Computer System Performance Analysis and Benchmarking WS 1999/2000

JVM Performance Analysis

Bernhard Egger, Daniel Schädler, Marcel Ziswiler

Inhaltsverzeichnis

1. Einleitung 3
2. Hardware Konfiguration 3
3. Die virtuellen Java-Maschinen (JVM)..3
   Sun JVM des jdk 1.1.8...4
   Sun JVM des jdk 1.2.x..5
   Sun JVM des jdk 1.3 Beta 5
   Netscape Navigator 4.7 mit JVM des Java 1.1.5..............................5
   Internet Explorer 5 mit Microsoft VM für Java 5.0 5
   Transvirtual Technologies Kaffe OpenVM 1.0.5 6
   Linux...6
4. Der JVM98 Java Virtual Machine Benchmark................................7
5. Testablauf...8
6. Auswertung..9
   Einfluss des Betriebssystems...9
   Sind 2 Prozessoren schneller als einer?..10
   Die Prozessorgeschwindigkeit 11
   Einfluss der Heapgrösse...12
   Einfluss des Just-in-Time-Compilers (JIT) 14
   Welches ist nun die Schnellste JVM auf einem Bestimmten System?.15
   Die jdks auf Windows 15
   Browser in Windows...16
7. Fazit..17
8. Referenzen...18

1. Einleitung

Ziel unseres Projektes ist es, zu einem gegebenen System die bestmögliche Java Umgebung zu finden. Der einzige hardwareseitig variierte Parameter ist deshalb auch die Taktrate des Prozessors. Softwareseitig werden die beiden Betriebssysteme Microsoft Windows NT 4.0 und Suse Linux 6.2 und natürlich verschiedene Java Virtual Machines (JVMs) getestet.

Zur Bestimmung der Leistung einer bestimmten JVM haben wir ausschliesslich den offiziellen SPEC JVM98 [2] verwendet.

2. Hardware Konfiguration

Für unsere Leistungsmessungen haben wir drei DELL Precision WorkStation 410 mit Intel Pentium II resp. III Prozessoren bei Taktraten von 400, 500 und 600MHz verwendet.

Die genaue Systemkonfiguration findet sich in Tabelle 1.

	System 1	System 2	System 3
CPU	Intel PII	Intel PIII	Intel PIII
Taktrate	400MHz	500MHz	600MHz
L1 Cache	32Kbyte	32Kbyte	32Kbyte
L2 Cache	512Kbyte	512Kbyte	512Kbyte
RAM	128MB	256MB	256MB
HDD	Seagate St39102LW	Seagate St39102LW	Quantum Atlas IV 9 WLS

Tabelle 1: Systemkonfiguration

3. Die virtuellen Java-Maschinen (JVM)

Die virtuelle Java-Maschine ist der Grundstein von Sun's Programmiersprache. Es ist die Komponente der Java-Technologie, welche für die Plattformunabhängigkeit, den kompakten Bytecode und die Fähigkeit den Benutzer vor schadhaften Programmen abzuschirmen, zuständig ist.
Die virtuelle Java-Maschine ist ein abstrakter Computer. Wie ein realer Computer hat auch sie einen Befehlssatz und benutzt verschiedene Speicherbereiche.

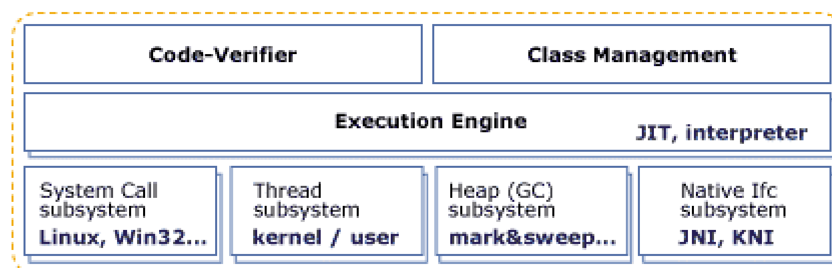

Abbildung 1 - Der modulare Aufbau einer JVM

Die virtuelle Java-Maschine geht nicht von einer bestimmten Implementierungstechnologie oder Wirtsplattform aus. Der Java-Bytecode muss aber nicht zwangsläufig interpretiert werden und kann durchaus zuerst auf den wirklichen Prozessor kompiliert werden, wie bei einer konventionellen Programmiersprache.
Diese Technologie wird JIT (just in time) Kompilierung genannt und konvertiert den Bytecode für die virtuelle Maschine vor dessen Ausführung in normale Instruktionen des Prozessors des Wirtssystems. Dies bewirkt unter Umständen eine kleine Wartezeit beim Starten und Laden von Klassen-Dateien (class files), kann aber auf der anderen Seite die gesamte Ausführungszeit um mehr als das Zehnfache verkürzen.
Die virtuelle Java-Maschine weiss dabei aber nichts über die Java-Programmiersprache. Sie kennt nur ein bestimmtes Dateiformat, das Format der Klassen-Dateien. Eine Klassen-Datei enthält Befehle der virtuellen Maschine, den sogenannten Bytecode und eine Symboltabelle, sowie weitere Hilfsinformationen.
Aus Sicherheitsgründen unterliegt der Code in den Klassen-Dateien strengen Format- und Strukturbeschränkungen.
Mittlerweile existieren unzählige Implementierungen verschiedenster Hersteller von virtuellen Java-Maschinen. Die von uns getesteten JVM Versionen werden nachfolgend unter die Lupe genommen:

Sun JVM des jdk 1.1.8

Die Windows Version enthält den Symantec JIT Bytecode Compiler, welcher die Ausführung von Java Programmen ungemein beschleunigt.
Es wurden in der Version 1.1 die folgenden Verbesserungen eingeführt:

- Verschiedene Teile der JVM, darunter die Interpreter Schleife, wurden bei Java 1.1 neu in Assembler realisiert. Die so entstandene virtuelle Maschine läuft bei bestimmten Operationen bis fünfmal schneller.
- Die Implementierung von Monitoren wurde beschleunigt, wodurch synchronisierte Methoden schneller und effizienter ablaufen.
- Die Speicherbereinigung von Klassen wurde erweitert, wodurch ungebrauchte Klassen automatisch gelöscht werden. Die dadurch verbesserte Speicherausnützung der JVM erlaubt es mit weniger Speicher effizienter zu arbeiten.
- Um höhere Performance zu erreichen wurden die AWT Partnerklassen komplett neu geschrieben um direkt auf der Win32 Schnittstelle aufzusetzen.
- Das plattformunabhängige JAR (Java Archive) Dateiformat wurde neu eingeführt um die Bündelung von Ressourcen in einer einzigen HTTP Transaktion zu erlauben. Somit können Klassen-, Bild- und Ton-Dateien komprimiert in einem einzigen Archiv abgelegt und vom Browser schneller über das Internet geladen werden. Zusätzlich kann der Autor individuelle Einträge in deiner JAR Datei digital unterschreiben, um dessen Herkunft zu beglaubigen.

Eine vollständige Zusammenstellung aller Geschwindigkeits-verbesserungen finden sie in [4].

Sun JVM des jdk 1.2.x
Änderungen seit Version 1.1.x:

- Änderungen in den Gleitkomma Funktionen ermöglichen grosse Effizienzgewinne.
- Die Speicherausnutzung geladener Klassen wurde optimiert. Konstante Zeichenketten werden von verschiedenen Klassen gemeinsam benutzt womit der Speicherverbrauch minimiert wird.
- Die Speicherzuordnung und –bereinigung wurde verbessert. Durch den im Thread lokal zwischengespeicherten Heap entfällt das dauernde locking bei Zuordnungen im Heap. Die Geschwindigkeit von Speicherzuordnungen wird drastisch erhöht. Die Pausen bei der Speicherbereinigung sind kürzer und der Garbage Kollektor konsumiert nicht mehr übermässig den C Stapelspeicher.
- Die Implementierung von Monitoren wurde abermals beschleunigt. Der im Thread lokale Monitor Zwischenspeicher erlaubt es den synchronisierte Methoden noch näher bei der Geschwindigkeit von normale Methoden zu laufen.

Eine Auflistung sämtlicher Beschleunigungen befindet sich in [5].

Sun JVM des jdk 1.3 Beta
Diese Version enthält zum ersten mal die Java HotSpot Client Virtual Machine. Sie ersetzt die klassische virtuelle Maschine und den JIT Compiler. Hauptmerkmale der HotSpot Technologie sind:

- Adaptive Kompilation, d.h. es werden nur performance-kritische Codeteile kompiliert.
- Dynamische Optimierung im Gegensatz zur simplen Kompilierung des JIT, z.B. durch Method-Inlining.

Weiter wurden die Ladezeit und der Speicherverbrauch von Klassen optimiert.
Auf die vollständige Auflistung kann unter [6] zugegriffen werden.

Netscape Navigator 4.7 mit JVM des Java 1.1.5
Der Navigator enthält die virtuelle Java-Maschine aus dem jdk 1.1.5 von Sun (mit integriertem JIT von Symantec), welche von der Implementierung der Version aus dem jdk 1.1.8 gleichzusetzen ist und keine spezielle Geschwindigkeitsverbesserungen enthält.

Internet Explorer 5 mit Microsoft VM für Java 5.0
Die extrem enge Verflechtung mit dem Internet Explorer Browser und der darunterliegenden Windows Plattform erlaubt ein effizientes Ausführen von Java Programmen.
Die virtuelle Maschine von Microsoft enthält eine Kollektion von profiling Schnittstellen, welche es dem Entwickler erlauben Programme in Bezug auf Geschwindigkeit und Speicherverbrauch speziell zu optimieren.

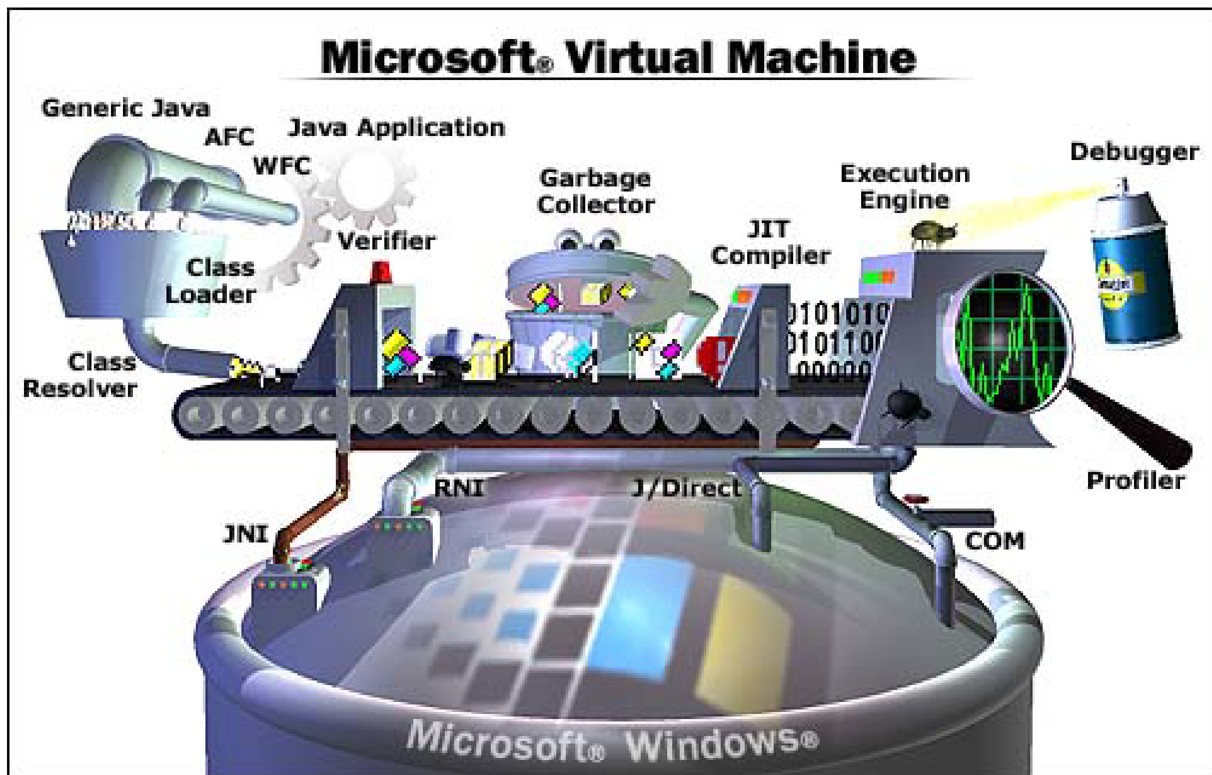

Abbildung 2 – Die Microsoft VM

Eine ausführliche Diskussion der virtuellen Maschine von Microsoft befindet sich in [7].

Transvirtual Technologies Kaffe OpenVM 1.0.5

Die Kaffe OpenVM ist eine unabhängige Implementierung einer virtuellen Java-Maschine mit eigenen JIT Compiler und geht bezüglich minimalem Speicherverbrauch neue Wege.
Eine ausführliche Beschreibung befindet sich im Datenblatt unter [8].

Linux

Leider enthalten sämtliche Implementierungen unter Linux entweder überhaupt keinen oder nur mässig optimierte JIT Compiler.

4. Der JVM98 Java Virtual Machine Benchmark

Benchmark	Beschreibung
_200_check	Testet die Java Virtual Machine um abzusichern, dass der JVM98 korrekt ausgeführt werden kann. Folgende Tests werden durchgeführt: • Array-Indexierung ausserhalb des Arrays • Kreiert Superklasse mit Subklasse und greift auf öffentliche, private und geschützte Variablen zu, zusätzlich werden Methoden überschrieben • Test, ob das Überschreiben der Superklasse korrekt funktioniert • Eine Test Kollektion namens PepTest, bei der einige if-then-else, Array und bitweise Operationen getestet werden (für Schleifen, Divisionen etc.). Dieser Test wird nur verwendet um die Funktionalität der JVM zu testen, die Resultate haben keinen Einfluss auf die SPECratio.
_201_compress	Modifizierter Lempel-Ziv Algorithmus (LZW), welcher nach häufigen Worteilen sucht und diese durch einen kürzeren Code ersetzt. Die Dekompression benötigt keine Eingabe Tabelle. Algorithmus aus: Terry A. Welch. *A Technique for High Performance Data Compression,* IEEE Computer Band 17, Nummer 6 (Juni 1984), Seiten 8-19. Dieser Test arbeitet mit echten Daten, nicht mit synthetischen. Benötigt 20MB Heap und alloziert 334MB Objekte
_202_jess	Jess ist das Java Experten Shell System basierend auf dem CLIPS Experten Shell System der NASA. Vereinfacht ausgedrückt führt ein Expertensystem viele if-then Entscheidungen, sogenannte Regeln, auf einer grossen Datenmenge, der Faktenliste, aus. Um auf längere Laufzeiten zu kommen wird die Berechnung iterativ immer auf eine grössere Faktenliste angewandt. Benötigt 2MB Heap und alloziert 748MB Objekte
_209_db	Führt viele Anfragen auf einer im Speicher gehaltenen Datenbank aus. Liest eine 1 MB grosse Datenbank, mit Namen, Adressen und Telefonnummern und führt darauf eine 19KB Datei mit Operationen aus. Die Operationen sind unter anderem: • Eine Adresse hinzufügen • Eine Adresse löschen • Nach einer Adresse suchen • Die Adressen sortieren Benötigt 16MB Heap und alloziert 224MB Objekte
_213_javac	Der Java-Compiler aus dem jdk 1.0.2 Benötigt 12MB Heap und alloziert 518MB Objekte
_222_mpegaudio	Diese Anwendung dekomprimiert Audio Dateien welche der ISO MPEG Layer-3 Audio Spezifikation entsprechen. Das Datenvolumen entspricht ungefähr 4MB. Belastet den Garbage Collector nur unbedeutend
_227_mtrt	Dies ist eine Variante des _205_raytrace, einem Strahlverfolgungs-berechnungsverfahren, dessen Szene einen Dinosaurier beschreibt. Zwei Threads berechnen jeweils die Szene aus einer 340KB Eingabe

	Datei. Benötigt 16MB Heap und alloziert 355MB Objekte
_228_jack	Ein Java Syntaxanalyse-Generator, basierend auf dem Purdue Compiler Construction Tool Set (PCCTS). Das ist eine frühe Version des heutigen JavaCC. Die Eingabedatei enthält Anweisungen zur Erstellung des Generators. Diese werden dem Generator gefüttert, womit der Syntaxanalysierer sich selbst mehrfach generiert. Benötigt 2MB Heap und alloziert 481MB Objekte

Tabelle 2: Bestandteile des jvm98

5. Testablauf

Der Benchmark wurde SPEC-konform (d.h. mit automatischer Anzahl Abläufen und ab einem Webserver – in unserem Fall der n.ethz.ch Server) ausgeführt. Der Test dauert zwischen 10 und 40 Minuten, je nach JVM. Er wurde jeweils als Applet im Browser, resp. im Appletviewer ausgeführt.

6. Auswertung

Einfluss des Betriebssystems

Man sollte eigentlich erwarten, dass Linux und WindowsNT etwa gleich schnell sind, Linux vielleicht sogar schneller – immerhin wurde Java für UNIX, dem Vorbild von Linux, entwickelt.

Stellt man die Beziehung zwischen den verschiedenen Java Virtual Machines und den Betriebssystemen jedoch grafisch dar zeiget sich schon deutlich, dass ein grosser Unterschied zwischen WindowsNT und Linux besteht:

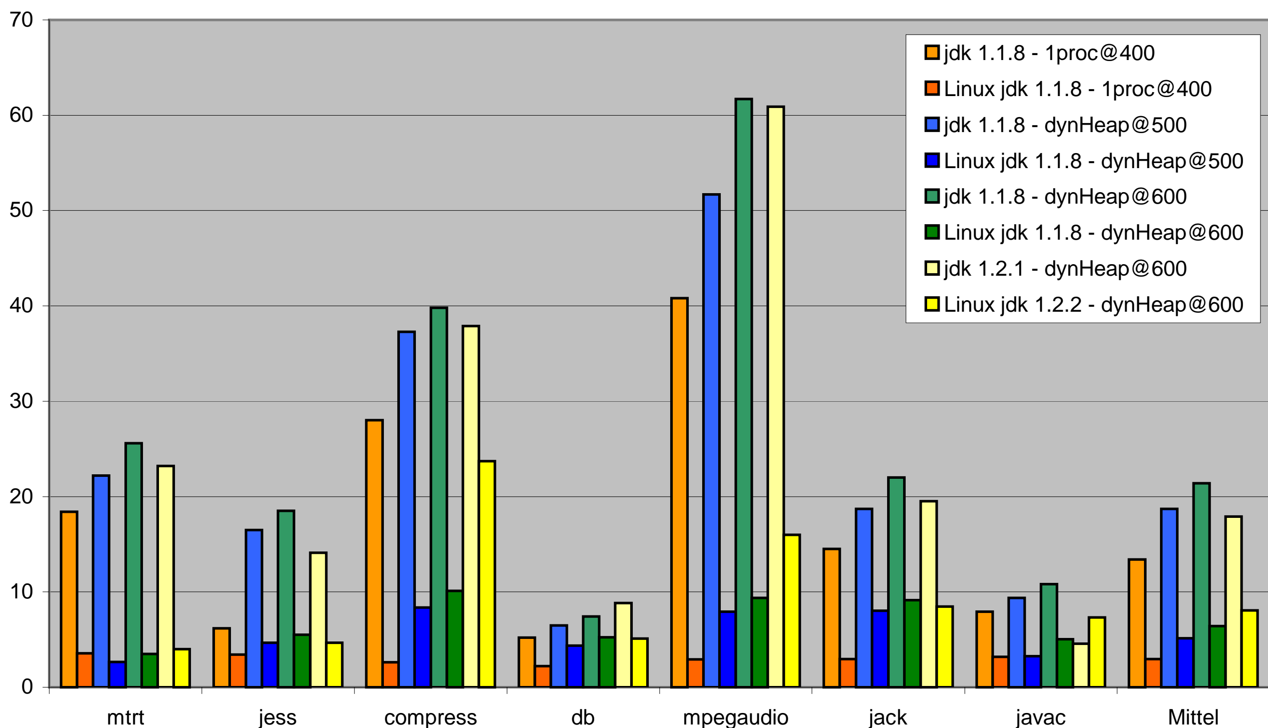

Diagramm 1 – Beziehung zwischen Betriebssystem und JVM

Unabhängig von der verwendeten JVM ist deutlich zu sehen, dass WindowsNT etwa um den Faktor 3-4 schneller arbeitet als Linux. Diese Beobachtung spiegelt sich auch in den einzelnen Benchmarks wider.

Die Bestätigung liefert die Berechnung der Variationen: [Berechnung im Anhang]

Variation abhängig vom Betriebssystem:	76.4%
Variation abhängig von der Java Virtual Machine:	21.4
unerklärt bleiben:	2.2%

Ein Grund dieser Unterschiede ist sicher, dass der JIT (Just im Time Compiler; siehe weiter unten) für Linux noch in der Entwicklung steckt und separat zum jdk dazugelinkt werden musste (Besserung ist in Sicht).

Sind 2 Prozessoren schneller als einer?
Davon auszugehen, dass 2 Prozessoren doppelt so schnell arbeiten wie einer alleine wäre im ersten Augenblick logisch, doch weil die JVM nur ein Prozess ist, darf das nicht erwartet werden:

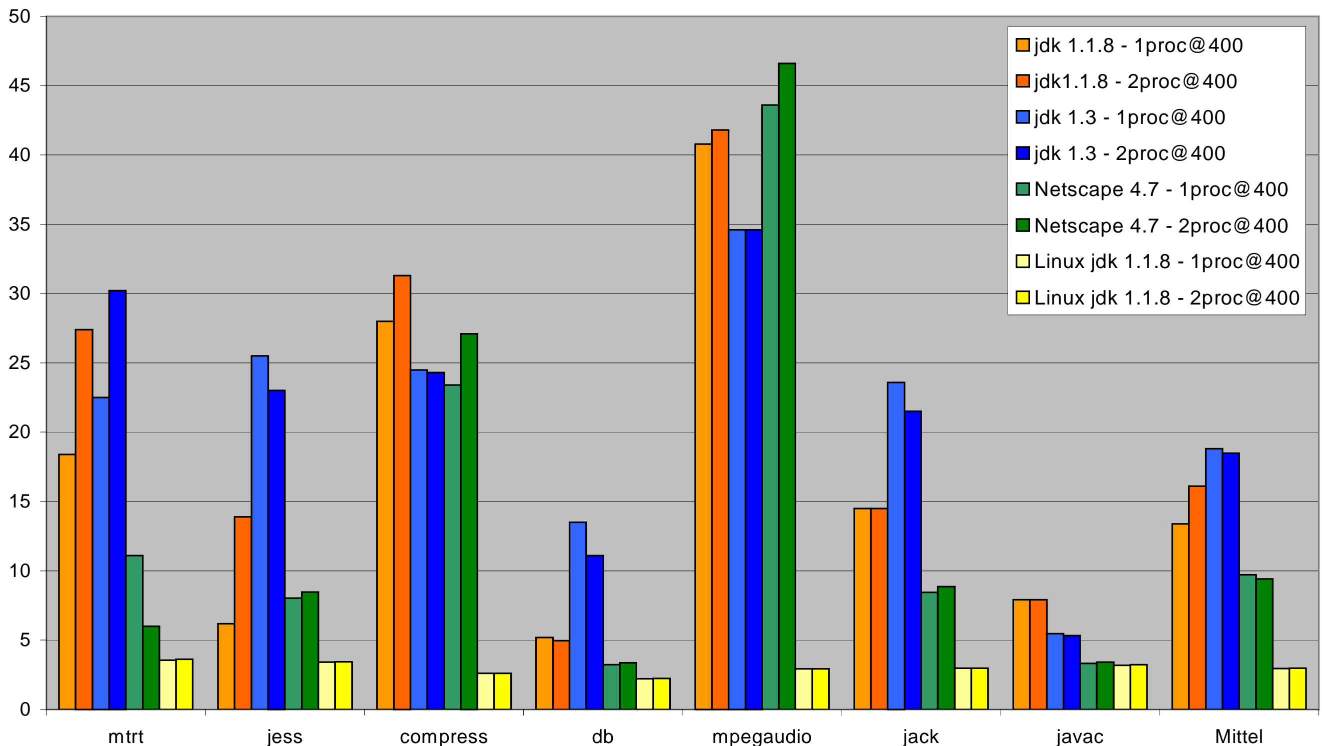

Diagramm 2 – Beziehung zwischen der Anzahl Prozessoren und der JVM

Das System mit 2 Prozessoren ist nicht schneller, teilweise sogar langsamer als das System mit einem Prozessor. Je nach JVM und Benchmark ist der Unterschied unterschiedlich gross; so ist zum Beispiel beim „mtrt" Benchmark mit jdk1.1.8 ein Gewinn ersichtlich, während Netscape mit 2 Prozessoren langsamer wurde. Bei den Meisten anderen ist hingegen kein wesentlicher Unterschied auszumachen. Linux bleibt etwa gleich schnell auf ein- und 2 Prozessorsystemen.

Die Berechnung der Variationen bestätigt obige Vermutungen: [Berechnung im Anhang]

Variation abhängig von der Anzahl Prozessoren:	0.1%
Variation abhängig von der Java Virtual Machine:	98.7%
unerklärte Variation:	1.2%

Die Prozessorgeschwindigkeit

Wir haben den JVM98 auf drei verschieden schnellen Prozessoren laufen lassen in der Absicht bei schnellerem Prozessor auch ein besseres Resultat zu erreichen.

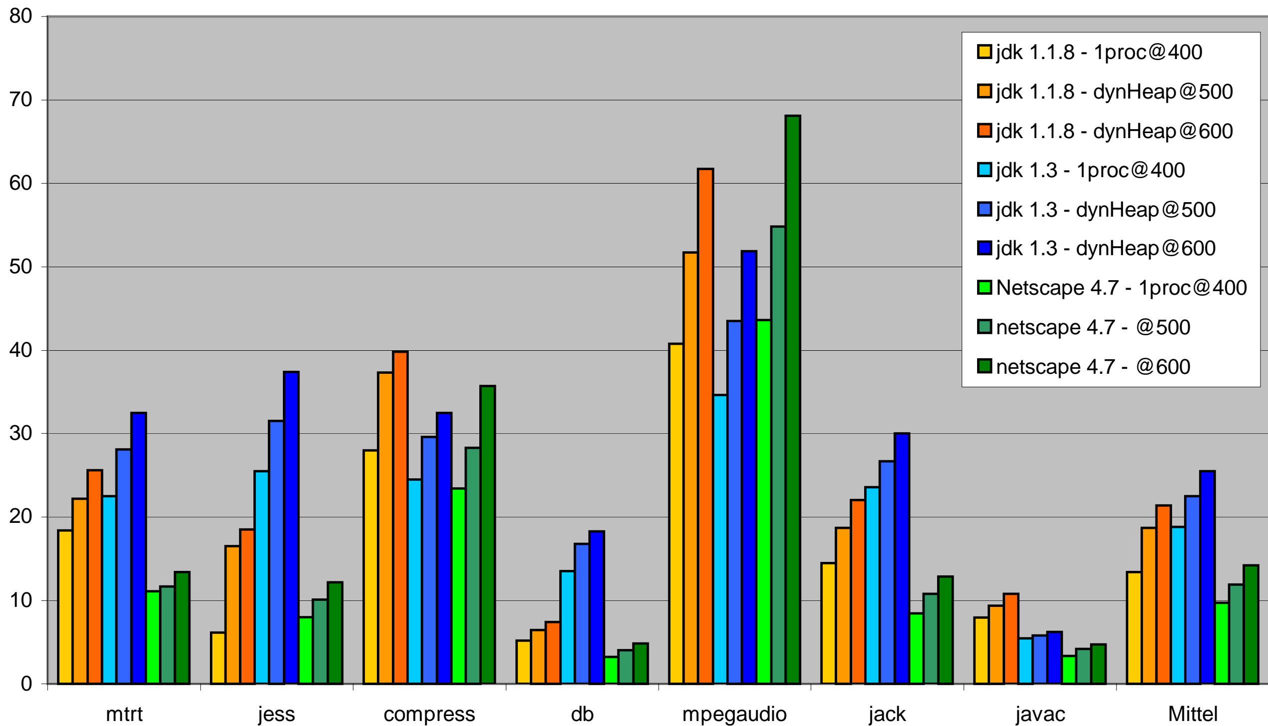

Diagramm 3 – Beziehung zwischen Prozessorgeschwindigkeit und JVM

In jedem Benchmark ist der schnellere Prozessor auch schneller. Es ist auch zu sehen, dass die Wahl der JVM einen etwas wichtigeren Faktor zu spielen scheint als die Prozessorgeschwindigkeit (gilt natürlich nur für die getesteten Konfigurationen). Das jdk1.3 ist im allgemeinen am schnellsten, doch beim „mpegaudio“ ist es am langsamsten. Zudem lässt sich feststellen, dass die Netscape JVM im allgemeinen am langsamsten ist, jedoch die anderen Beiden beim „mpegaudio“ hinter sich lässt.

Die genauen Verhältnisse sind folgende: [Berechnung im Anhang]

Variation abhängig von der Prozessorgeschwindigkeit:	30.2%
Variation abhängig von der JVM:	66.4%
unerklärte Variation:	3.4%

Einfluss der Heapgrösse

Zu erwarten wäre, dass eine dynamische Allokation nahezu optimal wäre, weil die JVM selbständig die Grösse des Heaps so setzt, dass der Garbage Collector möglichst wenig zu tun hat. Andererseits beginnt die dynamische Allokation bei einem bestimmten, kleinen Wert und muss schrittweise mehr Speicher anfordern. Trotzdem dachten wir, dass die Automatik besser sein würde als kleine, feste Angaben. Wir setzten bei den fixen Angaben die Grössen (Anfang und Maximal gleich) auf eher kleine Werte im Verhältnis zu der Datenmenge (teilweise mehr als 500Mb).

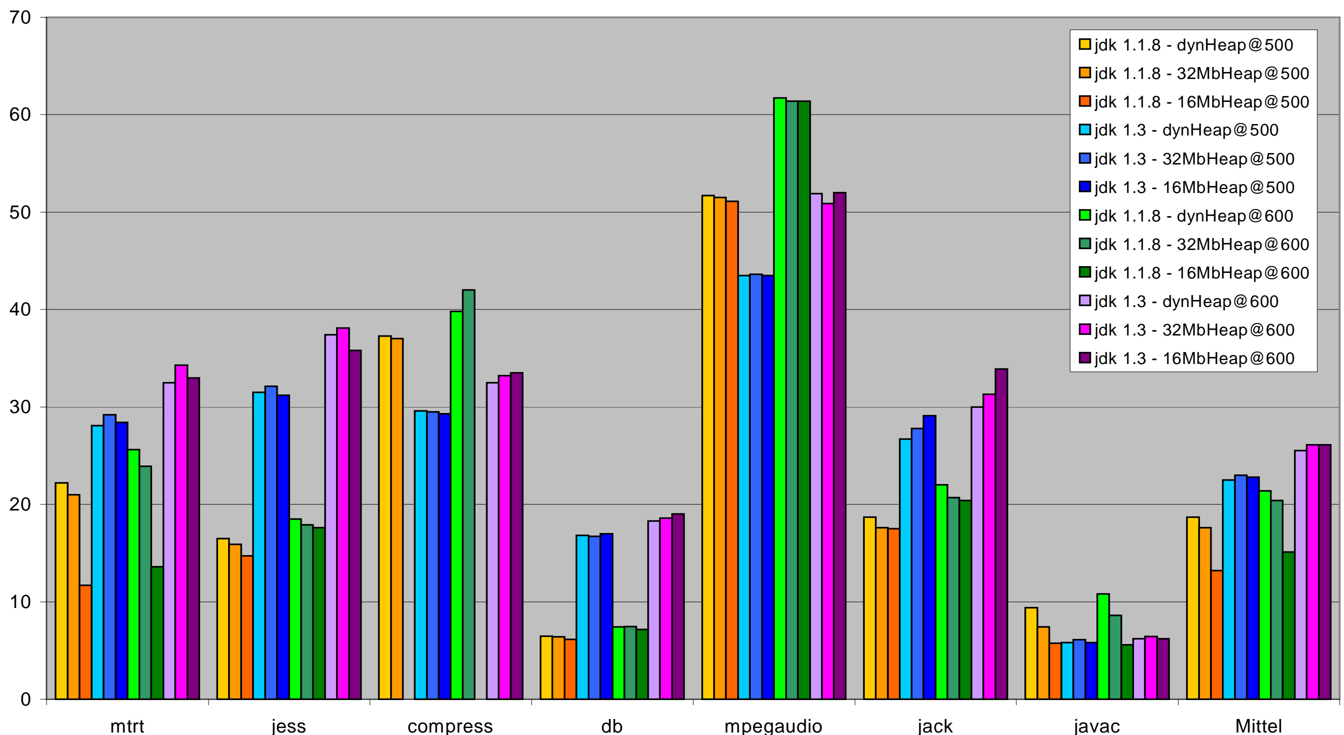

Diagramm 4 – Beziehung zwischen Heapgrösse und JVM

Solange der Benchmark keinen grossen Heap benötigt (siehe Benchmark Dokumentation), ist kein grosser Unterschied feststellbar. Wird jedoch die Heap-Maximalgrösse erreicht („mtrt"), wird eine deutliche Einbusse sichtbar, wird die Maximalgrösse überschritten steigt der Benchmark mit einem Fehler aus („compress"). Eine Ausnahme bildet jdk1.3. Es schaffte „mtrt" und „compress" ohne Einbussen.

Es lässt sich aber nicht endgültig abklären, ob die Heapgrösse im allgemeinen einen Einfluss hat; die Unterschied sind zu gering.

Die Berechnung der Variationen zeigt dies: [Berechnung im Anhang]

Variation abhängig von der Heapgrösse:	1.3%
Variation abhängig von der JVM:	96.3
Unerklärbare Variation:	2.4%

Beim MsInternet Explorer und beim Netscape liessen sich die Heapgroessen nicht einstellen

Sehr ähnlich sieht es bei den Linux jdks aus:

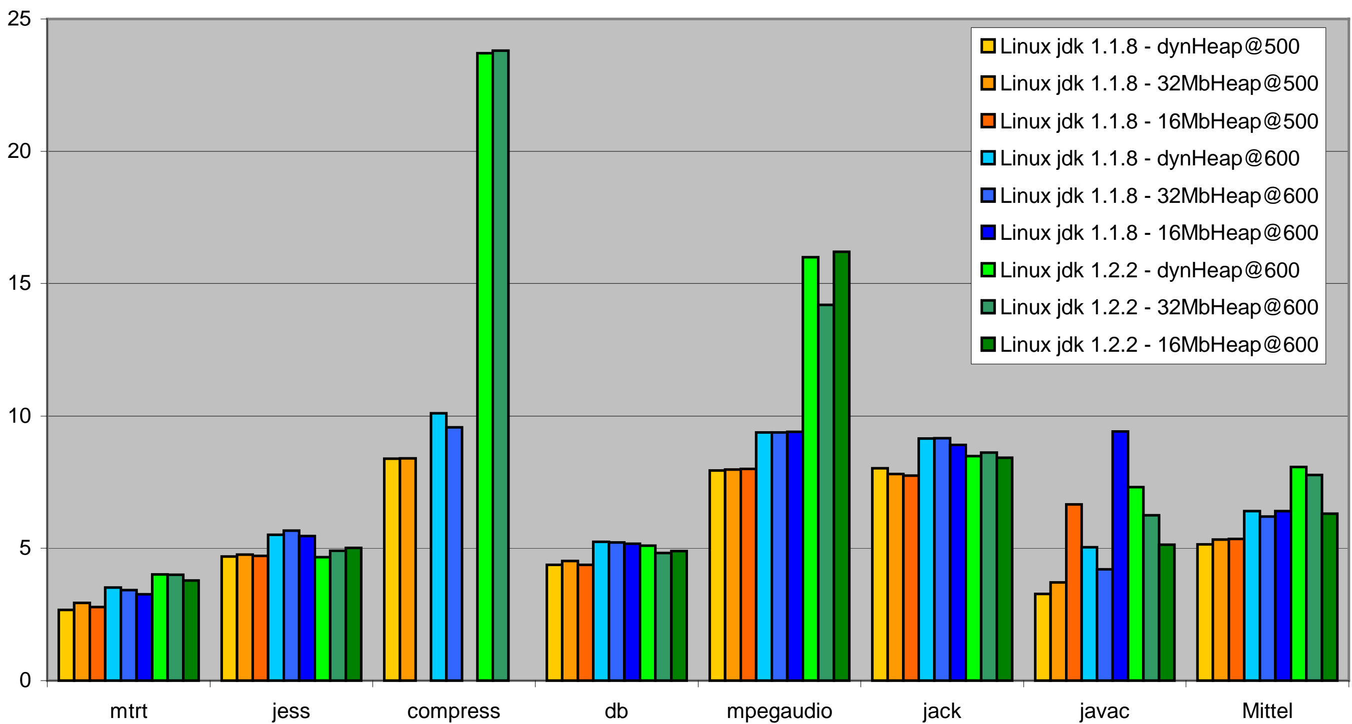

Diagramm 5 – Beziehung zwischen Heapgrösse und JVM unter Linux

Es gilt grundsätzlich das gleiche wie für Windows, doch der „javac" fällt aus dem Rahmen: das jdk1.1.8 wird schneller bei weniger Heap, das jdk1.2.2 aber langsamer.

Einfluss des Just-in-Time-Compilers (JIT)
Beim jdk1.1.8 lässt sich der JIT ausschalten (bei den neueren nicht mehr). Wir fragten uns nun wie gross der Einfluss des JIT wirklich sei. Ohne JIT wird der Javacode nur interpretiert, daher sollte der JIT einen Vorteil bringen.

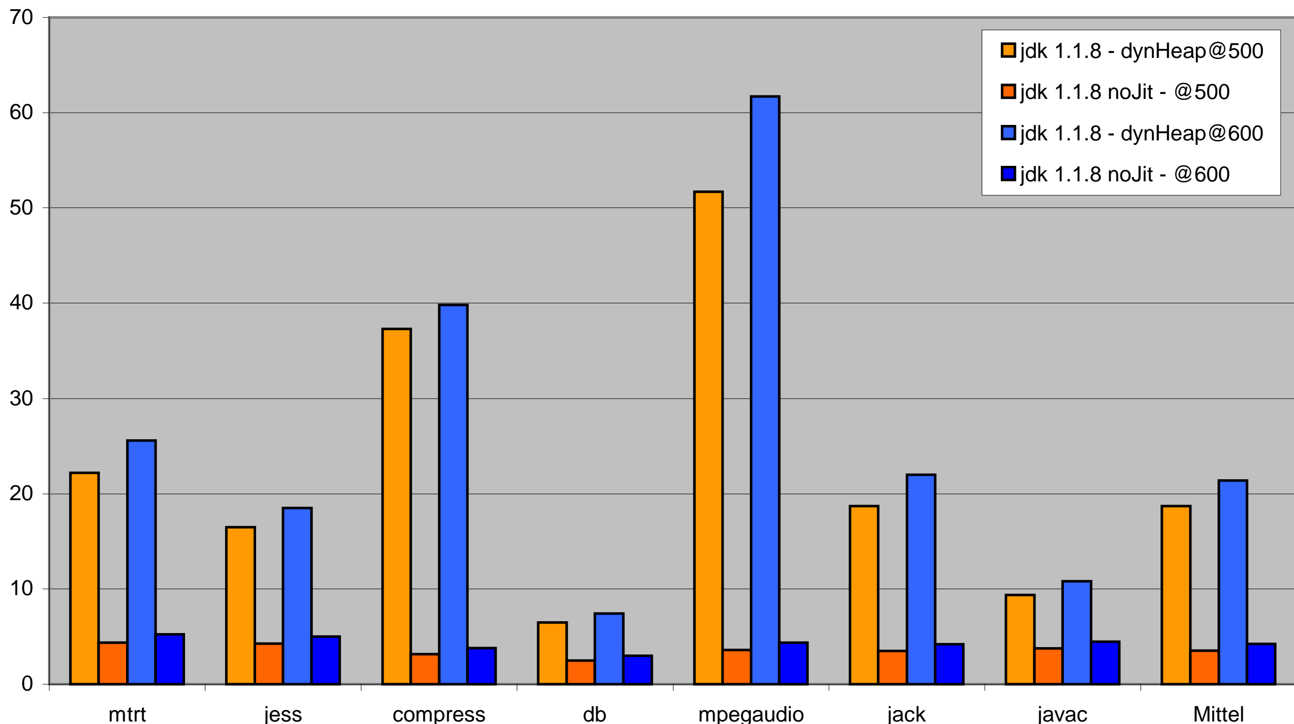

Diagramm 6 – mit/ohne Just-in-Time-Compiler (JIT)

Mit JIT laufen die Benchmarks im Schnitt ca. 5mal schneller als ohne. Bei den speicher- und datenintensiven Benchmarks („db“) fiel der Unterschied etwas kleiner aus (ca. Faktor 2), bei den rechenintensiven Tests („compress“, „mpegaudio“) dafür um so immenser (Faktor >10). Dieser Unterschied lässt sich dadurch erklären, dass datenintensive Programme den Prozessor weniger stark belasten, weil sie auf Disk- oder Speicherzugriffe warten müssen. Dadurch bringt die Codeoptimierung (Kompilation) weniger. Bei rechenintensiven Programmen wird hingegen der Prozessor stark gefordert und Codeoptimierungen bringt sehr viel.

Welches ist nun die Schnellste JVM auf einem Bestimmten System?
Es kann gesagt werden, dass Java auf Windows sicher wesentlich schneller läuft als auf Linux. Die genauen Unterschiede zwischen den beiden System sind uns leider unbekannt. Das Linux-System steckt noch in der Entwicklung – Besserung kann also erwartet werden.

Es ist sinnvoll zwischen den jdk's und den JVM in den Browsern zu unterscheiden. In Browsern laufen häufig Tools als Javaprogramme aus dem Internet, die etwas darstellen oder etwas kurzes berechnen. Auf den jdk's laufen häufiger grössere Applikationen – teilweise die Serverseite von Browserapplikationen - , die über einen längere Zeiträume aktiv sind.

Die jdks auf Windows
Noch einmal alle jdk's für Windows im Überblick:

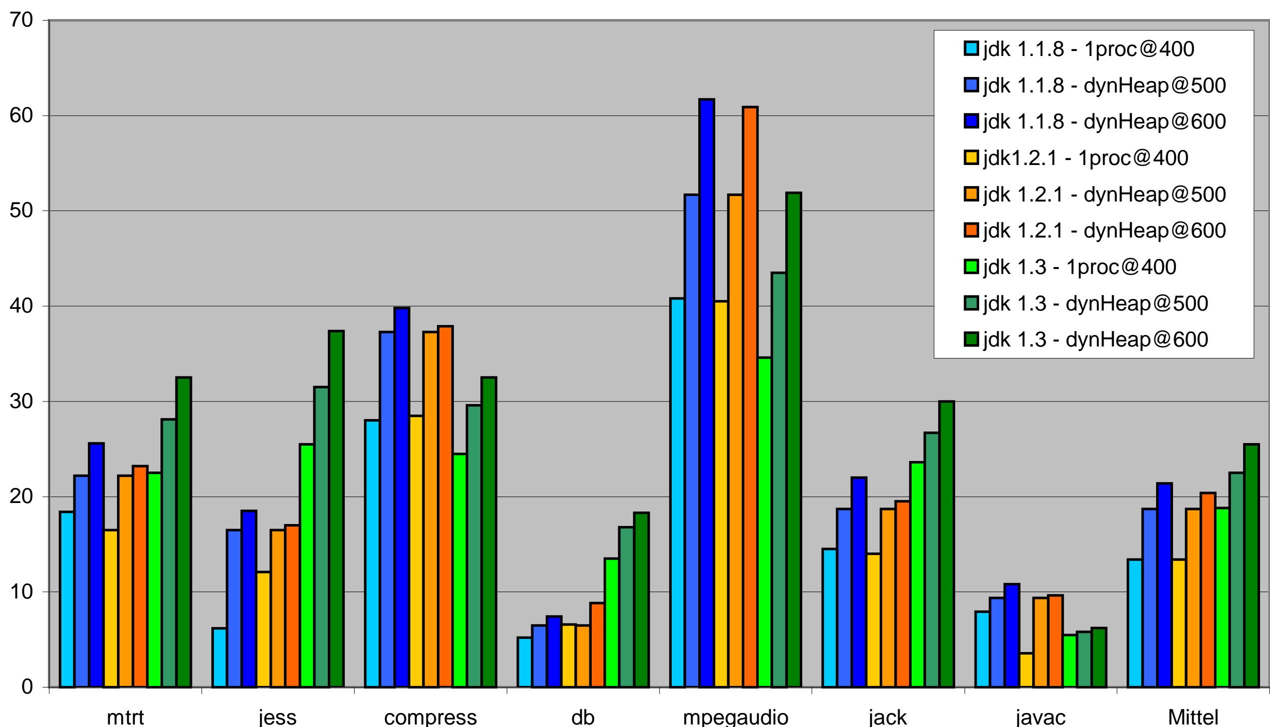

Diagramm 7 – Windows jdk's im Überblick

Die Wahl der JVM spielt eine etwas wichtigere Rolle als die Geschwindigkeit des Prozessors. Zudem ist die Wahl der jvm abhängig von der Aufgabe: während bei rechenintensiven Aufgaben („compress", „mpegaudio") das jdk1.1.8 am schnellsten ist (minimal schneller als jdk1.2.1), schneidet das jdk1.3 bei den restlichen Test am Besten ab, was sicher an der Einführung des Hot-Spot-Optimierers liegt. Allgemein kann also das jdk1.3 als Schnellste JVM gewählt werden. Die Anzahl der Prozessoren und die Heapgrösse sind nicht ausschlaggebend, sofern die Heapgrösse nicht unvernünftig klein gezwungen wird.

Browser in Windows

Ein kurzer Überblick über die getesteten Browsers:

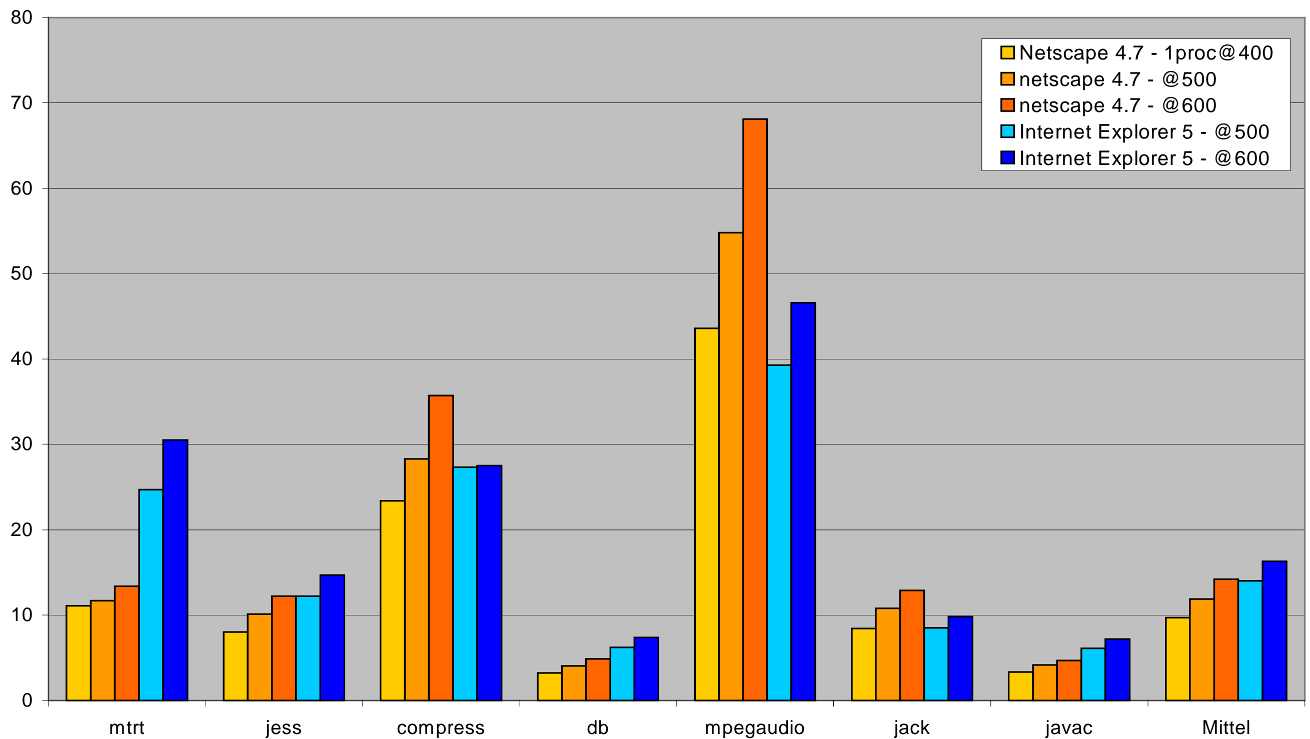

Diagramm 8 – Windows Browser im Überblick

Die Unterschiede zwischen der Netscape JVM und der MsInternet Explorer (IE) VM sind grösser als zwischen den jdk's. Wiederum kommt es auf die Art des Programms (speicherintensiv oder rechenintensiv) an, welche JVM schneller ist. Bei rechenintensiven Programmen („compress", mpegaudio") ist die Netscape JVM schneller, bei den anderen, speicherintensiven, der InternetExplorer. Allgemein ist die IE VM etwas schneller als die Netscape JVM.

7. Fazit

Unser Ziel war es, zu gegebener Hardware die beste JVM zu finden. Aufgrund der vorliegenden Resultate können wir also folgende Schlüsse ziehen:

- CPU: wie erwartet bringt der Einsatz einer zweiten CPU keinen Geschwindigkeitsvorteil. Der Hauptgrund dafür ist, dass die Benchmarks normale Prozesse sind und nicht mit mehreren Threads gearbeitet wird. Interessanter wird's bei der Taktrate: die SPEC ratio steigt nicht genau entsprechend dem Taktratenzuwachs, sondern der PIII@500MHz ist schneller als seine beiden Brüder mit 400 resp. 600 MHz ! Beim PII@400MHz ist dieser Unterschied auf die unterschiedliche Architektur des Prozessors zurückzuführen. Der PIII@600MHz hingegen wird durch Wartezeiten auf Daten aus dem RAM/Cache relativ gesehen stärker gebremst als der PIII@500Mhz.
- Betriebssystem: hier ist eindeutig Windows zu favorisieren. Die JVM unter Windows sind mindestens um Faktor 3 schneller als diejenigen unter Linux. Der Hauptgrund liegt sicher darin, dass die Linux jdks von Haus aus keinen JIT eingebaut haben, sondern wir von Hand einen Freeware-JIT angegeben haben, der offensichtlich nicht sehr effizient arbeitet.
- RAM: erstaunlicherweise bringt mehr RAM nicht unbedingt mehr Leistung. Die Tests mit fest zugewiesenem Heap von 16 und 32MB zeigen, dass die SPEC ratio gleich bleibt, obwohl der Garbage Collector bei 16MB ungefähr doppelt soviel zu tun hat wie bei 32MB. D.h. die verwendeten Garbage Collectoren sind äusserst effizient.
  Leicht langsamer ist die Variante mit dynamischem Heapspeicher. Dies ist darauf zurückzuführen, dass der Heap häppchenweise vom Betriebssystem angefordert wird und deshalb ein kleiner Geschwindigkeitsverlust resultiert. Im Weiteren haben wir festgestellt, dass die Tests mit dynamischem Heap nicht einfach RAM sammeln, bis keines mehr da ist, sondern ab jdk 1.2 sehr wenig Heap brauchen. Mehrere GC-Runs sind also einer zusätzlichen Speicherallokation vorzuziehen.
- JVM: Grundsätzlich kann man sagen, die neueste JVM ist auch die schnellste. Ab jdk 1.3 wird auch die Hot-Spot-Optimierung eingesetzt, was sich deutlich in der Leistung niederschlägt. Browser-JVMs sind etwas langsamer.

8. Referenzen

[1] Rai Jain. *The Art Of Computer Systems Performance Analysis.* John Wiley & Sons, 1991

[2] The Standard Performance Evaluation Corporation SPEC. *The JVM98 Java Virtual Machine Benchmark.* http://www.spec.org/osg/jvm98.

[3] Sun Javasoft. *The Java Hotspott Performance Engine Architecture.* http://java.sun.com/products/hotspot/whitepaper.html

[4] Sun Javasoft. *jdk 1.1 Performance Enhancements.* http://java.sun.com/products/jdk/1.1/docs/guide/performance/performance.html

[5] Sun Javasoft. *Performance Enhancements*. http://java.sun.com/products/jdk/1.2/docs/relnotes/features.html#performance

[6] Sun Javasoft. *Performance Enhancements.* http://java.sun.com/products/jdk/1.3/docs/guide/performance/index.html

[7] Microsoft Corporation. *Microsoft Virtual Machine Overview.* http://www.microsoft.com/java/resource/vm.htm, April 1999

[8] Transvirtual Technologies. *Where you want to run Java, Kaffe is there.* http://www.transvirtual.com/products/datasheet.pdf