Java und C# unter Windows und Linux: ein Geschwindigkeitsvergleich

Stefan Heimann (mail@stefanheimann.net)

6. März 2003

Zusammenfassung

Diese Untersuchung vergleicht die Performanz der Java–Entwicklungsplatform mit der des .NET Rahmenwerkes. Die Messungen wurden dabei unter Windows und Linux durchgeführt. Verwendet wurde die J2SE in der Version 1.4.1 von Sun, die .NET Platform Version 1.0 von Microsoft (unter Windows) und die .NET Platform Version 0.19 des Mono–Projekts (unter Linux). Die Benchmark–Suite besteht aus sechs selbstgeschriebenen Benchmarks und dem SciMark Benchmark. Damit werden häufig verwendete Teile des API abgedeckt. Java zeigte unter beiden Betriebssystemen eine recht ausgeglichene Performanz, während C# unter Windows deutlich schneller lief als unter Linux. Bei drei der sieben Benchmarks zeigte Java unter beiden Betriebssystem eine bessere Leistung, bei drei anderen Benchmarks erwies sich Java unter Linux als die schnellere Alternative, während unter Windows die C# Version des Benchmark besser war. Bei einem Benchmark zeigte C# unter Windows eine bessere Leistung, der Benchmark konnte aber unter Linux nicht ausgeführt werden. Man muss allerdings beachten, dass sich das Mono–Projekt derzeit noch im Beta–Stadium befindet.

Inhaltsverzeichnis

1 Einführung **4**

2 Benchmarks **6**

2.1 Messumgebung 6
2.2 Collection–Benchmark 6
2.3 String–Benchmark 8
2.4 Remoting–Benchmark 9
2.5 I/O–Benchmark 10
2.6 Reflection–Benchmark 11
2.7 Numerischer Benchmark 12
2.8 Phonecode–Benchmark 13
2.9 Vergleich Mono 0.19 mit Mono 0.20 15

3 Schlussfolgerung **15**

A Zusätzliche Daten und Abbildungen **17**

Akürzungsverzeichnis

ANOVA .... Analysis of Variance
API .... Application Programming Interface
CIL .... Common Intermediate Language
CLR .... Common Language Runtime
CLS .... Common Language Specification
J2EE .... Java 2 Enterprise Edition
J2ME .... Java 2 Micro Edition
J2SE .... Java 2 Standard Edition
JNI .... Java Native Interface
JVM .... Java Virtual Machine
MFLOPS .... Millions of Floating Point Operations Per Second
RMI .... Remote Method Invocation

1 Einführung

Java und C# sind mehr als nur moderne, objektorientierte Programmiersprachen. Beide sind Teil einer kompletten Entwicklungsplattform. Microsoft hat mit der .NET Platform ein Gegenstück zur Java–Platform (J2SE, J2ME und J2EE) von Sun geschaffen. Beide Platformen benutzen als Ausführungsmodell eine virtuelle Maschine; auf der Java–Platform ist dies die JVM, während im .NET Rahmenwerk diese Funktionalität die CLR bereitstellt. Die virtuelle Maschine ist nicht nur für das Ausführen von Programmen zuständig, sie ermöglicht darüberhinaus automatische Speicherverwaltung, Umsetzung von Sicherheitskonzepten und einiges mehr.

Auch bei der API hat Microsoft viele Konzepte der Java–Platform übernommen. Ein Unterschied besteht jedoch bei der Platformunabhängigkeit und der Sprachinteroperabilität. Die Java–Platform unterstützt lediglich die Sprache Java. Zwar steht über JNI die Möglichkeit zur Verfügung, Methoden in C zu implementieren und es gibt auch einige in Scriptsprachen, welche mit Java interagieren können (Jython, Peanuts, u.a.). Dennoch ist die Sprachinteroperabilität nicht im Kern der JVM verankert wie das beim .NET Rahmenwerk der Fall ist. C# ist zwar die Hauptsprache von .NET , jedoch ermöglicht die CLS eine nahtlose Integration anderer Programmiersprachen. Eine solche Integration ist zum Beispiel bereits für Visual Basic, C++, Eiffel, Scheme, Oberon u.a. vorhanden.

Bei der Platformunabhängigkeit jedoch hat die Java–Platform dem .NET Rahmenwerk einiges voraus. So gibt es Implementierungen der zur Zeit aktuellen Version 1.4 u.a. für Windows, Linux, Mac, BSD und Solaris, wohingegen .NET primär auf die Windows–Platform ausgelegt ist. Zwar hat Microsoft eine Portierung auf FreeBSD vorgestellt, jedoch ist diese Portierung eher als Machbarkeitsbeweis denn als in der Praxis einsetzbare Version gedacht.

Einen Teil dieser Lücke versucht das Mono–Projekt zu schließen. Mono ist eine Open–Source Implementierung der .NET Platform für Linux. Mono besteht im wesentliche aus einem C#–Compiler, einer Laufzeitumgebung für CIL Code und einer Implementierung der .NET API. Darüberhinaus stehen auch andere nützliche Werkzeuge zur Verfügung (zum Beispiel ein Debugger, ein Dokumentationstool u.a.). Momentan funktionieren der Compiler und die Laufzeitumgebung recht gut, ein großer Teil der API ist implementiert. Mit Gtk# ist auch die platformübergreifende Erstellung von graphischen Benutzeroberflächen möglich. Mono ist derzeit auf der x86 Architektur verfügbar, ein Port auf die PowerPC Architektur ist aber bereits in Arbeit.

In dieser Studie soll nun das Laufzeitverhalten untersucht werden, welches ein Entwickler zu erwarten hat, der mit Java oder C# seine Programme sowohl für Windows als auch für Linux entwickeln möchte. Da es sehr wenige frei verfügbare Benchmarks gibt, die sowohl für Java als auch für C# erhältlich sind (bzw. die mit vertretbarem Aufwand portiert werden können), wird dazu eine eigene Benchmark–Suite verwendet. Lediglich der numerische Benchmark stammt aus einer anderen Quelle (siehe [4] und [5]). Die Benchmark–Suite enthält neben dem

eigentlichen Benchmark Code auch ein in Python geschriebenes Steuerprogramm, welches das Compilieren, das Ausführen und das Protokollieren der Messergebnisse automatisiert. Die Benchmark–Suite kann von [1] heruntergeladen werden.

Die Benchmark–Suite setzt sich aus mehreren Benchmarks zusammen, die jeweils typische Gebiete der Anwendungsprogrammierung abdecken. Im einzelnen sind dies:

Collection: Hier werden Kollektionsdatentypen wie Hashtabellen und Listen untersucht.

String: Dieser Benchmark untersucht die Effizienz der String–Manipulations Funktionen.

I/O: Bei diesem Benchmark geht es um die Untersuchung der Geschwindigkeit von I/O Operationen.

Reflection: Die Fähigkeit der Analyse von Klassen zur Laufzeit wird als Reflection bezeichnet. Diese Fähigkeit ist Bestandteil dieses Tests.

Numerical: Berechnungen, die häufig im wissenschaftlichen Bereich auftreten, werden hier gemessen.

Remoting: In diesem Benchmark werden Methodenaufrufe über das Netzwerk getestet. Der Benchmark ist nur unter Windows durchgeführt worden, die zum Zeitpunkt der Messungen (Februar 2003) die Implementierung des entsprechenden Teils der API im Mono–Projekt noch nicht komplett vorhanden war.

Phonecode: Dies ist ein Applikationsbenchmark. Zu einer Liste von Wörtern und einer Telefonnummer werden alle möglichen Kodierungen der Telefonnummer durch die Wörter gesucht.

Als Metrik wurde die Ausführungszeit gemessen (bzw. davon abgeleitete Metriken wie Kilobytes pro Sekunde oder MFLOPS). Sicherlich sind auch andere Größen wie etwa Speicherverbrauch interessant. Gerade die Messung des Speicherverbrauchs gestaltet sich aber bei Laufzeitsystem mit integriertem Garbage Collector als nicht trivial, da das System prinzipiell erst einmal sämtlichen Speicher, welcher vom Betriebssystem bereitgestellt wird, verbrauchen kann. Auch ist es beispielsweise im .NET Rahmenwerk nicht möglich, die Heapgröße eines C# Programms zu begrenzen.

In den nachfolgenden Abschnitten wird nun jeder Benchmark vorgestellt und die gemessenen Daten analysiert. Aufgrund der Heterogenität der einzelnen Benchmarks wird auf eine Gesamtanalyse verzichtet. Das Zusammenfassen der Daten der einzelnen Benchmarks zu einer einzigen Zahl erscheint zwar verlockend, macht jedoch wenig Sinn und würde nur zu einem Resultat ohne Aussagekraft führen. Auch ist ein solche Zusammenfassung in vielen Fällen nicht sinnvoll, da nur wenige Applikation alle untersuchten Gebiete gleichermaßen abdecken.

2 Benchmarks

2.1 Messumgebung

Die gesamte Benchmark–Suite wurde unter Windows und Linux mit den Programmiersprachen Java und C# jeweils fünfmal durchgeführt. Als Testrechner stand dazu ein PC mit einem AMD Athlon Prozessor (700 MHZ) und 256 MB Arbeitsspeicher zur Verfügung. Auf diesem Rechner waren sowohl Windows 2000, SP 3 als auch Debian GNU/Linux, Kernel 2.4.20 installiert.

Zum Kompilieren und Ausführen der Java–Programme wurde der Compiler und die Virtual Machine aus der Version 1.4.1 der J2SE von Sun verwendet. Die Wahl fiel dabei auf die Client Version der Virtual Machine. Ein Vorabtest hatte ergeben, dass sich die Client und die Server Variante für diese Benchmark–Suite nicht signifikant unterscheiden. Unter Windows wurde die Version 1.0 des .NET Rahmenwerk von Microsoft verwendet, unter Linux kam die Version 0.19 des Mono–Projekts zum Einsatz. Die Optimierungflags der Compiler waren (soweit vorhanden) gesetzt.

Bei Betrachtung der nachfolgenden Messresultate sollte man im Auge behalten, dass es sich bei der Java Version von Sun und der .NET Ausgabe von Microsoft um ausgewachsene Produkte handelt, während sich die .NET Version des Mono–Projekts derzeit noch im Beta–Stadium befindet.

Nachfolgend ist nun eine Vorstellung und Analyse der einzelnen Teile der Benchmark–Suite zu finden. Aus Platz- und Übersichtlichkeitsgründen wurden dabei nicht alle Tabellen und Darstellungen an Ort und Stelle plaziert. Die ausgelassenen Tabellen und Darstellungen sind im Anhang zu finden. Bei allen Benchmarks außer dem Remoting–Benchmark wurde eine $2^k r$ Design mit $k = 2$ und $r = 5$ gewählt. Für eine nähere Beschreibung des $2^k r$ Designs siehe [2], Kapitel 18. Falls nicht anders vermerkt sind die Ausführungszeiten und die Effekte der Faktoren in Millisekunden angegeben.

2.2 Collection–Benchmark

Dieser Benchmark untersucht die Performanz der Kollektions–Datenstrukturen. Zwei sehr häufig verwendete Kollektions–Datenstrukturen sind Listen und Hashtabellen. Daher besteht der Benchmark aus typischen Operationen auf diesen beiden Datenstrukturen.

Die Unterschiede bei der Ausführungszeit sind dabei zu mehr als der Hälfte auf die verwendete Programmiersprache zurückzuführen, das Betriebssystem sowie die Interaktion zwischen der Programmiersprache und dem Betriebssystem sind jeweils zu etwa 1/4 verantwortlich (siehe Tabelle 2). Bei einer mittleren Ausführungszeit von knapp 13 Sekunden lässt sich durch Verwendung von Java eine Verbesserung von etwa 6.6 Sekunden erzielen. Wird der Benchmark unter Windows ausgeführt, verbessert sich die Ausführungszeit um knapp 4.6 Sekunden. Dies ist graphisch in Abbildung 1 dargestellt. Die Effekte der einzelnen Faktoren sind signifikant, was sich aus den Vertrauensintervallen in Tabelle 2

Mean t	I	Language	Platform	Language/Platform
6.544	1	-1 (Java)	-1 (Linux)	1
5.902	1	-1 (Java)	$+1$ (Windows)	-1
28.306	1	$+1$ (C#)	-1 (Linux)	-1
10.596	1	$+1$ (C#)	$+1$ (Windows)	1
Effect	12.837,0	6.614,0	-4.588,0	-4.267,0

Tabelle 1: *Analyse des Collection–Benchmarks (Raw Data).*

Factor	Effect	Percentage of Variation	Confidence Interval (90 %)
I	12.837,0	–	(12.784,572; 12.889,428)
Language	6.614,0	52,694 %	(6.561,572; 6.666,428)
Platform	-4.588,0	25,356 %	(-4.640,428; -4.535,572)
Language/Platform	-4.267,0	21,932 %	(-4.319,428; -4.214,572)
Error	–	0,017 %	–

Tabelle 2: *Analyse des Collection–Benchmarks*

ablesen lässt.

Allerdings sollte auch beachtet werden, dass unter anderem die schlechte Performanz von C# unter Linux dieses klare Resultat beeinflusst. So ist der Teil des Benchmarks, welcher den Listen–Datentyp testet, für C# unter Windows $1,5$ Mal schneller als mit Java. Allerdings ist auch die Hashtabellen Implementierung von C# unter Windows deutlich langsamer als die von Java. Für eine genaue Auflistung der einzelnen Ausführungszeiten sei auf die Tabellen 18 und 17 im Anhang verwiesen.

Da Tabellen ähnlich wie Tabelle 2 auch bei der Analyse der restlichen Benchmarks auftreten werden, sei hier noch eine kurze Erklärung der einzelnen Tabelleneinträge aufgeführt. Der Faktorname „I“ steht für „Identität“. In dieser Zeile ist in der Spalte *Effect* die mittlere Ausführungszeit in Millisekunden zu finden (in diesem Fall beträgt die mittlere Ausführungszeit also 12.837 ms). Die Spalte *Confidence Interval* enthält das Vertraunsintervall der angegebenen Sicherheit. In diesem Falle liegt also die Ausführungszeit mit 90–prozentiger Sicherheit in dem Intervall zwischen $12.784,572$ ms und $12.889,428$ ms. Die nächsten drei Zeilen enthalten die Daten für die primären Faktoren (Programmiersprache und Betriebssystem) sowie die Interaktion zwischen diesen beiden Faktoren. Die Spalte *Effect* gibt Aufschluss über die Laufzeitverbesserung bzw. Laufzeitverschlechterung abhängig vom Level des entsprechenden Faktors. Bezeichne etwa α den Effekt des Faktors A, so hat der Faktor A einen Einfluss von $l\alpha$ Millisekunden auf die Ausführungszeit, wobei l der Koeffizient des jeweiligen Levels ist (l ist also -1 oder 1). Eine Auflistung der Faktoren mit ihren Leveln und den dazugehörenden Koeffizienten findet sich in Tabelle 3. Die Spalte *Percentage of Variation* gibt an, wie stark sich der jeweilige Faktor auf die Variation der Ausführungszeit auswirkt. In der letzten Spalte finden sich wiederum die Vertrauensintervalle für die Effekt. In der letzten Zeile ist schließlich der prozentmäßige Anteil der Fehlers an der Variation zu finden.

Faktor	Level −1	Level 1
Programmiersprache	Java	C#
Betriebssystem	Linux	Windows

Tabelle 3: *Faktoren und Level des $2^k r$–Designs*

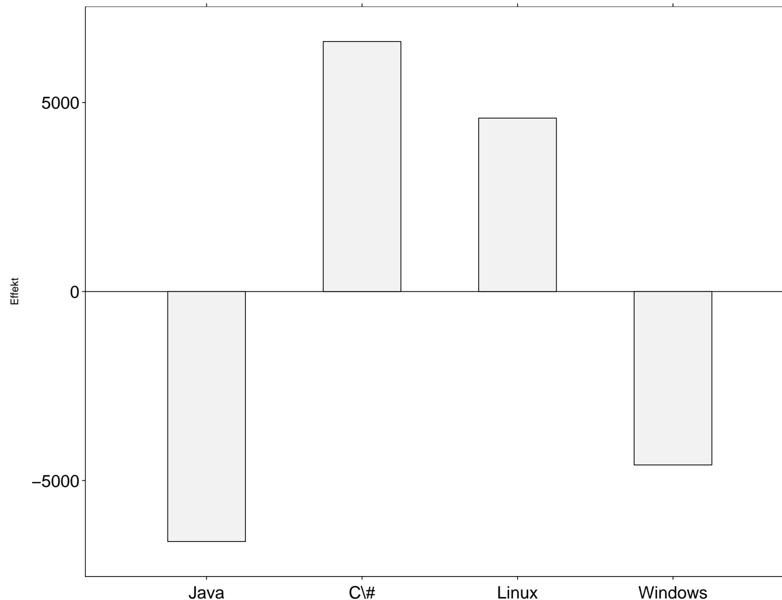

Abbildung 1: *Die Effekte der Hauptfaktoren im Collection–Benchmark*

2.3 String–Benchmark

Bei diesem Benchmark geht es um die Effektivität der String–Routinen. Stellvertretend werden in einem ersten Teil die Funktionen `equals` (vergleicht zwei Strings auf Gleichheit), `charAt` (liefert den Buchstaben an einer bestimmten Position), `indexOf` (liefert die Position des erstmaligen Auftretens eines Buchstabens), `substring` (extrahiert ein Teilwort des Strings) und `trim` (entfernt Whitespace am Anfang und Ende) getestet. Der zweite Teil des Benchmarks besteht aus der Anwendung von regulären Ausdrücken; dabei werden die regulären Ausdrücke `/*[^*]**+([^/*]**+)*/` (passt auf Kommentare wie sie zum Beispiel in C verwendet werden) und `"([^"\\]|\.)*"` (passt auf Text, welcher von Anführungszeichen umgeben ist) auf einen Beispieltext angewandt, wobei das erste Pattern im Text nicht vorkommt, das zweite hingegen schon.

Ein Blick auf Tabelle 5 zeigt ein sehr änliches Bild wie beim Collection–Benchmark: Die Ausführungszeit wird zu mehr als 50% von der Programmiersprache beeinflusst, die Wahl von Java bringt einen Geschwindigkeitsvorteil von etwas mehr als 6 Sekunden bei einer mittleren Ausführungszeit von 12,6 Sekunden. Allerdings sind die Ergebnisse beeinflusst vom schlechten Resultat des ersten Teils in der Kombination C#/Linux. Während dieser Teil unter Windows etwa 7,2 Sekunden benötigt, wurde unter Linux eine Zeit von etwa 24,4 Sekunden gemes-

Mean t	I	Language	Platform	Language/Platform
6.544	1	−1 (Java)	−1 (Linux)	1
6.252	1	−1 (Java)	+1 (Windows)	−1
27.084	1	+1 (C#)	−1 (Linux)	−1
10.835	1	+1 (C#)	+1 (Windows)	1
Effect	12.678,75	6.280,75	-4.135,25	-3.989,25

Tabelle 4: *Analyse des String–Benchmarks (Raw Data).*

Factor	Effect	Percentage of Variation	Confidence Interval (90 %)
I	12.678,75	–	(12.605,939; 12.751,561)
Language	6.280,75	54,418 %	(6.207,939; 6.353,561)
Platform	-4.135,25	23,59 %	(-4.208,061; -4.062,439)
Language/Platform	-3.989,25	21,954 %	(-4.062,061; -3.916,439)
Error	–	0,038 %	–

Tabelle 5: *Analyse des String–Benchmarks*

sen. Dennoch liefert die Analyse dadurch kein allzu verzertes Bild, die Tendenz wird lediglich verstärkt. Tabelle 4 ist zu entnehmen, dass die Ausführungszeit mit Java unter Windows etwa $6,2$ Sekunden, mit Java unter Linux etwa 6.5 Sekunden, mit C# unter Windows etwa $10,8$ Sekunden und mit C# unter Linux etwa 27 Sekunden beträgt.

2.4 Remoting–Benchmark

In diesem Benchmark geht es um Remote Procedure Calls. In Java spricht man hier von Remote Method Invocation (RMI), während im .NET Rahmenwerk einfach nur von Remoting die Rede ist. Neben der Geschwindigkeit der Netzwerkverbindung ist ein effizientes Marshalling und Unmarshalling der Argumente für eine gute Performanz unerlässlich. Da die Güte der Netzwerkverbindung von der Wahl der Entwicklungsplattform unabhängig ist, wurde in den Testläufen das Loopback–Interface verwendet, um eine mögliche Beeinflussung zu vermeiden.

Zur Geschwindigkeitsmessung der Remote–Aufrufe, startet der Benchmark einen Server, auf dem der Client dann entsprechende Methoden aufruft. Leider musste auf eine Durchführung unter Linux verzichtet werden, da die .NET Implementierung des Mono–Projekts momentan keine lauffähige Implementierung der entsprechenden Bibliotheken enthält. Zur Analyse wurde daher ein One–Factor Design verwendet; eine genauere Beschreibung findet sich in [2], Kapitel 20.

In Tabelle 6 sieht man die Ausführungszeiten der einzelnen Durchläufe, deren Mittelwert, die Effekte der beiden Programmiersprachen sowie die Vertrauensintervalle der Effekt mit 90–prozentiger Sicherheit. Man sieht, dass die Verwendung von C# einen positiven Effekt von etwa einer halben Sekunde hat. Tabelle 7 zeigt, dass die Variation der Ausführungsgeschwindigkeit zu mehr als 99% von der Wahl der Programmiersprache abhängt und dass das Ergebnis hochgradig signifikant ist.

	Java	C#	
	3.294	2.253	
	3.395	2.293	
	3.335	2.293	
	3.365	2.293	
	3.405	2.293	
Mean	3.358,8	2.285,0	2.821,9
Effect	536,9	−536,9	
Confidence Interval	(516,605; 557,195)	(-557,195; -516,605)	(2.801,605; 2.842,195)

Tabelle 6: *Auswertung des Remoting–Benchmark*

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F–Computed	F–Table
y	82.523.341,0					
$\hat{y}_{..}$	79.631.196,1					
$y - \hat{y}_{..}$	2.892.144,9		9			
Language	2.882.616,1	99,671	1	2.882.616,1	2.420,129	3,458
Errors	9.528,8	0,329	8	1.191,1		

Tabelle 7: *ANOVA Tabelle für Remoting–Benchmark*

2.5 I/O–Benchmark

Der I/O–Benchmark besteht aus dem Lesen und Schreiben von kleinen (etwa 4 KB) und großen (etwa 1,8 MB) Dateien; dabei wurden sowohl binäre Daten als auch textuelle Daten verwendet. Als Metrik wurde in diesem Benchmark die Einheit Kilobytes pro Sekunde verwendet. Die Angabe *n KB/s* bedeutet dann also, dass pro Sekunde n Kilobyte Daten gelesen und geschrieben wurde. Eine Analyse des $2^k r$ Designs brachte allerdings nicht den gewünschten Erfolg, da die Unterschiede zwischen C# auf Windows (etwa 8.880 KB/s) und C# auf Linux (etwa 1.376 KB/s) einfach zu gross waren. In Abbildung 2 sieht man, dass die Programmiersprache weniger als 1% der Variation ausmacht. Aus diesem Grund wurde eine getrennte Analyse für die beiden Betriebssysteme durchgeführt. Für das Design des Experiments wurde wiederum das bereits angesprochene One–Factor Design verwendet.

Unter Linux hat die Verwendung von Java bei einer durchschnittlichen Leistung von 3366 KBytes/Sekunde einen positiven Effekt von knapp 2000 KBytes/Sekunde wie in Tabelle 8 zu sehen ist. Während

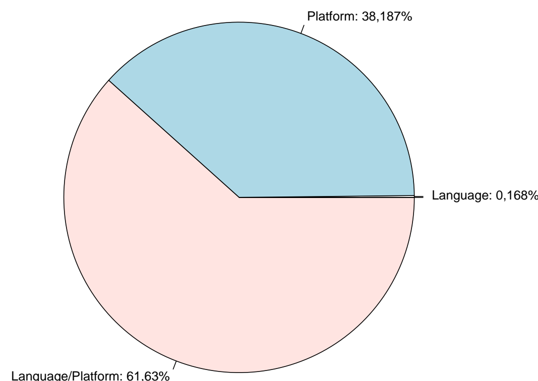

Abbildung 2: *Prozentualer Anteil der Faktoren an der Variation im I/O–Benchmark*

	Java	C#	
	5.346,087	1.377,581	
	5.407,304	1.372,363	
	5.372,092	1.377,332	
	5.372,935	1.377,332	
	5.282,571	1.378,719	
Mean	5.356,198	1.376,665	3.366,432
Effect	1.989,766	−1.989,766	
Confidence Interval	(1.970,383; 2.009,149)	(-2.009,149; -1.970,383)	(3.347,048; 3.385,815)

Tabelle 8: *One–Factor Design für den I/O–Benchmarks unter Linux (Metrik: KB/s)*

	Java	C#	
	4.498,271	8.951,848	
	4.454,377	8.893,711	
	4.448,589	8.870,668	
	4.460,18	8.802,248	
	4.451,481	8.882,175	
Mean	4.462,58	8.880,13	6.671,355
Effect	−2.208,775	2.208,775	
Confidence Interval	(-2.232,619; -2.184,931)	(2.184,931; 2.232,619)	(6.647,511; 6.695,199)

Tabelle 9: *One–Factor Design für den I/O–Benchmarks unter Windows (Metrik: KB/s)*

Java etwa 5350 KB/s schafft, werden mit C# nur etwa 1375 KB/s gelesen und geschrieben. Unter Windows zeigt sich hingegen das umgekehrte Bild (siehe Tabelle 9): Mit C# werden 8880 KB/s übertragen, während es bei Java nur 4462 KB/s sind. Dies entspricht einem Effekt von etwas mehr als 2200 KB/s bei einer Durschnittsleistung von 6671 KBytes/Sekunde. Beide Ergebnisse sind dabei hochgradig signifikant, was eine ANOVA gezeigt hat (siehe Tabellen 21 und 22 im Anhang).

2.6 Reflection–Benchmark

Mit *Reflection* bezeichnet man die Fähigkeit, zur Laufzeit Informationen über eine Klasseninstanz erhalten zu können. Dadurch lassen sich zum Beispiel Methoden aufrufen, deren Name erst zur Laufzeit feststeht. Zusammen mit dem dynamischen Laden von Klassen ergibt sich dadurch eine große Flexibilität, weshalb diese Technik auch häufig in der Praxis zu finden ist. Da Reflection jedoch unvermeidlich einen Performanzverlust beinhaltet, ist es umso wichtiger, diesen möglichst klein zu halten. Daher beinhaltet auch diese Studie einen Reflection–Benchmark.

Der Benchmark testet zum einen das dynamische Laden und Instanzieren einer Klasse und zum anderen den Methodenaufruf per Reflection. Es ist verwunderlich, dass die Geschwindigkeit weniger von der Programmiersprache als vielmehr vom Betriebssystem abängt. Tabelle 11 zeigt, dass mehr als 46% der Variation vom Betriebssystem abhängen, aber lediglich knapp 28% von der Programmiersprache. Zu beachten ist auch die hohe Interaktion der beiden Faktoren, die etwa $26,5\%$ der Variation ausmacht. In Tabelle 10 ist zu sehen, dass unter Windows die beiden Sprachen Java und C# beinahe gleichauf liegen. C# schneidet unter Linux sehr schlecht ab, aber auch Java ist unter Linux deutlich langsamer als unter Windows. Bei einer mittleren Ausführungszeit von etwas mehr als 10 Sekunden verbessert sich die Laufzeit unter Windows

	Java	C#
Windows	3.439 ms	3.686 ms
Linux	6.583 ms	26.858 ms

Tabelle 10: *Mittlere Ausführungszeit des Reflection–Benchmarks.*

Factor	Effect	Percentage of Variation	Confidence Interval (90 %)
I	10.155,0	–	(10.148,293; 10.161,707)
Language	5.117,0	27,709 %	(5.110,293; 5.123,707)
Platform	-6.565,5	45,617 %	(-6.572,207; -6.558,793)
Language/Platform	-5.020,5	26,674 %	(-5.027,207; -5.013,793)
Error	–	0,0 %	–

Tabelle 11: *Analyse des Reflection–Benchmarks*

um etwa $6,5$ Sekunden, während sie sich unter Linux eine Verschlechterung um denselben Wert ergibt. Java hat einen positiven Effekt von etwa 5 Sekunden, während C# also einen negativen Effekt von 5 Sekunden bewirkt. Die große Interaktion der beiden Faktoren zeigt sich in einem Effekt von ebenfalls etwa 5 Sekunden.

2.7 Numerischer Benchmark

Dieser Benchmark ist der einziger der Benchmark–Suite, der nicht vom Autor dieser Studie selbst erstellt wurde. Stattdessen wurde auf den SciMark Benchmark zurückgegriffen, der zuerst für Java geschrieben wurde [5] und später nach C# portiert worden ist [4]. Der Benchmark besteht aus fünf Teilen:

- Fast Fourier Transform
- Jacobi Successive Over–relaxation
- Monte Carlo integration
- Sparse matrix multiply
- dense LU matrix factorization

Die Performanz wurde bei diesem Benchmark in MFLOPS gemessen.

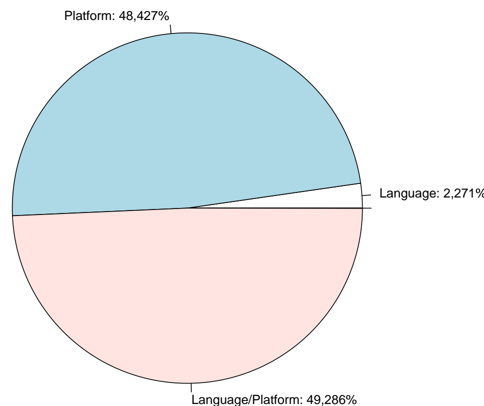

Abbildung 3: *Prozentualer Anteil der Faktoren an der Variation im Numeric–Benchmark*

	Java	C#	
	83,57	42,284	
	83,522	42,324	
	83,202	42,258	
	83,188	42,272	
	83,576	42,317	
Mean	83,412	42,291	62,851
Effect	20,56	−20,56	
Confidence Interval	(20,477; 20,644)	(-20,644; -20,477)	(62,768; 62,935)

Tabelle 12: *One–Factor Design für den Numeric–Benchmarks unter Linux (Metrik: MFLOPS)*

	Java	C#	
	83,825	110,569	
	82,908	109,467	
	82,979	109,603	
	82,931	109,612	
	82,933	109,253	
Mean	83,115	109,701	96,408
Effect	−13,293	13,293	
Confidence Interval	(-13,561; -13,025)	(13,025; 13,561)	(96,14; 96,676)

Tabelle 13: *One–Factor Design für den Numeric–Benchmarks unter Windows (Metrik: MFLOPS)*

Wie schon beim I/O–Benchmark spielt die Programmiersprache im Gegensatz zur Platform und der Interaktion der Faktoren kaum eine Rolle (siehe Abbildung 3). Daher wurde auch bei diesem Benchmark eine getrennte Auswertung für die beiden Betriebssysteme vorgenommen. In Abbildung 12 ist die Auswertung für Linux zu sehen. Java schneidet dabei deutlich besser ab; gegenüber dem Mittelwert von knapp 63 MFLOPS ist Java um etwa 20 MFLOPS besser, während C# mit etwa 42 MFLOPS eine deutlich schlechtere Leistung zeigt. Ein Blick auf Abbildung 13 zeigt, dass Java unter Windows fast die gleiche Leistung wie unter Linux liefert. C# hingegen ist unter Windows nicht nur deutlich schneller als unter Linux sondern auch mit fast 110 MFLOPS um einiges schneller als Java. Die Verwendung von C# unter Windows bringt also einen positiven Effekt von etwa 13 MFLOPS. Die Signifikanz dieser Ergebnisse wurde durch eine ANOVA nachgewiesen; die Daten hierzu sind in den Tabellen 26 und 27 im Anhang zu finden.

2.8 Phonecode–Benchmark

Bei diesem Benchmark handelt es sich um eine Applikations–Benchmark. Prechelt [3] benutzte den Benchmark für eine Vergleichsstudie zwischen verschiedenen Programmiersprache. Für den hier vorliegenden Benchmark wurden Prechelts funktionale Anforderungen in Java und C# umgesetzt. Auch die Testdaten wurden aus dieser Arbeit übernommen.

Bei dem Programm geht es um das Codieren von Telefonnummern durch Buchstaben. Dazu werden jeder Ziffer mehrere Buchstaben zugeordnet. Mit Hilfe eines Wörterbuchs soll dann zu einer gegeben Telefonnummer alle möglichen Wortfolgen gefunden wurden, die eine Codierung der Telefonnummer darstellen. Für eine genaue Spezifikation sei an die

Mean t	I	Language	Platform	Language/Platform
3.257	1	−1 (Java)	−1 (Linux)	1
2.898	1	−1 (Java)	+1 (Windows)	−1
4.049	1	+1 (C#)	−1 (Linux)	−1
2.127	1	+1 (C#)	+1 (Windows)	1
Effect	3.082,75	5,25	-570,25	-390,75

Tabelle 14: *Analyse des Phonecode–Benchmarks (Raw Data).*

Factor	Effect	Percentage of Variation	Confidence Interval (90 %)
I	3.082,75	–	(3.066,579; 3.098,921)
Language	5,25	0,006 %	(-10,921; 21,421)
Platform	-570,25	67,85 %	(-586,421; -554,079)
Language/Platform	-390,75	31,858 %	(-406,921; -374,579)
Error	–	0,286 %	–

Tabelle 15: *Analyse des Phonecode–Benchmarks*

Arbeit von Prechelt [3] verwiesen.

Das Wörterbuch liegt in Form einer Textdatei vor und umfasst 73.113 Wörter. Die zu codierenden Telefonnummern werden ebenfalls aus einer Textdatei gelesen; insgesamt sind 1.000 Nummern zu codieren. Das Lesen der Eingabedaten fordert also ein performantes I/O System. Zur internen Datenrepräsentation kommt bei beiden Implementierungen eine Hashtabelle zum Einsatz. Da zur Suche möglicher Wortfolgen ein rekursiver Algorithmus benutzt wird, finden recht viele Methodenaufrufe statt.

Die Programmiersprache für sich allein genommen hat keinen Einfluss auf die Ausführungsgeschwindigkeit. Der Effekt der Programmiersprache liegt mit 90 prozentiger Sicherheit zwischen -10,921 und 21,421 Millisekunden, man kann hierbei also nicht von einem signifikanten Effekt sprechen (siehe Tabelle 15) . Zum größten Teil verantwortlich für die Unterschiede bei der Ausführungsgeschwindigkeit ist das Betriebssystem. Der Benchmark läuft unter Windows bei einer durchschnittlichen Laufzeit von ungefähr 3 Sekunden mehr als eine Sekunde schneller als unter Linux. Das Betriebssystem ist damit für etwa 67% der Variation verwantwortlich. Ganz unwichtig ist die Programmiersprache allerdings auch nicht, da das Zusammenspiel zwischen Programmiersprache und Betriebssystem fast 32% der Variation ausmacht. Dies liegt daran, dass unter Windows die C#–Version des Benchmarks durchschnittlich um 771 Millisekunden schneller als die Java–Version ist, während unter Linux die Java–Version eine Verbesserung von 792 Millisekunden gegenüber C# bringt.

Da der Phonecode–Benchmark viel I/O erfordert, sind diese Ergebnisse nicht überraschend. Auch beim I/O–Benchmark war das Betriebssystem wichtiger für die Performanz als die Programmiersprache. Dieser Effekt ist beim Phonecode–Benchmark in abgeschwächter Form zu finden, da hier auch noch andere Faktoren in die Performanz eingehen.

	0.19	0.20
collection-list	9.443,4	9.475,333
collection-map	18.863,4	22.502,0
string-misc	24.392,8	27.212,333
string-regex	2.692,0	3.506,667
phonecode	4.049,4	4.589,0
reflection-invocation	13.757,6	12.681,667
reflection-instantiation	13.100,4	11.866,667
io-text-small	11.981,2	15.489,667
io-text-big	26.262,0	34.783,667
io-bin-small	6.405,4	34.738,0
io-bin-big	5.089,6	43.561,333
numerical-fft	24,011	24,927
numerical-lu	51,712	35,019
numerical-matmult	36,928	41,886
numerical-monte-carlo	10,883	10,566
numerical-sor	87,921	76,119

Tabelle 16: *Vergleich zwischen Mono Version 0.19 und Version 0.20. Zu sehen ist jeweils die mittlere Ausführungsgeschwindigkeit in Millisekunden, lediglich bei den numerischen Benchmarks ist die mittlere Performanz in MFLOPS angegeben.*

2.9 Vergleich Mono 0.19 mit Mono 0.20

Nachdem die Messungen und Analysen bereits durchgeführt waren, erschien am 25.2.2003 die Version 0.20 von Mono (Version 0.21 vom 27.2.2003 ist lediglich ein Bugfix Release). Daher ist in Tabelle 16 ein kurzer Vergleich der Laufzeiten der Benchmarks zwischen Mono 0.19 und Mono 0.20 zu finden.

Die Version 0.19 schneidet überall besser ab, lediglich beim Reflection–Benchmark ist die Version 0.20 geringfügig schneller. Auffallend sind desweiteren die zum Teil wesentlich schlechteren Resultate der Version 0.20, etwa bei *io-bin-small* und *io-bin-big*. Die Verwendung der Version 0.19 in dieser Untersuchung hat also sicher nicht zur Benachteiligung von Mono geführt.

3 Schlussfolgerung

Abschließend möchte ich die einzelnen Resultate zusammenfassen und versuchen, ein Fazit zu ziehen. Die Resultate der Benchmarks lassen sich grob in vier Gruppen einteilen:

Collection, String Bei diesen beiden Benchmarks schneidet Java deutlich besser ab als C#. Zwar verstärkt das sehr schlechte Abschneiden von C# unter Linux diesen Effekt, aber auch unter Windows ist C# langsamer als Java. Java zeigt unter beiden Platformen ein sehr ähnliches Laufzeitverhalten.

I/O, Numeric, Phonecode Hier hat das verwendete Betriebssystem

einen größeren Einfluss als die Programmiersprache. Beim I/O– und beim Numeric–Benchmark ist der Einfluss der Programmiersprache praktisch null, so dass hier eine nach Betriebssystem getrennte Analyse vorgenommen wurde. Bei allen drei Benchmarks schneidet Java unter Linux besser ab, unter Windows ist allerdings C# schneller. Java ist unter beiden Betriebssystemen in etwa gleichschnell, während C# unter Windows deutlich schneller als unter Linux ist.

Remoting Eine Messung unter Linux war nicht möglich, da die entsprechende Funktionalität in Mono noch nicht implemetiert ist. Unter Windows schneidet C# besser ab als Java.

Reflection Auffallend bei dem Reflection–Benchmark ist das deutlich schlechtere Abschneiden beider Programmiersprachen unter Linux. Unter Windows ist der Unterschied zwischen Java und C# marginal, unter Linux hingegen ist Java besser als C#.

Java zeigt unter beiden Betriebssystemen eine recht ausgeglichene Performanz, während C# unter Windows deutlich schneller läuft als unter Linux. Allerdings befindet sich das Mono–Projekt noch im Beta–Stadium, weshalb diese Aussage lediglich eine Momentaufnahme ist. Der Benchmarks sollte wiederholt werden, sobald Mono in Version 1.0 vorliegt, um eine größere Aussagekraft zu erzielen.

A Zusätzliche Daten und Abbildungen

	Windows									
	Java					C#				
	1	2	3	4	5	1	2	3	4	5
collection-list										
listsize	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
runs	3.488	3.488	3.488	3.488	3.488	3.488	3.488	3.488	3.488	3.488
time	2.794	2.804	2.754	2.794	2.754	1.822	1.852	1.852	1.842	1.842
collection-map										
mapsize	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
runs	3.359	3.359	3.359	3.359	3.359	3.359	3.359	3.359	3.359	3.359
time	3.074	3.115	3.184	3.124	3.115	8.702	8.762	8.782	8.822	8.702
io-bin-big										
filesize	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888
runs	12	12	12	12	12	12	12	12	12	12
time	2.884	2.924	2.904	2.914	2.914	1.772	1.792	1.812	1.802	1.802
io-bin-small										
filesize	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277
runs	4.030	4.030	4.030	4.030	4.030	4.030	4.030	4.030	4.030	4.030
time	3.465	3.485	3.525	3.465	3.495	2.433	2.453	2.463	2.493	2.443
io-text-big										
filesize	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888
runs	11	11	11	11	11	11	11	11	11	11
time	4.837	4.897	4.887	4.897	4.887	1.902	1.912	1.902	1.922	1.912
io-text-small										
filesize	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277
runs	1.835	1.835	1.835	1.835	1.835	1.835	1.835	1.835	1.835	1.835
time	4.036	4.066	4.076	4.076	4.086	1.542	1.542	1.542	1.562	1.552
numerical-fft										
mflops	34,674	34,228	34,455	34,238	34,335	77,479	76,661	76,932	76,932	76,661
time	6.339	6.409	6.379	6.399	6.390	5.628	5.698	5.688	5.688	5.698
numerical-lu										
mflops	126,701	125,306	125,306	125,352	125,306	183,249	181,782	181,783	181,783	181,299
time	5.418	5.488	5.478	5.488	5.488	7.450	7.530	7.540	7.530	7.530
numerical-matmult										
mflops	78,656	77,908	77,908	77,908	78,28	116,86	115,215	115,622	115,214	115,215
time	4.186	4.216	4.236	4.226	4.216	5.618	5.688	5.678	5.688	5.688
numerical-monte-carlo										
mflops	14,96	14,57	14,698	14,627	14,762	15,441	15,441	15,441	15,37	15,37
time	4.486	4.587	4.576	4.577	4.567	4.376	4.446	4.436	4.416	4.416
numerical-sor										
mflops	164,136	162,53	162,53	162,53	161,983	159,816	158,239	158,239	158,761	157,72
time	5.889	5.948	5.949	5.959	5.958	6.028	6.078	6.088	6.078	6.088
phonecode										
time	2.854	2.914	2.904	2.904	2.914	2.113	2.123	2.133	2.133	2.133
reflection-instantiation										
runs	160.580	160.580	160.580	160.580	160.580	160.580	160.580	160.580	160.580	160.580
time	1.903	1.933	1.933	1.933	1.932	1.191	1.191	1.201	1.211	1.201
reflection-invocation										
runs	269.614	269.614	269.614	269.614	269.614	269.614	269.614	269.614	269.614	269.614
time	1.552	1.562	1.572	1.572	1.573	2.463	2.493	2.493	2.483	2.503
remoting										
runs	700	700	700	700	700	700	700	700	700	700
time	3.294	3.395	3.335	3.365	3.405	2.253	2.293	2.293	2.293	2.293
string-misc										
runs	664.124	664.124	664.124	664.124	664.124	664.124	664.124	664.124	664.124	664.124

	Windows									
	Java					C#				
	1	2	3	4	5	1	2	3	4	5
time	3.115	3.154	3.145	3.145	3.154	7.200	7.280	7.270	7.270	7.270
string-regex										
runs	1.400	1.400	1.400	1.400	1.400	1.400	1.400	1.400	1.400	1.400
time	3.084	3.115	3.114	3.124	3.114	3.545	3.585	3.585	3.585	3.585

Tabelle 17: Daten der Benchmark–Suite Durchläufe unter Windows.

	Linux									
	Java					C#				
	1	2	3	4	5	1	2	3	4	5
collection-list										
listsize	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
runs	3.488	3.488	3.488	3.488	3.488	3.488	3.488	3.488	3.488	3.488
time	3.284	3.277	3.284	3.289	3.280	9.443	9.444	9.443	9.444	9.443
collection-map										
mapsize	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
runs	3.359	3.359	3.359	3.359	3.359	3.359	3.359	3.359	3.359	3.359
time	3.258	3.268	3.257	3.268	3.256	18.847	18.915	18.891	18.808	18.856
io-bin-big										
filesize	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888
runs	12	12	12	12	12	12	12	12	12	12
time	3.197	3.161	3.164	3.175	3.153	5.087	5.090	5.090	5.091	5.090
io-bin-small										
filesize	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277
runs	4.030	4.030	4.030	4.030	4.030	4.030	4.030	4.030	4.030	4.030
time	3.257	3.203	3.284	3.261	3.322	6.387	6.454	6.406	6.416	6.364
io-text-big										
filesize	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888	1.957.888
runs	11	11	11	11	11	11	11	11	11	11
time	3.127	3.084	3.090	3.086	3.085	26.255	26.312	26.251	26.238	26.254
io-text-small										
filesize	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277	4.277
runs	1.835	1.835	1.835	1.835	1.835	1.835	1.835	1.835	1.835	1.835
time	3.227	3.215	3.208	3.222	3.402	11.976	12.038	11.967	11.969	11.956
numerical-fft										
mflops	34,185	34,163	34,466	33,982	34,619	24,037	24,028	24,039	23,95	24,003
time	6.454	6.448	6.398	6.491	6.387	4.560	4.558	4.555	4.566	4.560
numerical-lu										
mflops	126,373	126,139	126,326	126,467	126,279	51,764	51,798	51,521	51,697	51,778
time	5.456	5.462	5.457	5.451	5.458	6.600	6.592	6.629	6.605	6.595
numerical-matmult										
mflops	77,723	77,723	75,817	75,782	77,576	36,93	36,96	36,89	36,949	36,909
time	4.248	4.247	4.353	4.355	4.254	4.441	4.436	4.447	4.438	4.442
numerical-monte-carlo										
mflops	14,873	14,887	14,873	14,9	14,88	10,802	10,937	10,939	10,794	10,943
time	4.517	4.516	4.516	4.527	4.517	6.242	6.206	6.204	6.405	6.287
numerical-sor										
mflops	164,697	164,697	164,528	164,81	164,528	87,888	87,895	87,902	87,968	87,954
time	5.874	5.871	5.876	5.868	5.877	5.489	5.489	5.490	5.486	5.487
phonecode										
time	3.294	3.252	3.254	3.244	3.242	4.051	4.057	4.048	4.043	4.048

	Linux									
	Java					C#				
	1	2	3	4	5	1	2	3	4	5
reflection-instantiation										
runs	160.580	160.580	160.580	160.580	160.580	160.580	160.580	160.580	160.580	160.580
time	3.304	3.309	3.301	3.299	3.301	13.090	13.099	13.101	13.106	13.106
reflection-invocation										
runs	269.614	269.614	269.614	269.614	269.614	269.614	269.614	269.614	269.614	269.614
time	3.274	3.279	3.283	3.280	3.285	13.753	13.751	13.742	13.776	13.766
remoting										
runs	–	–	–	–	–	–	–	–	–	–
time	–	–	–	–	–	–	–	–	–	–
string-misc										
runs	664.124	664.124	664.124	664.124	664.124	664.124	664.124	664.124	664.124	664.124
time	3.297	3.332	3.299	3.295	3.297	24.403	24.395	24.391	24.392	24.383
string-regex										
runs	1.400	1.400	1.400	1.400	1.400	1.400	1.400	1.400	1.400	1.400
time	3.240	3.239	3.241	3.240	3.243	2.583	3.124	2.584	2.584	2.585

Tabelle 18: Daten der Benchmark–Suite Durchläufe unter Linux.

Mean t	I	Language	Platform	Language/Platform
5.356,198	1	−1 (Java)	−1 (Linux)	1
4.462,58	1	−1 (Java)	+1 (Windows)	−1
1.376,665	1	+1 (C#)	−1 (Linux)	−1
8.880,13	1	+1 (C#)	+1 (Windows)	1
Effect	5.018,893	109,504	1.652,462	2.099,271

Tabelle 19: *Analyse des I/O–Benchmarks (Raw Data).*

Factor	Effect	Percentage of Variation	Confidence Interval (90 %)
I	5.018,893	–	(5.004,468; 5.033,318)
Language	109,504	0,168 %	(95,079; 123,93)
Platform	1.652,462	38,187 %	(1.638,037; 1.666,887)
Language/Platform	2.099,271	61,63 %	(2.084,845; 2.113,696)
Error	–	0,015 %	–

Tabelle 20: *Analyse des I/O–Benchmarks*

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F–Computed	F–Table
y	152.929.002,328					
$\hat{y}_{..}$	113.328.617,054					
$y - \hat{y}_{..}$	39.600.385,274		9			
Language	39.591.693,125	99,978	1	39.591.693,125	36.439,036	3,458
Errors	8.692,149	0,022	8	1.086,519		

Tabelle 21: *ANOVA Tabelle des One–Factor Designs für den I/O–Benchmark unter Linux (Metrik: KB/s)*

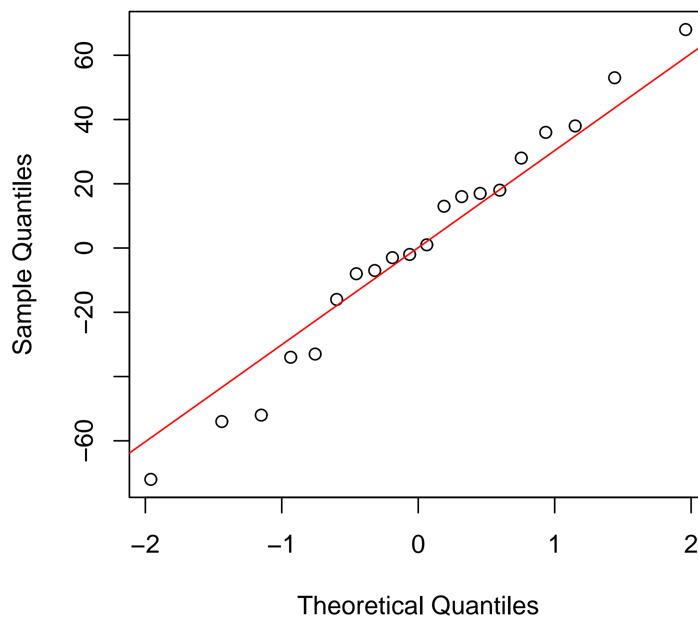

Abbildung 4: *Quantile–Quantile Plot für die Fehler des Collection–Benchmarks*

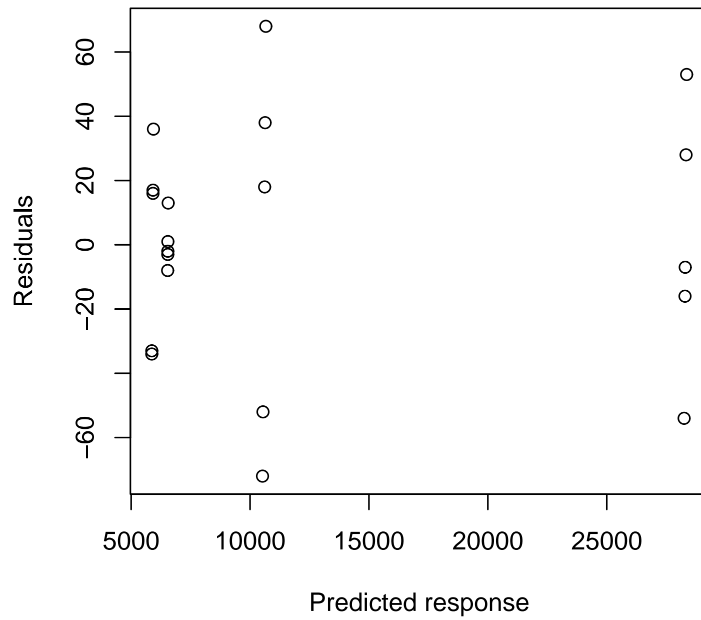

Abbildung 5: *Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim Collection–Benchmark*

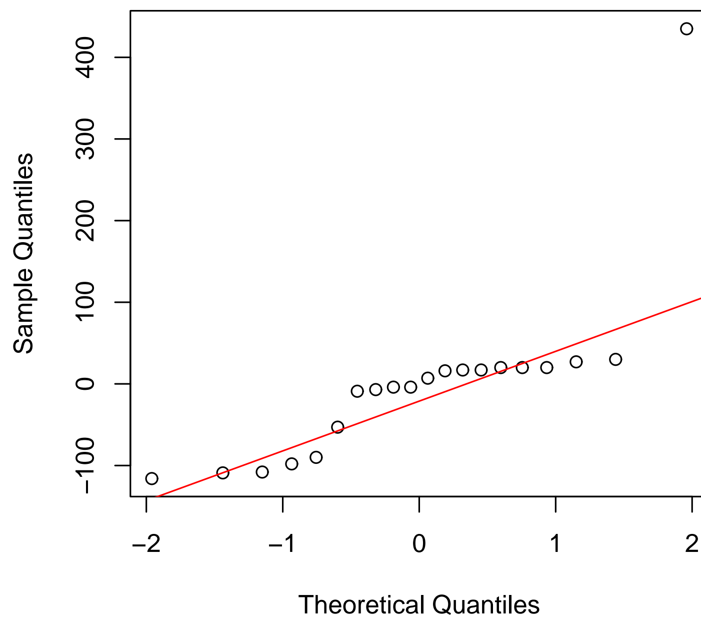

Abbildung 6: *Quantile–Quantile Plot für die Fehler des String–Benchmarks*

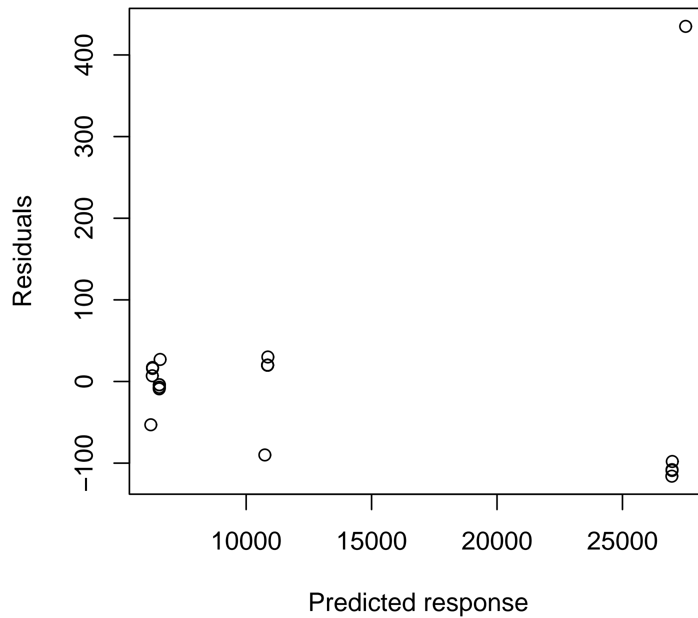

Abbildung 7: *Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim String–Benchmark*

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F–Computed	F–Table
y	493.869.778,616					
$\hat{y}_{..}$	445.069.752,547					
$y - \hat{y}_{..}$	48.800.026,07		9			
Language	48.786.872,948	99,973	1	48.786.872,948	29.673,182	3,458
Errors	13.153,122	0,027	8	1.644,14		

Tabelle 22: *ANOVA Tabelle des One–Factor Designs für den I/O–Benchmark unter Windows (Metrik: KB/s)*

Mean t	I	Language	Platform	Language/Platform
6.583	1	−1 (Java)	−1 (Linux)	1
3.493	1	−1 (Java)	+1 (Windows)	−1
26.858	1	+1 (C#)	−1 (Linux)	−1
3.686	1	+1 (C#)	+1 (Windows)	1
Effect	10.155,0	5.117,0	-6.565,5	-5.020,5

Tabelle 23: *Analyse des Reflection–Benchmarks (Raw Data).*

Mean mflops	I	Language	Platform	Language/Platform
83,412	1	−1 (Java)	−1 (Linux)	1
83,115	1	−1 (Java)	+1 (Windows)	−1
42,291	1	+1 (C#)	−1 (Linux)	−1
109,701	1	+1 (C#)	+1 (Windows)	1
Effect	79,63	-3,634	16,778	16,927

Tabelle 24: *Analyse des Numeric–Benchmarks (Raw Data).*

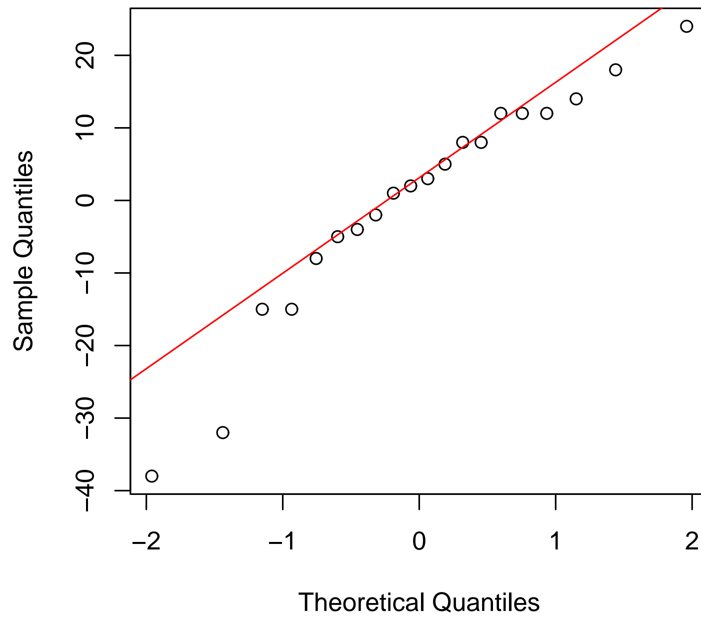

Abbildung 8: *Quantile–Quantile Plot für die Fehler des Reflection–Benchmarks*

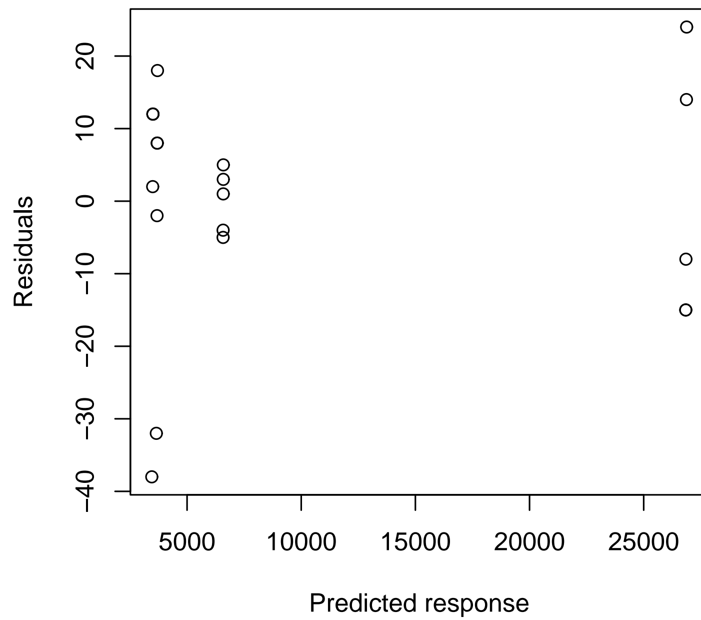

Abbildung 9: *Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim Reflection–Benchmark*

Factor	Effect	Percentage of Variation	Confidence Interval (90 %)
I	79,63	–	(79,498; 79,761)
Language	-3,634	2,271 %	(-3,765; -3,502)
Platform	16,778	48,427 %	(16,647; 16,91)
Language/Platform	16,927	49,286 %	(16,795; 17,058)
Error	–	0,016 %	–

Tabelle 25: *Analyse des Numeric–Benchmarks*

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F–Computed	F–Table
y	94.714,001					
$\hat{y}_{..}$	92.945,37					
$y - \hat{y}_{..}$	1.768,631		9			
Language	1.766,972	99,906	1	1.766,972	8.520,922	3,458
Errors	1,659	0,094	8	0,207		

Tabelle 26: *ANOVA Tabelle des One–Factor Designs für den Numeric–Benchmark unter Windows (Metrik: MFLOPS)*

Component	Sum of Squares	Percentage of Variation	Degrees of Freedom	Mean Square	F–Computed	F–Table
y	43.730,394					
$\hat{y}_{..}$	39.502,951					
$y - \hat{y}_{..}$	4.227,443		9			
Language	4.227,281	99,996	1	4.227,281	209.218,727	3,458
Errors	0,162	0,004	8	0,02		

Tabelle 27: *ANOVA Tabelle des One–Factor Designs für den Numeric–Benchmark unter Linux (Metrik: MFLOPS)*

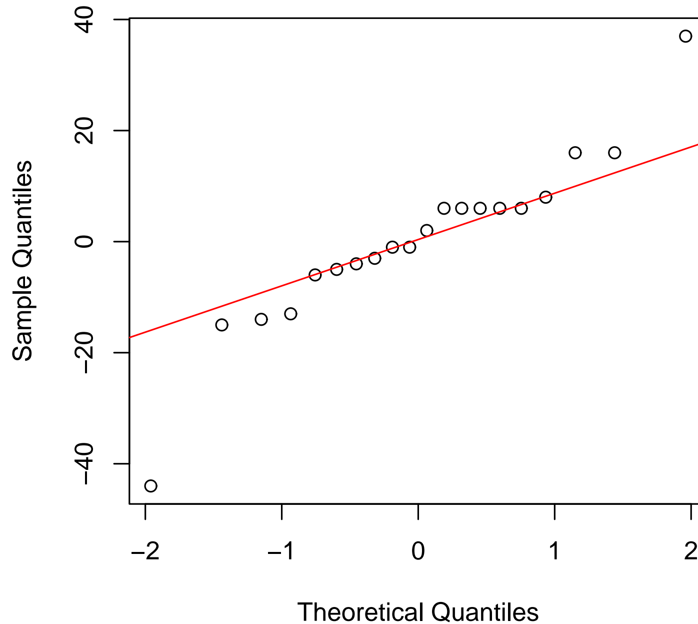

Abbildung 10: *Quantile–Quantile Plot für die Fehler des Phonecode–Benchmarks*

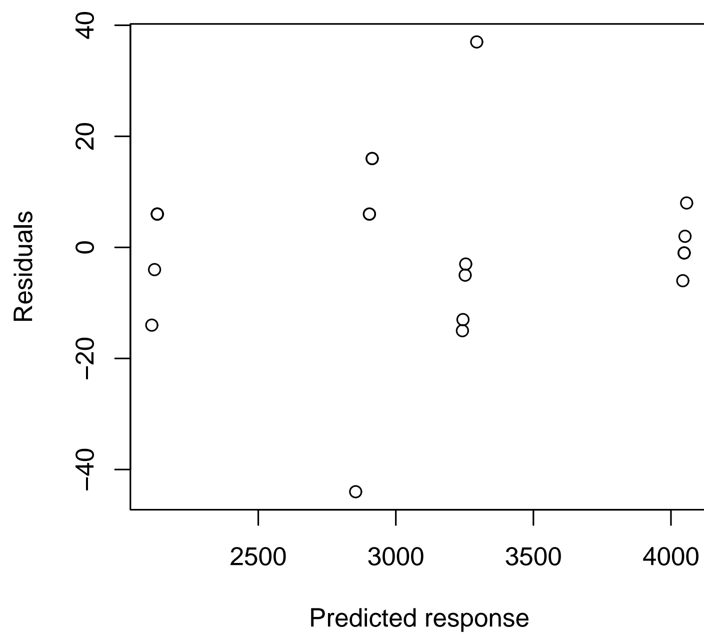

Abbildung 11: *Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim Phonecode–Benchmark*

Tabellenverzeichnis

1 Analyse des Collection–Benchmarks (Raw Data). 7
2 Analyse des Collection–Benchmarks 7
3 Faktoren und Level des $2^k r$–Designs 8
4 Analyse des String–Benchmarks (Raw Data). 9
5 Analyse des String–Benchmarks 9
6 Auswertung des Remoting–Benchmark 10
7 ANOVA Tabelle für Remoting–Benchmark 10
8 One–Factor Design für den I/O–Benchmarks unter Linux (Metrik: KB/s) 11
9 One–Factor Design für den I/O–Benchmarks unter Windows (Metrik: KB/s) 11
10 Mittlere Ausführungszeit des Reflection–Benchmarks. . 12
11 Analyse des Reflection–Benchmarks 12
12 One–Factor Design für den Numeric–Benchmarks unter Linux (Metrik: MFLOPS) 13
13 One–Factor Design für den Numeric–Benchmarks unter Windows (Metrik: MFLOPS) 13
14 Analyse des Phonecode–Benchmarks (Raw Data). 14
15 Analyse des Phonecode–Benchmarks 14
16 Vergleich zwischen Mono Version 0.19 und Version 0.20. Zu sehen ist jeweils die mittlere Ausführungsgeschwindigkeit in Millisekunden, lediglich bei den numerischen Benchmarks ist die mittlere Performanz in MFLOPS angegeben. 15
17 Daten der Benchmark–Suite Durchläufe unter Windows. 19
18 Daten der Benchmark–Suite Durchläufe unter Linux. . . 20
19 Analyse des I/O–Benchmarks (Raw Data). 21
20 Analyse des I/O–Benchmarks 21
21 ANOVA Tabelle des One–Factor Designs für den I/O–Benchmark unter Linux (Metrik: KB/s) 21
22 ANOVA Tabelle des One–Factor Designs für den I/O–Benchmark unter Windows (Metrik: KB/s) 23
23 Analyse des Reflection–Benchmarks (Raw Data). 23
24 Analyse des Numeric–Benchmarks (Raw Data). 23
25 Analyse des Numeric–Benchmarks 25
26 ANOVA Tabelle des One–Factor Designs für den Numeric–Benchmark unter Windows (Metrik: MFLOPS) 25
27 ANOVA Tabelle des One–Factor Designs für den Numeric–Benchmark unter Linux (Metrik: MFLOPS) 25

Abbildungsverzeichnis

1 Die Effekte der Hauptfaktoren im Collection–Benchmark 8
2 Prozentualer Anteil der Faktoren an der Variation im I/O–Benchmark 10
3 Prozentualer Anteil der Faktoren an der Variation im Numeric–Benchmark 12

4 Quantile–Quantile Plot für die Fehler des Collection–Benchmarks 21
5 Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim Collection–Benchmark 22
6 Quantile–Quantile Plot für die Fehler des String–Benchmarks 22
7 Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim String–Benchmark 23
8 Quantile–Quantile Plot für die Fehler des Reflection–Benchmarks 24
9 Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim Reflection–Benchmark 24
10 Quantile–Quantile Plot für die Fehler des Phonecode–Benchmarks 25
11 Höhe des Fehlerterms im Verhältnis zur Antwortzeit beim Phonecode–Benchmark 26

Literatur

[1] http://www.stefanheimann.net/benchmark.

[2] Raj Jain. *The Art of Computer Systems Performance Analysis.* John Wiley & Son, Inc., 1991.

[3] Lutz Prechelt. An empirical comparison of seven programming languages. 2000.

[4] SciMark für C#: http://rotor.cs.cornell.edu/SciMark/.

[5] SciMark Homepage: http://math.nist.gov/scimark2/.