es schwer den Programmfluss zu verfolgen
- Zeigt deutlich auf, wenn konkrete Statements den Code verlangsamen

## Profiling mit Jinsight

- Jinsight ist DAS Profiling Tool für Java unter z/OS und zLinux
- Kann zwischen Threads unterscheiden
- Gemessene Laufzeiten ist nicht repräsentativ → die Codezusammenhänge sind interessant
- Kaum Dokumentation; nach Eingewöhnungszeit jedoch gut benutzbar
- Suche nach Object-Konstruktoren
- Suche nach Wiederholungen im Code
- Die „Schnittstelle“ zwischen eigenem Code und Framework-Code sollte genau betrachtet werden → Was macht die kleine Framework-Methode im Hintergrund?
- vor Jinsight Profiling sollte die problematische Stelle im Programm möglichst gut eingegrenzt werden
- Jinsight Logs sind sehr groß (mehrere GBs)

## Java Performance Best Practices

- Zwei “Kategorien” der Performance Optimierung:
  - “guter” Programmierstil
  - “deep and dirty Hacks”
- “guter” Programmierstil
  - Kleine Kniffe, meist selbsterklärend
  - Sehr schnell umgesetzt
  - Keine Erhöhung des Wartungsbedarfs
  - Sollten generell angewendet werden
- “deep and dirty Hacks”
  - Große Eingriffe in den Code
  - Brechen oft mit “gutem” Stil und OO-Paradigmen
  - Erhöhter Wartungs- und Dokumentationsbedarf
  - Ergänzen Tuningmaßnahmen, „wenn es eng wird“

## Guter Programmierstil

- Code Motion
  - Auslagern von redundanten Berechnungen aus Schleifen
  - Generelle Vermeidung von Funktionsaufrufen mittels Caching
- Enge Bindung von Variablenerstellung und -Nutzung
  - Für lokale Variablen möglichst geringen Scope benutzen
- Vermeidung von Objekterzeugung
  - Objekte kosten dreifach: Anlegen, Initialisieren, Aufräumen
  - Wiederbenutzung von Objekten anstatt Wegwerfen und Neuerzeugen
- Vermeidung von finalizern
  - Finalizer benötigen 2(!) Garbage Collection Läufe
  - Es ist nicht garantiert, dass Finalize() ausgeführt wird
- Erster Bedingungsfall immer wahrscheinlichster
  - && und || Operatoren werten die Verknüpfte Bedingung nicht aus, wenn die erste Bedingung die Gleichung löst
- StringBuffer und StringBuilder anstatt „String + String“

## Deep and dirty Hacks

- Bevorzugung Lokaler Variablen vs. Objekt-, Klassen-, Arrayvariablen
  - Die Manipulation lokaler Variablen ist deutlich schneller, da sie auf dem Stack liegen
- Char[] statt Strings
  - Char-Arrays sind Strings in allen Performance-Belangen überlegen
- tail-Rekursion
  - Tail-Rekursion verringert das Stackwachstum
- Exception driven loop termination
  - Vermeidung von Bedingungen, Exception wird “hingenommen”
- Objekte übergeben und direkt manipulieren anstatt mit Return-Werten zu arbeiten
  - Objekt Initialisierungen verhindern
- Lazy vs. Early Initialisation
  - Initialisierung von Objekten am spätmöglichen Moment
  - Initialisierung von Objekten an einem Zeitpunkt, an dem die Maschine wenig ausgelastet ist

- I/O Engpässe sorgen für Auslastungsprobleme der CPU
- Programm wartet auf Datenbank Resultate
- Programm läßt Dateien langsam ein
- Programm wartet auf Netzwerk
- Stellschrauben:
  - So viel Logik wie möglich in der Datenbank
  - Wenn möglich nur Deltas verschicken
  - Komprimierte Daten lesen und senden

# Synchronisation

- Es gibt Java-Konstrukte, die nur von einem Thread gleichzeitig benutzt werden können
- Diese Konstrukte können dafür sorgen, dass trotz freier CPU die Anwendung schlicht weg auf Zugriffe auf diese Java-Konstrukte wartet
- Folge: die meisten Ressourcen langweilen sich (meistens ist jedoch genau ein CPU-Kern am Anschlag)
- Stellschrauben:
  - Synchronized Methoden vermeiden
  - Synchronized Klassen hinterfragen (StringBuffer, Vector)

# Fall-Beispiele



- Ausgangssituation:

„Hallo Marc,

*Wir haben hier eine Migration von Intel auf zLinux. Die Anwendung läuft schnell, jedoch bräuchten wir noch ein wenig mehr Power. Kannst du da was machen?“*

- Betrachten wir den GC-Trace mit PMAT (immer erster Schritt)
- Vergleich Sun-Parameter mit IBM-Parametern
- Optimierung der GC-Parameter
  - Richtige Wahl der Init- und Max-Size
  - Diskussion der GC-Policy
    - Optthruput für „Batch“-Workload (default)
    - Gencon für Transaktionalen Workload
- Vergleich von GC-Overhead:
  - Prozentuale Bewertung des Overhead ist fehleranfällig, da unterschiedliche Workload-Szenarien
  - Marcs Methode: [Weggeräumte Gbs/ verbrauchte Sekunden in GC]
  - Ist unabhängig von Workload-Szenarien (Tag/Nacht, Wochentag/Wochenende)

# Es ist langsam...

- Ausgangssituation:

„Hallo Herr Bauer,

*Wir haben heute eine Arbeitsgruppe zum Thema Anwendungs-Performance. Die Anwendung wird flächendeckend eingesetzt. Das Management ist schon ganz aufgeregt. Bitte unterstützen Sie uns für eine zügige Lösung.“*

- Garbage Collection war OK → schade
- Frage: Welches Modul ist langsam?
  - Vergleich der WebServer Access Logs und Daten der Request Metrics
  - Auswertung: Der WAS ist langsam... aber nur manchmal?!
- Bestimmte Request hängen >30 Sekunden in bestimmten Modulen fest
- Thread Dumps im 3 Sekunden Takt → längere Request sollten dort auftauchen
- Viele Monitor Locks zu erkennen immer im gleichen Modul; sogar gleiche Codezeile (Open Source)
- Analyse des Open Source Moduls

## Zuviele Objekte...

- Ausgangssituation:

„Hallo Marc,

Wir haben hier beim Kunden das Problem, dass bei jedem Login eine große Menge an Objekten erzeugt werden. Wir haben ohnehin sehr große Sessions pro Anwender(>8MB) und würden gern anfangen, das Objektaufkommen zu reduzieren. Kannst du nächste Woche mal vorbeikommen?“

- Analyse mit Jinsight
- Vector-Objekte werden unnötig erzeugt
- 1 neues IF-Statement: 90% weniger Objekte
- Seltsame blockende Requests im WebContainer

# Abstürze in den Griff bekommen

## Typischer Absturz einer JVM

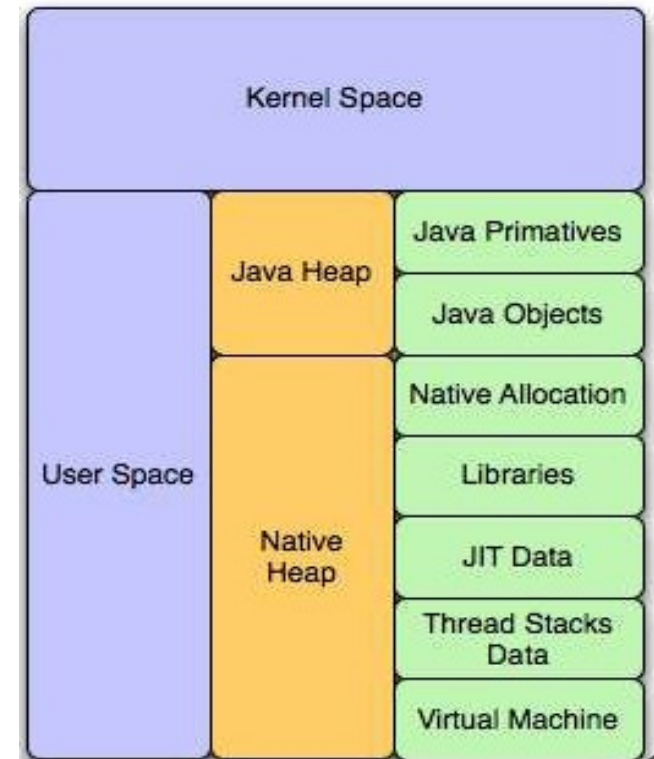
- Speicher voll → GC → Speicher voll → GC...
- OutOfMemory → Absturz
- Meist ist die JVM minutenlang vor dem eigentlichen Absturz unbenutzbar
- Der Prozess selber ist vorhanden
- Die Benutzer bekommen die Probleme vor dem Betrieb mit

## OutOfMemory Exceptions analysieren

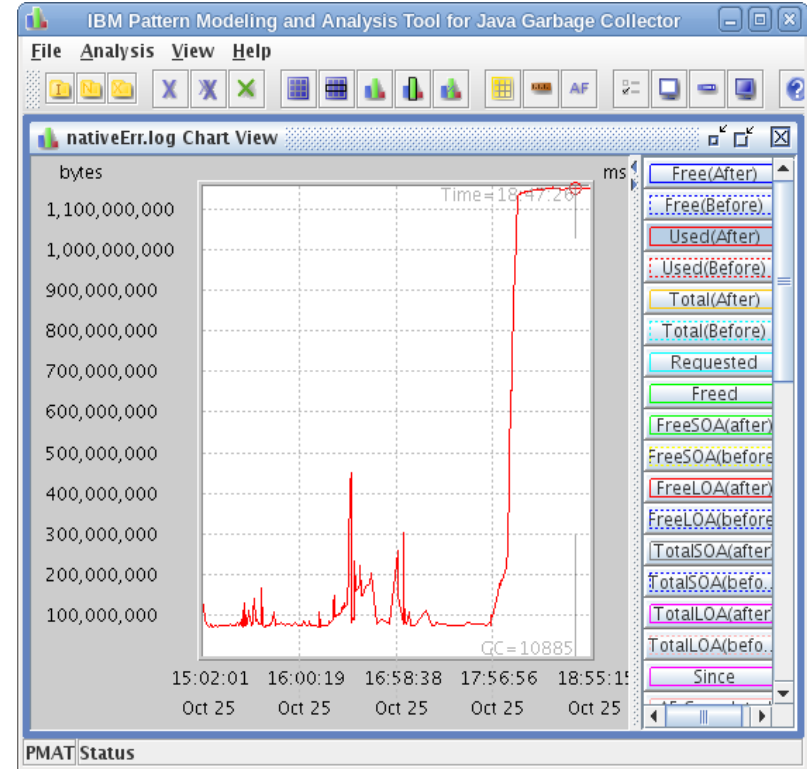
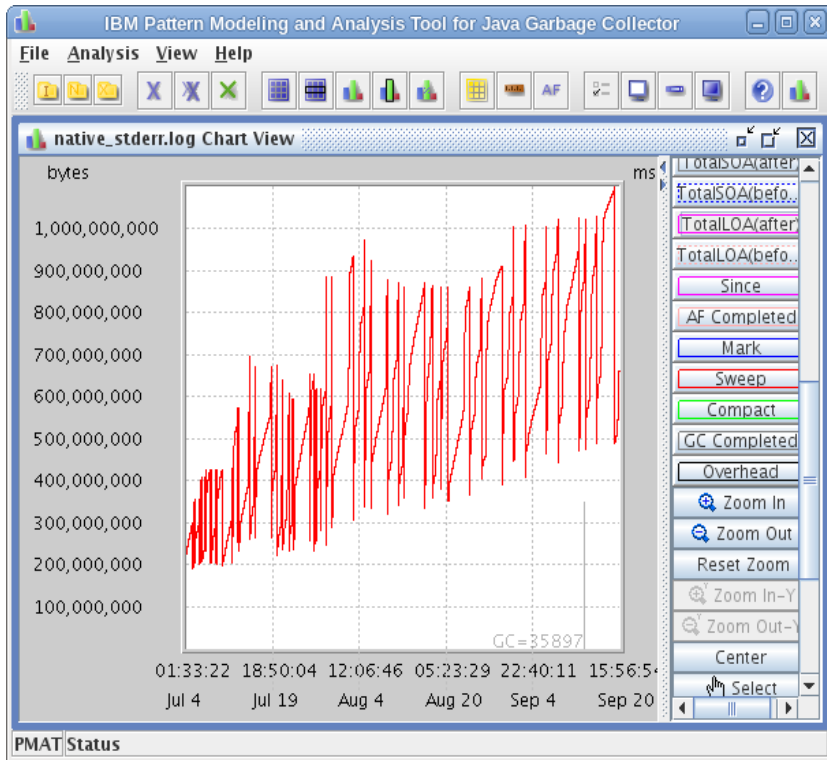
- Die mit Abstand häufigste Ursache für Abstürze von JVMs sind OutOfMemory-Exceptions
- Der dem Prozess zugewiesene Speicher ist komplett verbraucht und eine Speicheranfrage kann nicht befriedigt werden:
  - z.B. `Object o = new Object();` //im “Java Heap”
  - z.B. `malloc()` //im “Native Heap”
- Analyse:
  - Welcher Teil des Heaps ist betroffen? (verbose:gc-Trace)
  - Haben wir ein Memory Leak oder einen Doom-Switch? (verbose:gc-Trace)
  - Welche Objekte belegen den Heap? (Core- oder Heap-Dump)
  - Wo kommen diese Objekte her? (Java-Core-Dump)

## Aufbau des Heaps

- Der Adressraum ist in Java Heap und Native Heap eingeteilt
- Wichtig: Adressraumgröße = Java Heap – Native Heap
  - Bei 31 Bit: 2GB
  - Bei 64 Bit: 16EB
- Je kleiner der Java Heap, desto größer der Native Heap



# Memory Leak oder Doom Switch?





## Welche Objekte belegen den Heap?

- Egal ob Memory Leak oder Doom Switch, es haben sich unerwartet viele Objekte im Heap angesammelt
- Beim Doom Switch hängen diese Objekte meist direkt an einem Request

## Welcher Code hat die Probleme verursacht?

- JVM ist so konfiguriert, dass sie einen JavaCore bei einem OutOfMemory erzeugt
- Bei Memory Leak:
  - wegwerfen, da das Problem nichts mit dem zum Absturzzeitpunkt laufenden Code zu tun haben muss
- Bei Doom-Switch:
  - Reinschauen
  - Der Code der den Doom-Switch ausgelöst hat, muss noch aktiv gewesen sein
- Oft hat der Server lang genug um sein Leben gekämpft, dass der Thread Monitor angeschlagen hat
  - „Thread may be hung“-Meldung verrät uns, welcher Thread sehr lang gearbeitet hat
  - Sehr wahrscheinlich ist dies unser Kandidat
- Hier brauchen wir meist einen Anwendungsentwickler
- Standardproblem:
  - Suchen ohne Begrenzung der Ergebnisse
  - Reportgenerierung

# Fall-Beispiele

## „Server reagiert nicht mehr“

- Ausgangsszenario:

„Hallo Herr Bauer,

*am 07.02.2011 um 11:47 haben wir eine Meldung von einem unserer Keyuser erhalten, dass die Anwendung "stehen" würde. Aufgetreten sei das Verhalten zwischen ca. 11.15 Uhr und 12.00 Uhr. Können Sie an Hand der GC-Logs Rückschlüsse auf mögliche Ursachen geben?“*

- WAS ist noch da, aber Kunden klagen über schlechte bis gar keine Antwortzeit
- Sehr sehr starke GC-Aktivität
- Anwendungsthreads bekommen keine CPU mehr, bzw. dürfen keine Objekte anlegen

# Kunden-Szenario: Der Absturz

- Ausgangsszenario:

*„Hallo Herr Bauer,*

*Leider ist gestern die Anwendung abgestürzt und hat eine Menge an Dumps erzeugt. Können wir darin die Ursache für das Problem finden? Die Daten kann ich Ihnen über den alten PMR zur Verfügung stellen.“*

- GC-Trace weist auf Memory-Leak hin
- Heap-Dump-Analyse ergab:
  - Zu große Sessions
  - Zu viele Sessions
- Maximale Sessiongröße bestimmt maximale Sessionanzahl
- Lösung:
  - Weniger Sessions in den Heap lassen
  - „Least recent used“ Sessions persistieren

# Zusammenfassung

- Performance Probleme sind beherrschbar
- Abstürze sind beherrschbar
- IBM bietet viele Tools, um Internals der JVM zu überwachen
- Konfiguration spielt eine wichtige Rolle, um im Fehlerfall die richtigen Daten zu haben
- Anwendungsentwicklung kann viel für günstigen Betrieb tun
- Eigenen Code hinterfragen; Frameworks hinterfragen
- Kommen Sie auf uns zu, wir unterstützen gern bei Automatisierungen, Konfigurationen und natürlich Fehleranalysen plattformübergreifend!

## Links und Verweise

- IBM Support Assistant:
  - <http://www-01.ibm.com/software/support/isa/>
- Jinsight for IBM System z:
  - <http://www.alphaworks.ibm.com/tech/jinsightlive>
- Java Diagnostics Guide:
  - <http://www.ibm.com/developerworks/java/jdk/diagnosis/>
- Java Memories: Understanding and Analyzing the Java Heap
  - <http://www-01.ibm.com/support/docview.wss?uid=swg27019232>