Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX

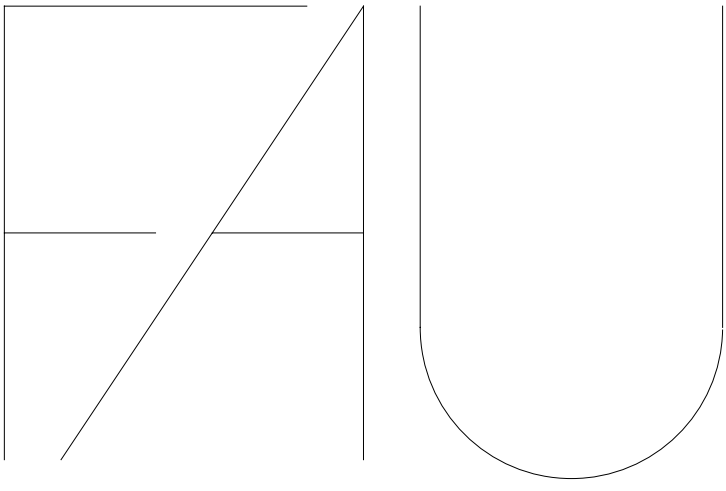
Christian Wawersich

April 2001 DA-I4-2001-05

Diplomarbeit

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)



Design und Implementierung eines Profilers und optimierenden Compilers für das Betriebssystem JX

Diplomarbeit im Fach Informatik

vorgelegt von

Christian Wawersich

geb. am 20.08.1970 in Landshut

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	***Prof. Dr. F. Hofmann***
	Dipl.-Inf. Michael Golm
Beginn der Arbeit:	1. Oktober 2000
Abgabe der Arbeit:	2. April 2001

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 02.04.2001 _____

Abstract

In most of the current operating systems memory protection is based on hardware, which is generally believed to have a better run time behavior than its software based alternative. However, software based memory protection provides a much more flexible way of allocating system resources. Therefore it can more than compare with traditional operating systems, provided a comparable effort in program and system optimization is made.

JX is an operating system whose major parts are implemented in the Java programming language. It simply uses the memory protection design of Java as a software based memory protection. This thesis introduces two tools, which were contributed to the JX project: the first tool supports the programmer in improving a Java program, the second tool improves the performance of the execution of the Java byte-code.

In order to compose faster code, it is necessary for the programmer to identify the most time consuming parts. These so called “hot spots of execution” often indicate poor code or bottle necks and can be found by the use of profilers. This thesis illustrates the iplementation of such a profiler for the JX system measuring the execution time of each method invocation.

The time logging is realized by counting each clock cycle of the CPU, using the time stamp counter of the computer’s Pentium processor. The time data is separately collected for each thread and can either be recorded to a file or examined via a terminal connection.

A common technique to decrease the execution time of applications is the translation of the Java byte-code into native code. For this task a translator was developed for the JX operating system. In contrast to conventional just-in-time (JIT) compilers, which aim at fast translation, the JX translator was designed to generate faster native code. Therefore it uses more expansive optimization techniques.

In exhaustive tests with the hardware based profiler it proved capable of depicting the most time consuming parts in an implementation of the Extended Two (ext2) filesystem, which is used by the JX operating system. Although the test results where no sufficient to achieve the same performance as with traditional operating systems, future work in this field will certainly lead to more promising results.

Kurzfassung

In heutigen Betriebssystemen kommen überwiegend hardwarebasierte Speicherschutzkonzepte zum Einsatz, da diese im Vergleich zu Systemen mit einem softwarebasierten Speicherschutz ein vermeintlich schnelleres Laufzeitverhalten haben. Der softwarebasierte Speicherschutz ermöglicht jedoch eine viel flexiblere Vergabe von Systemresourcen und könnte möglicherweise mit einem traditionellen Betriebssystem konkurrieren, wenn nur ein vergleichbarer Aufwand zur Programm- und Systemoptimierung zum Einsatz käme.

Das Java-Betriebssystem JX nutzt nur das Speicherschutzkonzept von Java als softwarebasierten Speicherschutz im System. Alle Anwendungen und große Teile vom System sind in Java geschrieben. Zur Optimierung des Systems werden in dieser Diplomarbeit zwei Hilfsmittel für JX vorgestellt. Das erste Hilfsmittel ist ein Profiler und soll dem Programmierer helfen, schneller Anwendungen zu schreiben. Das zweite Hilfsmittel ist ein Java-Bytecode-zu-Maschinencode-Übersetzer, welcher die Ausführungszeit von Java-Bytecode verbessert.

Um die Laufzeit von Programmen zu verbessern ist es wichtig, daß der Programmierer weiß, wo die Anwendung die meiste Zeit verbringt. Profiler werden benutzt, um diese Stellen eines Programms zu finden. Der Profiler für das JX-System bestimmt die Ausführungszeit für jeden Methodenaufruf. Für die Messung wird der Time Stamp Counter des Pentium-Prozessors verwendet, welcher die einzelnen Taktzyklen der CPU zählt. Die Daten werden für jeden Aktivitätsträger separat gespeichert und können entweder in eine Datei ausgegeben werden oder über eine Terminalverbindung ausgewertet werden.

Es ist eine allgemein gebräuchliche Technik, den Java-Bytecode vor der Ausführung in ein Maschinenprogamm zu übersetzen, um so die Ausführungszeit einer Java-Anwendung zu verbessern. Während das Ziel von JIT-Compilern eine möglichst schnelle Übersetzung ist, wurde bei dem Übersetzer für JX darauf abgezielt, ein möglichst schnelles Maschinenprogramm zu erzeugen.

Mit beiden Hilfsmitteln wurde eine deutliche Verbesserung am Beispiel einer ext2-Dateisystemimplementierung gezeigt. Das gesteckte Ziel mit einem hardwarebasierten System konkurrieren zu können wurde jedoch noch nicht erreicht.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Nachteile hardwarebasierter Speicherschutzkonzepten

In heutigen Betriebssystemen kommen überwiegend hardwarebasierte Speicherschutzkonzepte zum Einsatz. Beim hardwarebasierten Speicherschutz übernimmt ein in die Hardware integrierter Speichermanager, die sogenannte *Memory Management Unit (MMU)*, zusammen mit dem Betriebssystemkern den Speicherschutz. Die MMU stellt den einzelnen auf einem System laufenden Anwendungen getrennte Adressräume zur Verfügung. Der Adressraum einer Anwendung bildet dabei jeweils einen getrennten Schutzraum. Eine fehlerhafte oder böswillige Software kann nicht einfach auf die Daten einer anderen Anwendung zugreifen und diese zerstören.

Die einzige Software, die in diesen Systemen keinem Speicherschutz unterliegt ist in der Regel der Betriebssystemkern. Der Betriebsystemkern muß daher als vertrauenswürdig und fehlerfrei gelten. Um mögliche Fehlerquellen im Kern zu vermeiden, wurde versucht, die Betriebssystemkerne möglichst klein zu halten und möglichst viele Funktionen aus dem Kern heraus in separate Adressräume zu verlagern. Beispiele für Betriebsysteme mit sogenannten *Microkernels* sind MACH, GNU HURT und L4KA.

Diese Betriebsysteme leiden jedoch an einem der größten Nachteile von hardwarebasierten Speicherschutzkonzepten. Die Umschaltung zwischen den einzelnen Adressräumen ist relativ aufwendig und zeitraubend. Bei Betriebssystemen mit möglichst kleinem Systemkern kommt es zu häufigeren Umschaltungen, da typische Betriebssystemaufgaben zusätzlich in separaten Adressräumen liegen. Daher gelten sie im Vergleich zu Systemen mit grossen Systemkernen wie z.B. Unix oder Windows NT als langsam.

Ein weiterer Nachteil des hardwarebasierten Speicherschutzes ist die unflexible und grobe Einteilung der Schutzräume. So ist es zwar möglich, einzelne Speicherseiten in verschiedene Adressräume einzublenden, ohne diese kopieren zu müssen. Es ist jedoch nur unter großem Aufwand möglich, feinere und an den Grenzen von programmiersprachlichen Objekten orientierte Speicherbereiche in einen anderen Adressraum zu bringen. Auch die Schutzräume sind nicht an den Grenzen von Objekten orientiert, sondern gewähren nur Schutz zwischen relativ grossen Anwendungen.

1.2 Das Java-basierte Betriebsystem JX

Die Laufzeitumgebung von Java stellt einen softwarebasierten Speicherschutz zur Verfügung, welcher die einzelnen programmiersprachlichen Objekte schützt und es ermöglicht, die Zugriffe auf die Objektdaten flexibel zu kontrollieren. Das von Michael Golm entworfene Betriebssystem JX nutzt das Speicherschutzkonzept von Java als einzigen Speicherschutzmechanismus innerhalb des Systems.

Durch den rein softwarebasierten Speicherschutz hat JX nicht mit den beschriebenen Problemen von hardwarebasierten Systemen zu kämpfen. Über den Speicherschutz hinaus gewährleistet das System eine faire Verteilung von Speicher und Rechenzeit durch die Aufteilung der Resourcen in Domains. Eine Domain bekommt vom System einen Speicherbereich zugewiesen, in dem sie die Daten ihrer Objekte selber verwaltet. Auch die ihr zugestandene Rechenzeit verteilt jede Domain an ihre eigenen Aktivitätsträger. Einzelne Domains können über Portalaufrufe Dienste anderer Domains nutzen.

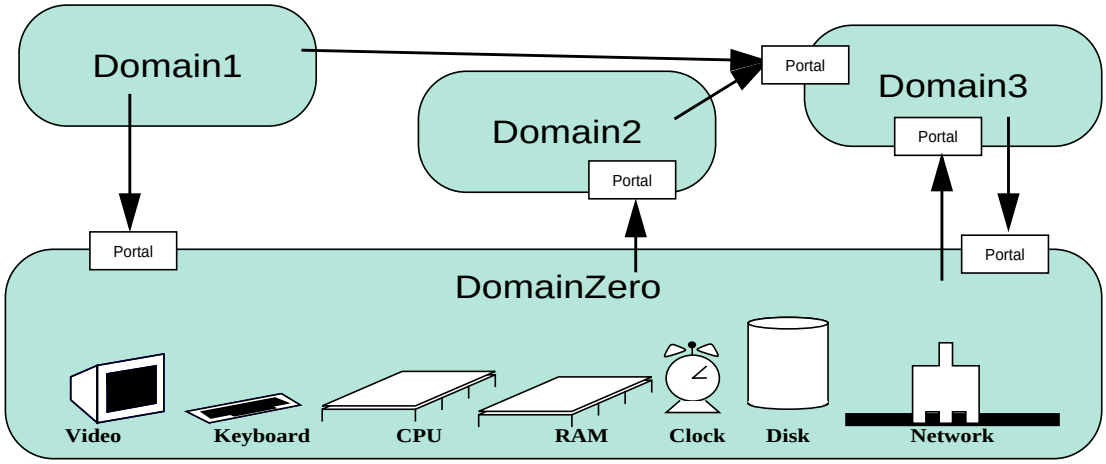


Abb. 1.1 Das Betriebsystem JX

Die in den Domains laufenden Anwendungen sind alle ausschließlich in Java geschrieben. Ausgenommen davon ist die DomainZero (siehe Abbildung 1.1). Die DomainZero stellt hardwareabhängige Dienste bereit und ist für diesen Zweck in C und Assembler geschrieben. Sie verfügt als einzige Domain über keinen Speicherschutz und ist mit einem Betriebsystemkern vergleichbar. Ihre Grösse ist mit der Grösse von Mikrokernen vergleichbar.

Ein softwarebasierter Speicherschutz erfordert einiges an zusätzlichen Aufwand zum Überprüfen von Speichergrenzen und Speicherzugriffen. Daher gelten Systeme mit einem softwarebasierten Speicherschutz gegenüber Systemen mit einem hardwarebasieten Speicherschutz als unterlegen. Anhand von JX wird sich zeigen, ob diese Annahme bei einem konsequent auf einen softwarebasierten Speicherschutz ausgelegeten System auch zutrifft, oder ob ein solches System mit MMU-basierten Systemen konkurrieren kann.

Um mit anderen Systemen konkurrieren zu können, ist es für JX wichtig, auch mit anderen Systemen vergleichbare Techniken für die Laufzeitoptimierung nutzen zu können.

Der erste Teil dieser Arbeit beschäftigt sich mit einem Profiler für das Betriebsystem JX. Als Profiler bezeichnet man ein Programm, welches dem Programmier hilft, das Laufzeitverhalten seiner Programme auszuwerten, um mit diesem Wissen die Programme gezielter verbessern zu können. Profiler stellen eine gängige Technik zur Programmoptimierung auf der Sprachebene dar.

Der zweite Teil der vorliegenden Arbeit beschäftigt sich mit der Funktionsweise eines optimierenden Compilers für JX. Optimierende Compiler sind in Systemen mit hardwarebasiertem Speicherschutz eine gängige Technik, um das Laufzeitverhalten einer Anwendung zu verbessern. Bei einem System mit softwarebasiertem Speicherschutz ergeben sich einige neue Optimierungen.

An einem für JX implementierten ext2-Dateisystem wird gezeigt, ob das Laufzeitverhalten von JX mit diesen Techniken verbessert wird.

2 Ein Profiler für JX

2.1 Einleitung

Um dem Programmierer einen Einblick in das Laufzeitverhalten seiner Anwendung zu ermöglichen sind spezielle Auswerteprogramme, so genannte *Profiler,* auf den verschiedenen Systemplattformen üblich. Ein Profiler mißt die zur Laufzeit verbrachte Zeit in einzelnen Teilbereichen einer Anwendung und macht dem Programmierer anschließend diese Zeiten einsehbar.

Der Programmierer kann an den Ergebnissen erkennen, an welchen Stellen eine Verbesserung seiner Anwendung von Hand sinnvoll ist. Da üblicherweise in 10% einer Anwendung 90% der gesamten Verarbeitungszeit verbracht wird, sollte diesen Programmabschnitten eine besondere Aufmerksamkeit gewidmet werden. Ein Programmierer ist gut beraten, wenn er sich nur auf die 10% des stark benutzten Programmcodes bei der Optimierung konzentriert. Nur in diesem Bereich hat er eine Chance, spürbare Geschwindigkeitsvorteile zu erzielen.

Das Laufzeitverhalten einer Anwendung hängt unter anderem auch von der Laufzeitumgebung ab. Ein für die JVM von Sun existierender Profiler kann damit durchaus zu anderen Ergebnissen kommen als ein Profiler unter JX. Einige Besonderheiten der aktuellen JX-Laufzeitumgebung möchte ich in Kapitel 2.4 erwähnen, um damit Programmierern für das JX-System spezifische Optimierungen aufzuzeigen.

Es ist jedoch meistens sinnvoller und auch erfolgsversprechender, langsame Algorithmen durch gleichwertige, jedoch schnellere zu ersetzen: Zum Beispiel kann eine lineare Suche über eine große Liste von Objekten in den meisten Fällen durch eine wesentlich schnellere Hashtabelle ersetzt werden. Derartige Optimierungen verbessern das Laufzeitverhalten von Anwendungen in der Regel wesentlich und sind auch in Zukunft gar nicht oder nur sehr eingeschränkt von optimierenden Compilern zu erbringen.

Um Programmierern die Möglichkeit zu eröffnen, für das JX-System erfolgreicher schnelle Anwendungen zu schreiben, ist auch ein Profiler für JX nötig. In den folgenden Kapiteln werden die Konzepte und die Details des von mir im Rahmen meiner Diplomarbeit implementierten Profilers beschrieben.

2.2 Zeitmessung

2.2.1 Zeitmessung pro Methode

Je feiner die vom Profiler gemessenen Teilbereiche einer Anwendung sind, desto genauer kann der Programmierer die zu optimierende Stelle eingrenzen. Oft sind es einzelne Programmschleifen, in denen eine Anwendung viel Zeit verbringt. Andererseits beeinflußt die Messung selber in der Regel das Laufzeitverhalten der zu messenden Anwendung, da die Messroutinen selbst wieder Rechenzeit und Speicherplatz beanspruchen. Es besteht daher ein Konflikt zwi-

schen einer möglichst feinen Messung und einem möglichst manipulationsfreien Eingriff in das zu messende System. Ein sinnvoller Kompromiß ist es, die Dauer einzelner Methodenaufrufe zu messen. Dieses hat auch den Vorteil, daß die gemessenen Bereiche durch die Methodennamen leicht vom Programmierer wieder gefunden werden können.

Der von mir implementierte Profiler für das Betriebsystem JX erfaßt darum die Laufzeit für einzelne Methodenaufrufe. Dazu nutzt er den *Time Stamp Counter* des Pentium Prozessors[Intel95]. Der *Time Stamp Counter* zählt die Prozessortakte seit dem Einschalten in einem 64 Bit breiten modellspezifischen Register. Dieses Register kann über einen speziellen Maschinenbefehl ausgelesen werden. Der im zweiten Teil meiner Diplomarbeit beschriebene JX-Translator fügt in den jeweiligen Methodenvorspann — den Prolog — und in den Methodenabspann — den Epilog — ein spezielles Maschinenprogramm ein, um diesen Taktzähler auszulesen. Im Epilog wird anschließend anhand der zwei unterschiedlichen Zählerstände die in der Methode verbrachte Zeit bestimmt und das Ergebnis gespeichert.

Das Einfügen der Messroutinen in Prolog und Epilog kann auch nur für ausgewählte Methoden erfolgen. Dies ermöglicht das Einschränken der Messung auf gezielte Bereiche im Programm und verringert dabei unvermeidbare Meßfehler.

2.2.2 Speichern der Meßdaten

Damit der für die Ergebnisse benötigte Speicherbereich nicht zu groß wird, werden die Zeiten für jedes Paar von aufrufender und aufgerufener Methode summiert und mit der Anzahl der Aufrufe gespeichert. Dies geschieht während der Messung und verlangt deshalb einen möglichst schnellen Zugriff auf die gespeicherten Datensätze.

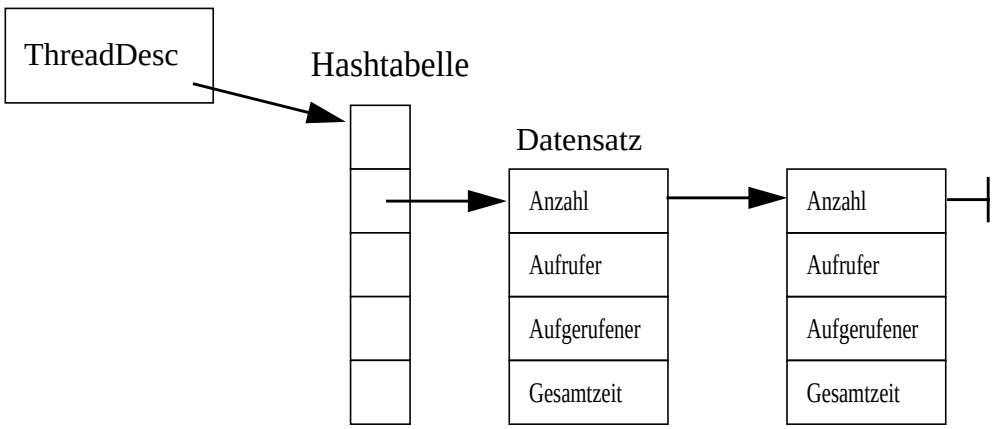


Abb. 2.1 Speicherung der Meßdaten

Die Datenstruktur zum Speichern der Meßdaten ist, wie in der Abbildung 2.1 gezeigt, an die im JX-Kern befindliche Struktur *ThreadDesc* gehängt. Diese Struktur speichert die Daten für jeweils einen Aktivitätsträger separat. Es ist daher nicht nötig, bei Datenzugriffen eine Synchronisation vorzunehmen. Das Speichern der Meßdaten erfolgt jeweils beim Verlassen einer Methode durch das eingefügte Maschinenprogramm im Epliog.

Da bei jedem Verlassen einer Methode der entsprechende Datensatz wieder gefunden werden muß, sind die Datensätze über eine Hashtabelle auffindbar. Die Hashtabelle besteht dabei aus einer Tabelle von Zeigern auf eine verkettete Liste von Datensätzen. Der zum jeweiligen Datensatz passende Tabellenindex wird aus den Adressen der aufrufenden und der aufgerufenen Methode berechnet. Das Einbeziehen beider Adressen ist notwendig, um trotz Sonderfällen, wie zum Beispiel Methoden die sehr häufig von unterschiedlichen Positionen aus aufgerufen werden oder Methoden die sehr viele andere Methoden aufrufen, eine möglichst gleichmäßige und breite Verteilung von Indizes zu bekommen. Abbildung 2.1 veranschaulicht die gespeicherte Datenstruktur für zwei Datensätze mit gleichem Index.

2.2.3 Kompensation von Meßfehlern

Obwohl beim Speichern der Meßdaten ein möglichst schnelles Verfahren gewählt wurde, dauert das Ausrechnen und Speichern der Ergebnisse im Vergleich zu den eigentlich gemessenen Methoden zu lange. Die in der Meßroutine verbrachte Zeit muß daher von der gemessenen Gesamtzeit wieder abgezogen werden. Zu diesem Zweck wird der Taktzähler am Ende der Meßroutine ein drittes Mal ausgelesen und die in der Meßroutine verbrachte Zeit bestimmt.

Die Abbildung 2.2 zeigt den zeitlichen Verlauf einer Messung. Die Zeiten in der Methode A und der Methode B sollen in diesem Beispiel gemessen werden. Betrachten wir zunächst die Methode A. Im Prolog der Methode wird der Taktzähler ausgelesen und auf dem Methodenstack gespeichert, anschließend wird der zu messende Methodenrumpf durchlaufen und am Ende im Epilog der Taktzähler erneut ausgelesen.

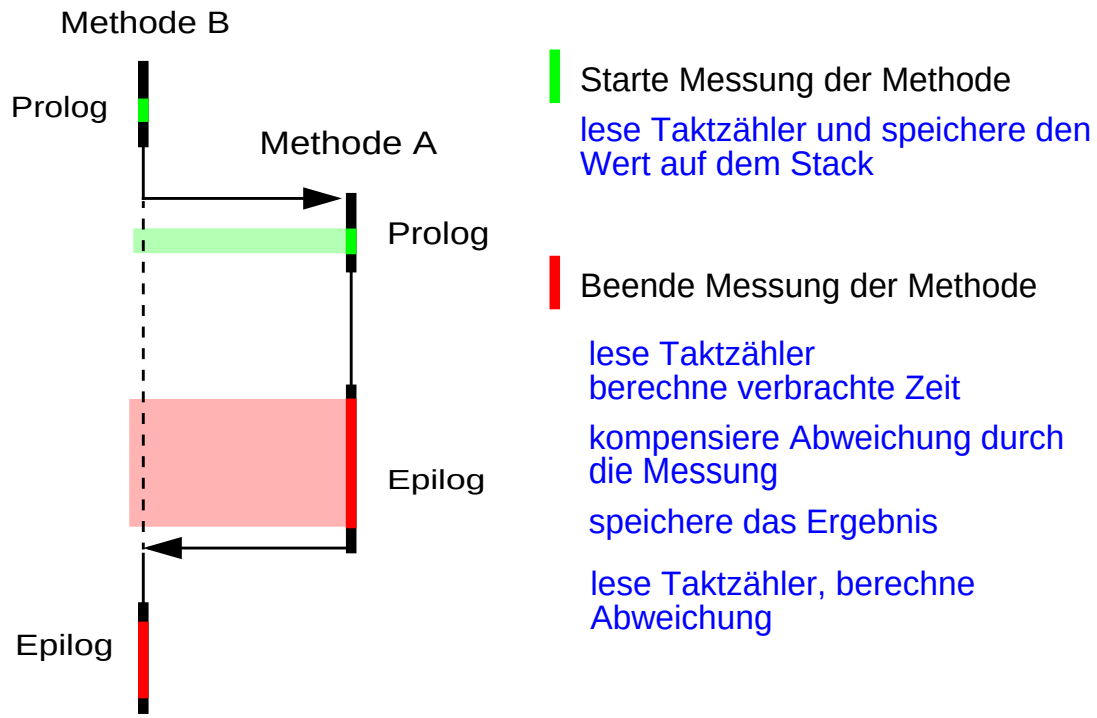


Abb. 2.2 Zeitlicher Verlauf der Messung

Da zwischen den zwei erfaßten Zeitpunkten, abgesehen vom Zwischenspeichern des ersten Zählerstandes auf dem Methodenstack, nur der Methodenrumpf liegt, ist in diesem Fall der Meßfehler sehr gering und man erhählt eine sehr genaue Aussage über die in der Methode verbrachte Zeit. Die verbleibende geringe Abweichung ist zudem konstant und kann, wenn sie einmal ermittelt wurde, von der gemessenen Zeit abgezogen werden.

Ein viel grösseres Problem ergibt sich bei Methoden, welche andere Methoden aufrufen. Dies ist bei der Methode B der Fall. Die Methode B ruft in unserem Beispiel die Methode A auf. Dadurch messen wir die verbrachte Zeit in Methode A und B, sowie die Zeit, die in der Meßroutine zum Berechnen und Speichern der Meßdaten der Methode A verbracht wird.

Die verbrachte Zeit in der Meßroutine wird durch ein weiteres Auslesen des Taktzählers am Ende des Epilogs ermittelt. Diese zweite Zeitspanne betrachten wir als Abweichung von einem Programmlauf ohne Messung und propagieren diese Abweichung weiter zur jeweils aufrufenden Methode. Der genaue Vorgang wird in Abbildung 2.3 an einem komplexeren Beispiel veranschaulicht.

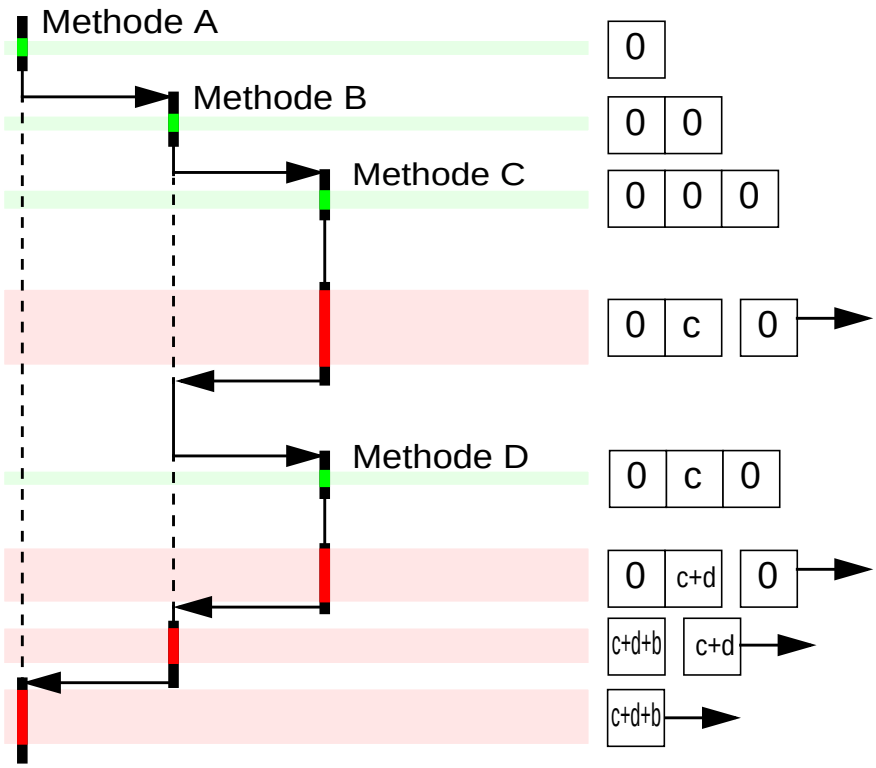


Abb. 2.3 Propagieren der zeitlichen Abweichung

Die Abbildung zeigt den Zeitverlauf von vier ineinander verschachtelten Methoden. Das Propagieren der Abweichung geschieht mit Hilfe eines separaten Stacks. Bei jedem Methodenaufruf legen wir einen neuen Zähler mit einem Initialwert von Null auf diesen Stack. Beim Verlassen einer Methode nehmen wir den obersten Zähler vom Stack und ziehen seinen Wert als Abweichung von der berechneten Zeit für den Methodendurchlauf ab. In den Methoden C und D ist dieser Wert jeweils null, da diese keine weiteren Methoden aufgerufen haben und demzufolge auch keine Abweichung innerhalb der Methoden angefallen ist.

Anschließend addieren wir die Abweichung und die in der Meßroutine benötigte Zeit auf den obersten Zähler. In der Methode C ist dies nur die verbrachte Zeit c in der Meßroutine von C, in der Methode D sind es schon der Zählerstand c und die in der Meßroutine von Methode D verbrachte Zeit d, in der Methode B sind es die Abweichung c+d und die in der Meßroutine von Methode B verbrachte Zeit b.

Mit dem beschriebenen Verfahren ist es möglich, den durch die Messung erzeugten Meßfehler zum Großteil wieder auszugleichen. Wir ergänzen das Verfahren noch um eine zusätzliche Erhöhung der Zähler um einen konstanten Wert. Diese Erhöhung ist nötig für einige nicht innerhalb der Meßroutine liegende zusätzliche Maschinenbefehle, welche zu einer zusätzlichen konstanten Abweichung führen. Der Wert für die konstante Abweichung kann durch eine gezielte Messung ermittelt werden. Zu diesem Zweck vergleicht man die Laufzeit einer Methode, welche eine Methode mit Meßroutine aufruft, mit der Laufzeit, die von der gleichen Methode benötigt wird, wenn sie diese Methode ohne Meßroutine aufruft.

Allem Aufwand zum Trotz bleibt jedoch eine Ungenauigkeit. Durch das Einfügen von zusätzlichen Maschinenbefehlen wird das Laufzeitverhalten der Anwendung nachhaltig verändert. Als Beispiel sei der L1-Befehlscache genannt. Durch eine Vergrößerung des Maschinenprogramms kommt es hier zu einem anderen Verhalten als bei einem Programmlauf ohne Messung. Das Gleiche gilt natürlich auch für den L2-Cache und die Tabellen für die Sprungvorhersage.

Das Ziel eines Profilers ist es jedoch nicht, eine auf den Takt genaue Aussage über die in den einzelnen Methoden verbrachte Zeit zu liefern. Wichtig ist, daß der Programmierer einen Einblick erhält, in welchen Methoden die meiste Zeit verbracht wird. Dieser Einblick kann aus den gesammelten Daten gewonnen werden.

Während der Arbeit am Profiler hatte Michael Golm parallel zu meiner Arbeit ein anderes Verfahren zum Profilen des Systems implementiert. Bei seinem Verfahren wurde einfach die Position des Befehlszeigers während einer gleichmäßig immer wieder auftretenden Unterbrechung des Programmlaufes gespeichert. Nach dem Programmlauf wurde bestimmt, an welchen Positionen diese Unterbrechungen stattgefunden hatten und wie häufig sie an diesen Stellen auftraten.

Dieses Verfahren verändert das Laufzeitverhalten einer Anwendung nicht, da keine zusätzlichen Maschinenbefehle eingefügt werden und die wiederkehrende Unterbrechung im JX-System schon aus einem anderen Grund nötig ist. Wir stellten fest, daß beide Verfahren zu vergleichbaren Aussagen führen.

2.2.4 Behandlung von mehreren Aktivitätsträgern

Wir sind bei der Zeitmessung bis jetzt stillschweigend davon ausgegangen, daß der Taktzähler nur für den aktuellen Aktivitätsträger exklusive die Takte zählt. Dies ist jedoch nicht der Fall. Findet im System eine Umschaltung des Aktivitätsträgers statt, läuft der Taktzähler weiter, ohne daß wir einen Fortschritt in dem von uns gemessenen Programm erzielen.

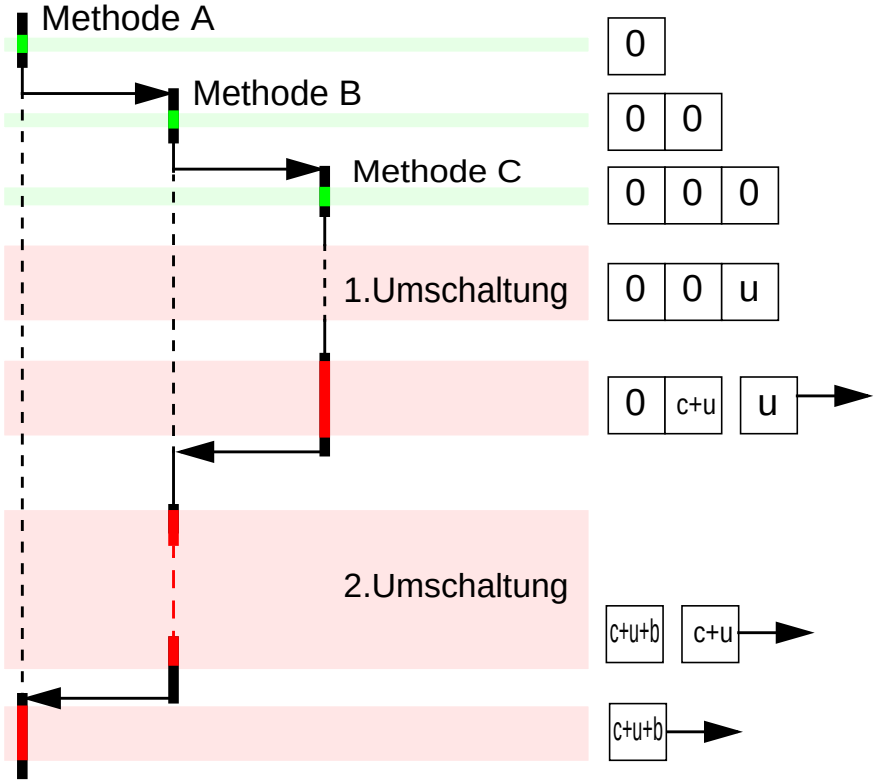


Abb. 2.4 Berücksichtigen von Umschaltungen des Aktivitätsträgers

Wir müssen also bei allen möglichen Umschaltungen die Anzahl der Takte bestimmen, welche nicht in unsererem Programmlauf verbracht wurden. Diese Zeitspanne betrachten wir wieder als Abweichung vom Programmlauf ohne Umschaltung und addieren den Wert auf den aktuellen Abweichungszähler. Abbildung 2.4 verdeutlicht am Beispiel der 1. Umschaltung dieses Vorgehen, wobei u für die Anzahl der Prozessortakte während der Unterbrechung steht. Die Abweichung wird wie in Kapitel 2.2.3 beschrieben mit der durch die Meßroutine erzeugten Abweichung weiter propagiert.

Zum Ermitteln der Abweichung ist der JX-Profiler auf die Mithilfe des JX-Kerns angewiesen. Bei der Umschaltung muß der JX-Kern für den angehaltenen Aktivitätsträger den Taktzähler sichern und für den fortsetzenden Aktivitätsträger die Abweichung bestimmen und dem JX-Profiler mitteilen. Der Profiler stellt für diese Aufgabe zwei Routinen zur Verfügung, welche während einer Messung aktiviert werden.

Ein weiteres Problem wird in der Abbildung 2.4 bei der 2. Umschaltung verdeutlicht. Eine Umschaltung des Aktivitätsträgers kann auch auftreten, während wir uns in einer Meßroutine befinden. In diesem Fall kompensieren wir die Abweichung bereits und ein erneutes Aufaddieren der durch die Umschaltung entstandenen Abweichung würde diese Zeit doppelt zählen. Die vom Profiler vorgesehenen Routinen berücksichtigen diesen Fall und ermitteln keine Abweichung, falls die Umschaltung innerhalb einer Meßroutine stattfindet.

2.2.5 Unterstützung von Multiprozessorsystemen

Während meiner Diplomarbeit begann Meik Felser, das JX-Betiebssystem mit einer Unterstützung für Multiprozessorsysteme auszustatten. Da die für das Profilen verwendeten Taktzähler für jeden Prozessor separat zählen und es nicht ausgeschlossen ist, daß die Aktivitätsträger während einer Umschaltung nicht den Prozessor wechseln, führt der Einsatz des Profilers auf Multiprozessormaschinen vorraussichtlich zu unbrauchbaren Ergebnissen. Es ist in Zukunft auch nicht geplant, den Profiler für ein Multiprozessorsystem anzupassen.

2.3 Auswertung

2.3.1 Einleitung

Damit der Programmierer aus den in Kapitel 2.2 gesammelten Daten einen Nutzen ziehen kann, müssen diese übersichtlich dargestellt werden. Der JX-Profiler bietet mehrere Zugriffsmöglichkeiten auf die Daten. Der schnellste Zugriff ist über den Monitor des JX-Kernes. In Kapitel 2.3.2 werden die unterschiedlichen Fähigkeiten dieser Schnittstelle beleuchtet. Es ist auch möglich, den Profiler über Portalaufrufe von einer anderen JX-Domain aus zu steuern. Damit beschäftigt sich das Kapitel 2.3.3. Im letzten Kapitel wird besprochen, wie die Meßdaten auch außerhalb von JX ausgewertet werden können.

2.3.2 Auswerten im Monitor

Der JX-Kern besitzt einen Monitor, der die Untersuchung wichtiger Systemzustände über die serielle Schnittstelle ermöglicht. Dieser Monitor dient während der Entwicklung von JX zur Fehlersuche und für Testzwecke. Der Monitor kann sowohl in einem Fehlerfall als auch über einen Portalaufruf an der DomainZero aufgerufen werden.

Da das Herrunterladen der Meßdaten und das Auswerten auf einem anderen Rechner zeitraubend ist, wurde in diesen Monitor ein kleine Benutzerschnittstelle für den Profiler integriert. Aus dem Monitor kann durch das Kommando **profile** zu dieser Benutzerschnittstelle gewechselt werden.

Aus den gespeicherten Datensätzen berechnet der Profiler drei Zeitwerte pro Methode. Die erste Zeit ist die Gesamtzeit aller Methodendurchläufe. Dieser Wert wird als **total** bezeichnet. Der zweite berechnete Wert ist die durchschnittlich verbrachte Zeit in einer Methode und wird als **average** bezeichnet. Der dritte Wert wird als **method** bezeichnet und ist die Gesamtzeit abzüglich der in von der Methode aufgerufenen Unterroutinen verbrachten Zeit.

Die Benutzerschnittstelle besitzt eine Reihe von Kommandos, über die der Benutzer diese Werte einsehen kann. Die Einabe des Befehls **help** listet alle anwendbaren Kommandos auf. Die zwei wichtigsten Kommandos möchte ich an einem Beispiel erläutern und damit die Funktionalität des Profilers demonstrieren.

Das erste Kommando ist **sort** und bekommt als Argument entweder **total**, **average** oder **method** übergeben. Das Kommando sortiert die Methoden nach dem durch das Argument ausgewählten Zeitwert und gibt anschließend die ersten 10 Einträge aus. Abbildung 2.5 zeigt dies an einem realen Profilerlauf für das ext2-Dateisystem von JX.

```
Monitor: profile
scan profile data of domain Test
make method list

Profiler (Test): sort method
calls        total      average    in method method
213977    6367886us          29us    1189193us javafs.FileInode.read(Ljx/zero/Memory;II)I
909513    1585481us           1us     747154us buffercache.BufferCache.brelse(L...;)V
909551    1348087us           1us     488394us buffercache.BufferCache.findBuffer(II)L...;
374485    2095229us           5us     454593us javafs.InodeImpl.blockGetBlk(L...;IZII)L...;
    24     405041us       16876us     405041us buffercache.BufferHashtable$1.advance()V
535012     342478us           0us     342478us jx.buffer.BufferList.removeElement(L...;)V
535051     447076us           0us     282526us buffercache.BufferCache.putLastFree(L...;)V
535002    3315531us           6us     238518us javafs.InodeImpl.getBlk(IZ)L...;
909552     238070us           0us     238070us buffercache.BufferHashtable.get(I)L...;
535003     981782us           1us     233323us javafs.InodeImpl.inodeGetBlk(IZI)L...;
Profiler (Test): _
```

Abb. 2.5 Die Profiler Benutzerschnittstelle im Monitor (sort)

Die erste Spalte der Ausgabe enthält die Anzahl der Aufrufe, die zweite die Gesamtzeit in Mikrosekunden, die dritte die durchschnittlich benötigte Zeit in Mikrosekunden, die vierte die in der Methode verbrachte Gesamtzeit in Mikrosekunden und in der letzten Spalte befindet sich der Methodenname. Die Liste ist nach dem Argument von **sort** sortiert. Das Argument **method** wählt dabei die Zeiten aus, die in der vierten Spalte stehen.

Wir erkennen nun, daß in der Methode **javafs.FileInode.read** relativ viel Zeit verbracht wird. Um mehr Informationen über diese Methode zu erhalten, benutzen wird das Kommando **show**. Das **show**-Kommando bekommt als Argument den Methodennamen übergeben.

```
Profiler (Test): show javafs.FileInode.read(Ljx/zero/Memory;II)I
Method    : javafs.FileInode.read(Ljx/zero/Memory;II)I
calls 213977 total time  6367886 us, average time       29 us, time in method  1189193 us = 21%
callers:
213977  100% 4292745248 us test.fs.ReRead.<init>(Ljx/fs/FS;)V
callees:
213977    0%      17253 us test.fs.DummyClock.getTimeInMillis()I
534957   52%    3313426 us javafs.InodeImpl.getBlk(IZ)Ljx/fs/buffercache/BufferHead;
534957    1%     104589 us buffercache.BufferCache.updateBuffer(L...;)V
534957   17%    1137509 us buffercache.BufferCache.brelse(Ljx/fs/buffercache/BufferHead;)V
213977    6%     414952 us javafs.InodeImpl.setDirty(Z)V
213978    1%      88380 us javafs.InodeImpl.permission(I)Z
615196    1%      80652 us javafs.BufferHeadAccess.readInt(I)I
534957    0%      21928 us jx.buffer.BufferHead.getData()Ljx/zero/Memory;
Profiler (Test): _
```

Abb. 2.6 Die Profiler Benutzerschnittstelle im Monitor (show)

Die Abbildung 2.6 zeigt die Ausgabe des Kommandos **show** für die Methode **javafs.FileInode.read**. In den ersten Zeilen werden die schon bekannten Informationen angezeigt, der Methodenname, die Zahl der Aufrufe und die Zeiten. Anschließend folgt eine Liste aller Aufrufer, in unserem Beispiel wird die Methode nur von der Methode **test.fs.ReRead.<init>(Ljx/fs/FS;)V** aufgerufen. Danach folgt eine Liste der von der Methode aufgerufenen Methoden. Dieser Liste kann die Anzahl der Aufrufe, der prozentuale Anteil an der Methodenlaufzeit und die Gesamtzeit der aufgerufenen Methode entnommen werden.

Das Ergebnis zeigt, daß die Methode selber mit 21%, die Methoden **javafs.InodeImpl.getBlk(IZ)Ljx/fs/buffercache/BufferHead;** mit 52% und vielleicht noch die Methode **buffercache.BufferCache.brelse(Ljx/fs/buffercache/BufferHead;)V** mit 17%, genauer untersucht werden sollten. Alle anderen Methoden spielen für die Gesamtlaufzeit in diesem Beispiel eine untergeordnete Rolle.

2.3.3 Auswerten über Protalaufrufe

Das Auswerten der Daten über den Systemmonitor ist in der frühen Phase des JX-Projekts und beim Profilen von Systemkomponenten sicher eine schnelle und bequeme Vorgehensweise. Für das Profilen von Anwendungen in einem laufenden JX-System ist es jedoch sinnvoll, die Daten des Profilers direkt auslesen zu können.

```
Profiler p = (Profiler) domainZero.lookup("Profiler");
p.restartMeasurement();
... der zu messende Programmabschnitt ...
p.stopMeasurement();
p.dumpResults();
```

Abb. 2.7 Steuern des Profilers über Portalaufrufe

Eine elegante Lösung zum Steuern und Abfragen des Profilers stellt ein Profiler-Portal an der DomainZero dar. Durch die Anbindung des Profilers über ein Portal ist es auch möglich, die Messung nur für bestimmte Abschnitte zu starten. Abbildung 2.7 veranschaulicht die Benutzung des von DomainZero implementierten Portals.

2.3.4 Auswerten mit fremden Anwendungen

Die vom Profiler gesammelten Daten können in einer Datei gespeichert werden, um sie anschließend mit anderen Anwendungen auszuwerten. Die JVM von Sun besitzt eine vergleichbare Funktionalität und erzeugt, wenn sie mit der Option -d aufgerufen wird, ebenfalls Daten für das Profilen von Anwendungen. Die vom JX erzeugten Profiledaten werden dabei auf eine vergleichbare Weise gespeichert wie die Daten der JVM von Sun.

So können auch Anwendungen, welche für das Profilen mit der Sun JVM gedacht waren, im Zusammenhang mit dem JX-Profiler genutzt werden und umgekehrt.

Abb. 2.8 Externer Profiler für JX

Die Abbildung 2.8 zeigt ein einfaches Auswerteprogramm für die Profiledaten. Es entstand im Rahmen meiner Diplomarbeit für einige kleine Testzwecke.

2.4 JX-spezifische Besonderheiten

2.4.1 Allgemeines zur Programmoptimierung

Hat man sich mit Hilfe eines Profilers ein Bild über das Laufzeitverhalten seiner Anwendung gemacht, besteht der nächste Schritt darin sich Gedanken über mögliche Verbesserungen zu machen. Die hier genannten Besonderheiten von JX sollten dabei nicht die erste Wahl für mögliche Lösungsansätze darstellen. Es ist deutlich effektiver, sich über das Design der Anwendungen und über die verwendeten Algorithmen Gedanken zu machen.

2.4.2 Optimierungen des Bytecode-Übersetzers

Der im zweiten Teil dieser Arbeit beschriebene Bytecode-Übersetzer implementiert einige Optimierungtechniken. Diese Techniken müssen zum Teil noch explizit vom Programmierer aktiviert bzw. eingesetzt werden. Techniken wie zum Beispiel die In-Line Expansion oder der Einsatz von Compiler-Plugins erscheinen vielversprechend. Auch der Blick auf den aus der Zwischenrepräsentation erzeugten pseudo Programmcode bringt einen Aufschluß darüber, was der Übersetzer aus der Anwendung macht und wo Optimierungen vielleicht noch nicht zum Einsatz kommen.

2.4.3 Zugriffzeiten auf die Variablen

Die von der Java-Anwendung benutzten Variablen werden im Speicher sehr unterschiedlich adressiert.

Variablen, die nur innerhalb einer Methode sichtbar sind, werden auf dem Stack der Methode oder auf ein Register abgebildet. Der Zugriff auf diese Variablen ist sehr schnell, darum sind diese wenn möglich zu bevorzugen, besonders im Zusammenhang mit Programmschleifen.

Variablen, die innerhalb eines Objekts sichtbar sind, liegen frei im Speicher und werden immer über den Referenzzeiger des Objekts adressiert. Unter Umständen wird dabei eine Referenzüberprüfung durchgeführt. Der Zugriff auf diese Varibalen ist also bereits um einiges aufwendiger.

Die Klassenvariablen sind für alle Objektinstanzen innerhalb einer Domain gleich. Um die Speicheradresse dieser Variablen zu bestimmen, wird zur Zeit die DomainZero befragt. Der Zugriff auf diese Variablen ist daher teuer und sollte wenn möglich vermieden werden.

3 Ein Java-Bytecode-nach-Maschinencode Übersetzer für JX

3.1 Einleitung

3.1.1 Ziele der Bytecode-Übersetzung

Ein maschinenunabhängiger Bytecode wie der Java-Bytecode hat viele Vorteile. Eine Anwendung ist dadurch zum Beispiel auf unterschiedlicher Hardware lauffähig, solange für diese eine Java-Laufzeitumgebung (JVM) existiert. Frühe JVM waren jedoch um ein Vielfaches langsamer als speziell für diese Hardware übersetzter Maschinencode. Ein Hauptgrund dafür war, daß der Bytecode auf der Zielplattform interpretiert wurde und nicht direkt von der jeweiligen Hardware ausgeführt wird. Aktuelle Laufzeitumgebungen übersetzen daher den Java-Bytecode vor der Ausführung in Maschinencode. Diese Technik wird als Just-In-Time-Übersetzung bezeichnet.

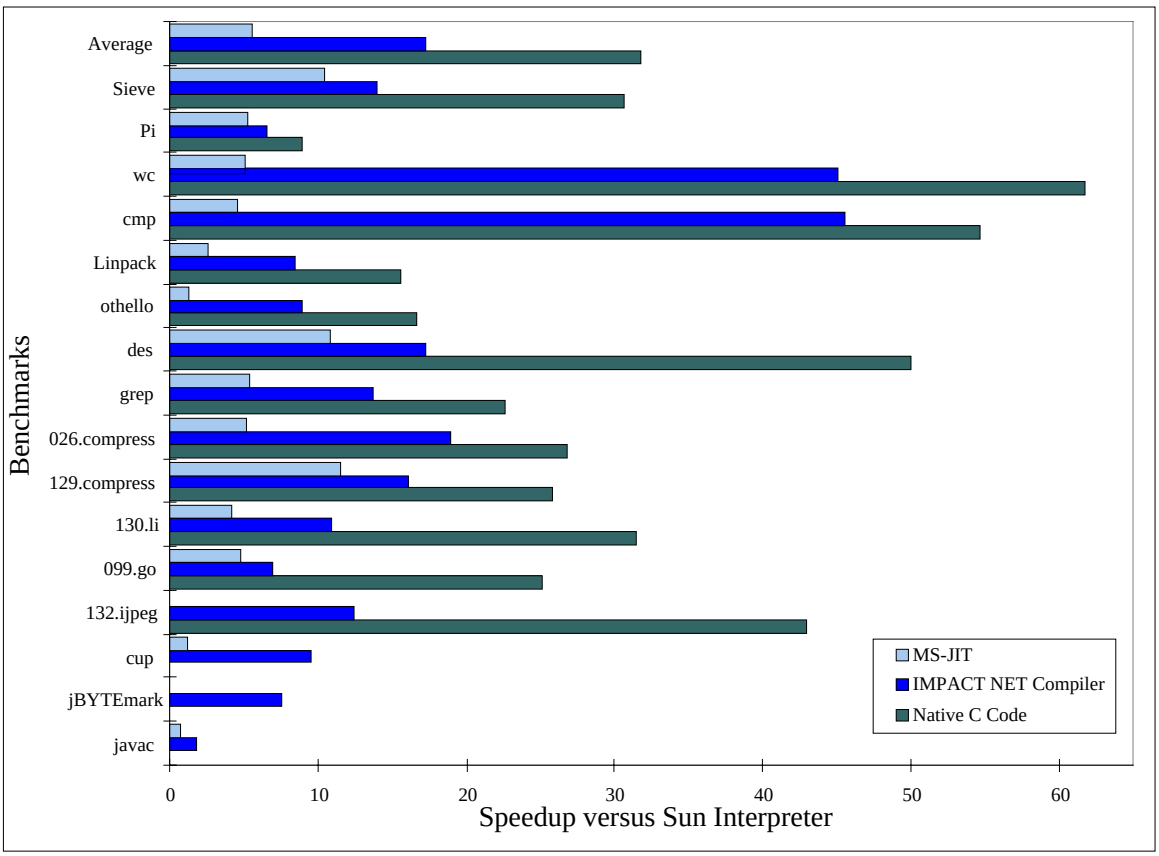


Abb. 3.1 Laufzeitverhalten von Java

Laufzeitumgebungen mit JIT-Compilern sind zwar schon deutlich schneller als solche, die Bytecode nur interpretieren, sie sind jedoch immer noch um Faktoren langsamer als für die Maschine direkt übersetzte Programme. Dieses hat verschiedene Gründe, einerseits kommt zur

Ausführungszeit der reinen Anwendung auch noch der Zeitbedarf zur Übersetzung des Bytecodes hinzu, zum anderen optimieren JIT-Compiler im Vergleich zu statischen Compilern den Programmcode nicht oder nur geringfügig. Eine zeitaufwendige Optimierung zur Laufzeit könnte die Ausführungszeit einer Anwendung möglicherweise auch verlängern und kommt für einen JIT-Compiler somit nicht in Frage.

Übersetzt man den Java-Bytecode vor der Ausführung und hält den übersetzten Maschinencode für die anschließende Ausführung bereit, können auch aufwendigere Optimierungsschritte eingesetzt werden. In der Arbeit "Using NET to Capture Performance in Java-Based Software"[HCJ+97], wurde gezeigt, daß durch den Einsatz eines als *Native Executable Translator* (NET) bezeichneten optimierenden Übersetzers nochmals eine deutliche Verbesserung der Laufzeit von Java-Anwendungen erzielt werden kann. Die Abbildung 3.1 stammt aus dieser Arbeit. Die Grafik zeigt den Geschwindigkeitszuwachs bei Anwendungen, welche von drei verschiedenen Verfahren zur Maschinenprogrammerzeugung erstellt wurden, im Vergleich zum Java-Interpreter von Sun. Zur Maschinenprogrammerzeugung kamen der Microsoft JIT-Compiler, der besagte NET und ein optimierender C-Compiler zum Einsatz.

3.1.2 Ziele des JX Bytecode Translators

Um das Laufzeitverhalten eines Betriebssystems mit einem auf softwarebasierenden Speicherschutzkonzept wie JX und einem Betriebssystem mit hardwarebasiertem Speicherschutz vergleichen zu können, müssen vergleichbare Compiler-Techniken verwendet werden.

Der von mir implementierte Bytecode-Übersetzer für JX implementiert einige bekannte Optimierungen von modernen Compilern. Der erzeugte Maschinencode ist für PCs mit Pentium Prozessor optimiert. Die verwendeten Optimierungen stellen nur einen kleinen Teil der bekannten Möglichkeiten dar. Ich habe mich bei der Auswahl auf die erfolgversprechendsten Optimierungstechniken beschränkt.

Zu den bekannten Verfahren kommen noch neue Techniken hinzu, welche nur bei einem flexiblen softwarebasierten Speicherschutzkonzept wie dem von JX möglich sind. Der softwarebasierte Speicherschutz ist zwar flexibler, erfordert jedoch einen nicht unerheblichen Teil an zusätzlicher Rechenzeit. Es wird sich zeigen, ob die neu gewonnene Flexibilität diesen Nachteil ausgleichen kann.

Der Translator ist selbst in Java geschrieben und kann auch auf dem JX-Betriebssystem selber laufen. Es ist angedacht, daß Java-Klassen in Zukunft beim Laden ins JX-Betriebssystem übersetzt werden und daß Klassen bei Bedarf vom System auch neu übersetzt werden können. Der Translator kann für den Zweck einer möglichst schnellen Übersetzung auch ohne zeitaufwendige Optimierung benutzt werden.

3.2 Der JX-Translator als Teil des JX-Betriebssystems

3.2.1 Allgemeines

Der JX-Translator ist ein wichtiger Teil des JX-Betriebssystems, dies wird schon dadurch begründet, daß der Speicherschutz des Systems zum grossen Teil durch den Translator gewährleistet wird. Der Translator muß den für eine JVM spezifizierten Speicherschutz durch das Einfügen entsprechender Überprüfungen zur Laufzeit gewährleisten, geschieht dies nicht, ist im JX-System auch kein Speicherschutz mehr vorhanden.

Weitere Aspekte, bei denen der Bytecode-Translator eng mit dem System zusammenarbeiten muß, sind das automatische Freigeben von unbenutztem Speicher (Garbage Collection), beim Verwalten von Metainformation über die geladenen Klassen, beim Referenzieren von Objekten und Methoden, sowie bei direkten Speicherzugriffen. Auf einige dieser Aspekte möchte ich nachfolgend noch genauer eingehen.

3.2.2 Der softwarebasierte Speicherschutz

Das Speicherschutzkonzept von Java verbietet eine freie Zeigerarithmetik, wie sie in Programmiersprachen wie C oder C++ möglich ist. Dies wird bereits vom Java-Compiler beim Erzeugen des Java-Bytecodes berücksichtigt und wird von einem Bytecode-Verifier beim Laden des Bytecodes überprüft. Ein fehlerhafter oder böswillig erzeugter Bytecode kann so schon vor der Laufzeit erkannt und abgelehnt werden.

Der Bytecode-Verifier kann jedoch nicht feststellen, ob zur Laufzeit ungültige Referenzen verwendet werden oder Speicher- bzw. Stackgrenzen überschritten werden. Für diese Fälle muß die Java-Laufzeitumgebung, oder in unserem Fall das Java-Betriebssystem JX, zur Laufzeit entsprechende Überprüfungen vornehmen.

Der JX-Translator fügt für diese Überprüfungen spezielle Maschinenbefehle in die erzeugten Maschinenprogramme ein. Einige der vom JX-Translator vorgenommenen Optimierungen beschäftigen sich ausschließlich mit diesen Tests, da besonders die Referenzüberprüfungen sehr häufig vorkommen.

Die Referenzüberprüfung besteht aus einem einfachen Testen der Referenz auf Null. In diesem Fall wird in den JX-Kern gesprungen und eine **NullPointerException** ausgelöst. Bei den Arraygrenzen ist die Überprüfung ähnlich einfach. Die Größe eines Arrays wird in der Objektstruktur des Arrays gespeichert. Vor dem Zugriff wird überprüft, ob der Index in den Grenzen des Arrays liegt, ansonsten wird eine **ArrayIndexOutOfBoundException** ausgelöst.

Um bei der Stackgrenzenüberprüfung nicht bei jedem Methodenaufruf die Obergrenze eines Stacks ermitteln zu müssen, bedient sich JX eines Tricks. Alle Methodenstacks bestehen aus Speicherblöcken fester Größe, in der Regel 4096 bytes. Diese Speicherblöcke sind auf Speichergrenzen, die einem Vielfachen der Blockgrösse entsprechen, ausgerichtet. Dadurch ist es möglich durch einen Vergleich der führenden Bits des aktuellen Zeigers auf den Anfang des Stack-

rahmens (Framepointer) und einem Zeiger auf das Ende des benötigten Stackrahmens zu ermitteln, ob dieser noch in den aktuellen Stackblock paßt. Sind diese Bedingungen nicht erfüllt wird eine Ausnahme erzeugt, und das System kann mit dem Anhängen eines weiteren Blocks darauf reagieren.

Die Abbildung 3.2 veranschaulicht den Vergleich der beiden Zeiger. Da der Stack auf Speichergrenzen von, in unserem Beispiel 4096 Bytes, ausgerichtet ist, besitzen alle Adressen, die im Speicherbereich des Stacks liegen, die gleichen führenden 20 Bits. Die führenden Bits können als Identifier für den Stack aufgefaßt werden, während die letzten 12 Bits als ein Index betrachtet werden können. Bei einem Methodenaufruf wird nun das Ende des neuen Stackrahmens berechnet. Liegt der Zeiger auf dieses Ende außerhalb des Stacks, dann unterscheiden sich die führenden 20 Bits im Vergleich zum alten Zeiger und es wird eine entsprechende Ausnahme erzeugt.

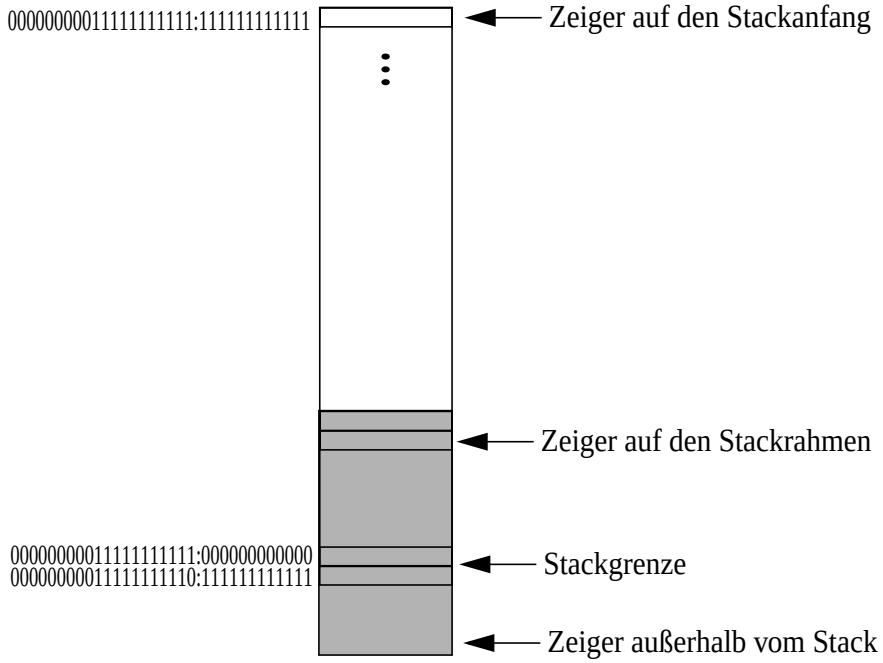


Abb. 3.2 Überprüfung der Stackgrenzen bei JX

3.2.3 Das Speicherlayout von Objekten

Für alle Objekte und deren Klassen fallen im laufenden System spezifische Daten an. Während die Objektzustände und Klassenzustände im Speicher der jeweiligen JX-Domain gespeichert werden und damit Domain spezifisch sind, werden die Progammdaten für alle Domains gemeinsam genutzt. Der Speicherschutz sorgt dafür, daß die Programmdaten nicht verändert werden können.

Um die Daten von Objekten, Klassen und Programmen richtig zu referenzieren, muß der Bytecode-Translator deren Organisation kennen, deswegen ist es nötig noch genauer auf die aktuelle Objekt-Struktur der Daten einzugehen. Die Abbildung 3.3 zeigt den Aufbau eines Objekts im

Speicher. Die Objektreferenz zeigt direkt auf die objektspezifischen Daten. Diese Daten umfassen einen Zeiger auf die Klassenbeschreibung, einen Zeiger auf eine Methodentabelle und direkt anschließend den Speicherbereich für die Objektdaten. Durch die direkte Referenzierung ist ein schneller Zugriff auf Objektvariablen und die Methodentabelle möglich. Der Verweis auf die Klassenbeschreibung wird unter anderem für die Typüberprüfung zur Laufzeit benötigt.

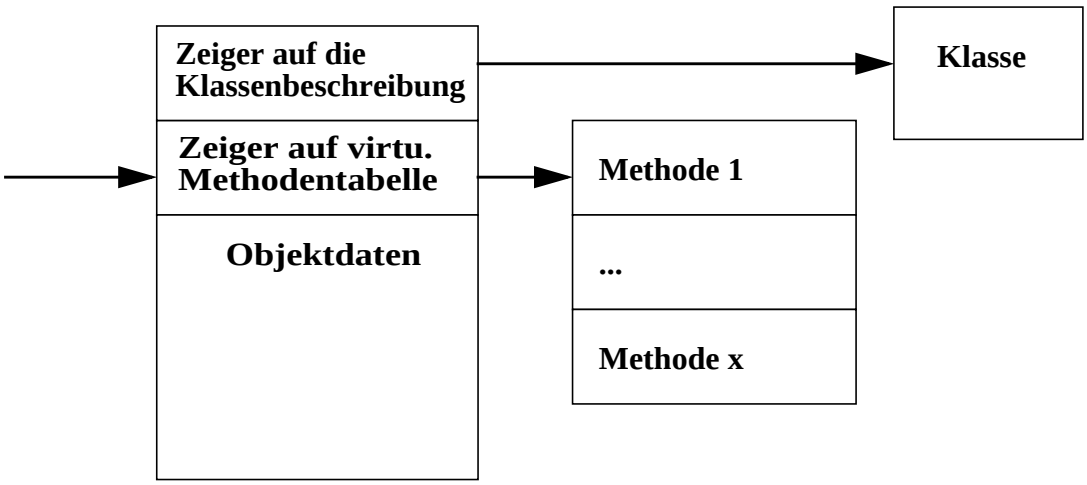


Abb. 3.3 Speicherlayout von Objekten in JX

Wichtiger als die Klassenbeschreibung ist die Methodentabelle. Im Java-Sprachkonzept sind fast alle Methodenaufrufe *virtuell.* Virtuell bedeutet in diesem Zusammenhang, daß die Methode der zur Laufzeit aktuellen Klasse aufgerufen wird und nicht die Methode der zur Übersetzungszeit bekannten Klasse. Dieses Verhalten entspricht dem objektorientierten Programmiermodell, hat jedoch das Problem, daß zum Übersetzungszeitpunkt die Sprungadresse für den Methodenaufruf nicht bekannt ist.

Um nun zur Laufzeit die gültige Sprungadresse schnell bestimmen zu können, wird eben die besagte Methodentabelle benutzt. In ihr sind die gültigen Sprungadressen gespeichert und über einen festen Index kann zur Laufzeit die benötigte Adresse ermittelt werden. Solche Methodentabellen werden auch oft als virtuelle Methodentabelle oder vtable bezeichnet. Der JX-Translator erzeugt für die Methodentabellen bei der Übersetzung die passenden Aufrufe.

Auch wenn ein Methodenaufruf über eine Methodentabelle relativ schnell ist im Vergleich zu einer Suche nach einem Methodennamen, ist er dennoch langsamer als ein statischer Methodenaufruf. Dieses ist oft mit ein Grund, warum ein in Java geschriebenes Programm im Vergleich zu Programmen mit überwiegend statischen Methodenaufrufen, wie z.B. bei C++, langsamer ist.

3.2.4 Die automatische Speicherfreigabe von JX

Die JVM besitzt eine sogenannte *garbage collection*, welche den Speicher von nicht mehr benutzten Objekten wieder freigibt. Eine Domain im JX-Betriebssystem ist aus der Sicht einer Java-Anwendung eine JVM und muß diese Funktionalität auch bieten.

Der Speicher ist im JX-System auf die einzelnen Domains aufgeteilt. Jede dieser Domains besitzt einen eigenen *garbage collector*. Dieser wird nach Bedarf aktiviert, um den Domain eigenen Speicher nach nicht mehr referenzierten Objekten zu durchsuchen. Objekte, die nicht mehr referenziert sind, sind auch der laufenden Anwendung nicht mehr bekannt und können folglich freigegeben werden.

Das JX-System muß dem *garbage collector* zum Erfüllen seiner Aufgabe die Möglichkeit bieten, alle aktuellen Objektreferenzen in einer Domain zu finden. Dafür gibt es in aktuellen JVMs zwei verbreitete Ansätze. Beim konservativen Ansatz wird für jede Objektreferenz ein Objekthandle eingeführt, über welches das Objekt adressiert wird. Dieser Ansatz hat den Vorteil, daß man Objekte auch besonders leicht im Arbeitsspeicher verschieben kann, er verlangsamt jedoch die laufende Anwendung, da alle Speicherzugriffe eine zusätzliche Dereferenzierung benötigen.

Bei JX wurde darum der zweite Ansatz gewählt. Bei diesem werden die Objekte direkt adressiert. Für diesen Ansatz genügt es jedoch nicht, wenn der *garbage collector* die Objektstruktur genau kennt, ihm muß auch die Position von Objektreferenzen auf dem Methodenstack bekannt sein. Für die Position von Referenzen auf dem Methodenstack erzeugt der JX-Translator sogenannte *Stackmaps*, dabei handelt es sich um eine statische Typinformation über die auf dem Stack gespeicherten Daten. Diese *Stackmaps* werden für feste Zeitpunkte eines Programmlaufes erzeugt. Soll die automatische Speicherfreigabe in einer Domain gestartet werden, so läßt das JX-System die Aktivitätsträger einer Domain noch bis zum nächsten Zeitpunkt mit einer bekannten *Stackmap* weiterlaufen und kann dem *garbage collector* anschließend die nötigen Informationen liefern.

Der JX-Translator erzeugt *Stackmaps* für alle Zeitpunkte von Methodenaufrufen, alle Sprünge in den JX-Kern und alle Rückwärtssprünge, wie sie z.B. in Programmschleifen vorkommen können. Dadurch wird gewährleistet, daß ein möglicher Zeitpunkt zum Freigeben des Speichers auf jedem Fall eintritt.

Neben dem Methodenstack und dem Speicherbereich der Objekte können sich aktive Objektreferenzen auch noch in Registern befinden. Aus diesem Grund müssen sich alle Register, welche Referenzen enthalten, zu dem Zeitpunkt einer aufgezeichneten *Stackmap* auf dem Stack befinden.

Bei Methodenaufrufen und Sprüngen in den JX-Kern ist dies sowieso nötig, da die aufrufende bzw. die aufgerufene Methode nicht wissen kann wie die jeweils andere Methode die Register nutzt. Die normalerweise gebräuchliche Aufrufsemantik, nach der ein Teil der Register von der aufgerufenen Methode gesichert werden, ist in JX nicht möglich. Die Register %ecx, %edi und %esi werden darum auch vom Aufrufer gesichert und nicht vom Aufgerufenen. Bei Rückwärtssprüngen müssen in Registern befindliche Referenzen zusätzlich gesichert werden.

3.3 Interne Darstellung

3.3.1 Bytecode-Objekte

Ein Java-Bytecode-Übersetzer unterscheidet sich in vielen Bereichen nur geringfügig von einem gewöhnlichen Compiler. Der Hauptunterschied liegt beim Einlesen des Progammcodes und der Syntaxanalyse, da der Java-Bytecode ein strenges Klassendateiformat besitzt, ist das Einlesen der Symbole und des Bytecodes im Vergleich zum Einlesen einer Programmiersprache einfach. In meiner Implementierung konnte ich zum Einlesen der Klassendateien auf eine bereits bestehende Klassenbibliothek zurückgreifen.

Für die einzelnen Java-Bytecodebefehle legt der Tranlator jeweils ein Bytecodeobjekt an, wobei gleichartige Befehle in gemeinsamen Klassen zusammengefaßt werden. Dies entspricht dem Vorgehen des von Hans Kopp implementierten Just-in-Time-Compilers für Java[Kopp98].

```
private final int const_value = 5;
private int member;

final public int methodForInlining(int a,int b) {
        a = a + member;
        if (a<0) {
                return const_value+b-a
        } else {
                return const_value+b+a;
        }
}

public int testMethod(int c) {
        int d = methodForInlining(c,7) + 5;
        return d;
}
```

```
method methodForInlining(II)I {
        START:
                v1=v1 + v0.member;
                if v1>=0 goto B10;
        B4:
                return 5+v2-v1;
        B10:
                return 5+v2+v1;
        END:
}

method testMethod(I)I {
        START:
                v2 = v0.methodForInlining(v1,7)+5;
                return v2;
        END:
}
```

Abb. 3.4 Ausgabe der Zwischenrepräsentation

Die eingelesenen Bytecode-Objekte werden anschließend mit Hilfe einer Programmflußanalyse und einer Datenflußanalyse um weitere Informationen ergänzt und in einer Zwischenrepräsentation gespeichert. Zu den ergänzenden Informationen zählen unter anderem das Auffinden der Operanten und die Unterteilung der Methode in Basisblöcke. Die Datenflußanalyse dient dazu, Operanten auch über Basisblockgrenzen hinweg richtig abzubilden. Eine aufwendigere Analyse mußte aus Zeitgründen entfallen. Die in der Zwischenrepräsentation gespeicherten Daten können in einer pseudo Programmiersprache als Zwischenergebnis ausgegeben werden. Abbildung 3.4 stellt als Beispiel zwei Java-Methoden der vom JX-Translator erzeugten Zwischenrepräsentation gegenüber.

3.3.2 Basisblöcke

Für die Optimierung der Methode ist es notwendig, diese in Basisblöcke zu unterteilen. Ein Basisblock ist dabei eine Folge von einzelnen Anweisungen, welche immer in einer zusammenhängenden Folge ausgeführt werden. Es gibt keine Sprünge in diese Folge hinein außer zum Anfang und es gibt keine Sprünge aus dieser Folge heraus außer am Ende[GE99].

Die Basisblockgrenzen werden in der Zwischenrepräsentation durch das Einfügen einzelner Basisblock-Objekte gekennzeichnet. Ihre Notwendigkeit für die späteren Optimierungsverfahren ist im Drachenbuch[ASU86] ausführlich beschrieben. Auch der virtuelle Operantenstack, welcher im JX-Translator zum Ermitteln der Operanten verwendet wird, benötigt dieses Wissen.

3.3.3 Virtueller Operantenstack

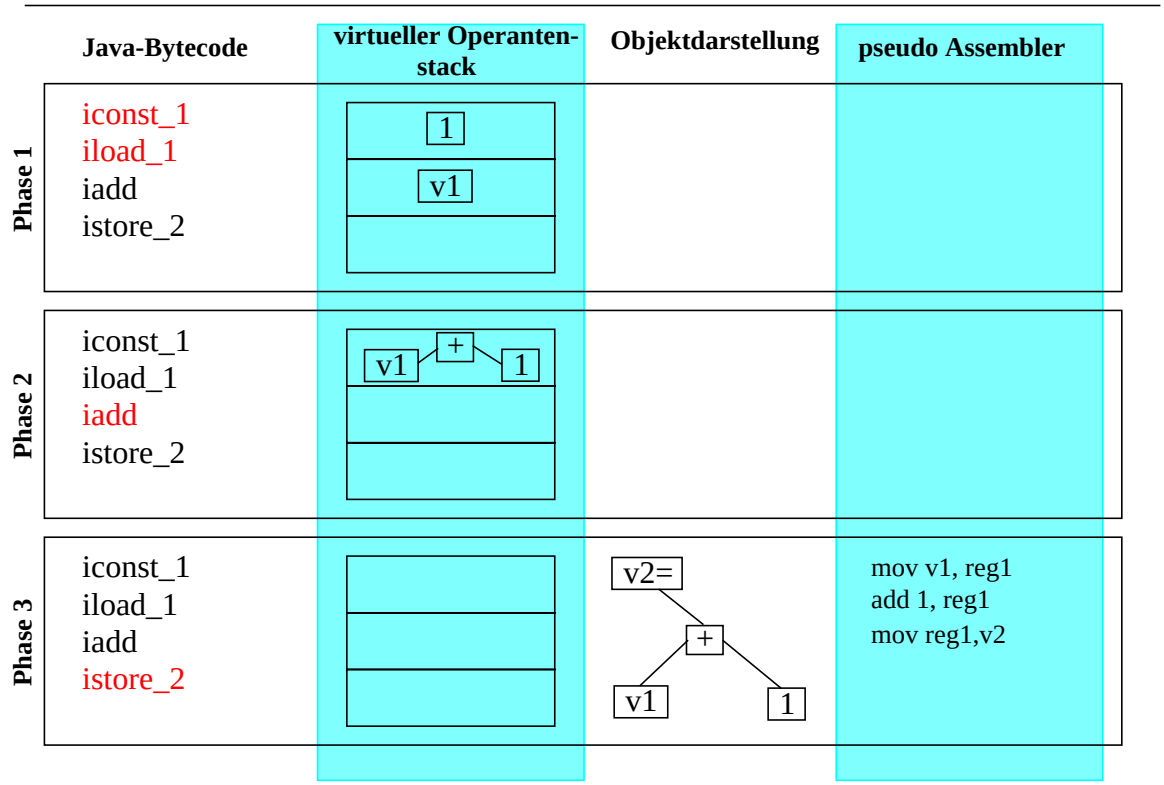


Abb. 3.5 Funktionsweise des virtueller Operantenstack

Beim Java-Bytecode handelt es sich um einen stackorientierten Befehlssatz[LY99], welcher die Operanten für die einzelnen Befehle von einem Operantenstack bezieht. Im Gegensatz dazu besitzt der Pentium einen Befehlssatz mit vorwiegend zwei Operanten, wobei die einzelnen Operanten entweder Register, über Register adressierte Speicheradressen oder Konstanten sein können. Um schnellen Maschinencode zu erzeugen ist es nötig, die Operanten für die jeweiligen Maschinenbefehle zu bestimmen, um nicht zur Laufzeit den Operantenstack nachbilden zu müssen. Dieses geschieht beim JX-Translator mit Hilfe eines sogenannten *virtuellen Operan-*

tenstacks. Die Verwendung eines *virtuellen Operantenstacks* wird in der Literatur auch für JIT-Compiler beschrieben[CFM+97]. Abbildung 3.5 veranschaulicht an einem Beispiel von Bytecodebefehlen die Funktionsweise des *virtuellen Operantenstacks* im JX-Translator.

In **Phase 1** werden die Befehlsobjekte für die Bytecodebefehle **iconst_1** und **iload_1** in der Reihenfolge ihres Auftretens auf einen Stack gelegt. Der Befehl **iconst_1** steht für das Laden der Konstante 1 und der Befehl **iload_1** für das Laden des Inhaltes der zweiten lokalen Variablen auf den Operantenstack der JVM, siehe auch hierzu die JVM Spezifikation [LY99].

Beim Auftreten einer Operation, wie zum Beispiel einer Addition (**iadd**) in **Phase 2**, werden die Befehlsobjekte vom Stack genommen und dem Befehlsobjekt der Addition als Operanten zugewiesen. Anschließend wird das Befehlsobjekt für die Addition auf den virtuellen Stack gelegt.

In **Phase 3** wird der Befehl **istore_2** angenommen, welcher die oberste Stackposition in der JVM einer lokalen Variablen zuweist. An dieser Stelle wird das Additionsobjekt mit den zwei Operanten vom *virtuellen Operantenstack* genommen und mit dem neuen Zuweisungsobjekt verknüpft. Das Resultat wird nicht wieder auf den Stack gelegt, sondern für die weitere Verarbeitung in der Zwischenrepräsentation gespeichert.

An dieser Stelle könnte auch schon Maschinencode für einen Prozessor wie den Pentium erzeugt werden, da alle Operanten bekannt sind. Der pseudo Assembler in der letzten Spalte der Abbildung zeigt eine mögliche Codesequenz für das Beispiel.

3.3.4 Syntaxbäume

Die vom *virtuellen Operantenstack* erzeugten Objektbäume entsprechen Syntaxbäumen, wie sie auch von Compilern beim Einlesen von Programmiersprachen erzeugt werden. Die Maschinencodeerzeugung aus solchen Syntaxbäumen ist bekannt und zum Beispiel auch im Drachenbuch[ASU86] beschrieben.

3.3.5 Behandlung vom Operantenstack an Basisblockgrenzen

Der Aufbau von Syntaxbäumen mit Hilfe des *virtuellen Operantenstacks* wie in Kapitel 3.3.3 beschrieben funktioniert nur innerhalb eines Basisblocks. Am Ende eines Basisblocks muß der Inhalt des Stacks auf ein reales Stackmodell zur Laufzeit abgebildet werden.

Das liegt daran, daß der Anfang eines Basisblocks von verschiedenen Stellen aus angesprungen werden kann und der Operantenstack dabei unterschiedliche Zustände enthalten könnte. Man kann diese Stackzustände auch als Aufruf- bzw. Rückgabeparameter für den jeweiligen Basisblock betrachten.

Der JX-Tranlator führt dafür weitere Variablen auf dem Methodenstack ein. Diese werden im weiteren als Blockvariablen bezeichnet. Jede beim Überschreiten der Basisblockgrenze genutzte Stackposition entspricht dabei einer Blockvariablen. Diesen Blockvariablen weißt der zu verlassende Basisblock die aktuellen Inhalte zu und dem aufgerufenen Basisblock werden die entsprechenden Blockvariablen auf den *virtuellen Operantenstack* gelegt.

3.4 Verwendete Optimierungen

3.4.1 Einführung

Die verschiedenen Optimierungstechniken lassen sich zum einen in lokale und zum anderen in globale Optimierungen unterteilen. Lokal ist in diesem Zusammenhang eine Optimierung, die nur innerhalb eines Basisblocks angewandt wird. Eine Optimierung ist global, wenn sie über Basisblockgrenzen hinweg geschieht. Außerdem ist eine Unterteilung in maschinenabhängige und maschinenunabhängige Optimierung sinnvoll. Bei einer maschinenunabhängigen Optimierung ist es, im Gegensatz zur maschinenabhängigen Optimierung, unwichtig für welche Zielplattform der Maschinencode am Ende erzeugt werden soll.

Die maschinenunabhängige Optimierung findet beim JX-Translator auf der in Kapitel 3.3 beschriebenen Zwischenrepräsentation statt. In den nachfolgenden Kapiteln werden die im JX-Translator implementierten maschinenunabhängigen Optimierungen nach lokaler und globaler Anwendung beschrieben.

Die maschinenabhängigen Optimierungen geschehen beim JX-Translator während der Erzeugung des Maschinenprogrammes. Dies hat den Vorteil, daß keine weitere Zwischendarstellung nötig ist, auf der eine solche Optimierung stattfinden könnte. Ein Beispiel für eine solche Optimierung ist die *Peephole*-Optimierung wie in dem Buch “Übersetzerbau”[GE99] von Güting und Erwig oder dem Drachenbuch[ASU86] beschrieben.

Der Verzicht hat jedoch auch den Nachteil, daß die Maschinenprogrammerzeugung komplexer wird und meistens ineffizientere Maschinenprogramme erzeugt. Im Kapitel 3.4.4 werden die Maschinencodeerzeugung und die maschinenabhängigen Optimierungen des JX-Translators genauer beschrieben.

3.4.2 Lokale maschinenunabhängige Optimierungen

3.4.2.1 Frühe Ausführung von Stackoperationen

Eine erste lokale Optimierung findet schon während der Erzeugung der Zwischenrepräsentation statt. Im Java-Bytecode gibt es Befehle, die nur den Zustand des Operantenstacks verändern. Beispiele für solche Befehle sind **pop**, **dup** und **swap**. Diese Befehle können zum Teil schon vom *virtuellen Operantenstack* ausgewertet werden und entfallen somit in der späteren Codeerzeugung völlig.

Der Befehl **swap** als Beispiel vertauscht in der JVM die obersten Operanten auf dem Operantenstack. Im JX-Translator werden dafür einfach die obersten Positionen auf dem *virtuellen Operantenstack* vertauscht, und damit die Operanten für die folgenden Befehle. Im Gegensatz zur JVM geschieht dies schon zur Übersetzungszeit und nicht erst zur Laufzeit.

3.4.2.2 Entfernen von unnötigen Referenzüberprüfungen

Beim Zugriff auf eine Objektreferenz muß von der JVM immer überprüft werden, ob diese Referenz nicht null ist. Ist eine Referenz null, so wird eine **NullPointerException** erzeugt und die Programmroutine zur Ausnahmebehandlung aufgerufen.

Der JX-Translator fügt zum Erkennen von ungültigen Referenzen entsprechende Überprüfungen ein, welche die jeweilige Referenz auf null testet und gegebenenfalls die genannte Ausnahme auslöst. Diese Überprüfung kann entfallen, wenn zu einem früheren Zeitpunkt die Referenz bereits überprüft wurde. Der JX-Translator implementiert dafür zwei unterschiedliche Verfahren.

Das einfachere Verfahren merkt sich für jede Referenz, ob bereits eine Überprüfung stattgefunden hat und verzichtet anschließend bei nachfolgenden Referenzzugriffen solange darauf bis die Referenz verändert wird. Beim Überschreiten von Basisblockgrenzen werden diese Informationen jedoch ungültig, da hier eine Aussage darüber, ob die Referenz bereits überprüft wurde oder nicht, nicht mehr so leicht getroffen werden kann und eine umfangreiche Datenflußanalye nötig wäre.

Beim zweiten Verfahren findet die Entfernung unnötiger Referenzüberprüfungen auf globaler Ebene statt und wird in Kapitel 3.4.3.3 beschrieben.

3.4.2.3 Entfernen von unnötigen Überprüfungen der Speichergrenzen

Die JVM Spezifikation schreibt vor, daß bei allen Arrayzugriffen überprüft wird, ob der Zugriff nicht die Speichergrenzen des Arrays überschreitet. Der JX-Translator fügt bei der Maschinencodeerzeugung für alle Arrayzugriffe einen entspechenden Maschinencode ein. Dies kann jedoch dann entfallen, wenn bei einem früheren Array-Zugriff für den gleichen oder einen größeren Index schon eine Überprüfung stattgefunden hat, wie zum Beispiel bei folgenden Programmzeilen.

```
a[1] = a[1] + 2;
b[i] = b[i] + 2;
```

Um in solchen Fällen unnötige Überprüfungen feststellen zu können, wird für jede Array-Referenz der höchste bereits überprüfte Index mitgeführt und die Referenz der zuletzt überprüften Indexvariablen. Ist bei einem nachfolgenden Array-Zugriff nun ein Index kleiner oder gleich dem mitgeführten Index, oder die Indexvariable die selbe und wurde in der Zwischenzeit nicht verändert, so kann auf eine Überprüfung verzichtet werden.

Das Entfernen von unnötigen Arrayüberprüfungen wird nur innerhalb eines Basisblocks vorgenommen. Beim Überschreiten einer Basisblockgrenze werden die gesammelten Informationen ungültig, da diese Einsprungspunkte von unterschiedlichen Stellen aus erreicht werden können.

Um das Verfahren auch über Basisblockgrenzen hinweg nutzen zu können, ist eine aufwendige Datenflußanalye nötig.

3.4.2.4 Beschleunigte Typüberprüfung

Neben der Überprüfung von Referenzen und Speichergrenzen muß die Java-Laufzeitumgebung an einigen Stellen zur Laufzeit Typüberprüfungen vornehmen. Es gibt dafür zwei sehr ähnliche Operationen. Eine dient der Abfrage ob eine Objektreferenz von einem bestimmten Typ ist und wird vom Programmierer gezielt durch die Anweisung **instanceof** vorgenommen. Der zweite Fall ist eine explizite Typumwandlung, bei der zur Übersetzungszeit nicht klar ist, ob die Referenz den gewünschten Typen entspricht.

Beide Operationen sind in der Mehrzahl aller Fälle sehr einfach zu beantworten. Immer dann wenn die zu überprüfende Referenz genau dem geforderten Typ entspricht oder der Referenzwert null ist, kann die Aufgabe von einer aus wenigen Maschinenbefehlen bestehenden Routine beantwortet werden. In allen anderen Fällen wird eine Routine in der DomainZero aufgerufen, welche überprüfen muß, ob es sich um einen kompatiblen Typ handelt oder nicht.

Der JX-Translator fügt für die einfachen Fälle das eben beschriebene kleine Maschinenprogramm ein, welches nur bei komplizierten Fällen in die DomainZero springt.

3.4.2.5 Konstantenpropagation und Konstantenfaltung

Bei der *Konstantenpropagation* werden Variablen, welche nachweislich einen konstanten Wert beinhalten, durch die entsprechende Konstante ersetzt. Dies alleine bringt nur selten einen Geschwindigkeitsvorteil. Es erhöht jedoch die Wahrscheinlichkeit, daß alle Operanten eines Ausdruckes konstant werden. Ein Ausdruck, dessen Operanten konstant sind und deren Wert schon zur Übersetzung bekannt ist, können zur Übersetzungszeit bereits ausgewertet werden. Dieser Vorgang wird als *Konstantenfaltung* oder *partielle Auswertung* bezeichnet. Eine *Konstantenfaltung* kann wiederum zu einer möglichen *Konstantenpropagation* führen, darum ist es sinnvoll, diese Verfahren mehrmals hintereinander anzuwenden.

Die *Konstantenpropagation* und *-faltung* kann auch von Java-Compilern beim Erzeugen des Java-Bytecodes erfolgen, und erscheint darum bei einer Bytecode-zu-Maschinencode-Übersetzung als nicht mehr sehr sinnvoll. Nach der in Kapitel 3.4.3.5 auf Seite 32 beschriebenen In-Line Expansion kann es durch das Propagieren von konstanten Methodenparametern jedoch wieder zu sinnvollen Kandidaten für die *Konstantenfaltung* kommen. Dadurch wird die In-Line Expansion auch von größeren Methoden vielversprechender.

Im JX-Translator ist die allgemeine Form der *Konstantenpropagation* und *-faltung* auf Basisblöcke begrenzt. Nur im Rahmen der In-Line Expansion werden konstante Parameter auch über Basisblockgrenzen hinweg propagiert.

3.4.3 Globale maschinenunabhängige Optimierungen

3.4.3.1 Allgemeines zu globalen Optimierungen im JX-Tranlator

Für die globalen Optimierungsmethoden greift der JX-Translator auf Ergebnisse des Bytecode-Verifiers zurück. Der Bytecode-Verifer überprüft den Java-Bytecode bevor er vom JX-Translator verarbeitet wird und muß zu diesem Zweck eine Programm- und Datenflußanalyse vornehmen. Während der Überprüfung erzeugt der Verifer einige zusätzliche Daten, welche vom Übersetzer benötigt werden. Diese werden in der Klassenstruktur gespeichert. Zum Übersetzungszeitpunkt kann der JX-Translator anschließend auf diese Informationen zurückgreifen. Diese Vorgehensweise erspart dem Gesamtsystem viel doppelte Arbeit, und beim JX-Translator die Implementierung einer globalen Programmfluß- und Datenflußanalyse.

3.4.3.2 Entfernen von unnötigen Stacküberprüfungen

Die nötige Überprüfung der Stackgrenzen, wie sie in Kapitel 3.2.2 auf Seite 21 ausführlich beschrieben ist, erhöht die Kosten für einen Methodenaufruf nicht unerheblich. Besonders bei kleinen Methoden, welche nur wenige Operationen ausführen, kann so der Aufwand für den Methodenaufruf die Rechenzeit der Methode dominieren.

Die In-Line-Expansion, wie im nachfolgenden Kapitel beschrieben, löst zwar auch dieses Problem, kann jedoch nicht für alle Methoden sinnvoll angewendet werden. Es stellt sich darum die Frage, ob es Methoden gibt, für die eine Stackgrenzenüberprüfung entfallen kann.

Eine Überprüfung kann dann entfallen, wenn die Stackgrenzenüberprüfung der aufrufenden Methode auch sicherstellt, daß der Stackrahmen der aufgerufenen Methode noch genügend Platz auf dem Stack findet. Dies wird dadurch sichergestellt, daß alle Methoden auf einen deutlich größeren Bereich testen als sie selber benötigen. Das Sicherstellen dieser Platzreserven ist auch aus einem anderen Grund nötig. Bei Sprüngen in den JX-Kern, zum Beispiel beim Auslösen einer Ausnahme, muß für die Arbeiten im Kern auch noch genügend Platz auf dem Stack vorhanden sein.

Wir können daher die Stackgrenzenüberprüfung immer dann entfallen lassen, wenn eine Methode relativ wenig Speicher auf dem Stack benötigt und keine weiteren Methoden aufruft. In diesen Fällen können wir sicherstellen, daß eine aufrufende Methode auch die Stackgrenze für die aufgerufene Methode überprüft hat.

Die Information, ob eine Methode diese Bedingungen erfüllt, könnte der JX-Translator beim Übersetzen selber ermitteln. Es ist jedoch sehr geschickt, diese Information schon vor dem Übersetzungsvorgang zu kennen. Der Bytecode-Verifier, welcher vor dem JX-Translator läuft, liefert daher dem JX-Translator diese Information.

3.4.3.3 Entfernen unnötiger Referenzüberprüfungen

Durch eine globale Datenflußanalyse, wie sie vom Bytecode-Verifier vorgenommen wird, können im Vergleich zu einer lokalen Betrachtung wie im Kapitel 3.4.2.2, noch wesentlich mehr Referenzüberprüfungen entfallen.

Der JX-Translator arbeitet bei dieser Optimierung wieder mit dem JX-Bytecode-Verifier zusammen und greift auf dessen in der Klassenbeschreibung gespeicherte Daten zurück.

3.4.3.4 Statische Methodenaufrufe

Bei Java sind virtuelle Methodenaufrufe die Regel. Im JX-System werden diese virtuellen Methodenaufrufe mittels Methodentabellen realisiert. Eine genauere Beschreibung hierzu befindet sich unter Kapitel 3.2.3 auf Seite 22.

In vielen Fällen sind virtuelle Methodenaufrufe jedoch gar nicht nötig. Da zur Übersetzungszeit schon bekannt ist, welche Methode als einzige aufgerufen werden kann, ist es möglich, den virtuellen Methodenaufruf durch einen statischen Aufruf zu ersetzen.

Dies ist unter Anderem immer dann der Fall, wenn die Methode einer Klasse als privat (**private)** oder abgeschlossen (**final)** gekennzeichnet ist. Diese Methoden können von keiner Unterklasse mehr überschrieben werden und sind durch die zur Übersetzungszeit bekannte Klasse hinreichend beschrieben.

Die vom Programmierer als **private** oder **final** gekennzeichneten Methoden sind jedoch nicht die einzigen Kandidaten für einen statischen Methodenaufruf. Da das JX-System einen Überblick über alle im System befindlichen Klassen besitzt, kann es bestimmen, welche Methoden zum Zeitpunkt der Übersetzung in keiner der bekannten Unterklassen überschrieben werden und für diese ebenfalls statische Methodenaufrufe benutzen. Diese Methoden sind sozusagen "systemweit abgeschlossen" und werden im System als **system final** gekennzeichnet.

Wird in einer Methode der Methodenaufruf einer systemweit abgeschlossenen Methode als statischer Aufruf übersetzt, muß diese Methode möglicherweise beim Nachladen von Klassen erneut übersetzt werden. Aus diesem Grund kennzeichnet der JX-Translator die entsprechenden Methoden. Die Information darüber, welche Methoden systemweit abgeschlossen sind, bekommt der JX-Translator vom JX-Verifier.

3.4.3.5 In-Line Expansion von Methoden

Das objektorientierte Programmiermodell von Java führt zu vielen kleinen Methoden. Insbesondere treten dabei Methoden auf, welche nur den Inhalt einer Objektvariablen auslesen bzw. setzen oder welche lediglich einen weiteren Methodenaufruf tätigen.

Bei diesen Methoden ist der Zeitaufwand für den Methodenaufruf größer als die anschließend für die eigentliche Arbeit aufgewandte Zeit. Eine für diese Art von Methodenaufrufen vielversprechende Optimierungstechnik ist die sogenannte In-Line Expansion.

Bei der In-Line Expansion wird der Methodenaufruf durch den Inhalt der aufzurufenden Methode ersetzt. Werden häufig wiederverwendete große Programmabschnitte auf diese Weise in Methoden eingefügt, führt dies zum einen zu einem größeren Speicherbedarf für das Maschinenprogramm, zum anderen kann es aufgrund der Cache-Architektur des Prozessors auch zu einer Verlangsamung kommen. Der JX-Translator versucht deshalb von sich aus nur bei kleinen Methoden die In-Line Expansion anzuwenden.

Wie beim Ersetzten von virtuellen Methodenaufrufen durch statische Methodenaufrufe, ist auch die In-Line Expansion von virtuellen Methodenaufrufen nicht ohne Einschränkung möglich. Für die In-Line Expansion muß zur Übersetzungszeit die aufzurufende Methode auch eindeutig feststehen. Der JX-Translator benutzt hierfür die gleichen Kriterien wie beim Einsatz von statischen Methodenaufrufen.

```
method methodForInlining(II)I {
        START:
                v1=v1 + v0.member;
                if v1>=0 goto B10;
        B4:
                return 5+v2-v1;
        B10:
                return 5+v2+v1;
        END:
}

method testMethod(I)I {
        START:
                v2 = v0.methodForInlining(v1,7)+5;
                return v2;
        END:
}
```

```
method testMethod(I)I {
        START:
                v3 = v1;
        START_methodForInlining:
                v3 = v3 + v0.member;
                if v3>=0 goto B10;
        B4:
                v4 = 5+7-v3
                 goto END_methodForInlining;
         B10:
                v4 = 5+7+v3;
         END_methodForInlining:
                v2 = v4+5;
                return v2;
         END:
}
```

Abb. 3.6 Beispiel: In-Line Expansion von Methoden

Die Abbildung 3.6 veranschaulicht an einem einfachen Beispiel die In-Line Expansion von Methoden. Die Methoden sind in einer Pseudoprogrammiersprache verfaßt, welche aus der Zwischenrepräsentation des JX-Translators erzeugt wurde. Die Lokalenvariablen werden als **v0** bis **v4** bezeichnet, die Methodennamen werden den Symbolnamen aus der Java-Klassendatei entsprechend gewählt.

In der linken Spalte wird in der Methode **testMethod** die Methode **methodForInlining** aufgerufen. Die lokale Variable **v0** enthält dabei einen Zeiger auf das eigene Objekt, entspricht also dem **this**-Zeiger in der Java-Programmiersprache. Die Methode wird mit drei Parametern aufgerufen: Einerseits mit der lokalen Variable **v1** und einer Konstanten mit dem Wert **7** und zum anderen implizit mit dem Objektzeiger **v0**.

Die rechte Spalte zeigt die Methode **testMethod** nach der In-Line Expansion. Die Parameter **v0** und die Konstante können die lokalen Variablen **v0** und **v2** in der eingefügten Methode ersetzen, da diese innerhalb der Methode nicht verändert werden. Der Parameter **v1** muß jedoch zuerst

kopiert werden, um das gleiche Verhalten wie vor dem Einfügen der Methode zu erhalten. Hierfür wird eine neue lokale Variable **v3** eingeführt und ihr der Wert von **v1** zugewiesen. Die Variable **v3** ersetzt anschließend die Variable **v1** in der eingefügten Methode.

Für den Rückgabewert der Methode wird auch eine neue Variable eingeführt. In unserem Beispiel ist es **v4**. Ihr wird, bei Rücksprüngen aus der Methode, der Rückgabewert zugewiesen und die Anweisung **return** wird durch einen Sprung zum Ende der eingefügten Methode ersetzt.

Das Propagieren von konstanten Parametern zeigt in unserem Beispiel, daß dadurch zum einen Zuweisungen wegfallen können, zum anderen entstehen dadurch auch Kandidaten für die im Kapitel 3.4.2.5 beschriebene Konstantenfaltung.

3.4.3.6 Methodenspezifische Übersetzung

Für einige systemnahe Klassen ist ein erweiterter Zugriff auf die Hardware nötig. Zum Beispiel bei einer Klasse für den PCI-Bus oder den Videospeicher. Die nötige Funktionalität stellt in der JX-Systemarchitektur der JX-Kern zur Verfügung. Der JX-Kern ist als DomainZero für alle anderen Domains über normale Portale zugänglich und kann so transparent diese Dienste bieten.

Ein Portalaufruf ist im Vergleich zu einer IPC in anderen Betriebssystem recht schnell, er dauert auf einem Pentium III etwas weniger als 650 Taktzyklen[GKB01], was ungefähr dem Zwanzigfachen eines einfachen Methodenaufruf entspricht. Auf dem gleichen Prozessor benötigt eine Implementierung des schnellen L4KA-Microkernels immerhin 800 Taktzyklen für eine IPC[GJP+00].

Für einige Anwendungsfälle dauert dies jedoch schon entschieden zu lange. Der JX-Translator bietet für diese Fälle die Möglichkeit, einzelne Methoden durch spezielle Erweiterungen erzeugen zu lassen. Diese Erweiterungen werden im weiteren auch als *Plugins* bezeichnet.

Die Methoden werden wie bei einem Portalaufruf als Java-Interface beschrieben. Während des Übersetzungvorgangs wird anstelle der Erzeugung eines Methodenaufrufs die Maschinenprogrammerzeugung dem Plugin überlassen. Auf diese Weise ist es möglich, anstelle von Portalaufrufen gleich entsprechende Maschinenbefehle einzufügen.

Anwendungsbeispiel hierfür sind das Sperren von Interrupts, das Auslesen der Systemzeit oder einfach das Vertauschen der Byteorder. Für all diese Beispiele werden nur wenige Maschinenbefehle benötigt und sie rechtfertigen nicht den Aufwand für einen Portalaufruf.

Ein anderes Beispiel für den Einsatz von Plugins ist die Implementierung eines speziellen Memory-Objekts im JX-System. Dieses Memory-Objekt ermöglicht Java-Anwendungen den direkten Speicherzugriff für definierte Speicherbereiche, wie es für einige Hardwarezugriffe notwendig ist und für Anwendungen die dem Netzwerktreiber oder dem Dateisystem sinnvoll erscheinen.

Durch den Einsatz eines Plugins für einige Methoden des Memory-Objekts wird zum Beispiel das einfache Setzen von Bytes nur durch eine Bereichsüberprüfung und den entsprechenden Maschinenbefehl übersetzt. Im Idealfall dauert dies nur wenige Taktzyklen.

Die Plugins sind weder hardwareunabhängig noch unterliegen sie dem Speicherschutzkonzept von JX. Sie müssen darum für das System als vertraulich betrachtet werden.

3.4.4 Maschinenabhängige Optimierung

3.4.4.1 Der Pentium Prozessor

Um die vom JX-Translator vorgenommenen maschinenabhängigen Optimierungen besser verstehen zu können, ist es wichtig, die Funktionsweise des Pentium Prozessors genauer zu betrachten. Das *Intel Architecture Optimization (Reference Manual)* [Intel99], beschreibt umfassend den Aufbau und die Areitsweise des Pentium Prozessors. Im folgenden sollen nur die wesentlichen Abläufe erläutert werden.

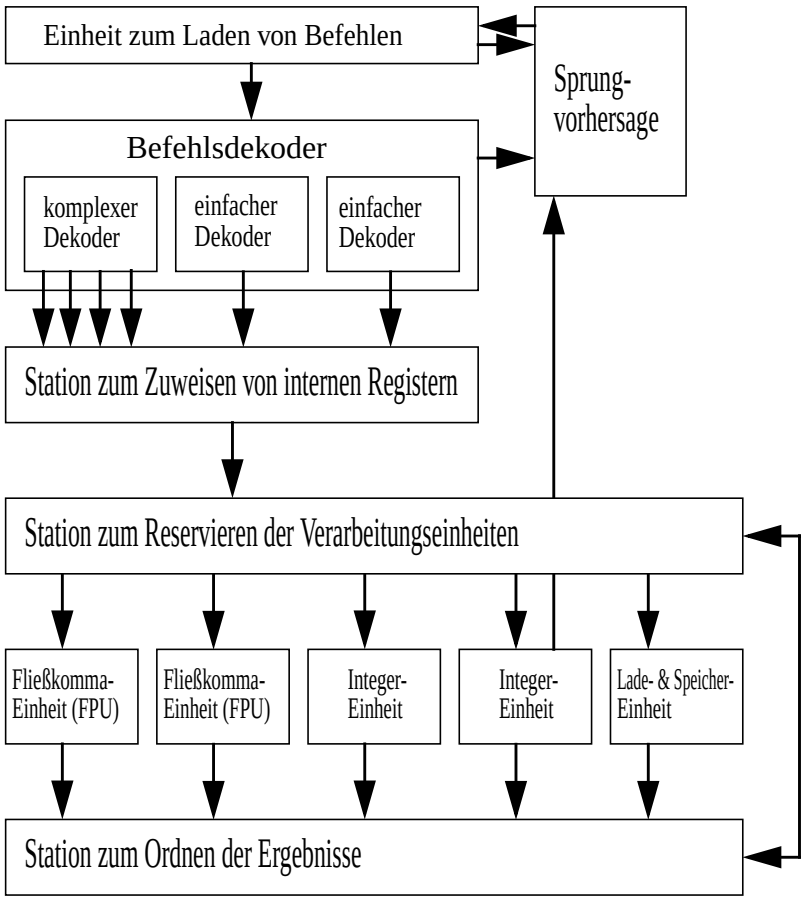


Abb. 3.7 Der Pentium Prozessor

Der Pentium besitzt einen für heutige Prozessoren ungewöhnlich komplexen Befehlssatz mit auffallend wenigen Registern. Der Grund dafür liegt in dem Bestreben, den Prozessor zu seinen Vorgängermodellen jeweils binärkompatibel zu halten. Obwohl der Pentium einen komplexen Befehlssatz besitzt, ist er jedoch kein reiner CISC-Prozessor mehr, sondern implementiert viele

der von RISC-Prozessoren bekannten Techniken. Die Entwickler des Pentium Prozessors zerlegten zu diesem Zweck die komplexen Befehle in sogenannte Mikrooperationen (yOps), die anschließend wie in einem RISC Prozessor verarbeitet werden.

Das Abarbeiten der Maschinenbefehle erfolgt beim Pentium in insgesamt 12 aufeinander folgenden Arbeitsschritten. Die einzelnen Arbeitsschritte von aufeinanderfolgenden Befehlen werden dabei, ähnlich wie bei einem Fließband, überschneidend ausgeführt. Diese Technik wird nach dem englischen Begriff für Fließband, der sogenannten “pipeline”, auch als Pipelining bezeichnet. Mit Hilfe von Piplining ist es möglich, pro Prozessortakt einen Maschinenbefehl fertigzustellen, auch wenn jeder einzelne Befehl mehrere Takte benötigt.

Um die Verarbeitungsgeschwindigkeit des Prozessors weiter zu steigern, setzt der Pentium auf eine Technik, die zuerst in RISC-Prozessoren angewandt wurde. Die einzelnen Mikrooperationen können auf mehreren Verarbeitungseinheiten parallel ausgeführt werden. Der Prozessor besitzt insgesamt fünf Einheiten: zwei Fließkommaeinheiten, zwei Integereinheiten und eine Einheit zum Laden und Speichern von Operanten. Um Abhängigkeiten zwischen den einzelnen Maschinenbefehlen aufzulösen können die 8 Register des IA32-Befehlssatzes intern auf wesentlich mehr Register abgebildet werden. Nach dem Abarbeiten der Befehle in einer losen Reihenfolge werden die Ergebnisse in einem letzten Arbeitsschritt wieder in die richtige Arbeitsreihenfolge gebracht.

Um die Verarbeitungseinheiten mit genügend Mikrooperationen zu versorgen, kann die Dekodereinheit bis zu drei Maschinenbefehle gleichzeitig dekodieren. Dafür besitzt sie einen vollwertigen Dekoder, welcher auch komplexe Befehle dekodieren kann und zwei einfache, welche nur Befehle verarbeiten können, die lediglich eine Mikrooperation benötigen. Auf diese Weise können bei einer idealen Anordnung von Maschinenbefehlen bis zu 6 Mikrooperationen pro Takt in die Pipeline gestellt werden.

Damit die einzelnen Stationen bei der Verarbeitung von Maschinenbefehlen fortlaufend mit Arbeit versorgt werden können, ist es normalerweise notwendig, den nachfolgenden Maschinenbefehl genau zu kennen. Dies ist jedoch bei bedingten Sprüngen nicht der Fall, wenn diese von Ergebnissen in Bearbeitung befindlicher Maschinenbefehle abhängen. Der Prozessor versucht in einem solchen Fall das Sprungverhalten zu “erraten” und macht mit einem der zwei möglichen Sprungziele weiter. Im Falle eines richtig vorhergesagten Sprungzieles, bleibt auf diese Weise die Pipeline gefüllt, bei einer falschen Vorhersage wird die bis zu diesem Zeitpunkt geleistete Arbeit verworfen und mit dem richtigen Befehl wieder neu gestartet. In einem solchen Fall dauert es 9-26 Takte bis wieder ein Befehl fertig wird.

Der Prozessor benutzt für die Vorhersage eine eigene Tabelle, in der protokolliert wurde ob der Sprungbefehl beim letzten Besuch genommen wurde oder nicht genommen wurde. Von dem Inhalt dieser Tabelle wird die aktuelle Entscheidung abhängig gemacht. Kommt der Sprungbefehl jedoch das erstemal zur Ausführung, dann wird anhand der Sprungrichtung eine Entscheidung gefällt. Bei Rückwärtssprüngen, wie sie bei Schleifen oft vorkommen, wird davon ausgegangen, daß der Sprung genommen wird. Bei Vorwärtssprüngen dagegen wird der Sprung nicht genommen.

3.4.4.2 Registerbelegung

Das möglichst optimale Abbilden von lokalen Variablen auf die Register von Prozessoren ist in der Literatur schon oft untersucht worden. Eine optimale Abbildung gilt als NP-Vollständig[ASU86] und ist somit für einen Übersetzer, der auf eine möglichst kurze Übersetzungszeit achtet, nicht zu empfehlen.

Das Erzeugen eines Maschinenprogramms aus einem Syntaxbaum mit einer minimalen Anzahl von benötigten Registern zum Speichern von Zwischenergebnissen ist dagegen in linearer Zeit zu bewerkstelligen. Ausgehend von dem durch den virtuellen Operantenstack erzeugten Syntaxbaum in Kapitel 3.3.3 auf Seite 26 wird jeweils der Zweig mit dem größeren Bedarf an Registern zuerst übersetzt[ASU86].

Der JX-Translator achtet bei diesem Vorgehen auch darauf, daß bereits in Register befindliche lokale Variablen nicht erneut geladen werden. Das Vorgehen in dieser Form ist noch weit von einer wirklich guten Registerbelegung, wie sie in heutigen Compilern Verwendung findet, entfernt. Es erzeugt jedoch für kleine Methoden schon recht akzeptable Maschinenprogramme.

3.4.4.3 Entfernen der Ausnahmenbehandlung aus dem Programmfluß

Der JX-Translator fügt mehrere Referenz- und Speicherüberprüfungen in das Maschinenprogramm ein. Bei jedem Test wird entweder in einen Aufruf zum Auslösen einer Ausnahme verzweigt, oder wie in den meisten Fällen das Programm normal fortgesetzt.

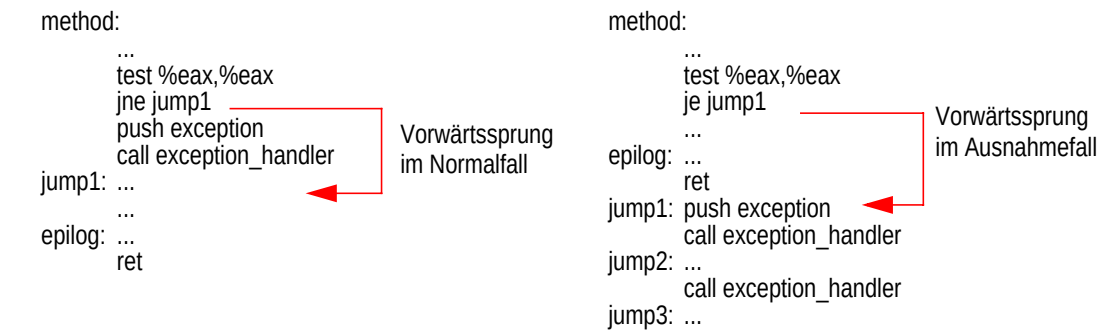


Abb. 3.8 Entfernen der Ausnahmebehandlung aus dem Programmfluß

Berücksichtigt man bei den Programmverzweigungen die Sprungvorhersage des Prozessors und ordnet das Maschinenprogramm so, daß nur im Ausnahmefall der Vorwärtssprung genommen wird, so verbessert man den Durchsatz des Prozessors schon allein dadurch, daß die Sprünge nun richtig vorhergesagt werden.

Die Abbildung 3.8 zeigt die zwei unterschiedlichen Vorgehensweisen. Links werden die Maschinenbefehle zum Erzeugen einer Ausnahme im fehlerfreien Fall übersprungen. Es findet dabei im Normalfall ein Vorwärtssprung satt, welcher durch die Sprungvorhersage des Prozessors beim ersten Auftreten falsch vorhergesagt wird, also mindestens 9 Prozessortakte kostet. In der

rechten Lösung sind die Befehle an das Ende der Methode angehängt und werden im Fehlerfall angesprungen. Der Vorwärtssprung findet demzufolge nur im Ausnahmefall satt. Im Normalfall wird mit dem von der Sprungvorhersage richtig ausgewählten Befehl weitergearbeitet.

Die rechte Lösung hat noch einen weiteren Vorteil. Im fehlerfreien Fall müssen deutlich weniger Programmdaten aus dem Hauptspeicher geladen werden, da die Befehle zum Erzeugen der Ausnahme nicht direkt dem Sprung folgen, werden sie nicht versehentlich in den Befehlscache des Prozessors geladen.

Der JX-Translator ordnet bei allen von ihm eingefügten Referenz- und Speichertests die Ausnahmeaufrufe, wie in der rechten Lösung beschrieben, an das Ende einer Methode. Die Vorteile dieses Vorgehens sind auch in einer Veröffentlichung über schnelle und effiziente Maschinenprogrammerzeugung für JIT-Compiler[ACL+98] beschrieben. Eine weitere Auswertung von bedingten Sprüngen und eine entsprechende Anordnung wird nicht vorgenommen. Hoch optimierende Compiler für C und C++ nutzen für entsprechende Optimierungen in Testläufen gesammelte Daten und ordnen anschließend möglichst viele Sprünge in einer idealen Folge.

3.4.4.4 Optimierte Befehlsauswahl

Der Pentium Prozessor verfügt über einen recht komplexen Befehlssatz. Die Ausführungszeit der einzelnen Befehle ist dabei sehr unterschiedlich. Die Unterschiede hängen sowohl von der Befehlslänge, welche stark variiert, als auch von der Anzahl der zu erzeugenden Mikrooperationen ab. Wie in Kapitel 3.4.4.1 beschrieben, kann der Befehlsdekoder nur jeweils einen aus mehreren Mikrooperationen bestehenden Maschinenbefehl pro Takt dekodieren. Bei besonders komplexen Maschinenbefehlen benötigt der Dekoder sogar mehrere Takte. Von Maschinenbefehlen, welche lediglich eine Mikrooperation benötigen, kann der Dekoder jedoch zwei zusätzliche pro Takt dekodieren.

Das *Intel Architecture Optimization (Reference Manual)* [Intel99] listet für die Maschinenbefehle auf, welche Befehle als komplex gelten, also mehrere Takte zum Dekodieren benötigen und wie viele Mikrooperationen für welchen Befehl benötigt werden. Zum Optimieren des Maschinenprogramms wird empfohlen, auf die komplexen Befehle zu verzichten und auch möglichst auf kurze Befehle zu achten. Darüber hinaus werden einige einfache Alternativen vorgeschlagen.

Bei der Maschinenprogrammerzeugung berücksichtigt der JX-Translator einige dieser Vorschläge und verzichtet weitgehend auf den Einsatz von komplexen Maschinenbefehlen. Beispiele für komplexe Maschinenbefehle sind **enter, leave** oder **loop.** Diese Befehle benötigen mehrere Mikrooperation und/oder mehrere Taktzyklen zum Dekodieren.

Ein anderes Beispiel für geschicktere Befehlsauswahl ist der Einsatz des **xor**-Befehls, zum Löschen von Registerinhalten, anstelle eines **mov**-Befehls. Verknüpft man den Inhalt eines Registers über die xor-Funktion mit dem gleichen Register, so ist das anschließend im Register gespeicherte Ergebnis null. Dieses Vorgehen bewirkt zum einen, daß der **xor**-Befehl kürzer ist, zum anderen hat es auch noch den Vorteil, daß dieser Befehl auf einer der zwei Integereinheiten parallel ausgeführt werden kann.

Beim Pentium werden bedingte Verzweigungen in der Regel mit zwei Maschinenbefehlen kodiert. Einer Vergleichsoperation, entweder **cmp** oder **test**, folgt eine Sprunganweisung, zum Beispiel **je**. Hierbei können alle Vergleiche mit dem Wert Null durch eine **test**-Anweisung ersetzt werden. Ähnlich wie der **xor**-Befehl, der beim Löschen eines Register angewandt wird, ist diese Anweisung kürzer. Es geht jedoch noch kürzer. Die Vergleichsoperation kann völlig entfallen, wenn der zu vergleichende Wert das Ergebnis einer Integeroperation ist. Diese Maschinenbefehl setzen bereits die Flags für die Sprunganweisung.

Der JX-Translator berücksichtigt diese Beispiele und versucht, wo es möglich ist, konstante Werte von Operanten als konstante Parameter den Maschinenbefehlen zu übergeben. Das erzeugte Maschinenprogramm wurde auf Grund dieser Maßnahmen und der geschickteren Registerbelegung um ein Drittel kleiner.

3.5 Ergebnisse

Die vom JX-Translator erzielten Verbesserungen zeigten sich an einer realen Anwendung für das Betriebsystem JX. In der Arbeit von Andreas Weissel wurde das von Linux bekannte ext2-Dateisystem und der IOZone-Benchmark für JX implementiert[Weissel00]. Beim IOZone-Benchmark werden für unterschiedliche Block- und Dateigrößen der Durchsatz des Dateisystems für Lese- und Schreibzugriffe ermittelt. Für eine Vergleichsmessung kommt der JIT-Compiler von Hans Kopp zum Einsatz[Kopp98]. Der ursprünglich für MetaXa entwickelte JIT-Compiler wurde von Michael Golm auf die Bedürfnisse von JX angepaßt und stellte vor dem JX-Translator den einzigen Java-Bytecode-zu-Maschinencode-Übersetzer des Systems dar.

Die Abbildung 3.9 zeigt die Ergebnisse eines IOZone-Benchmarks für Blockgrößen von 4 bis 128 kbyte und Dateigrößen von ebenfalls 4 bis 128 kbyte. Für die Graphik wurden die Ergebnisse von zwei unterschiedlichen Durchläufen benutzt. Ein Durchlauf wurde mit einem vom JX-Translator übersetzten Dateisystem erstellt, der zweite mit einem Dateisystem, welches vom JIT-Compiler übersetzt wurde. Die ersten zwölf Spalten von links zeigen die Ergebnisse vom Schreibdurchsatz beim wiederholten Schreiben derselben Datei. Die ersten sechs Ergebnisse repräsentieren das Dateisystem mit dem JX-Translator, die nächsten sechs Ergebnisse stehen für den Lauf mit dem JIT-Compiler. Anschließend folgen die jeweils sechs Ergebnisse für das wiederholte Lesen einer Datei.

Durch das wiederholte Schreiben beziehungsweise Lesen der Daten befinden sich diese mit großer Wahrscheinlichkeit im Cache des Prozessors. Wir messen bei kleinen Dateien daher nicht den Durchsatz von Festplatten beziehungsweise Hauptspeicherzugriff, sondern den Durchsatz beim Kopieren im Cache und den durch das Dateisystem verursachten Verwaltungsaufwand.

Während das Kopieren der Daten im Cache sich bei beiden Messungen kaum unterscheiden dürfte, hängt die benötigte Zeit für den Verwaltungsaufwand von der Qualität des erzeugten Maschinenprogamms ab.

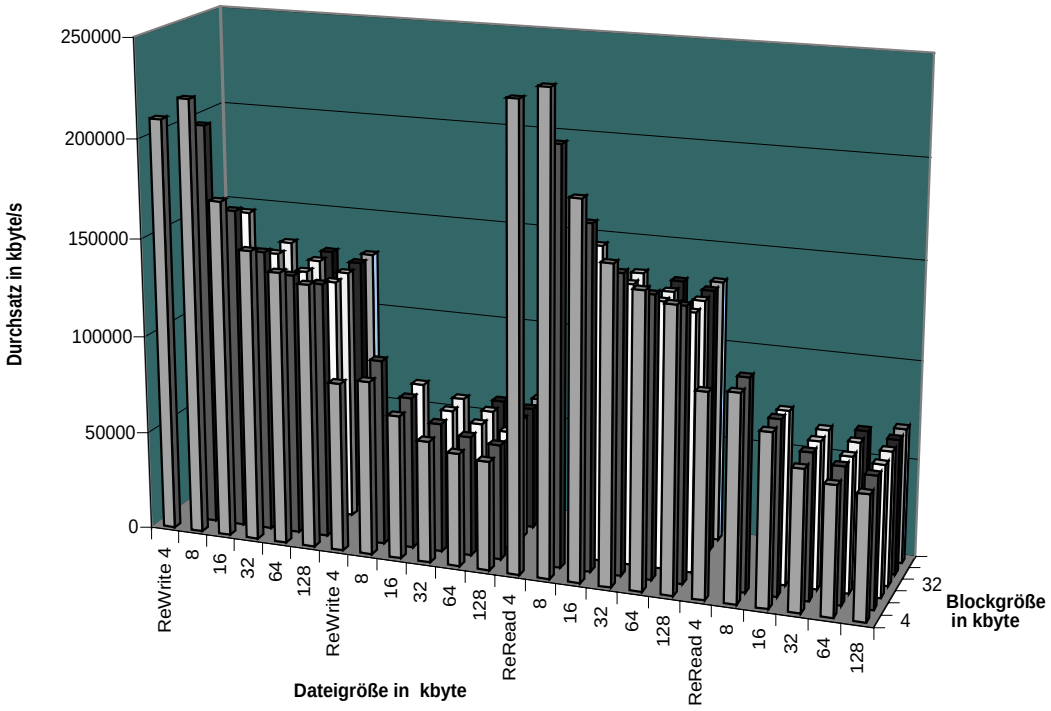


Abb. 3.9 Ergebnisse vom IOZone Benchmark

Von den beschriebenen Optimierungstechniken des JX-Translators kamen bei der Messung nicht alle zum Einsatz. Da die globalen Techniken vom Ergebnis des Bytecode-Verifiers abhängen und dieser noch nicht zur Verfügung stand, fehlen alle globalen Techniken, soweit sie auf Ergebnisse des Verifiers angewiesen sind. Die In-Line Expansion war nur für sehr kleine Methoden aktiv. Eine methodenspezifische Übersetzung wurde für die wichtigsten Methoden des Memory-Objektes benutzt, das beim Kopieren der Daten und beim Lesen der Verwaltungsstruktur zum Einsatz kommt.

Der Spitzendurchsatz mit dem neuen Übersetzer lag bei 221 Mbyte/s für den schreibenden Zugriff bzw. 239 Mbyte/s beim lesenden Zugriff. Beide Werte wurden bei einer Dateigröße von 8 kbyte und einer Blockgröße von 4 kbyte erzielt. Im Vergleich dazu kam der JIT-Compiler ohne Optimierungen nur auf 94 bzw. 106 MByte/s. Die Abbildung 3.10 stellt die Ergebnisse für eine feste Blockgröße von 4 KByte separat dar.

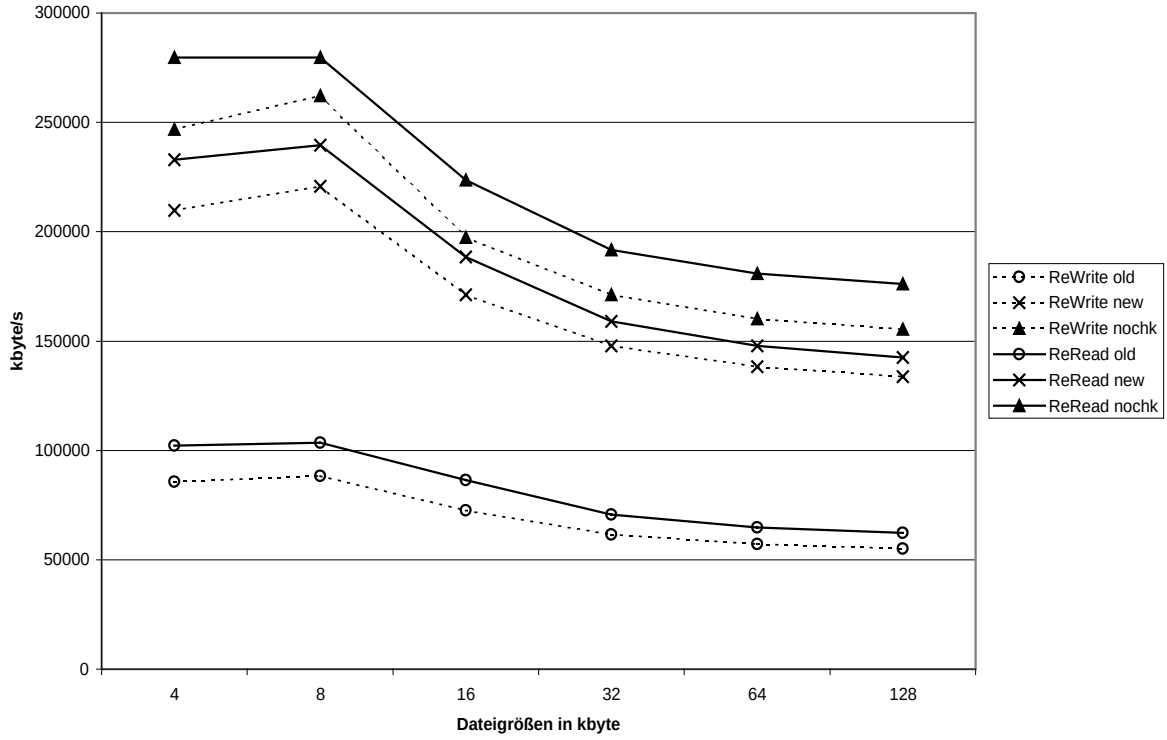


Abb. 3.10 Ergebnisse vom IOZone Benchmark im Vergleich

Die mit einem Kreis gekennzeichneten Meßwerte stammen aus der Messung mit dem JIT-Compiler. Die Werte vom JX-Translator sind durch ein Kreuz gekennzeichnet. Neben den Ergebnissen aus den ersten beiden Messungen sind noch Ergebnisse einer weiteren Messung eingetragen, die durch Dreiecke markiert wird. In dritten Messung wurden alle für den Speicherschutz benötigten Überprüfungen entfernt. Diese Messung markiert die Maximalwerte der für die auf dei Überprüfungen ausgerichteten Optimierungen. Bei diesen Optimierungen wäre also noch eine maximale Verbesserung von weniger als 20% möglich. Die erzielte Verbesserung liegt unabhängig von der Dateigröße immer weit über 100% gegenüber des nicht optimierenden JIT-Compilers.

4 Zusammenfassung und Ausblick

Durch den Einsatz des Profilers entdeckten wir eine fehlerhaft implementierte Hashtabelle, welche in der Pufferverwaltung des ext2-Dateisystems benutzt wurde. Auf diese Weise wurde der Durchsatz des ext2-Dateisystems von anfänglich nur 10 Mbyte/s auf etwa 100 Mbyte/s verbessert.

Die Implementierung eines Profilers für JX hat damit wesentlich dazu beigetragen, daß das Betriebsystem bei Dateizugriffen mit einem Spitzendurchsatz von mehr als 200 MByte/s schon 50% des Durchsatzes von Linux mit 400 MByte/s erreicht. Beide Messungen fanden auf dem gleichen System, einem PC mit Pentium III 500 MHz, statt[GKB01].

Der Pentium Prozessor bietet neben dem vom Profiler genutzten Taktzähler noch zwei weitere Register zum Auswerten von Programmläufen. In ihnen kann zum Beispiel das Auftreten von falsch vorhergesagten Sprüngen gezählt werden. Mit einem relativ geringen Aufwand könnte der Profiler so erweitert werden, daß auch diese Zähler ausgewertet werden können, um so weitere aufschlußreichere Hinweise zur Optimierung von Programmen zu finden.

Der Java-Bytecode-zu-Maschinencode-Übersetzer implementiert erst ein kleines Spektrum von möglichen Optimierungstechniken. Auch ist noch nicht der gesamte Funktionsumfang von Java-Bytecodebefehlen implementiert. Es gibt an dieser Stelle folglich noch einen großen Entwicklungsbedarf für den Übersetzer. Eine Vergleichsmessung des Übersetzers zu anderen optimierenden Compilern konnte aus Zeitgründen leider nicht mehr erfolgen. Eine solche Messung könnte im Vergleich zum Impact-NET[HCJ+97] oder zu anderen optimierenden Übersetzern das noch zu erwartende Verbesserungspotenzial aufzeigen.

Die bis jetzt durchgeführten Messungen zeigen zwar eine erfolgreiche Verbesserung des erzeugten Maschinenprogramms, sie können aber noch keine Aussage darüber geben, ob mit nur ausreichend guter Optimierung ein System mit softwarebasiertem Speicherschutz mit einem System mit hardwarebasiertem Speicherschutz konkurrieren kann. Da ein Entfernen aller Laufzeitüberprüfungen nur noch eine Verbesserung von 20% bringt, zeigt sich, daß der softwarebasierte Speicherschutz den noch vorhandenen Unterschied nicht alleine verschuldet.

Die Möglichkeit, ein System viel flexibler an die gestellten Herausforderungen anpassen zu können, läßt die Hoffnung zu, daß ein System mit softwarebasierten Speicherschutz wie JX sich am Ende gegenüber einem traditionellen System als das schnellere System erweist.

5 Literaturverzeichnis

ACL+98 Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, James M. Stichnoth: Fast, Effective Code Generation in a Just-In-Time Java Compiler. Intel Corporation, Santa Clara, CA 95052, 1998

ASU86 Alfred V.Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers - Principles, Techniques, and Tools. Addison-Wesley, 1986

CFM+97 Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, Mario Wolczko: Compiling Java Just in Time. IEEE, 1997

GE99 Ralf H. Güting, Martin Erwig: Übersetzerbau - Techniken, Werkzuege, Anwendungen. Springer-Verlag Berlin ISBN 3-540-65389-9, Heidelberg, 1999

GJP+00 A. Gefflaut, T. Jaeger, Y. Park, J. Liedtke, K. Elphinstone, V. Uhlig, J.E. Tidswell, L. Deller, and L. Reuther. The SawMill Multiserver Approach. In *Proc. of the 9th SIGOPS European Workshop*, Sep. 2000.

GKB01 Michael Golm, Jürgen Kleinöder, Frank Bellosa: Beyond Address Spaces - Performance, Protection and Resource Management in the Type-Safe JX Operation System. Technical Report TR-I4-01-04, Universität Erlangen, 2001

HC00 Nathan M. Hanish, William Cohen: Hardware Support for Profiling Java Programs. University of Alabama, Huntsvile, 2000

HCJ+97 Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C.Gyllenhaal, Wen-mei w. Hwu: Using NET to Capture Performance in Java-Based Software. Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign IL 61801, 1997.

Intel95 Intel: Pentium Processor Family Developres's Manual - Volume 3: Architecture and Programming Manual. Intel Corporation, USA P.O. Box 7641, 1995

Intel99 Intel: Intel Architecture Optimization (Reference Manual). Intel Corporation, USA P.O. Box 7641, 1998/1999

Kopp98 Hans Kopp: Design und Implementierung eines maschinenunabhängigen Just-in-Time-Compilers für Java. Diplomarbeit, Universität Erlangen, IMMD4, Oktober 1998

LY99 Tim Lindholm, Frank Yellin: The Java Virtual Machine Specification (2nd Edition). Addison-Wesley ISBN 0201432943, 1999

Weissel00 Andreas Weissel: Ein offenes Dateisystem mit Festplattensteuerung für metaXaOS. Studienarbeit, Universität Erlangen, IMMD4, Feb. 2000