JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen

Georg Acher

Lehrstuhl für Rechnertechnik und Rechnerorganisation

JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen

Georg Acher

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Hans Michael Gerndt

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Arndt Bode

2. Univ.-Prof. Dr. Eike Jessen, em.

Die Dissertation wurde am 18.6.2003 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 6.10.2003 angenommen.

JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen

Georg Acher

Kurzfassung

Durch die wachsende Leistungsfähigkeit von Prozessoren für eingebettete Systeme wird der Einsatz von Java für diesen Bereich immer vielversprechender. Allerdings sind die von normalen Desktop- oder Serversystemen bekannten Techniken zur Beschleunigung der Java Virtual Machine (JVM), wie z.B. die Just-in-Time-Compilation (JIT), bei den sehr begrenzten Ressourcen in dieser Geräteklasse nur schwer einsetzbar oder erfordern eine Bindung auf eine bestimmte CPU-Architektur.

Diese Arbeit stellt das JIFFY-Konzept vor, das den Ablauf und die Integration eines vollständigen JIT-Compilers für die JVM in einem frei programmierbaren Logikbaustein (FPGA, Field Programmable Gate Array) beschreibt. Durch den Einsatz des FPGAs wird eine sehr hohe Übersetzungsgeschwindigkeit erreicht, wobei die Qualität des erzeugten Compilats die von einfachen, softwarebasierten JIT-Compilern mindestens erreicht und oft übersteigt. Durch ein spezielles Konzept ist der gesamte Übersetzungsvorgang im FPGA und damit die synthetisierte Gatterlogik unabhängig von der Zielarchitektur und erlaubt eine flexible Nutzung auch für typische heterogene Mehrprozessorsysteme in Kommunikationsanwendungen.

Neben der Beschreibung des Übersetzungskonzepts werden auch die Auswirkungen des FPGA-basierten Ansatzes auf das Java-Laufzeitystem und mögliche Abwägungen in der Implementierung von Hardware und Software qualitativ und quantitativ an zwei exemplarischen Modellierungen für 80586- und AlphaCPU-basierte Systeme verglichen.

Abstract

The steadily growing performance of processors for embedded systems make the usage of the platform independent Java system more and more attractive. However, the usual techniques known for acceleration of the Java Virtual Machine, widely used on desktop computers, don't apply well in general to this class of devices, the most prominent example is the Just-in-Time-Compilation (JIT). This is caused by the tight resource constraints of these systems and their wide range of processor architectures.

This thesis presents the JIFFY concept, which describes the integration of a complete JIT-compiler for the JVM into an Field Programmable Gate Array (FPGA). By using the FPGA, a very high translation speed can be achieved, the code quality and thus execution speed reaches or exceeds simple, software based JITs. Based on a layered concept, the translation process in the FPGA und therefore the synthesized gate logic is independent of the target CPU architecture. This feature allows a very flexible usage of the FPGA even for heterogenous multiprocessor systems, typically found in modern communication applications.

Besides the detailed explanation of the translation process itself, the impacts of the FPGA-based approach on the Java runtime system and possible considerations in the implementation of the hardware and software are also described. To qualify and quantify the resulting properties for CISC and RISC CPUs, the system is modeled for two typical architectures (80586 and AlphaCPU).

Danksagung

Diese Dissertation ist nur durch die Unterstützung einiger Menschen durchführbar geworden, bei denen ich mich an dieser Stelle bedanken will.

Professor Dr. Arndt Bode ermöglichte mir in den vergangenen Jahren am Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR) der Technischen Universität München ein hervorragendes Arbeitsumfeld, und das sowohl in wissenschaftlicher, als auch in organisatorischer und kollegialer Hinsicht. Ebenso möchte ich Professor Dr. Eike Jessen für seine Bereitschaft als Zweitgutachter danken, er hat wertvolle Hinweise zur vorliegenden Arbeit gegeben.

Herzlicher Dank gebührt auch dem Leiter der Architekturgruppe am LRR, Wolfgang Karl, ohne dessen zahlreichen Vorschläge und Diskussionen die Arbeit mit diesem Thema wohl gar nicht entstanden wäre.

Von den zahlreichen hilfsbereiten Kollegen am LRR möchte ich besonders Rainer Buchty, Michael Eberl, Detlef Fliegl und Robert Lindhof erwähnen, die neben wissenschaftlicher oder administrativer Unterstützung auch in meiner unmittelbaren Umgebung für Spass an der Arbeit gesorgt haben.

Der TU München möchte ich meinen Dank aussprechen, da sie mir in den ersten beiden Jahren der Arbeit ein Promotionsstipendium gewährt hat.

Zum Schluss möchte ich mich bei meinen Eltern bedanken, die viel Verständnis für meine technischen Basteleien aufgebracht haben und mich auf meinem gesamten Ausbildungsweg immer unterstützt haben.

in a jiffy, [’ji-fE], engl.: “im Nu”

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1

Einführung und Motivation

Seit Beginn des Computerzeitalters steht ein Satz
für alle Forschungsbemühungen:

Es muss schneller werden.

Seit Beginn des Computerzeitalters steht ein Wort
für die Ergebnisse der Forschungsbemühungen:

Inkompatibilität.

1.1 Java und die Konsequenzen

Wurde am Anfang der Computerzeit noch mit Maschinencode und hardwarenah programmiert, setzte sich aufgrund von immer wieder veränderter Maschinenarchitektur bald die Erkenntnis durch, dass eine gewisse Abstraktion von der Maschine zwar die Ausführungsgeschwindigkeit verringert, dafür aber die Wiederbenutzbarkeit der mühsam geschriebenen Programme stark verbessert. Basierend darauf wurden Betriebssysteme und Hochsprachen entwickelt, die es dem Programmierer ermöglichten, den gewünschten Algorithmus einfacher und übersichtlicher zu beschreiben. Dabei war stets ein Kompromiss zwischen Kompatibilität und Geschwindigkeit zu finden, der meistens zu Gunsten der Geschwindigkeit entschieden wurde.

Als Beispiel sei die Programmiersprache “C” genannt, die seit Anfang der 70er Jahre des 20ten Jahrhunderts starke Verbreitung gefunden hat, obwohl C eher als guter Makroassembler bezeichnet werden kann. Selbst wenn UNIX als relativ einheitliches Betriebssystemkonzept gewählt wird, ist doch für jede neue Prozessor- oder Betriebssystemplattform mindestens eine Neukompilation nötig,

manchmal auch Quellcodeänderungen. Das Ergebnis ist aber eine im Durchschnitt sehr gute Rechnerleistung.

Als Alternative dazu existiert inzwischen eine Unzahl von Interpretersprachen, die auf unterschiedlichen Systemen unverändert ablaufen können. Gerade im Bereich des "Rapid Prototyping" sind sie sehr verbreitet, da Änderungen im Programmcode sehr schnell ausgetestet werden können. Auf der anderen Seite sind sie normalerweise auf ein spezielles Einsatzgebiet beschränkt (z.B. Perl für Stringverarbeitung) und (bis auf Forth [56]) auch für allgemeine Anwendungen nicht besonders performant.

Die Idee der Firma Sun bei der Konzeption von Java war es, eine stark an die bekannte C/C++-Struktur angelehnte Sprache mit einer Vielzahl von sehr flexibel einsetzbaren Bibliotheksfunktionen und einem virtuellem Ablaufsystem zu verknüpfen. Damit entsteht eine universelle Sprache, die auf beliebigen Systemen ablaufen kann, da der Java-Maschinencode zunächst von der Zielmaschine interpretiert werden muss.

Seit der "Erfindung" von Java hat sich das gesamte Java-System weit verbreitet. Zu Beginn wurde hauptsächlich das Umfeld des World Wide Web (WWW) als Einsatzbereich angesehen, inzwischen ist die Grenze zwischen "Web"-Anwendung und "normaler" Anwendung verwischt und Java wird zunehmend auch in anderen Einsatzgebieten benutzt, was nicht zuletzt auch an der Vielzahl der bereits existierenden und quasi-standardisierten Bibliotheken liegt. Trotzdem ist Java keine völlige Neuerfindung, sondern eher eine geschickte Kombination vieler, teilweise schon lang bekannter Techniken und Erkenntnisse.

Einer der Hauptvorteile von Java, die Plattformunabhängigkeit, ist zugleich auch einer der größten Nachteile. Ein Java-Programm läuft (im Prinzip) ohne Neukompilation auf jeder beliebigen Hard- und Softwarearchitektur. Allerdings wird durch die notwendige Nachbildung der sog. Java Virtual Machine (JVM), die wie bei jedem Computer als eine Prozessor-ähnliche Ausführungseinheit die Grundlage des Java-Systems bildet, ein nicht unbeträchtlicher Teil der verfügbaren Rechenleistung nicht für die eigentliche Ausführung des Programms benutzt. Die Emulation der JVM durch Interpretation ergibt nur 1-10% der theoretisch möglichen Ausführungsgeschwindigkeit eines z.B. in C programmierten äquivalenten Algorithmus.

Der dadurch entstehende Leistungsverlust ist auch heute noch zu spüren, allerdings konnte er durch zahlreiche Verbesserungen in der Ausführung der JVM, wie z.B. die Just-in-Time-Compilation, im Vergleich zur ersten interpretierten Java-Version stark verringert werden.

Ein Großteil dieser Verbesserungstechniken ist aber auch nur durch die zunehmende Leistungsfähigkeit der Hauptprozessoren und der zur Verfügung stehenden Menge an Hauptspeicher ermöglicht worden. Während zu Beginn der Java-Entwicklung die Taktfrequenz von Arbeitsplatz-PCs unter 100MHz und der

Hauptspeicher unter 32MByte lagen, sind heute Taktfrequenzen von 2 GHz, Speicher von über 1GByte und CPUs mit Leistungsaufnahmen von mehr als 60Watt keine Seltenheit mehr. Dies ermöglicht natürlich wesentlich umfangreichere Initialaufwendungen für eine mögliche Beschleunigung, die dann im Gesamtablauf nicht mehr so stark ins Gewicht fallen.

1.2 Einordnung der Arbeit

Trotz aller Euphorie über die inzwischen erreichbare Java-Geschwindigkeit wird in diesem Zusammenhang aber die immer wichtiger werdende Gruppe der eingebetteten Systeme (Embedded Systems) vergessen. Inzwischen verrichten diese, teils für den Benutzer völlig unsichtbar oder zumindest unauffällig und nicht "als Rechner erkennbar", in vielen Geräten ihren Dienst.

Diese stark ressourcenbeschränkte Systeme können bislang davon nur zum Teil von der beeindruckenden Leistungsentwicklung im PC-Bereich profitieren. Preisdruck und Stromverbrauch erlauben dabei nicht so auffällige (und wie im PC-Bereich auch teilweise unnötige) Leistungsverbesserungen für "Consumer"-Geräte.

Eines der zur Zeit am stärksten verbreitetesten Geräte in dieser Klasse, der PalmPilot von Palm Computing, arbeitet mit einem Motorola 68000-Derivat mit Taktfrequenzen zwischen 16 und 32MHz, einem Hauptspeicher von 8MByte und mit zwei Microbatterien. Mit diesen "harten" Anforderungen verlieren viele der JVM-Beschleunigungstechniken ihre Grundlage, und so bleibt im Endergebnis nur wieder die alte, langsame JVM-Interpretation zur Ausführung von Java-Programmen übrig.

Gerade in letzter Zeit rückt aber die Ausführung von Java auf diesen Systemen immer weiter in den Vordergrund, als weitere Beispiele seien hier nur Set-Top-Boxen für digitalen Rundfunk (z.B. DAB/DVB), Mobiltelephone oder (allgemeiner) mobile und portable Datenterminals genannt. In Zukunft wird auch die Möglichkeit eines universellen Updates dieser Geräte für neue Übertragungsprotokolle ebenfalls eine große Rolle spielen.

Die Notwendigkeit von Hardwarebeschleunigern, die bei der Umsetzung der JVM helfen, wurde natürlich auch von der Industrie erkannt. Es existieren inzwischen eine Reihe von "Java"-Coprozessoren, die für verschiedene Anwendungs- und Einsatzgebiete entwickelt wurden, einige als Prozessorersatz, andere als Hilfsmittel zur Übersetzung (sog. "Binary Translation"). Diese Systeme sind dabei meist relativ inflexibel und sind nicht an bestimmte, abweichende Anwendungsszenarien anpassbar.

Mit dem JIFFY-System wird nun in dieser Arbeit eine Möglichkeit beschrieben, eine starke Ausführungsbeschleunigung für diese Anwendungsgebiete zu er-

reichen, ohne z.B. wie bei anderen Systemen an eine bestimmte Zielarchitektur gebunden zu sein. Schwerpunkt ist in diesem Zusammenhang nicht der Bereich der Compilertechniken an sich, sondern die effiziente Umsetzung und Einbindung eines Übersetzungskonzepts in spezifische Hardware-Architekturen. Besonders interessant ist hier die Abwägung verschiedener Vorgehensweisen und deren Auswirkungen auf die Komplexitäten und Laufzeiten in Soft- und Hardware.

Das vorgestellte System JIFFY stützt sich dabei auf rekonfigurierbare Logikbausteine (FPGAs), die in einem beliebigen Prozessorsystem als externer JVM-Coprozessor arbeiten. Im Unterschied zu anderen Systemen ist JIFFY aber **keine** JVM-Emulation in Hardware, sondern ein Just-In-Time-Compiler (JIT) im FPGA. Dieser gänzlich andere Ansatz ermöglicht eine Geschwindigkeitssteigerung, der zukünftige Technologiefortschritte sowohl bei den Mikroprozessoren als auch bei FPGAs ausnutzen kann, und damit sehr gut skaliert. Zusätzlich bietet die Rekonfigurierbarkeit des FPGAs eine große Flexibilität bei der Einbindung des FPGAs und seiner Nutzung im Gesamtsystem. Dies erlaubt ein nicht mehr änderbares, kundenspezifisches IC (ASIC) nicht.

Ein grober Überblick über die Abläufe und die wesentlichen Bestandteile im JIFFY-System ist in in Abb. 1.1 gegeben. Prinzipiell basiert der Ansatz auf dem bekannten Prinzip der Übersetzung in eine Zwischensprache, allerdings wurden alle Abläufe und Schnittstellem auf die explizite Integration in programmierbarer Logik ausgelegt. Damit ergeben sich gegenüber Software-JIT-Compilern wesentlich andere Gesichtspunkte und Eigenschaften.

1.3 Forschungsbeitrag

Die vorliegende Arbeit liefert aufgrund der beschriebenen Merkmale des Übersetzungskonzepts folgende Beiträge in wichtigen Gebieten der “Binary-Translation” und der eingebetteten Systeme:

- Umfangreiche Zusammenstellung und Evaluierung von existierenden Java-Beschleunigerkonzepten

  Die wichtigsten akademischen und kommerziellen Konzepte zur Beschleunigung von Java werden besprochen und ihrem Ansatz nach eingeordnet und (soweit möglich) bewertet.

- Detailierte Untersuchungen zur Geschwindigkeit von einfachen, Hardware-basierten Übersetzungsmechanismen und ihre Auswirkungen auf die Integration in ein Laufzeitsystem.

  Das entwickelte Übersetzungssystem hat durch seine auf den FPGA-Einsatz festgelegte Struktur bestimmte Schwächen in der Codeerzeugung, bzw. er-

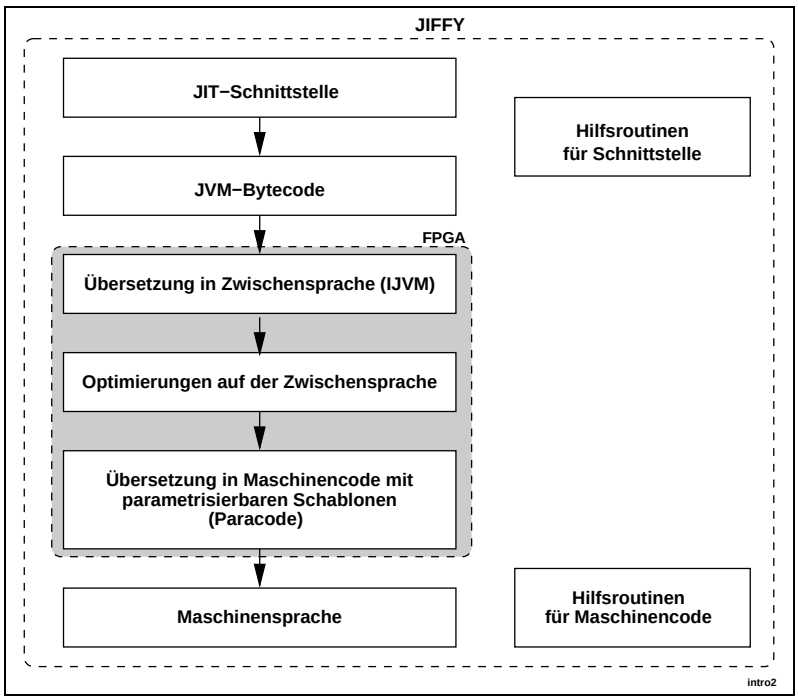

Abbildung 1.1: Grober Überblick über die JIFFY-Abläufe

zwingt bestimmte Vorgehensweisen im Laufzeitsystem. Anhand von konkreten Beispielen werden diese Punkte näher untersucht, bewertet und ihre Auswirkungen für das Gesamtsystem erläutert.

- Beschreibung einer universellen Optimierungs- und Codeerzeugungsstruktur, ausgerichtet auf die Eigenschaften von FPGAs.

  Die Grundlage des JIFFY-Systems bildet eine, speziell auf die Logik- und Verdrahtungsmöglichkeiten von FPGAs ausgerichtete Hardwarestruktur. Diese besitzt als “Backend” eine Art frei parametrisierbaren Assembler für nahezu alle CPU-Architekturen, der ebenfalls vollständig in Hardware abläuft. Dieser Hardware-Assembler arbeitet zwar auf Basis von Tabellen, erlaubt aber trotzdem eine sehr große Flexibilität in der Codeerzeugung, die weit über die Codeerzeugung mit normalen Tabellen hinausgeht. Dieser Teil des JIFFY-Systems ist nicht an die JVM gebunden und kann auch für andere Anwendungen im Bereich der “Binary Translation” als Hardwarebeschleuniger benutzt werden.

1.4 Aufbau der Arbeit

Obwohl JIFFY "nur" eine Beschleunigung von Java-Programmen als Ziel hat, berühren, wie in Abb. 1.2 angedeutet, Entwicklung und Implementierung mehrere Fachgebiete. Um die Herleitung des JIFFY-Konzepts und seiner Besonderheiten zu begründen, werden die Teilbereiche, die JIFFY beeinflussen, etwas genauer dargestellt.

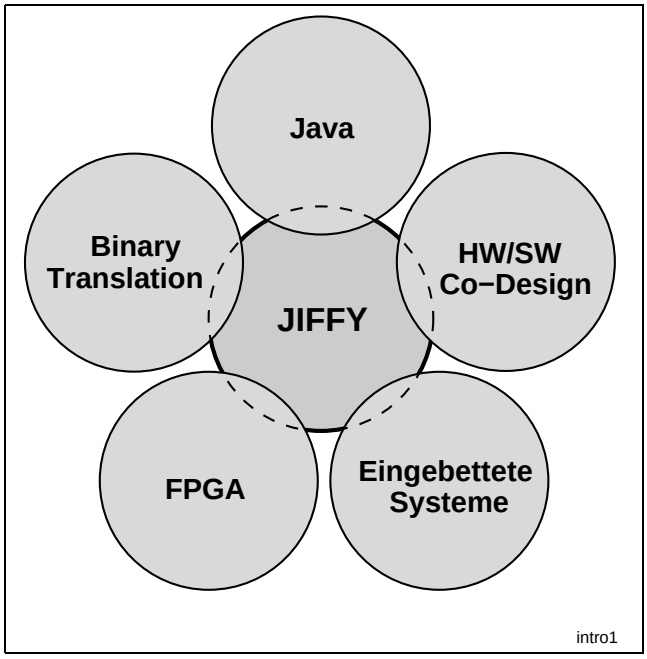

Abbildung 1.2: Einordnung von JIFFY

Der Aufbau dieser Arbeit ist wie folgt: Nach den Grundlagen von Stackcomputern, der "Binary Translation" und anderen als "plattformunabhängig" bezeichneten Systemen in Kapitel 2, wird in Kapitel 3 der Aufbau des Java-Systems von Sun erklärt. Dieses besteht aus vielen Komponenten, die manchmal auch einzeln als "Java" bezeichnet werden, was aber im strengen Sinne nicht korrekt ist. Die verschiedenen Ausführungs- und Beschleunigungskonzepte der Java Virtual Machine werden dargestellt und bezüglich ihrer Vor- und Nachteile genauer analysiert.

Kapitel 4 beschreibt das ins Auge gefasste Einsatzgebiet von JIFFY, also eingebettete Systeme und die Anforderungen in diesem Bereich. Als Beispiele werden die Prozessorarchitekturen einiger weit verbreiteter Systeme vorgestellt, um die Breite der Anforderungen an das Übersetzungssystem von JIFFY zu demonstrieren. Anschließend werden die Realisierungsgrundlagen von JIFFY, die sog. Field Programmable Gate Arrays (FPGAs) und darauf aufbauend, die rekonfi-

gurierbaren Systeme behandelt. Neben den Anwendungen der Rekonfiguration sind hier insbesondere die Unterschiede zu “normalen” oder kundenspezifischen Schaltkreisen und deren Einfluss auf den Entwurf von JIFFY von Interesse.

Kapitel 5 stellt einige existierende Java-Beschleuniger und JVM-Implementierungen in Soft- und Hardware vor, die bei der Auswahl des Übersetzungskonzepts betrachtet wurden.

In Kapitel 6 wird die dem JIFFY-System zugrunde liegende Architektur aus den sich in den Kapiteln 2 bis 5 ergebenden Konsequenzen entwickelt und deren Auswirkungen auf das Übersetzungskonzept dargelegt.

Um das Konzept in seiner Leistungsfähigkeit zu verifizieren, wurden sowohl ein Software-Modell als auch eine FPGA-Logik für JIFFY erstellt. Die dabei gewonnenen Implementierungsmöglichkeiten werden in Kapitel 7 beschrieben.

In Kapitel 8 werden die Messergebnisse mit anderen System zur Java-Ausführung verglichen und die Stärken und Schwächen des realen JIFFY-Systems anhand detaillierterer Einzelbetrachtungen analysiert.

Eine Zusammenfassung der Arbeit wird in Kapitel 9 gegeben, ein Ausblick auf weitere Forschungsaspekte und zukünftige Anwendungsgebiete folgt in Kapitel 10.

2

Stackmaschinen, Binary Translation und plattformunabhängige Systeme

> "When I use a word," Humpty Dumpty said, in rather a scornful tone,
> "it means just what I choose it to mean – neither more nor less."
> *Lewis Carroll, Through the Looking Glass*

Vor der Beschreibung des eigentlichen Java-Systems in Abschnitt 3 werden in diesem Kapitel wichtige Grundlagen und dazu verwandte Techniken beschrieben. Dies sind Kellermaschinen, die binäre Übersetzung und plattformunabhängige, nicht auf Java-basierende Systeme. Während die Kellermaschine das Java-Ausführungsmodell bestimmt, sind für eine effiziente Ausführung Prinzipien der binären Übersetzung notwendig.

2.1 Kellermaschinen

2.1.1 Klassifizierung

Der Keller bzw. Stack, ein selbstorganisierender Speichertyp mit LIFO-Prinzip (Last In, First Out) wurde am Anfang des Computerzeitalters zur einfacheren und schnelleren Abarbeitung von Hochsprachen benutzt, um z.B. Rekursionen und Zwischenspeicher für lokale Variablen zu ermöglichen. Die Stackfunktionalität kann zwar auch ohne Unterstützung in Hardware ermöglicht werden (wie es heute bei RISC-Prozessoren wieder üblich ist), effizienter wird es mit einer dedizierten Stackverwaltung auf dem Prozessor selbst, die z.B. Post-Inkrement bzw. Pre-Dekrement eines Adressregisters implementiert.

Der Stack kann für verschiedene Betriebsarten genutzt werden:

- Auswertung von Ausdrücken ("Expression Evaluation Stack")

In diesem Fall dient der Stack als Zwischenspeicher bei der Auswertung von Ausdrücken, die bei der Berechnung anfallen und z.B. aus Platzmangel nicht in einem Register gespeichert werden können. Durch die Selbstorganisation können mindestens soviele Zwischenergebnisse abgelegt und wieder zurückgeholt werden, wie Stackelemente vorhanden sind.

- Speicherung der Rückkehradresse (“Return Stack”)

  Bei vielen Problemen lässt sich die Implementierung durch das Einführen von Unterroutinen (“Subroutines”) und Rekursion stark vereinfachen. Dies bedingt aber die Speicherung der Rückkehradresse der aufrufenden Funktion. Da (bis auf Ausnahmen) die letzte gespeicherte Rückkehradresse auch die nächste benötigte ist, bietet sich hier der Stack zur automatischen Verwaltung der Adressen an,

- Speicher von lokalen Variablen (“Local Variable Stack”)

  Viele prozedurale Funktionen besitzen einen internen Status, der bis zum endgültigen Verlassen der Funktion speicherbar und veränderbar sein soll. Durch die Möglichkeit, dass diese Funktion direkt oder indirekt wieder aufgerufen werden kann, scheiden feste Speicheradressen für diese Variablen aus, ansonsten wäre die Funktion “non-reentrant”. Auf einem Stack werden für diese Anwendung temporäre Speicherplätze eingerichtet, auf die mithilfe einer speziellen Adressierungsart (basisrelativ) zugegriffen werden kann.

- Übergabe von Funktionsparametern (“Parameter Stack”)

  Ähnlich den lokalen Variablen erlaubt die Nutzung als Parameterstack das Reservieren prinzipiell beliebig vieler Speicherzellen, um darüber die Parameter an die aufgerufene Funktion weiterzugeben. Zwar ist diese Übergabeart auf Registerprozessoren langsamer als die über Register, andererseits ist sie von der Anzahl der Parameter unbeschränkt und erlaubt einen Zugriff auf Parameter ähnlich wie auf lokale Variablen.

Weitere Funktionen wie z.B. die Kontextspeicherung bei Unterbrechungen oder Taskwechseln sind mit der Speicherung von lokalen Variablen vergleichbar und somit ebenfalls mit dem Stack einfach möglich.

Zugriffe auf den Stack erfolgen üblicherweise über Befehle ohne Adressen, da die Stack-Hardware die Abbildung auf den Speicher selbst verwaltet. Als Befehle sind PUSH (Wert auf den Stack legen) und POP (obersten Wert vom Stack holen) üblich, dabei wird, wie in Bild 2.1 gezeigt, das aktuell als oberstes liegende Element TOS (Top Of Stack) und das darunter liegende NOS (Next Of Stack) benannt. Wird der Stack über Speicheradressen angesprochen, wird dazu ein sog.

Stackpointer (SP) benutzt, der je nach Implementierung auf TOS oder die erste, nicht benutzte Speicherzelle zeigt.

Bei Benutzung als Returnstack sind die Befehle CALL ("PUSH PC; JUMP Ziel") und RET ("POP PC") verbreitet, die implizit auf den Stack zugreifen. Zur Adressierung von lokalen Variablen oder Funktionsparametern muss eine Stackadresse zwischengespeichert werden (BP "Base Pointer" oder FP "Frame Pointer") und die Speicherzugriffe in der Funktion erfolgen relativ zu diesem Wert.

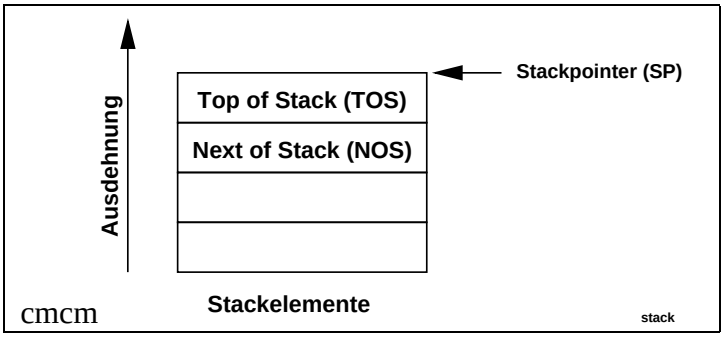

Abbildung 2.1: Der Stack

Auf den meisten Prozessoren muss der Stack obige Anwendungsmodi gemischt verwalten, üblichweise existiert nur ein gemeinsamer Stack für alle vier Varianten, da auch die speziellen Adressierungsarten (PUSH, POP, CALL, RET, basisrelativ) nur für das Stackregister vorhanden sind. Ausnahmen hiervon sind RISC-CPUs ohne spezielle Stackbefehle und die Motorola 68k-Serie, die neben dem Hauptstack für CALL/RET die Adressierungsarten PUSH/POP/basisrelativ für 7 weitere Registerzeiger erlaubt. Damit ist eine Trennung von Return- und Variablenstack einfacher möglich (z.B. in der 68k-Variante von gforth [29]) und kann auch von gemeinsamer Stacknutzung hervorgerufene Fehlersituationen besser vermeiden. Häufige Fehler sind dabei die sog. "Stack-Overflows", die üblicherweise zu gravierenden und schwer auffindbaren Fehlern führen, aber auch gefährliche Sicherheitslücken in sich bergen [156].

2.1.2 Explizite Stackarchitekturen

Stackmaschinen [86] besitzen im Gegensatz zu registerbasierten Architekturen keine expliziten Maschinenregister zur Verarbeitung von Daten, somit entfällt die Spezifizierung der nötigen Operandenregister im Befehl und man erhält eine 0-Operanden-Architektur[1].

[1] Befehle zum Laden von Literalen (Konstanten) bzw. Adressen, oder Sprünge fallen zwar nicht in diese Kategorie, sind der Effizienz halber aber immer nötig und bilden daher eine Ausnahme.

Alle Operationen beziehen sich damit auf TOS, NOS und evtl. tieferliegende Stackelemente. Der Vorteil dieser Adressierungsart liegt in der möglichen Verkürzung der Befehlslängen, üblicherweise reicht ein Byte aus. Von Nachteil sind teilweise komplizierte Befehlsfolgen beim Zugriff auf Datenstrukturen zur Berechnung von Adressen und Inhalt. Zusätzlich sollten Vorkehrungen getroffen sein, auf tieferliegende Stackelemente zuzugreifen.

Diese Nachteile sind mit der Ergänzung einiger 1-Operand-Befehle zu umgehen, die auf TOS und einem direkt adressierbaren Operanden arbeiten. Die Programme werden damit kürzer, allerdings wird die ausführende Hardware etwas verkompliziert.

2.1.3 Konkrete Stackarchitekturen und Sprachen

2.1.3.1 Forth

Die Programmiersprache Forth [56, 50] wurde um 1965 von Chuck Moore für Echtzeitanwendungen und Steuerungen entwickelt. Der Name soll andeuten, dass es sich hierbei um eine Sprache für die vierte Generation von Computersystemen handelt, die sich nach Moores Auffassung zu kleineren und verteilten Systemen entwickeln würden.

In Forth wird der Stack als Hauptarbeitsplatz benutzt, alle Befehle arbeiten implizit auf dem Stack. Für Prozeduraufrufe und die Verschachtelung von Schleifen steht ein zusätzlicher Returnstack zur Verfügung. Dabei ist sowohl die Interpretierung der Befehle als auch Compilation in nativem Maschinencode möglich. Die Interpretierung wird normalerweise für kleinere Systeme benutzt, da sie nur ein relativ kleines Laufzeitsystem erfordert. Üblicherweise sind Interpreter, Compiler, Laufzeitsystem und oft auch ein Betriebssystemkern integriert, d.h. auf der Zielplattform kann direkt entwickelt werden.

Die Sprache Forth ist, verglichen mit C oder Java, nicht besonders bekannt, wird aber aufgrund der Fähigkeit des “Rapid Prototyping” besonders bei der Entwicklung von Steuerungsaufgaben (z.B. bei der NASA [108]) eingesetzt.

Forth ist nur in Grundzügen standardisiert (ANSI Forth) und ist daher einer kontinuierlichen Weiterentwicklung unterworfen. Durch das modulare Konzept von Interpreter, Compiler und Laufzeitsystem und sind beliebige Spracherweiterungen möglich und auch zur Laufzeit einbindbar, wie z.B. Erweiterungen für objektorientierte Programmierung [134].

Eine Forth-Variante (OpenBoot [145]) wird z.B. als Systemmonitor im ROM von Sun eingesetzt.

2.1.3.2 PostScript

PostScript ([3]) ist eine von Adobe entwickelte Seitenbeschreibungssprache, die in vielen Druckern und auch zur Displaysteuerung (Display PostScript) eingesetzt wird.

PostScript ist sehr ähnlich zu Forth, allerdings bringt es bereits eine Reihe von speziellen Funktionen zur Erzeugung und Manipulation von Grafikdaten mit und besitzt keinen Compiler.

Als Stack stehen ebenfalls ein Operandenstack und ein Returnstack zur Verfügung. Aufgrund der Anwendung zur unmittelbaren Erzeugung von Grafikdaten in eingebetteten Systemen (z.B. Drucker) werden zur Ausführung üblicherweise nur Interpreter benutzt.

2.1.3.3 INMOS-Transputer

Anfang 1980 entwickelte die Firma INMOS mit dem sog. “Transputer” [74] einen Prozessortyp, der für die damalige Zeit sehr schnell war. Die Transputer basieren hauptsächlich auf einer vereinfachten Stackarchitektur. Der Hardwarestack kann nur drei 32Bit Werte beinhalten, mehr Daten sind nur durch explizites Stackspilling speicherbar. Die Befehle sind dabei CISC-ähnlich aufgebaut, unterschiedlich lang und enthalten verschiedene Präfixe zur Befehlserweiterung und Konkatenation von Konstanten.

Durch integrierte serielle Kommunikationskanäle ist es möglich, mehrere Transputer zu einem Cluster zu verbinden (Multiprozessorunterstüzung). Aus heutiger Sicht ist der Transputer durch die Vielzahl der bereits integrierten Peripherie (Speicheransteuerung, Timer, Eventcontroller) in die Klasse der eingebetteten Controller zu zählen.

2.2 Grundlagen der binären Übersetzung

Als “Binäre Übersetzung” bzw. “Binary Translation” wird die wie auch immer geartete Übersetzung der Befehle (des Binärcodes) einer Prozessorarchitektur (Instruction Set Architecture, ISA) in Befehle einer anderen ISA (Zielarchitektur) bezeichnet [12]. Üblicherweise wird diese Übersetzung unternommen, um mit nur einem Prozessortyp Programme anderer, evtl. nicht mehr existenter Systeme auszuführen.

Im technischen Sprachgebrauch wird die Maschinensprache der Zielarchitektur of auch als native Sprache bezeichnet, also die Sprache, die das System ohne weitere Übersetzungen direkt versteht.

Übersetzungen in die native Sprache könne dabei prinzipiell in drei Grundformen vorkommen:

- Emulation
- Statische Übersetzung/Static Translation
- Dynamische Übersetzung/Dynamic Translation

2.2.1 Emulation

Emulation ist die Interpretierung der Befehle, d.h. ein zu übersetzender Befehl wird in Software geholt, dekodiert und die damit verbundenen Aktionen aufgerufen. Damit wird also der Befehlsholzyklus einer CPU mit Software (tw. auch Mikroprogrammierung) nachgebildet, dies benötigt allerdings relativ wenig Programmcode. Der durch den Emulations-Aufwand dieser "virtuellen Maschine" (VM) vorhandene Leistungsverlust ist dabei aber erheblich. Zum Beispiel wurde für die stackorientierte Sprache Forth, die über relativ einfache Primitive ähnlich einer normalen CPU verfügt, in [52] eine Häufigkeit der Befehle `call` und `return` zwischen 26% und 34% ermittelt.

Damit sind die Einsatzgebiete der reinen Emulation relativ klar abgegrenzt: Vorteile ergeben sich nur dann, wenn die Emulation häufig wechselnden Code ausführt, der Optimierungaufwand also zu stark wird, oder wenn der verfügbare Speicherplatz sehr beschränkt ist.

2.2.2 Static Translation

Static Translation ist die Übersetzung der Binärcodes **vor** der Ausführung (Ahead-of-Time, AOT). Dies ist ähnlich der Methode für die Übersetzung der Programmiersprachen C und Fortran. Sie erlaubt umfangreiche und globale Optimierungen, da alle Informationen zur Verfügung stehen und diese Compilation nicht in bestimmter Zeit abgeschlossen sein muss. Durch diese Optimierungen ist Geschwindigkeit zur Laufzeit sehr gut.

Nachteil der statischen Übersetzung ist der große Zeitbedarf und die fehlende Adaptationmöglichkeit auf das ausführende System. Sowohl die Zielarchitektur als auch alle während der vollen Laufzeit benötigten Codemodule müssen zur Zeit der Compilation bekannt sein, selbstmodifizierender Code ist ebenfalls möglich.

Dadurch ist die statische Übersetzung nicht oder nur schlecht für das dynamische Nachladen von Codeteilen oder für heterogene Ausführungsplattformen geeignet.

2.2.3 Dynamic Translation

Dynamic Translation oder auch Just-In-Time-Compilation benötigt die Übersetzeralgorithmen zur Laufzeit und übersetzt Funktionen erst, wenn sie aufgerufen werden.

Durch die Integration von Compiler und Laufzeitsystem ist die dynamische Übersetzung relativ aufwendig, die nur schwer möglichen globalen Optimierungen erlauben keine in der Laufzeit so effiziente Compilierung wie bei der statischen Übersetzung. Zusätzlich geht die Zeit für die Compilation in die Ausführungsgeschwindigkeit mit ein, daher führen umfangreiche Optimierungen im Normalfall eher zu einer Verringerung der Ausführungsleistung.

Die Schwierigkeit der dynamischen Übersetzung besteht also darin, passende Übersetzungsalgorithmen zu finden, die relativ schnell relativ guten Code erzeugen. Damit fallen einige gebräuchliche Optimierungstechniken aus, da sie ein nicht vorhersagbares Zeitverhalten haben. Weiterhin sind Cache-Konzepte für mehrfach aufgerufene Funktionen notwendig.

2.3 Existierende "Binary Translation" Systeme

Dieser Abschnitt beschreibt einige bekannte "nicht-Java"-Systeme, die Binary Translation zur Ausführung einer anderen Systemarchitektur bzw. zur Optimierung des laufenden Codes benutzen.

2.3.1 Forschungssysteme

2.3.1.1 RePLay und I-COP

rePLay [121] ist ein Hardware-orientiertes Konzept zur dynamischen Optimierung von Programmcode. Während der Laufzeit des Programms sammelt eine auf der CPU angebrachte Einheit häufig benutzte Start- und Endpunkte im Maschinencode. Diese Laufzeitinformationen werden zu einer Optimierung des Programmablaufs innerhalb dieser Blöcke herangezogen. Die dabei entstehenden, zusammenhängenden ("Single-Entry/Single-Exit") Codeblöcke werden anschließend in einem sog. Frame-Cache gespeichert, um schnell wiederverwendet werden zu können. Die Entscheidung, welcher Ablaufplan ausgeführt wird, wird zwar spekulativ gefällt, basiert aber auf dem bisherigen Programmlauf mit einer einstellbaren "Vorgeschichte". Damit sind Trefferraten von 61 bis 90% erreichbar.

Die bislang verfügbaren Informationen betreffen nur die Effizienz der Erzeugung dieser Codeblöcke, für die dann anwendbaren Optimierungen und deren Auswirkungen auf den Programmablauf konnten noch keine Ergebnisse gefunden werden.

Das Konzept des Instruction Path Coprocessor (I-COP) [38] nutzt einen im VLIW-Prozessor vorhandenen Co-Prozessor zur Vorverarbeitung der Befehle. Damit wird eine dynamische Code-Modifikation möglich, also auch eine Übersetzung der Befehle und einfache Optimierungen darauf.

2.3.1.2 Dynamo/HP

Dynamo [26, 27] ist ein Optimierungssystem, das PA-8000-Binärcode auf einer PA-8000-Architektur interpretiert. Dabei wird durch die Interpretation ein Ablaufschema (Trace) erstellt, ohne den Code zu instrumentieren. Dieser Ablauf wird analysiert und die häufigsten Verzweigungen (bedingte Sprünge, Unterprogramm) als Ausgangspunkt für eine Linearisierung des Ablaufs verwendet. Damit werden zum einen evtl. Verluste bei indirekten oder bedingten Sprüngen (PA-8000-spezifisch) und Unterprogrammaufrufen verringert, andererseits erlaubt der lineare Ablauf ohne Verzweigungen wiederum Optimierungen, die im Originalcode nicht erlaubt wären.

Basis der Optimierung ist die Selektion von Codefragmenten im Programmlauf während der Interpretation, die in einem Fragmentcache gesammelt werden, anschließend nahtlos verbunden werden und in der Ausführung den Originalcode ersetzen.

Gegenüber der normalen Ausführung fällt zunächst der Interpreter-Overhead an, mit zunehmender Laufzeit kann der Geschwindigkeitsgewinn aufgrund fehlender Pipeline-Stalls durch falsche Sprungvorhersage bis zu 20% erreichen.

2.3.1.3 Sun Walkabout

Walkabout [39] ist ein Framework zur dynamischen Übersetzung verschiedender Architekturen. Das System besteht aus einem Interpreter, der zur Erkennung oft benutzter Codeteile (“Hot Path”) instrumentiert ist. Aufbauend auf diesen Ergebnissen wird ein optimierter Code generiert, wobei Verzweigungen eliminiert und ähnlich Dynamo die Fragmente verbunden werden. Damit wird der Zusatzaufwand des Interpreters miniminiert. Da allerdings außer der Vermeidung von Sprüngen und “gefädeltem Code” (noch) keine weiteren Optimierungen vorgenommen werden, liegt die erreichte Geschwindigkeit zwischen der des reinen Interpreters und der des nativen Codes.

2.3.1.4 IBM Daisy

DAISY [47] ist ebenfalls wie das später besprochene CodeMorphing-Konzept von Transmeta der Versuch, eine VLIW-Architektur als Basis zur Ausführung beliebiger anderer ISAs einzusetzen. Als Basisarchitektur wurde eine mit dem PowerPC

vergleichbare CPU verwendet (RS/6000). Die VLIW-Zielarchitektur wurde in verschiedenen Parallelitätsgraden simuliert.

Neben einem effizienten Code-Scheduling bestehen hier die Schwierigkeiten darin, die Eigenschaften der virtuellen Speicherverwaltung und einiger "restart"-barer CISC-Befehle während und nach einer Ausnahmebehandlung zu implementieren, ohne die Gesamtleistung zu beeinträchtigen.

Das Ergebnis der Übersetzung ist sehr effizient, einige Benchmarks waren sogar schneller als das Ergebnis eines direkten VLIW-Compilers. Durchschnittlich wurden ca. 4000 RS/6000-Befehle für die Übersetzung eines Befehls verwendet. Im Vergleich dazu wird *gcc* mit über 60000 Befehlen pro Ausgabebefehl angegeben, damit ist auch die Übersetzung selbst mit geringen Laufzeitkosten verbunden. Als erreichbare Parallelität auf VLIW-Befehlsebene (ILP, Instruction Level Parallelism) wurden Werte zwischen 3.0 und 6.5 ermittelt.

2.3.2 Kommerziell eingesetzte Systeme

2.3.2.1 FX!32 von Digital

FX!32 von Digital [70, 40, 46] wurde entwickelt, um auf Systemen mit dem Alpha-Prozessor unter nativem Windows NT4.0Alpha auch Programme ablaufen zu lassen, die für die NT4.0 auf Intel x86-Systemen compiliert wurden. Damit sollte die Verbreitung von Alphasystemen erleichtert werden[2].

FX!32 kann nur Anwenderprogramme aber keine Systemtreiber übersetzen, dies schränkt die Praktikabilität wieder ein.

Die Übersetzung läuft zunächst in einem Interpreter ab. Dabei wird ein Ablaufprofil erstellt, das am Programmende abgespeichert wird.

Während der Emulation werden einige Techniken zur beschleunigten Abarbeitung benutzt, unter anderem eine "lazy condition code"-Auswertung, welche die bei fast jedem x86-Befehl nötige Änderung der Statusregister erst bei einer Abfrage der Flags gezielt ausführt.

Ein timergesteuerter Daemon liest das bei der Emulation erzeugte Ausführungsprofil ein und erzeugt parallel zum x86-Programm ein optimiertes Alpha-Programm. Die Optimierung verbessert sich bei häufigen Programmstarts noch etwas. Der übersetzte Code läuft bis zu zehnmal schneller ab als die vorherige Interpretierung.

Alle diese Vorgänge sind für den Benutzer transparent, eine Unterscheidung zwischen Interpretation und übersetztem Code ist (außer anhand der Geschwindigkeit) nicht möglich.

Insgesamt kann mit einer Leistung von 30-60% gegenüber einem gleich schnell getakteten Pentium II gerechnet werden. Allerdings ist die Schwankungs-

[2] Für 16Bit x86-Programme wurde von Microsoft bereits ein Emulator mitgeliefert.

breite recht groß, da einige Operationen auf dem Alpha wesentlich schneller (FPU), andere wesentlich langsamer (Bytezugriffe) ablaufen.

Nach [70] bestand eine der Hauptschwierigkeiten bei der Entwicklung von FX!32 darin, dass dies ohne Quellcodemodifikation der WindowsNT-Umgebung geschehen sollte und große Teile der nötigen Informationen dazu von Microsoft nicht dokumentiert waren. Damit ist FX!32 stark von der jeweiligen Betriebssystemversion abhängig.

Eigene Tests ergaben eine nach mehreren Optimierungsläufen akzeptable Rechenleistung, allerdings war die Stabilität bzw. Windowskompatibilität von FX!32 nicht perfekt. Einige Anwendungsprogramme stürzten entweder ab oder erzeugten reproduzierbare Fehler.

2.3.2.2 Dynamic Recompiling Emulator – Der 68k→PPC-Konverter von Apple

Der Dynamic Recompiling Emulator (DR Emulator) von Apple [15] wurde bei der Einführung der PowerPC-basierten Macintosh-Systeme benötigt, da zu diesem Zeitpunkt noch kaum Anwendungen im nativen PPC-Format vorlagen.

Der DR Emulator ist ein Just-In-Time-Compiler, der beim Einsprung in eine noch nicht übersetzte Methode diese zuerst übersetzt, den erzeugten nativen Code abspeichert und dann ausführt. Es bildet sich also eine parallele Ansammlung von nativen PowerPC-Code für das Programm. Anders als FX!32 erfolgt aber keine weitere Optimierung dieser Routinen bei späteren Läufen.

Im Gegensatz zu FX!32 kann der DR Emulator auch alle Befehle und Abläufe im Supervisormode der 68040-Programme emulieren, unter anderem auch Interrupts. Damit ist es möglich, auch “Legacy”-Kerneltreiber auf dem PowerPC-System zu benutzen. Diese Besonderheit wurde z.B. beim SCSI-Treiber ausgenutzt, deshalb war dieser in den ersten PPC-Systemen nicht besonders schnell.

2.3.2.3 80586-kompatible Desktopprozessoren

Alle leistungsfähigen, zur 80586-Architektur kompatiblen CPUs (AMD K5, K6, Athlon, Intel Pentium2/3/4) arbeiten mit einem internen, superskalaren RISC-Prozessor. Dieser wird über eine interne Übersetzungseinheit (hart codiert oder über Tabellen) mit aus den x86-Befehlen erzeugten RISC-Befehlen (bei AMD ROP, bei Intel μOP genannt) versorgt [95]. Durch diese Umsetzung auf Grundbefehle wie Load, Store, ALU oder Branch können die x86-Befehle effizienter auf die einzelnen Einheiten verteilt werden und damit die Parallelität besser ausnützen. Komplexere Operationen werden über ein Mikroprogramm ausgeführt.

Diese Übersetzung in ROPs/μOPs wird “just-in-time” im engeren Sinne des Wortes ausgeführt, d.h. sie ersetzt die normale Dekodierphase und ist deshalb re-

lativ zeitkritisch. Aus diesem Grund können bereits dekodierte Mikrobefehle im L1-Cache zwischengespeichert werden. Außer der Bestimmung der Befehlsabhängigkeiten und dem Scheduling finden aber keine weiteren Optimierungen auf Mikroprogrammebene statt, da diese ebenfalls zuviel Zeit bzw. eine zu hohe Pipelinelatenz erzeugen würden.

2.3.2.4 Der Crusoe-Prozessor von Transmeta

Der Crusoe-Prozessor von Transmeta [84] ist ein für den Anwender x86-kompatibler Mikroprozessor, der intern allerdings auf einer völlig anderen Architektur beruht, um für den Notebookeinsatz besonders stromsparend zu sein. Die Übersetzung des x86-Codes erfolgt mit sog. “Code-Morphing”, das von der Hardware **und** Software unterstützt wird.

Dazu wird ein Systemspeicherbereich reserviert, der als Zwischenspeicher für die übersetzten Routinen dient. Eine für den Anwender nicht sichtbare Firmware übersetzt dynamisch x86-Code in das interne VLIW-Maschinenformat. Als Basisarchitektur steht ein VLIW-System mit zwei Integereinheiten, einer FPU, einer Load/Store-Einheit und einer Branch-Unit zur Verfügung, die ein 128Bit großes Instruktionswort mit 4 parallelen Befehlen besitzt.

Die Übersetzung selbst erfolgt in Software, die sich dabei zur Leistungssteigerung des compilierten Codes auf bestimmte Funktionseinheiten in der Hardware stützen kann. Dies sind z.B. eine Einheit zur Erkennung von Aliasoperationen bei Out-Of-Order-Ausführung, und eine MMU-Erweiterung zur Erkennung von selbst modifizierendem Code.

Messungen wie in [154, 113] bescheinigen dem Transmeta-Konzept allerdings nur unterdurchschnittliche Leistungswerte. Die beabsichtigte Stromeinsparung ist zwar vorhanden, wirkt sich aber im Einsatz mit anderen Notebook-Komponenten (Hintergrundbeleuchtung, Festplatte etc.) nicht so signifikant aus, wie erwartet.

2.4 Weitere plattformunabhängige Systeme

Als plattformunabhängige Systeme werden im folgenden Sprachen und Laufzeitumgebungen bezeichnet, die bei der Ausführung nicht auf bestimmte Prozessoren und Betriebssysteme (Plattformen) festgelegt sind.

2.4.1 vmgen

Eine allgemeine Methode zur Ausführung virtueller Maschinen und der automatischen Erzeugung der Laufzeitsysteme ist mit `vmgen` in [51] beschrieben.

`vmgen` erzeugt aus einer Spezifikation der VM-Befehle C-Code zur Ausführung/Interpretierung der Instruktionen, zur Erzeugung der Laufzeitumgebung und zum Debugging bzw. Profiling. Für stack-basierte VMs sind spezielle Vorkehrungen zur Beschleunigung der Interpretation integriert, wie z.B. die Speicherung von TOS in einem Register oder die Nutzung einer Peepholeoptimierung zur Zusammenfassung von mehreren Instruktionen zu einer Superinstruktion.

2.4.2 C#

Nachdem Microsoft gescheitert ist, durch Windows-spezifische Modifikationen Java zu erweitern, wurde die Entwicklung eines eigenen Systems mit ähnlichen Eigenschaften forciert. Dieses System mit dem Namen C# ("C-Sharp") stellt die Basis für das neue .NET-Konzept dar.

C# als Programmiersprache [98, 99] ist wie Java sehr eng mit C/C++ verwandt und besitzt unter anderem folgende Eigenschaften:

- Garbage Collection
- Typensichere Programmierung
- Automatische Initialisierung von Variablen
- Versionsverwaltung
- Pointer und native-API-Zugriffe nur in speziellen "unsafe" Regionen erlaubt
- Mehr Basistypen als Java
- Unterstützung von Ereignissen ("Events")

Wie auch Java basiert das Ausführungsystem von C# auf einer virtuellen Machine, der MSIL (Microsoft Intermediate Language). Als Ausführungsmodell wird nur die JIT-Compilation benutzt, Interpretierung oder ein gemischter Modus sind nicht vorgesehen. Zur Zeit arbeitet die virtuelle Maschine noch ohne adaptive Optimierungen, die auf Laufzeitinformationen basieren (Inlining etc.).

2.4.3 Weitere plattformunabhängige Sprachen

Im Bereich der interpretierbaren Sprachen gibt es eine große Auswahl an verschiedenen Konzepten.

Die Sprache BASIC (Beginner's All Purpose Symbolic Instruction Code) wurde 1964 mit der Absicht entwickelt, eine leicht erlernbare Sprache für Ausbildungszwecke zu erhalten. Obwohl bereits 1978 ein ANSI-Standard für ein minimales BASIC entstand, waren die Sprachvarianten teilweise sehr unterschiedlich, da sie üblicherweise auch spezielle Eigenschaften der jeweiligen Computerplattform ausnutzten. In seiner Grundform hat BASIC keine Records, keine Konstrukte für abstraktere Programmierung (wie Prozedurnamen statt Zeilenummern) und auch keine Objektorientierung. Diese Eigenschaften wurden ab 1983 auf verschiedene Weise implementiert, wie z.B. in Super-BASIC (Sinclair QL), GFA-BASIC (Atari ST) oder noch später in Visual BASIC (Microsoft). Die Ausführungsart war ursprünglich die Interpretierung eines teilweise tokenisierten Programms, später kamen Precompiler hinzu, die die Ausführungsgeschwindigkeit stark erhöht haben.

Skriptsprachen sind ursprünglich für die Ausführung von Kontrollabläufen und kleineren Programmen entwickelt worden. Hauptmerkmal dieser Skriptsprachen sind umfangreiche Methoden zur Stringverarbeitung, wie beispielsweise reguläre Ausdrücke. In [126] wird anhand einer Untersuchungsreihe gezeigt, dass für Anwendungen mit vielen Stringoperationen Skriptsprachen eine wesentlich schnellere Programmierung des Codes erlauben. Dies wird mit nur unwesentlich höheren Laufzeiten bzw. Speicheranforderungen gegenüber normalen Compilersprachen erkauft.

Sehr verbreitet sind in diesem Bereich Shellskript-Sprachen, die eine Erweiterung der Kommandozeilenfunktion sind. Die Sprachen der verschiedenen Shells (z.B. bash oder csh) sind sehr ähnlich und unterscheiden sich hauptsächlich in der Syntax. Die Ausführungsbasis ist ausschließlich Interpretierung.

Perl [118] ist eine Skriptsprache speziell für Stringverarbeitung und kann sowohl interpretiert als auch precompiliert ablaufen.

Die Programmiersprache Pascal [114] wurde 1970 von N. Wirth entwickelt. Das Compilersystem bot zur Vereinfachung der Portierung auf verschiedene Architekturen die Möglichkeit, den Quelltext für eine virtuelle Maschine (P-code) zu übersetzen. Diese war ähnlich der Java Virtual Machine stackorientiert und konnte auf beliebigen Zielsystemen mit einem P-code-Interpreter ausgeführt werden.

2.5 Zusammenfassung

Stackmaschinen besitzen als Arbeitsgrundlage keine individuell adressierbaren Register, sondern einen oder mehrere Stacks. Die Vorteile bestehen hauptsächlich im wesentlich kompakteren Befehlsformat und in einer einfachen Umsetzung einer Hochsprache auf dieses Format. Rein stackbasierte CPUs sind nur noch wenig

verbreitet, stackbasierte Sprachen werden in speziellen Einsatzgebieten benutzt, wo es auf Interaktivität (“Rapid Prototyping”) und Portabilität ankommt.

Binary Translation beschreibt allgemein die möglichst transparente Umsetzung des Maschinenprogramms einer Prozessorarchitektur auf eine andere Zielarchitektur. Benutzt wird dies hauptsächlich zur Wahrung der Kompatibilität und Weiterverwendbarkeit von Programmen oder als Grundlage zur weiteren impliziten Beschleunigung einer Prozessorarchitektur. Ein Hauptbestandteil der Ausführung ist dabei eine virtuelle Maschine (VM), die für die Nachbildung der Systemarchitektur auf dem Zielsystem zuständig ist. Je nach Anforderungen an die Geschwindigkeit kann diese VM auf Interpretierung oder verschiedenen Arten der Compilierung beruhen.

3

Java und Java Virtual Machine

Kipp die Tradition!
Nescafé XPress

3.1 Hintergründe zur Java-Entwicklung

Das Java-Konzept wurde Anfang der 90er Jahre zunächst als eine Plattform (Projektname “Green”) für Consumergeräte entwickelt, die Programmiersprache war zuerst unter dem Namen “Oak” bekannt [55][61]. Mit dem Erscheinen des World Wide Web (WWW) und der universellen Layout”sprache”[1] HTML wurde bei Sun auch die Wichtigkeit einer plattformunabhängigen Programmiersprache erkannt und die Entwicklung von Java auf den Einsatz im WWW ausgerichtet. Mit der Java-Unterstützung durch den damals weit verbreiteten Webbrowser Netscape Navigator konnte diese Technologie sehr schnell Verbreitung finden.

Trotz einiger Geschwindigkeitsprobleme trat Java einen Siegeszug wie bei kein anderes System durch die gesamte Computerindustrie an und ermöglichte durch die Plattformunabhängigkeit eine Reihe von neuen Softwarekonzepten. Obwohl diese Plattformunabhängigkeit das erklärte Ziel von Java war und ist (“Write once, run everywhere”), wurden aber nach der Einführung auch die Probleme dieser Philosophie deutlich:

- Nachdem Microsoft Java zunächst ablehnend gegenüberstand, erwarb die Firma bald eine Javalizenz und entwickelte eine sehr schnelle Laufzeitumgebung für Java[2]. Zusätzlich erweiterte Microsoft mit ihrem Java Development Kit allerdings die Javaspezifikation um Elemente, die nur unter Windows lauffähig waren. Nach diversen Gerichtsverfahren unterstützt Microsoft Java nicht mehr direkt, sondern hat einen konkurrierendes Konzept (C#, siehe 2.4.2) entwickelt.

[1] HTML ist keine turingmächtige Sprache
[2] Die MS-JVM waranfangs bei einigen Optimierungen allerdings etwas zu aggressiv[112]

- Für freie Betriebssysteme (z.B. Linux oder FreeBSD) ist zwar von Anfang an eine Java-Unterstützung vorhanden gewesen, allerdings kann diese in den Bereichen Kompatibilität, Stabilität und Geschwindigkeit immer noch ganz nicht mit den “kommerziellen” Entwicklungen mithalten. Erst die Freigabe des Sourcecodes des Java-Laufzeitsystems (JRE) von Sun hat diese Situation deutlich verändert.

Im folgenden wird nun das gesamte Java-Konzept vorgestellt, beginnend mit einem Überblick über das System, gefolgt von genaueren Beschreibungen der Programmiersprache Java, der Bibliotheksklassen und der Grundlage von Java, der Java Virtual Machine (JVM). Anschließend werden die verschiedenen möglichen Ausführungsarten und konkrete Implementierungen in Software und Hardware mit ihren Vor- und Nachteilen erläutert.

3.2 Beschreibung von Java

3.2.1 Übersicht

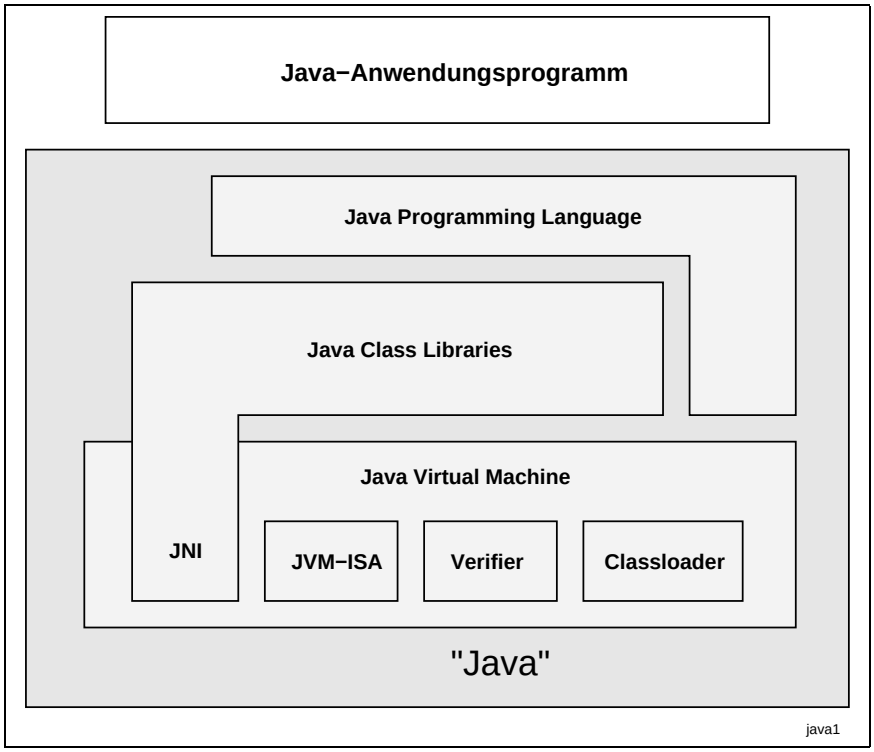

Abbildung 3.1: Java Systemsicht

Wie in Abb. 3.1 zu erkennen ist, beruht das Java-Konzept auf drei Grundbestandteilen, die, obwohl völlig unterschiedlich, im normalen Sprachgebrauch gerne vermischt bzw. verwechselt werden.

1. Die Programmiersprache Java

   Dies ist eine an C/C++ eng angelehnte, objektorientierte Programmiersprache. Beim Sprachentwurf wurde dabei geachtet, dass die C/C++-üblichen Fehlerquellen (wie z.B. Pointernutzung), erschwerte Lesbarkeit (z.B. durch Präprozessoranweisungen) und schwierig kompilierbare Konstrukte (z.B. Templates) nicht mehr möglich sind.

2. Java-Klassenbibliotheken

   Die Java-Klassenbibliotheken beinhalten wesentlich mehr als die in den C/C++ Standardbibliotheken "üblichen" Funktionen zur Ein-/Ausgabe und für mathematischen Routinen. Es gibt z.B. Methoden für HTTP-Zugriffe, Listenverwaltung, grafische Benutzerschnittstellen (GUI-Toolkits) und vieles mehr.

3. Java Virtual Machine (JVM)

   Die JVM ist die Grundlage jedes Java-Systems, sie bildet einen abstrakten Prozessor (JVM Instruction Set Architecture, JVM-ISA) nach, der mit definierten JVM-Befehlen auf einem realen Prozessor emuliert wird. Dabei wird sehr viel Komplexität auf die Emulation der virtuellen Maschine verschoben, der JVM-Maschinencode (auch Bytecode genannt) ist z.B. für Funktionsaufrufe und Speicherverwaltung abstrakt gehalten. In der JVM sind unter anderem auch Elemente enthalten, die für eine gewisse Ablaufsicherheit (Bytecodeverifier) sorgen, oder Betriebssystemfunktionen übernehmen (Classloader für dynamisches Nachladen von Klassen, Java-Native-Interface JNI zum Aufruf von Systemfunktionen).

Wenn also von einem Java-Programm gesprochen wird, handelt es sich um ein in der Programmiersprache Java geschriebenes Programm, dass unter Benutzung der Java-Klassenbibliotheken auf einer Java Virtual Machine läuft. Bibliotheken und JVM werden zusammen als Java-Laufzeitsystem (Java Runtime Environment, JRE) bezeichnet.

Während die JVM als ausführender Prozessor auch mit anderen Programmiersprachen benutzt werden könnte (schließlich ist sie ja in einer Maschinensprache programmierbar), sind die Programmiersprache Java und die Bibliotheken sehr stark von der JVM abhängig. Der JVM kommt also besondere Bedeutung zu, sie ist sowohl für die Ausführungsgeschwindigkeit als auch für die Sicherheit (siehe 3.2.4.2) der ausgeführten Programme zuständig.

Datentyp	Inhalt	Standardwert	Größe	Wertebereich
`boolean`	`true`, `false`	`false`	1Bit	
`char`	Unicode Zeichen	0	16Bit	$0_{16} \ldots FFF_{16}$
`byte`	signed integer	0	8Bit	-128 ... 127
`short`	signed integer	0	16Bit	-32768 ... 32767
`int`	signed integer	0	32Bit	$-2^{31} \ldots 2^{31}-1$
`long`	signed integer	0	64Bit	$-2^{63} \ldots 2^{63}-1$
`float`	IEEE754 float	0.0	32Bit	
`double`	IEEE754 float	0.0	64Bit	

Tabelle 3.1: Java Basisdatentypen

3.2.2 Die Programmiersprache Java

Java ist **eine** Möglichkeit, Programme auf der JVM-ISA auszuführen, aber im Prinzip für das Java-System nicht erforderlich. Ebensogut sind andere Sprachen und dazugehörige Compiler in den Maschinencode für die JVM denkbar und existieren bereits auch (z.B. Basic [36]). Da allerdings JVM und Java miteinander und füreinander entwickelt wurden, ist die Sprache Java genau auf die JVM abgestimmt und nutzt alle ihre Möglichkeiten bzw. die JVM bietet Mechanismen an, die sich für die Sprache Java besonders gut eignen.

Wie schon erwähnt, wurde Java in Grundzügen an die bekannten Sprachen C und C++ angelehnt. Da die Java allerdings völlig neu entwickelt wurde, konnten einige Erfahrungen mit C einfließen, die sich sowohl auf die Fehleranfälligkeit als auch die Wiederverwendbarkeit auswirken.

Java besitzt die in Tabelle 3.1 dargestellten Basisdatentypen, allerdings ist z.B. der `boolean`-Typ auf Maschinenebene auf `int` abgebildet. Diese Basistypen können mit Objekten oder mit Arrays angeordnet werden.

Der in C/C++ fast nicht vermeidbare Datentyp “Pointer” fehlt in Java hingegen ganz, zumindest ist er nicht mehr direkt sichtbar. Da die JVM keinen wahlfrei adressierbaren Speicher im herkömmlichen Sinne besitzt und damit auch die Speicherverwaltung (z.B. `malloc()` und `free()`) wegfällt, sind natürlich auch direkte Speicherzugriffe über Pointer nicht mehr möglich.

Die Stelle des Pointer-Typs nimmt der Typ “Reference” (Referenz) ein, der eine Art abstrakten Zeiger auf einen definierten Objekttyp darstellt. In C übliche und fehlerträchtige Pointerarithmetik ist nicht mehr möglich, auch das “wilde” Transformieren von Zeiger auf verschiedene Objekte (Cast) wird damit unterbunden. Auf Referenzen kann nur noch definiert über Objektelemente bzw. Arrayindizes zugegriffen werden, fehlerhafte Zugriffe können bereits während der Kompilation oder während der Ausführung innerhalb von Java mit Exceptions (siehe 3.2.4.6) abgefangen werden.

```
if(...){...}else{...}
for(...){...}
do{...}while(...)
while(...){...}
switch(...){case ....}
```

Tabelle 3.2: Java Grundkonstrukte

Die Hochsprachenkonstrukte zur Programmablaufsteuerung, die in Java benutzt werden können, sind in Tabelle 3.2 aufgeführt.

Im Gegensatz zu C besitzt Java keinen `goto`-Befehl, dagegen ist es in Java möglich, bei dem `break`-Konstrukt ein Ziel anzugeben. Damit sind die üblichen und sinnvollen Anwendungsgebiete von `goto` ebenfalls erfasst.

Für threadorientierte Programmierung sind bereits in der Sprache Java Möglichkeiten zum exklusiven Zugriff auf Objekte in kritischen Bereichen ("critical sections") mit dem `sychronized(object){...}` integriert.

3.2.2.1 Objekte und Klassen

Durch das Fehlen von Speicherzeigern nehmen Java-Objekte einen sehr großen Stellenwert ein. Grundsätzlich wird fast alles in Java (bis auf die Basistypen) an Objekte gebunden. Ein Java-Objekt als Instanz einer Klasse ist also eine Zusammenfassung und Zugriffsmöglichkeit auf klassenspezifische und auch statische Methoden, Arrays, Records und wiederum andere Objekte.

Jedes Objekt gehört dabei einer Klasse an und kann wie in C++ Eigenschaften (Methoden und Variablen) einer Superklasse "erben" und sie überschreiben oder neu definieren. Im Gegensatz zu C/C++, wo es globale Variablen und Methoden gibt, die keiner Klasse angehören müssen, ist dies in Java nicht möglich. Auch statische Methoden, also Methoden, die in Instanzvariablen keine Daten speichern und auch nicht darauf zugreifen müssen, werden in Klassen gekapselt, und "verschmutzen" damit nicht den Namensraum ("Name Space"). Ein Beispiel dafür ist die mathematische Bibliothek mit den "üblichen" Funktionen (`java.lang.Math.sin()`, `java.lang.Math.cos()` ...).

Objekte werden über einen Konstruktor einer Klassendefinition mit `new` erzeugt. Im Gegensatz zu C/C++ gibt es kein `delete`, nicht mehr referenzierte und damit auch definitiv nicht mehr benutzte Objekte werden mittels einer automatischen Garbage-Collection, deren Algorithmus in der JVM-Spezifikation nicht vorgeschrieben ist, gefunden und gelöscht. Speicherlecks wie in C aufgrund vergessener `free()` oder `delete()`-Aufrufe sind damit nicht mehr möglich. Vor dem eigentlichen Löschen des Objekts erfolgt noch ein Aufruf der

`finalize()`-Methode des Objekts, die mit der Destruktor-Funktion in C++ vergleichbar ist.

Die Vererbung von Eigenschaften einer Klasse erfolgt über das `extends`-Schlüsselwort in der Klassendefinition. Die Vererbung ist dabei nur von einer Superklasse aus möglich, eine Vererbung von Eigenschaften mehrerer Klassen ("multiple inheritance") wie in C++ ist nicht erlaubt. Als nahezu gleichwertiger Ersatz sind dafür sog. "Interfaces" vorgesehen, die weniger Seiteneffekte haben ([55], Section 3).

Ein Überladen von Methoden gleichen Namens ist in Java möglich, wie auch die Definition von (statischen) Klassenvariablen, die damit eine Art Ersatz für globale C-Variablen sind. Wie bei statischen Methoden in Java verschmutzen sie aber nicht den Namensraum, da sie nur unter dem vollständigen Klassennamen zugreifbar sind.

3.2.3 Die Java-Klassendateien und Bibliotheken

Der Java-Compiler (`javac` oder JVM-Assembler (z.B. jasmin [97]) erzeugt für jede Objektklasse ein sog. Classfile, in dem alle Informationen zu dieser Klasse enthalten sind. Dazu zählen hauptsächlich (Bild 3.2):

- Konstantenpool: Enthält Namen, Signaturen und Initialisierungswerte aller verwendeten Objekte (Methoden oder Variablen dieser oder anderer Klassen).
- Interfaces: Eine Liste aller Interfacemethoden, referenziert werden Konstantenpooleinträge.
- Feldelemente: Beschreibung der Zugriffsattribute der klassenspezifischen Felder, also statischer Variablen und Instanzvariablen. Indiziert wird dabei immer ein Feldname im Konstantenpool.
- Methoden: In diesem Teil sind die Zugriffsattribute der benutzen Methoden abgelegt. Name und Signatur werden über je einen Index im Konstantenpool referenziert.
- Attribute: Dieser Bereich speichert variable Attribute und ihre Werte. Besonders wichtig ist das "Code Attribute", das den eigentlichen Bytecode zu einer Methode enthält und somit dem `.text`-Bereich anderer Binärdateien entspricht.

Im Normalfall werden die Klassendateien einzeln geladen, allerdings können auch mehrere davon in einer sog. JAR-Datei (Java-Archive, JAR) zusammengefasst und komprimiert als ZIP-Archiv abgespeichert werden. Damit wird neben

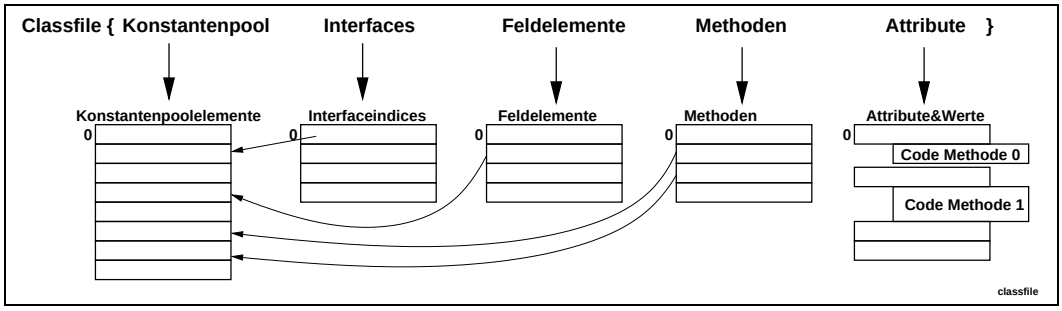

Abbildung 3.2: Aufbau der .class-Datei

der einfacheren Handhabung der Speicherbedarf der Klassen verringert und für Netzwerkanwendungen die Übertragungsgeschwindigkeit erhöht.

3.2.3.1 Java-Bibliotheken

Die mit dem Java-System mitgelieferten Klassenbibliotheken sind sehr umfangreich und bestehen z.B. im JDK1.3 aus über 1200 verschiedenen Klassen mit insgesamt ca. 16000 Methoden für das Basissystem. Dazu kommen weitere 5000 Klassen mit Erweiterungen (z.B. SWING, Corba, etc.).

Durch die unmittelbare Verfügbarkeit dieser Klassen, die unter anderem Methoden zur einfachen Netzwerkkommunikation (TCP, HTTP, FTP, ...) und auch Klassen zum direkten Parsen bzw. Erzeugen von Java und JVM-Code ("Reflection") beinhalten, wird die Entwicklung von komplexen Programmen stark vereinfacht.

Sowohl die Systembibliotheken als auch eigene Klassen werden vom Laufzeitsystem erst dann geladen, wenn Objekte daraus benötigt werden, d.h. es findet eine dynamische Bindung statt.

3.2.4 Die Java Virtual Machine

Die Java Virtual Machine ist der (virtuelle) Prozessor, auf dem Java-Programme ablaufen. Dazu werden diese Programme mit einem Compiler (z.B. `javac`) auf die Maschinenspracheebene übersetzt. Diese ist bei der JVM der sog. Bytecode bzw. JVM-Bytecode.

Im Gegensatz zu "normalen" Mikroprozessoren arbeitet die JVM nicht auf einer registerorientierten, sondern auf einer stackbasierten Architektur. Dieses Konzept wurde auch deshalb gewählt, weil durch den Wegfall von Registerfeldern in den Opcodes Platz eingespart wird. Damit werden die ausführbaren Programme relativ klein, was gerade für die dynamische Übertragung von Programmcode sehr vorteilhaft ist. Auf der anderen Seite ergibt sich damit eine Inkompatibilität zu den

üblicherweise registerbasierten Systemen, auf denen die JVM ausgeführt wird, da auf dem Zielsystem dieser Stack nachgebildet werden muss.

Nachdem die JVM keinen frei adressierbaren Speicher kennt, können Daten nur an definierten Orten temporär oder für längere Zeit gespeichert werden. Die Grundtypen werden in sog. lokalen Variablen gespeichert, diese sind allerdings nur als temporärer Speicher innerhalb einer Methode gedacht und außerhalb dieser nicht zugreifbar. Eine weitaus umfangreichere, dauerhafte Speichermöglichkeit steht mit neu erzeugten Objekten zur Verfügung. Diese können neben den Grundtypen weitere Objekte und damit indirekt Zeiger auf Methoden beinhalten.

3.2.4.1 Befehlsstruktur

Die JVM besitzt ca. 200 Maschinenbefehle, von denen die meisten mit ihren Parametern und Ergebnissen auf dem Stack arbeiten und damit nur ein Byte lang sind. Befehle, die auf lokale Variablen bzw. Methoden zugreifen, besitzen einen zusätzlichen 8Bit Parameter, der durch einen Präfixcode (“wide”) auf 16Bit erweiterungsfähig ist.

Von dieser relativ einfachen Struktur weichen nur die komplexen Verzweigungsbefehle `tableswitch` und `lookupswitch` ab. Sie besitzen die 32Bit-Vergleichswerte und die dazugehörigen Sprungziele direkt als Parameterblock hinter dem Befehlsbyte.

Konstanten (Literale) sind, bis auf wenige Ausnahmen für häufige Werte wie 0 oder 1, nur indirekt über den Konstantenpool zu laden, d.h. der JVM-Bytecode selbst enthält in den Parametern keine benutzerdefinierten Konstanten.

Ähnlich den Konstanten erfolgt der Zugriff auf Klassen- oder Instanzvariablen nur anhand eines Index in den Konstantenpool. Darüber wird die eigentliche Variable mit ihrem Namen und der Klassenzugehörigkeit identifiziert.

Unterroutinensprünge sind mit `jsr` und `return` zwar möglich, werden in der Programmiersprache Java nur für das `finally`-Konstrukt einer Ausnahmebearbeitung benutzt. Methodenaufrufe erfolgen über spezielle Befehle, die ebenfalls die Methode mit ihrem Namen anhand eines Konstantenpooleintrags indizieren. Anhand des verwendeten Befehls wird die Art der Funktion unterschieden. Es sind statische und virtuelle Methoden sowie Interfacemethoden möglich. Durch den Umweg über den Konstantenpool sind relativ einfach Zugriffsrechte und korrekte Parameterübergabe überprüfbar.

Weitere komplexe Funktionen sind für die Erzeugung von Feldern und Objekten vorhanden, ebenso wie Funktionen zur Überprüfung der Verwandschaft von Objekten.

3.2.4.2 JVM-Ausführungsmodell – Das Sandkastenprinzip

Wenn, wie beim Einsatz im WWW, fremde und unbekannte Programme auf einem Rechner ablaufen sollen, sind besondere Sicherheitsmechanismen notwendig, um negative Seiteneffekte auf die Integrität und Zuverlässigkeit des Rechners zu vermeiden. Das Java-Programm soll in einem sicheren und abgeschlossenen System, dem sog. Sandkasten (Sand Box) arbeiten und keinerlei Zugriffe auf Benutzerdaten und kritische Systemressourcen haben. Diese Sand Box stellt also eine Barriere dar, die von keinem Javaprogramm überschritten werden darf, egal ob unabsichtlich (Programmierfehler) oder absichtlich (Angriffsversuche, Denial-of-Service, etc.).

Das Prinzip der emulierten JVM-Architektur ist ein wichtiger Bestandteil für dieses Sicherheitskonzept, aber nicht der Einzige. Durch das Fehlen einer direkten Ablaufmöglichkeit ist es (bei korrekter Implementierung des Laufzeitsystems) nicht möglich, über Bufferüberläufe oder ähnliche Angriffe "feindlichen" Code auszuführen. Zusätzlich überprüft der Bytecode-Verifier[3] vor jeder Methodenausführung die Korrektheit des Bytecodes. Unerlaubte Typkonversionen oder Stackmanipulationen sind damit im Prinzip unmöglich.

Der Classloader überprüft ebenfalls bei jedem neuen Laden einer "fremden" Klasse die Konsistenz und die in der Klassendefinition spezifizierten Zugriffsregeln auf Methoden und Variablen. Daher sind auch die Klassen untereinander vor nicht gewollten Zugriffen geschützt und erlauben somit eine weitergehende Sicherheit.

Zusätzlich zum Schutz der eigentlichen JVM besitzen die verschiedenen Klassenbibliotheken weitere Einschränkungen, die je nach Ablaufart den Zugriff auf das Dateisystem oder auf Netzwerkverbindungen untersagen. Somit dürfen z.B. Applets (d.h. Java-Anwendungen, die in einem Webbrowser ablaufen) nur Netzwerkverbindungen zu dem Rechner herstellen, von dem sie heruntergeladen wurden.

Ausnahmen von diesen Regeln sind nur bei lokalen Anwendungen oder über Zertifikate möglich.

3.2.4.3 Ausführungsstack

Fast alle Befehle arbeiten auf dem Stack, haben also eine implizite Adressierung. Bis auf wenige Ausnahmen (z.B. `iinc`) müssen für alle Befehle die Parameter erst auf diesen Stack geladen werden (z.B. aus lokalen Variablen, siehe 3.2.4.5).

[3]Der Begriff Verifizierer sollte nicht mit dem gleichlautenden Begriff aus der theoretischen Informatik verwechselt werden. Es handelt sich dabei nicht um den formalen Beweis von Eigenschaften eines Javaprogramms, sondern nur um Überprüfungen einiger in [94] spezifizierter Eigenschaften.

Zwar sind die Werte auf dem Stack selbst typlos, allerdings wird vor Ausführung der Methode die Korrektheit bezüglich der Typen mit dem Bytecode-Verifier ([55], Section 4.9.2, siehe auch 3.2.4.2) überprüft. Dieser erkennt unter anderem, falls illegale Typkonvertierungen vorgenommen werden und ob der Stackfüllstand für jeden Befehl unabhängig vom Ausführungsfluss konstant ist. Damit werden unter anderem auch interne Rekursionen, die nicht durch einen expliziten Methodenaufruf erzeugt werden, verhindert.

Typkonvertierungen sind explizit nur mit JVM-eigenen Befehlen (z.B. `i2l` oder `i2f`) oder mit externen Konvertierungsmethoden durchführbar.

Wie in anderen stackbasierten Systemen (z.B. Forth, 2.1.3.1), existieren auch in der JVM Befehle zur Stackmanipulation, also Duplizierung des obersten Stackelements (TOS) und Vertauschen des TOS mit darunterliegenden Elementen. Allerdings sind diese typgebunden und erlauben keine unbeabsichtigte Vertauschung verschiedener Basistypen, was die Integrität der Ausführung beeinträchtigen könnte.

3.2.4.4 Der Konstantenpool

Zu jedem Classfile gehört ein sog. Konstantenpool (Constant Pool, CP), der zu jedem Objekt und jeder Methode verschiedene Attribute speichert. Die dabei erlaubten Einträge sind in Tabelle 3.3 aufgelistet. Einige Typen (z.B. Char-Array) haben eine beliebige Länge, daher kann der Konstantenpool nicht als ein Feld mit direkten Einträgen im Speicher abgelegt werden.

Wie z.B. anhand des Methodendeskriptors erkennbar ist, werden über diese Indizes Klassen und Methodennamen, Signaturen (d.h. Aufrufparametertypen und Rückgabetyp) und weitere Angaben verbunden. Wenn in der Programmiersprache Java auf ein Objekt oder eine Methode zugegriffen wird, ist der dazugehörige JVM-Befehl nur mit einem Index auf den Deskriptor im CP versehen. In der sog. Auflösungsphase (Resolving) werden die zum Index gehörenden CP-Einträge gelesen und die damit referenzierten Klassen überprüft und geladen. Statische (d.h. klassenglobale) und instanzgebundene Variablen sind ebenfalls im CP mit einer Referenz auf den Typ abgelegt, damit ist implizit auch der Speicher reserviert.

Die Informationen über Methoden und andere Objekte eines Index sind durch Zeiger auf anderen Indexeinträge verteilt und nur als Zeichenketten für Klasse, Namen und Signatur beschrieben. Somit sind zur Auflösung mehrere, teilweise komplexere Zugriffe und Vergleiche notwendig.

Bei der Auflösung eines CP-Indices müssen evtl. mehrere Klassen nachgeladen werden. Diese werden wiederum initialisiert, bevor auf ihre Objekte zugegriffen wird. Je nach Abhängigkeitsverhältnis der Klassen kann damit eine einzelne Auflösung sehr zeitaufwendig werden.

CP-Eintrag	Nutzung	Inhalt [1)]
Felddeskriptor	Klassen/Instanzvariable	Typ-und-Name (I)
Methodendeskriptor	Methoden	Klassen (I), Typ-und-Name (I)
Interfacedeskriptor	Interface	Klasse (I), Typ-und-Name (I)
Typ-und-Name	Signatur und Name	Deskriptor (I) und Name (I)
Klasse	Klassenbezeichnung	Name (I)
String	String-Initialisierung	Stringindex (I)
Klassenname	Benamung	Char-Array
Name	Benamung	Char-Array
Deskriptor	Datentypbeschreibung	Char-Array
Integer, Float	Wertdefinition	4Byte
Long, Double	Wertdefinition	8Byte
Char-Array	Wertdefinition	n Bytes

[1)] (I): Index auf weiteres CP-Element

Tabelle 3.3: Einträge im Konstantenpool

Da in der JVM sehr häufig objektorientierte Befehle vorkommen, hat die Auflösung der Indices einen großen Einfluss auf die Effizienz der Abarbeitung. Fast alle heutigen JVM-Implementierungen (auch Interpreter) arbeiten daher mit einer Art CP-Cache, die die Daten bereits aufgelöster Objekte im CP zwischenspeichert. Damit sind sie beim nächsten Zugriff schon vorhanden und der nötige Überprüfungsaufwand wird eingespart oder zumindest stark reduziert. Einige ziemlich triviale[4] Algorithmen sind dazu sogar patentiert [149, 110, 132].

3.2.4.5 Lokale Variablen und Funktionsparameter

Methodeneigene Variablen auf Basistypen, die temporärer Natur sind, werden in der JVM als sog. Lokale Variable (LV) gespeichert. Funktionsparameter werden als LV übergeben und sind für die aufgerufene Funktion als zusätzliche LV zugreifbar.

Die Zugriffsart der LV erfolgt wiederum über Indices, die das n-te Element im Speicherbereich der lokalen Variablen adressieren. Falls Übergabeparameter vorhanden sind, belegen sie die ersten m Einträge in diesem Bereich, die methodeneigenen Variablen schließen sich an.

Obwohl die JVM von Grund auf neu entwickelt wurde, finden sich einige Unregelmässigkeiten[5] auch bei der Organisation der lokalen Variablen: Variablen, die einen 64Bit-Typ beinhalten (also `long` und `double`) belegen **zwei** Index-

[4]Trivial, da einfache Übertragung des bekannten Cacheprinzips auf Konstantenpoolzugriffe

[5]Auch in anderen Bereichen der JVM-Spezifikation erscheint oft der Ausdruck “due to historical reasons”

nummern, wovon auf die höhere nicht zugegriffen werden darf. Dies trifft aber nur für Variablen zu, die nicht auch Aufrufparameter der Methode sind. Damit wird die Implementierung der Zugriffe auf lokale Variablen erschwert, da damit für Parameter eine methodenabhängige Umrechnungstabelle von dem LV-Index auf den Offset im Stackrahmen nötig ist.

Im Entwurf des Bytecodes spiegelt sich auch die Annahme wieder, dass Funktionen mit besonders vielen lokalen Variablen selten sind. Daher gibt es zum Laden und Speichern der ersten 4 lokalen Variablen eigene Byte-Befehle. Für die Indices größer 3 sind allgemeine Befehle mit einem oder zwei Zusatzbytes vorhanden, somit können maximal 65536 lokale Variablen adressiert werden. Diese "Kompression" sorgt für eine kleinere Codelänge und damit schnellere Übertragung als auch für eine (zumindest bei Interpretern) beschleunigte Ausführung.

3.2.4.6 Ausnahmebehandlung

Wie auch in C++, gibt es in Java und der JVM Ausnahmen (Exceptions), die entweder abgefangen werden oder an die aufrufende Methode weitergeleitet werden.

Bei den Ausnahmen sind zwei Arten zu unterscheiden:

- JVM-generierte Ausnahmen

  Die JVM erzeugt selbstständig Ausnahmen, z.B. beim Aufruf von nicht kompatiblen Klassen, bei Zugriffen über Arraygrenzen hinaus und bei Division durch Null.

- Programmerzeugte Ausnahmen

  Der JVM-Befehl `throw` erlaubt es, benutzerdefinierte Ausnahmen zu generieren.

Jedem Code-Bereich innerhalb einer Methode können im Classfile "Handler" zum Abfangen von aufgetretenen Exceptions zugewiesen werden. Zur codesparenden Realisierung der `finally`-Funktion sind in der JVM noch die Befehle `jsr` (Jump Subroutine) und `ret` (Return) implementiert. Damit lässt sich die `finally`-Funktion in eine Unterroutine auslagern, die sowohl vom Exceptionhandler als auch beim normalen Durchlauf des `try`-Konstruktes angesprungen wird. Diese Unterroutinenaufrufe sind nur für die `finally`-Behandlung gedacht und werden in Java anderweitig nicht benutzt.

3.3 Java Ausführungsarten

Die JVM stellt ein virtuelles und teilweise abstrahiertes Prozessorsystem dar und kann daher nicht direkt auf herkömmlichen Systemen ausgeführt werden.

Um dennoch JVM-Programme ablaufen zu lassen, ist es nötig, das JVM-System nachzubilden. Die Implementierung der Klassenbibliotheken und der allgemeinen JVM-Verwaltungsfunktionen (z.B. Resolving, Classloader, etc.) sind dabei getrennt von der eigentlichen Ausführung zu sehen, da sie allgemein gehalten werden können.

Für die Ausführung des Bytecodes stehen jedoch mehrere Möglichkeiten zur Verfügung, die in den Abschnitten 3.3.1 bis 3.3.4.4 besprochen werden. Jede der Möglichkeiten besitzt dabei spezifische Vor- und Nachteile. Diese werden anhand von realen Implementierungen ab 5.3 näher ausgeführt und verglichen.

3.3.1 Interpretierung der JVM in Software

Die einfachste Möglichkeit der Ausführung von JVM-Bytecode besteht in der sequentiellen Interpretierung der einzelnen Opcodes. Üblicherweise sind diese Interpreter in anderen Hochsprachen (z.B. C) geschrieben und werden dann für das jeweilige Zielsystem kompiliert.

Dieser Weg erlaubt zwar eine einfache, schnelle und besonders portable Implementierung, verursacht aber durch die Nachbildung des Stacks und des Funktionsverteilers (Dispatchers) sehr viel Overhead. Gerade moderne Prozessoren haben durch die Sprungfolge des Dispatchers Probleme mit der Sprungvorhersage [161]. Diese Nachteile entstehen durch die nicht vorhersagbaren Sprünge bei der Dispatcherabarbeitung selbst und durch viele indirekte Sprünge bei der Behandlung. Zur effizienteren Ausführung sind daher auch schon Änderungen bzw. Ergänzungen für normale Prozessoren vorgeschlagen worden [92].

Inzwischen sind auch Ansätze für eine etwas effizientere Implementierung entwickelt worden [63]. Sie kommen aber nur in Betracht, wenn andere Systeme einen zu großen Ressourcenbedarf besitzen.

3.3.2 Umgehung der JVM-Ebene – Direkte Compilation

Bei einigen Anwendungsgebieten ist es durchführbar, das Javaprogramm direkt aus dem Quellcode ohne Umweg über die JVM in den nativen Maschinencode zu übersetzen (siehe z.B. gcj, 5.3.3). Dadurch ist eine sehr gute Geschwindigkeit möglich, allerdings werden dadurch viele Vorteile von Java aufgegeben: Beispielsweise geht zwangsläufig die Plattformunabhängigkeit verloren und es ist unmöglich, während der Laufzeit dynamisch Klassen im JVM-Format nachzuladen.

3.3.3 Just-In-Time-Compiler (JIT) und Compile Ahead

Im Unterschied zur direkten Compilation wird hier der JVM-Bytecode erst unmittelbar vor der Ausführung in ein natives Maschinencodeprogramm übersetzt. Bei

diesem Ansatz sind verschiedene Abstufungen in der “Komplettheit” der Übersetzung möglich:

- Vorcompilierung/Compile-Ahead

  Hierbei wird der gesamte gegebene JVM-Bytecode auf einmal (vor bzw. beim Start des Programms) übersetzt. Dies hat den Vorteil, dass während der Ausführung selbst keine Wartezeiten mehr entstehen. Ebenso sind aufwendigere, globale Optimierungen realisierbar. Andererseits ist die Startzeit relativ hoch, insbesondere wenn Methoden übersetzt werden, die im späteren Programmlauf gar nicht benötigt werden. Zusätzlich muss Speicherplatz für alle übersetzten Methoden vorhanden sein, was bei großen Anwendungen durchaus ins Gewicht fallen kann.

- Just-In-Time (JIT)

  Beim Just-In-Time-Ansatz wird ein kleiner Teil des Codes (üblicherweise der einer Methode) erst dann in das native Format übersetzt, wenn durch die Ausführung feststeht, dass dieser Teil überhaupt benötigt wird. Dadurch verkürzen sich die Startzeiten und auch der benötigte Speicherverbrauch, allerdings kann auch hier durch die Übersetzung von Methoden, die kaum benutzt werden, ein großer Zeitverlust entstehen. Gerade bei interaktiven Anwendungen kann die Reaktionszeit (Responsivität) stark unter den andauernden Übersetzungsläufen leiden. Globale Optimierungen sind durch den Sichtbereich auf Methoden stark eingeschränkt bzw. nicht durchführbar.

  Der nach der Übersetzung erzeugte Maschinencode wird üblicherweise gespeichert, somit kann die Methode beim nächsten Aufruf wesentlich schneller ausgeführt werden.

Für den Einsatz bei Java, das auch Klassen nur auf Anforderung nachlädt, hat sich das Just-In-Time-Prinzip als besser geeignet herausgestellt, die Nachteile müssen aber toleriert bzw. über eine Mischform (z.B. Hotspot, siehe 5.3.2) weiter reduziert werden.

3.3.4 Ausführung in Java-CPU bzw. mit Co-Prozessor

Die effizienteste Methode der Java-Ausführung ist die Verlagerung der JVM-Emulation in Hardware. Hierbei sind bislang vier Ansätze realisiert, die im folgenden besprochen werden.

3.3.4.1 Eigenständiger Java-Prozessor

Diese Lösung implementiert eine vollständige JVM als Prozessor (sog. “Java-CPU”), auch alle Aktionen neben der eigentlichen Codeausführung (z.B. Garbage Collector, Class Loader etc.) laufen auf diesem System ab. Ein solcher Java-Prozessor kann die Grundlage für ein autonomes System bilden und entspricht damit anderen registerbasierten Mikroprozessoren.

Da die JVM allerdings selbst keine Möglichkeiten zur hardwarenahen Programmierung bietet, muss sie in diesem Bereich ergänzt werden. Dies kann z.B. durch Zusatzbefehle geschehen, die nur in einem “Supervisor”-Modus ausführbar sind.

Nachteilig ist im praktischen Einsatz der Java-CPUs die völlig neue Architektur, die sowohl bei der Software-Entwicklung (neue Tools, neue Fehler) als auch für die Hardwareentwicklung (neue HW-Schnittstellen, neue Hardwaretreiber) Probleme aufwirft. Eine einfache Migration bzw. Erweiterung einer schon existierenden SW- und HW-Infrastruktur ist damit nicht gegeben.

3.3.4.2 JVM-Coprozessor

Ein JVM-Coprozessor benötigt weiterhin einen herkömmlichen Prozessorzum Laden von Klassen bzw. zur Auflösung von Konstantenpooleinträgen, arbeitet aber sonst relativ autonom eine bestimmte Methode ab.

3.3.4.3 Java-Unterstützung für existierende Prozessoren

Dieser Ansatz fügt einem herkömmlichen Prozessor intern (oder seltener extern) eine Zusatzlogik zur schnelleren Abarbeitung von JVM-Befehlen hinzu. Damit kann z.B. ein JVM-Interpreter auf spezielle, besonders effiziente Befehle zur Nachbildung der JVM-ISA zurückgreifen. Möglich ist auch die unmittelbare Übersetzung des JVM-Befehlsstromes in Mikrocodeanweisungen oder CPU-Steuersignale (z.B. ARM Jazelle, 5.4.10). Die JVM-Ausführung läuft aber weiterhin auf dem Hauptprozessor. Problematisch an diesem Ansatz ist die sehr CPU-spezifische Schnittstelle, die nur eine möglichst geringe Latenz in der Befehlsverarbeitung hinzufügen sollte.

3.3.4.4 HW-Befehlsübersetzung in natives Maschinenformat

Dieser Ansatz ist eine Mischform zwischen Software und Hardware und besteht in einer Hardwareunterstützung zur JIT-Übersetzung in das native Maschinecodeformat. Ziel ist es, Vorteile der beiden Ausführungsebenen zu kombinieren und

Nachteile abzuschwächen bzw. zu vermeiden. Wesentlicher Unterschied zur "einfachen" Befehlsübersetzung im Befehlsstrom ist hier die Möglichkeit zu umfangreicheren Optimierungen, da ein größerer Kontext sichtbar ist.

Das in dieser Arbeit besprochene JIFFY-System basiert auf diesem Konzept, das allerdings selten eingesetzt wird.

3.3.5 Allgemeiner Vergleich der JVM-Ausführungsarten

Ein inzwischen etwas veralteter[6] in [87] kommt zu dem Ergebnis, dass Softwareübersetzer für die meisten Anwendungsgebiete bessere Leistungen erwarten lassen. Allerdings stützt sich diese Annahme nur auf wenige Beispiele, die nicht alle zur Zeit verfügbaren Konzepte umfassen und ist daher unvollständig. Weiterhin werden nur Systeme mit ausreichenden Ressourcen (im Grunde also PCs) betrachtet und andere Plattformen außer acht gelassen.

Eine von mir vorgenommene grobe Einordnung der Verfahren mit ihren Vor- und Nachteilen ist in Tabelle 3.3.5 zu finden. Die Bewertungen entstanden dabei aus der Synthese einschlägiger Einzelvergleiche und eigener Tests und Überlegungen. Ein Gesamturteil ist dabei nicht möglich, da der Einsatz eines bestimmten Konzepts aus einer gewichteten Bewertung der Einzelkriterien bestimmt wird, die vom Einzelfall abhängen.

Die in 3.3.4.4 erwähnte Methode der JIT-Compilation in Hardware ist dabei nicht in der Tabelle aufgeführt, da sie außerhalb des JIFFY-Systems bis auf ein weiteres System (siehe 5.4.11) noch nicht realisiert wurde und somit keine sinnvollen und umfassenden Vergleichsdaten zur Verfügung stehen. Die nur für das JIFFY-System geltenden Anforderungen und Bewertungen sind in Abschnitt 6.2 aufgeführt, wobei diese dann auch mit anderen realen Implementierungen verglichen wird.

3.4 Java Benchmarks

Die Messung der Geschwindigkeit ("Performance") von Ausführungsplattformen für Java bringt wie alle sog. Benchmarks die "üblichen" Probleme: Es können nur exemplarische Geschwindigkeitsvergleiche mit zwar typischen, aber doch willkürlichen Anwendungen gemacht werden. Eine Kompression der Ergebnisse auf eine leicht vergleichbare Zahl wird zwar oft benutzt, ist aber nicht wirklich sinnvoll. Beispielsweise wird in [139] eine Reihe von Java-Benchmarks auf verschiedenen Ausführungssystemen getestet und verglichen. Bis auf die Gemeinsamkeit, dass die Interpretierung fast immer die langsamste Ausführungsart ist, sind die

[6] Der Vergleich stammt aus dem Jahr 1998 und hat damit nur wenig konkrete Vergleichssysteme

	Interpreter	Direkte Compilation	Compile Ahead
Portabilität	⊕⊕	⊖⊖	⊖
Speicherverbrauch	⊕⊕	◯	⊖⊖
Ablaufeffizienz	⊖⊖	⊕⊕	⊕
Gesamteffizienz	⊖⊖	⊕⊕	⊕
SW-Aufwand	⊕⊕	⊖	⊖
HW-Aufwand	–	–	–
Praktikabilität	⊕	⊖⊖	⊖

	JIT	Java-CPU	JVM-Coprozessor	JVM-Erweiterung
Portabilität	⊖	⊕⊕	⊖	⊖⊖
Speicherverbrauch	⊖	⊕⊕	⊕⊕	⊕
Ablaufeffizienz	⊕	⊕	⊕	⊕
Gesamteffizienz	◯			
SW-Aufwand	⊖⊖	⊕	◯	◯
HW-Aufwand	–	⊖⊖	⊖	⊖
Praktikabilität	⊕⊕	⊖	◯	◯

Tabelle 3.4: Vergleich der JVM-Ausführungsarten

Ergebnisse je nach Benchmark sehr gestreut und nicht in ein Gesamturteil überführbar.

Wie bei anderen Systemen, existieren eine Reihe von Java-Benchmarks, die über synthetische Microprogramme [60] bis zu Anwendungsprogrammen (SpecJVM98[42], SciMark2.0[125]) reichen. Für eingebettete Systeme existiert auch eine speziell angepasste Benchmarksammlung JavaEEMBC [41].

Ein interessanter Ansatz für einen plattform-unabhängigen Vergleich von Java-Programmen ist in [64] beschrieben. Dort wird versucht, direkt aus den Eigenschaften des Bytecodes, der von verschiedenen Java-Compilern erzeugt wurde, die Effizienz des Bytecodes herauszuarbeiten.

3.5 Eigenschaften von JVM-Bytecode

Ein Übersetzungskonzept von einer ISA in eine andere hängt in der resultierenden Effizienz stark von den Eigenschaften beider Architekturen ab. Wird, wie bei JIFFY, zusätzlich eine Zwischenschicht eingeführt, sind Optimierungen bereits auf dieser Ebene möglich. Die Qualität und der dazu nötige Aufwand sind dabei mit den Eigenschaften der Ausgangsarchitektur verbunden. Neben den implizi-

ten Eigenschaften der JVM, wie der Stacknutzung und den möglichen Befehlen, ist es aber auch wichtig, die reale Nutzung zu überprüfen. Dies erlaubt eine Abschätzung über den Sinn von Optimierungsmethoden und den Aufbau der zu entwickelnden Zwischensprache.

Diese Eigenschaften sind natürlich abhängig vom individuellen Programm und der Erzeugung des Bytecodes aus Java. Daher lassen sich ähnlich der Benchmarks nur statistische Werte für eine Menge von repräsentativen Methoden und Klassen angeben.

Die dabei gefundenen "typischen" Eigenschaften lassen sich dabei in folgende Arten einteilen:

- "Typisch" durch Java-Entwicklungsmethodik

  Java-Methoden sind relativ klein und bislang in der Länge auf max. $2^{16} - 1$ Bytes begrenzt. Die JDK1.3-Klassenbibliotheken mit ca. 16300 Methoden haben eine durchschnittliche Methodenlänge von 57Byte. Diese Methoden haben üblicherweise wenig Argumente und wenige lokale Variablen, nicht skalare Typen werden nur als Referenzen auf Objekte gespeichert.

- "Typisch" durch JVM-Architektur

  - Die Stackbenutzung in Kombination mit dem Vorhandensein lokaler Variablen führt zu einem "flachen" Stack, d.h. der Stack wird nur selten als Zwischenergebnisspeicher für mehr als 2-3 Ergebnisse benutzt. Dies wird auch durch den Bytecodeverifier bedingt, der zu einem Befehl in einer Methode keine unterschiedlichen Stackfüllstände zulässt (Kap. 4.9.2 in [94]), und somit implizite Rekursionen oder variable Datenspeicherung verbietet.
  - Durch das Fehlen des direkten Einfügens vom Konstanten in den Bytecode (außer wenigen Literalen im Bereich -1 bis 4), führen Konstantenzugriffe über den Befehl `ldc` zu Zugriffen auf den Konstantenpool.

- "Typisch" durch Compilerbenutzung

  - Java-Compiler (wie z.B. `javac`) erzeugen für Java-Konstrukte passende Schablonen im Bytecode, wie z.B. für Schleifen. Weiterhin führt die nicht orthogonale JVM-Befehlsstruktur für bestimmte Typen (z.B. `long`) teilweise zu einer Kaskadierung von komplexen Befehlen, nur um einen Befehl nachzubilden. Diese "Schablonen"-Nutzung ist relativ einfach zu erkennen.
  - Es ist davon auszugehen, dass durch den Java-Compiler unnötige Codeteile bereits weggelassen wurden ("Dead-Code-Elimination").

3.5.1 Anzahl der lokalen Variablen und Stacktiefe

Um Anhaltspunkte zu finden, wurden in [163] verschiedene Programme und Klassen einschließlich die des Java-Laufzeitsystems untersucht, um ihre Eigenschaften zu ermitteln. Diese statistischen Werte mit ihren Extremwerten sind zusammengefasst in Tabelle 3.5 zu finden. 60% der Methoden benötigen dabei maximal zwei lokale Variablen.

Anzahl	Lokale Variablen (%)	max. Stackelemente (%)
0	0.3 - 11.8	1.6 - 7.6
1	18.8 - 42.9	2.5 - 19.2
2	23.4 - 32.1	29.4 - 28.8
3	0.7 - 9.6	3.6 - 34.2
4	1.0 - 21.3	2.8 - 20.7
5	6.2 - 46.3	9.9 - 55.6
6	1.3 - 7.2	0.6 - 4.2
7	0.0 - 1.6	0.0 - 3.4
8	0.0 - 3.0	0.0 - 35.2
9	0.0 - 1.1	0.0 - 0.4
>9	0.0 - 1.3	0.0 - 0.6

Anmerkung zu Lesart der Tabelle:
Beispielsweise schwankt der Anteil der Methoden mit einer lokalen Variable bei den untersuchten Klassen zwischen 18.8 und 42.9%

Tabelle 3.5: Anzahl von lokalen Variablen und Stacknutzung (nach [163])

Eigene Tests an den JDK1.3-Klassenbibliotheken, die in Tabelle 3.6 gezeigt sind, bestätigen diese Messungen im Grundsatz. Es gab unter den $>$16300 untersuchten Methoden nur eine Methode, die 53 lokale Variablenelemente benutzt[7] und die davon bereits 30 Variablen als 30 Methodenparameter benötigt.

Die bei Methodenaufrufen verwendeten Argumente sind nach [163] zu ca. 60% Integerwerte, zu 20% Referenzen. Der verbleibende Anteil verteilt sich auf Array, Double, Byte und Char (in dieser Reihenfolge). Dieses Ergebnis ist aufgrund der Konzentration der JVM-Befehle auf Integer- und Referenzmanipulation nicht überraschend.

Eine Untersuchung auf die Häufigkeit der verwendeten JVM-Befehle in [164], ergab, dass allein Lesezugriffe auf lokale Variablen ca. 30% aller benutzten Befehle ausmachen, weitere 15% sind Zugriffe auf Objektelemente. Komplexe Befehle (invoke*, Arrayallokierung) sind mit weniger als 6% vertreten.

[7] Es handelt sich um `sun/awt/geom/Curve/findIntersect`

Anzahl der LV	Anzahl der Methoden absolut	in %	Anzahl der LV	Anzahl der Methoden absolut	in %
0	478	2.9	19	11	0.1
1	4845	29.4	20	13	0.1
2	4331	26.3	21	14	0.1
3	2291	13.9	22	20	0.1
4	1383	8.4	23	9	0.0
5	904	5.5	24	7	0.0
6	542	3.2	25	6	0.0
7	378	2.3	26	7	0.0
8	288	1.8	27	6	0.0
9	253	1.5	28	3	0.0
10	148	0.9	29	5	0.0
11	150	0.9	30	5	0.0
12	95	0.6	31	3	0.0
13	77	0.5	33	1	0.0
14	55	0.3	37	2	0.0
15	40	0.2	38	1	0.0
16	37	0.2	40	2	0.0
17	24	0.1	48	1	0.0
18	20	0.1	53	1	0.0

Tabelle 3.6: Anzahl von lokalen Variablen (LV) im JDK1.3 (eigene Messung)

3.5.2 Anzahl der Sprünge

Bei einer JIT-Übersetzung ist es nötig, dass die Adressen von Sprüngen im Bytecode auf die nativen Adressen übersetzt bzw. gebunden werden (“Linker”). Als Sprünge gelten in diesem Zusammenhang Befehle, die den Befehlsablauf innerhalb einer Methode verändern. Dies sind “normale” Sprünge, aber auch die einzelnen Ziele der `tableswitch`- und `lookupswitch`-Befehle. Jeder dieser Sprünge benötigt Platz in der Linkertabelle, die zu erwartende Anzahl der Sprünge bestimmt also den Linkeralgorithmus und die Datenstrukturen im Speicher (siehe Kap. 6.5.9).

Die maximale Anzahl von Sprüngen ist durch die Größenbeschränkung von Javamethoden selbst gegeben, womit nicht mehr als ca. 21000 Sprungbefehle mit 3Byte Befehlslänge möglich sind. Bei Verwendung von `tableswitch` bzw. `lookupswitch` reduziert sich dies auf max. 16000 bzw. 8000 Sprünge. Diese Zahlen sind natürlich theoretische Maximalwerte, da diese Methoden keinerlei Funktionalität aufweisen würden.

JDK1.3-Klassenbibliotheken	
Methoden:	ca. 16300
Sprünge	max. 137 pro Methode
Sprungziele	max. 136 pro Methode
Methode mit maximaler Anzahl:	`java/lang/String/getKeyText()`
Durchschnittswerte:	
Sprünge	2.34 pro Methode
Sprungziele	1.85 pro Methode
JDK1.3 und CaffeineMark 2.5	
Sprünge	max. 187 pro Methode
Sprungziele	max. 173 pro Methode
Methode mit maximaler Anzahl:	`logicmark()`
Durchschnittswerte:	
Sprünge	2.35 pro Methode
Sprungziele	1.87 pro Methode

Tabelle 3.7: Typische Anzahl der Sprungbefehle und Sprungziele

Um an realistische Zahlen zu gelangen, wurden eigene Untersuchungen an den mit dem JDK1.3 mitgelieferten Klassenbibliotheken (incl. AWT) und einigen Java-Benchmarks bezüglich der Häufigkeiten von Sprungbefehlen und Sprungzielen durchgeführt. Diese Messungen ergaben die in Tabelle 3.7 gezeigten, typischen Höchst- und Durchschnittswerte.

Wie anhand der ermittelten Durchschnittswerte zu sehen ist, ist die Benchmarkmethode aus dem CaffeineMark 2.5 mit 187 Sprungbefehlen und 173 Sprungzielen eine Ausnahme, übliche Javamethoden benötigen wesentlich weniger Sprünge.

3.6 Zusammenfassung

Java ist eine Kombination aus Sprache, Ausführungssystem und umfangreichen Bibliotheken. Die virtuelle Maschine (JVM) basiert auf einer “synthetischen” und relativ abstrakten Stackarchitektur, die in realen System mit verschiedenen Möglichkeiten nachgebildet werden kann.

Die Art der Nachbildung (Emulation) bestimmt die Leistungsfähigkeit des ganzen Java-Systems. Die am häufigsten eingesetzten Emulationsarten sind In-

terpretation, die portabel, aber auch sehr langsam ist, und die Just-In-Time-Compilation (JIT), die sehr schnellen Code erzeugt, aber im allgemeinen nicht einfach auf andere Zielsysteme portierbar ist. Für eingebettete Systeme sind inzwischen auch vollständige, integrierte JVM-CPUs bzw. Co-Prozessoren erhältlich, die im Einsatzgebiet aber relativ festgelegt sind und den Entwurf solcher Systeme verkomplizieren.

4

Eingebettete Systeme, FPGAs, Rekonfigurierbare Systeme

4.1 Definition von eingebetteten System

Unter eingebetteten Systemen werden im Allgemeinen Rechensysteme verstanden[1], die auf einen speziellen Anwendungsbereich zugeschnitten und in einen technischen Kontext eingebunden sind. Im Gegensatz dazu stehen die universal verwendbaren Systeme, wie z.B. Arbeitsplatzrechner (PCs), die vom Hard- und Softwareentwurf auf keinen bestimmten Anwendungszweck festgelegt sind.

Eingebettete Systeme haben unter anderem folgende Eigenschaften [157]:

- Echtzeitfähigkeit

  Eingebettete Systeme werden häufig in zeitkritischen Anwendungen (z.B. Prozess-Steuerungen) eingesetzt, bei denen die Antwort auf externe Eingänge innerhalb bestimmter Zeitspannen erforderlich ist, um das ganze System in einem fehlerfreien Zustand zu halten. Hierbei wird noch zwischen "harter" und "weicher" Echtzeitfähigkeit unterschieden (siehe Abschnitt 4.4).

- Heterogenität

  Eingebettete Systeme bestehen oft aus unterschiedlichen Hardware- und Softwarekomponenten. Besonders bei Kommunikationsanwendungen sind sehr häufig mehrere Prozessoren (DSPs, Mikrocontroller etc.) bzw. prozessorartige Hardware (ASICs, FPGAs etc.) zu einem Gesamtsystem zusammengeschaltet. Sie empfangen, reagieren und erzeugen unterschiedliche Datenarten, z.B. einzelne, asynchrone Ereignisse oder kontinuierliche Datenströme.

[1] Es gibt mehrere, mehr oder weniger strikte Definitionen bezüglich der Einbettung in physikalische Prozesse

- Verteilte Implementierung

  Die Komponenten eines eingebetteten System kommunizieren oft über ein oder mehrere gemeinsame Kommunikationsysteme unter Einsatz von Protokollen. Diese Kommunikationskanäle besitzen teilweise hohe Bandbreiten und sind stark zeitgebunden.

- Begrenzte Ressourcen

  Durch den festgelegten Anwendungsbereich und aus betriebswirtschaftliche Gründen (z.B. Bauteilpreis) sind die in einem eingebetteten System verfügbaren Ressourcen (Stromverbrauch, Rechenleistung, Speichergröße, mechanische Abmessungen) entweder hart begrenzt oder nur mit großem Aufwand erweiterbar. Abschnitt 4.2 beschreibt diese Einschränkungen genauer.

Übliche Anwendungsgebiete eingebetteter Systeme sind z.B. Prozess-Steuerungen in der Industrie und anderen technischen Bereichen, Kontroll- und Betriebssteuerung von Kommunikationseinrichtungen (Router, Mobiltelefone) oder “unsichtbare” Computer in Consumergeräten und Alltagsgegenständen (z.B. Autos).

4.2 Eigenschaften und Einschränkungen

4.2.1 Speicherarchitektur

Da ein eingebettetes System schon beim Entwurf auf ein bestimmtes Anwendungsgebiet angepasst wird, ist auch üblicherweise der Systemspeicher in seiner Größe und Anschlussart definiert.

Während bei PC-ähnlichen Systemen auch der Systemspeicher ähnlich genutzt wird, also ein “Chipsatz” den Speicher (z.B. SDRAM-Module) mit dem Prozessor verbindet, ist bei “kleineren” Systemen die Speicheransteuerung teilweise direkt in den Prozessor integriert. Die dafür benötigten internen Adressdekoderbänke führen häufig zu Einschränkungen bezüglich des maximal nutzbaren Speicherbereichs und auch der Speichertypen (ROM, SRAM, SDRAM, etc.)

Einige Prozessoren besitzen bereits internen Speicher (ROM und/oder RAM), der besonders schnell ansprechbar ist. Maskenprogrammierbares bzw. nur einmal programmierbares ROM ist bei diesen Systemen nur selten vorhanden, typisch ist Flash-EPROM mit Größen zwischen 16KByte und 1MByte. Als Arbeitsspeicher ist üblicherweise SRAM mit Größen von 4KByte bis 128KByte integriert, seltener auch größere eingebettete DRAM-Blöcke (Embedded DRAM). Einige Systeme erlauben dabei die Nutzung des internen Cache-RAMs als echten Hauptspeicher und sparen somit in einigen Anwendungsbereichen externes RAM ein.

4.2.2 Leistungsaufnahme

Eingebettete Systeme sind in ihrer elektrischen Leistungsaufnahme üblicherweise durch ihr Einsatzgebiet eingeschränkt. Neben den direkten Einschränkungen in der Energieversorgung, wie z.B. Batteriebetrieb oder Größen- oder Kostenbeschränkung des Netzteils, ist die Einschränkung der Kühlmöglichkeiten ebenfalls bestimmend: Nahezu die gesamte aufgenommene elektrische Energie wird wieder als Abwärme abgegeben.

Für tragbare Geräte ist damit meist nur noch eine passive Kühlung über das Gehäuse möglich. Für industrielle Umgebungen ist mit 85° eine wesentlich höhere Lufttemperatur erlaubt als bei Consumergeräten. Damit wird die maximal zulässige lokale Aufheizung herabgesetzt und bedingt eine Vergrößerung des Kühlaufwands.

Daher sind besonders Prozessoren für eingebettete Systeme mit verschiedenen Methoden zur Minderung der Leistungsaufnahme ausgestattet. Bei CMOS-Bausteinen wird diese Verlustleistung hauptsächlich durch die statischen Leckströme der MOSFET-Transistoren und die dynamischen Umladungsverluste bestimmt. Im Gegensatz zu den Leckströmen[2] steigt die dynamische Leistungsaufnahme nahezu linear mit dem Takt. Daher sind die am häufigsten eingesetzten Methoden zur Verminderung der Verlustleistung eine kontrollierte Herabsetzung des Taktes über eine PLL[3] mit gleichzeitiger Verminderung der Versorgungsspannung oder das Abschalten des Taktes für bestimmte Einheiten auf dem Chip (“Clock Gating”).

Neben dem Prozessor, dessen Leistungseffizienz üblicherweise in mW/MHz verglichen wird, ist auch der Speicher in der Leistungsaufnahme zu berücksichtigen. Dabei besitzen statische Speicher (SRAM, EPROM) im Ruhezustand im allgemeinen nur eine vernachlässigbare Verlustleistung. Ausnahmen bilden hier nur die Cache-RAMs. Dagegen benötigen dynamische Speicher (DRAM, SDRAM) eine kontinuierliche Auffrischung, die entweder extern oder intern (Self Refresh) erzeugt wird. Der dabei benötigte Strom ist zwar gering, aber um Größenordnungen höher als der von SRAM. Einige Beispiele der Leistungsaufnahmen von zur Zeit üblichen Speichertypen- und Größen sind in Tabelle 4.1 gezeigt.

DRAM und SDRAM sind, auf die Speichergröße bezogen, zwar sparsamer als SRAM oder EPROM, benötigen aber neben zusätzlicher Logik zur Ansteuerung auch vergleichsweise viel Leistung in inaktiven Phasen bzw. während des Self-Refreshs. Cache-SRAM mit Zugriffszeiten $<$25ns ist meistens nicht auf einen stromsparenden Standby-Betrieb ausgelegt und benötigt auch im inaktiven Zustand (Chip-Select inaktiv) relativ viel Strom.

[2] Die Leckströme steigen mit wachsender Transistoranzahl und kleineren Strukturgrößen.

[3] Phase Locked Loop, Einheit zur quarzstabilen Erzeugung nahezu beliebiger Taktverhältnisse

Typ	Größe	t_{access} [1)]	Refresh	Standby	Active
SRAM					
MCM6246 [104]	512KB	20ns	–	50mW	1000mW
AS6VA5128 [9]	512KB	55ns	–	55μW	132mW
Flash EPROM					
W28J800B [166]	1MB	90ns	–	45μW	100mW
DRAM					
MT4C1M16-6 [153]	2MB	35-130ns	900μW	1-3mW	500mW
SDRAM					
HYB39S128400-7 [73]	16MB	15-50ns	2-4.5mW	135mW	510mW

Anmerkung: 1) Bei (S)DRAM Angabe der minimalen Column- bzw. Row-Zugriffszeiten

Tabelle 4.1: Leistungsaufnahmen verschiedener Speichertypen

4.3 Mikroprozessoren für eingebettete Systeme

Dieser Abschnitt beschreibt einige oft in eingebetteten Systemen verwendete Mikroprozessoren, auf denen das JIFFY-Konzept arbeiten soll. Bis auf wenige Ausnahmen basieren alle auf RISC-ähnlichen Architekturen. Sie enthalten neben dem eigentlichen Prozessorkern interne Speicherblöcke und noch weitere Peripherie und sind damit streng genommen Microcontroller. Auf ausführliche Vergleiche der Leistungsfähigkeit der Prozessoren wurde hier verzichtet, da sich die Benchmarkproblematik im Bereich der eingebetteten Systeme weiter vergrößert und so ein sinnvoller Vergleich für den allgemeinen Fall sehr schwierig wird [91].

4.3.1 80386-Kompatible

Der 80386 von Intel ist ein 32Bit-CISC-Prozessor mit 4 Allzweckregistern und einigen weiteren Registern zur Speicheradressierung, die in Grenzen auch für “normale” Rechenaufgaben benutzbar sind. Der Befehlssatz ist nicht orthogonal, da er aus dem 16Bit-Befehlssatz des 8086 entstanden ist, der wiederum vom 8Bit-Prozessor 8080 abgeleitet wurde. Insbesondere “komplexere” Rechenbefehle (wie z.B. Schieben und Multiplizieren) sind nicht mit allen Registern durchführbar.

Die FPU-Architektur ist nicht registerbasiert, sondern beruht auf einem Wertestack mit 8 Einträgen, auf dem die Befehle üblicherweise implizit arbeiten. Dieses Vorgehen liegt darin begründet, dass die FPU zunächst nicht in den Prozessor integriert war sondern extern als Coprozessor angebunden war. Damit wird durch die Stackbefehle eine aufwendige Operandenübermittlung über den Systembus eingespart. Dies kommt zwar der Ausführungsgeschwindigkeit zugute, verkompliziert aber die Programmierung. Neuere Erweiterungen der Architektur (MMX, SSE) erlauben allerdings inzwischen eine registerbasierte Adressierung.

Für eingebettete Systeme sind die Desktop-Prozessoren der Intel-Pentium2/3/4-Serie oder der AMD-Athlon-Serie weniger geeignet. Der Leistungsverbrauch liegt teilweise über 80W und erfordert damit eine aufwenige Kühlung. Notebookprozessoren (z.B. Mobile Pentium) verbrauchen im Vergleich dazu weniger Leistung. Damit eignen sie sich für Systeme mit kleinem Formfaktor (z.B. PC/104), mit Einschränkungen sind sie auch im mobilen Betrieb nutzbar. Allerdings benötigen sie weiterhin einen sog. “Chipsatz”, der die CPU an den Speicher und weitere Peripherie anbindet.

Eine vollständige Integration einer 80586-kompatiblen CPU und des Chipsatzes stellt z.B. der AMD Elan SC520 [5] dar. Dieser Prozessor arbeitet mit max. 100MHz, besitzt MMU, FPU, SDRAM-Speichercontroller und einen PCI-Bus. Deshalb benötigt er nur zusätzlichen Speicher und ist damit ein vollständiges Prozessorsystem. Durch die Verfügbarkeit von bekannten Betriebssystemen wie z.B. Linux, sind die Einsatzgebiete dieses Systems vor allem Netzwerkkomponenten (z.B. WLAN-Access Points) und andere Anwendungen, für die ein “echter” PC zu groß wäre. Durch die Binärkompatibilität besteht die Möglichkeit, Anwendungen zunächst auf dem PC zu entwickeln und zu testen. Mit AMD Alchemy [4] oder dem National Geode [109] stehen weitere, noch wesentlich leistungsfähigere 80586-kompatible CPU mit bis zu 500MHz Taktfrequenz und diversen integrierten Peripherieeinheiten zur Verfügung.

4.3.2 ARM

Die am weitesten verbreiteten Prozessoren für eingebettete Systeme beruhen auf der ARM-Familie der Firma “Advanced RISC Machines”. Die Systeme basieren auf einer RISC-Architektur und sind sowohl als einzelne Prozessoren (z.B. ARM720, ARM920, StrongArm1100) erhältlich, als auch als “Cores” eingebettet in kundenspezifische Peripherie.

Die ARM-Architektur beruht aus einem RISC-Prozessor mit 32 32Bit-Allzweckregistern. Je nach Kern kommen Instruktions- und Datencache, MMU oder DSP-Erweiterungen hinzu.

Die Firma Psion benutzt den nicht MMU-fähigen ARM5-Prozessor in ihren auf dem Betriebssystem EPOC basierenden Organizern, in Mobiltelefonen sind Basisbandprozessoren ebenfalls sehr oft mit einem ARM-Kern ausgestattet (z.B. bei Nokia und Trium).

In vielen Windows-CE und Linux-basierten PDAs (HP/Compaq Ipaq, Sharp Zaurus) wird die von Digital (jetzt Intel) enwickelte ARM-Variante StrongArm SA-1100 [79] eingesetzt. Dieser Standardchip bietet neben ARM-Kern bereits “PDA-übliche” Peripherie wie LCD-Controller, USB-Device-Unterstützung und UARTs.

Der ARM-Befehlssatz hat eine Besonderheit, die ihn von anderen RISC-basierten System abhebt. Neben dem normalen Befehlssatz mit 32Bit Länge pro Befehl gibt es noch einen sog. Thumb-Code mit 16Bit pro Befehl. Dieser besitzt zwar einige Einschränkungen in der Zahl der adressierbaren Register, erlaubt jedoch platzsparenden Code bei nur geringfügig verminderter Leistung. Die Umschaltung des Modus geschieht bei Sprungbefehlen, somit können Unterroutinen in einem anderen Modus ablaufen.

Ein Beispiel für die Core-Integration ist der von Texas Instruments entwickelte AV7200 [155]. Dieser Chip ist für DVB-Anwendungen (Digital Video Broadcasting) als Controller für Settop-Boxen gedacht und integriert neben DVB-spezifischen Funktionen (Paketfilter, Entschlüsselungshardware, On-Screen-Display-Controller, Videoein/ausgänge) und MPEG-Video-Decoder einen ARM925-Core (mit MMU) als Hauptprozessor, einen ARM720-Core, einen TMS32C540-DSP und verschiedene weitere Peripherie (PCI/IDE-Controller, Timer, UARTs). Spätere Versionen sind mit der Jazelle-Erweitung (siehe 5.4.10) für den ARM925 geplant.

4.3.3 Motorola 68k

Aus der 68k-Linie von Motorola gingen Mitte der 80er Jahre eine Reihe von Prozessoren für eingebettete Systeme hervor. Unter den am häufigsten eingesetzten 68k-Derivaten sind der MC68331 und MC68360 [82]. Die gesamte 68k-Familie besitzt eine relativ einheitliche Architektur. Somit können Benutzerprogramme auf allen Prozessoren meist unverändert ablaufen, Systemprogramme benötigen nur kleine Anpassungen. Mit 8 Daten- und 7 Adressregistern zu je 32Bit ist eine sehr effiziente Programmierung möglich, dies wird durch komplexe Adressierungsarten sehr unterstützt. Prinzipiell haben alle Prozessoren der 68k-Familie einen Adressraum von 4GB, dieser kann aber durch die zur Verfügung stehenden Anschlüsse eingeschränkt sein (z.B. auf 16MB).

Der MC68331 ist ein auf dem “CPU32”-Kern aufbauender Prozessor, unterstützt also auch eine virtuelle Speicherverwaltung. Zusätzlich sind noch Befehle für effizienten Table-Lookup und Werteinterpolation vorhanden. Der CPU32-Kern wird über ein System Integration Module (SIM) an weitere interne Module (wie serielle Einheiten für SCI, SPI & UART, und eine Timereinheit) sowie an einen externen Bus angebunden. Über das SIM sind weitere Funktionen wie interne und externe Chipselects als auch ein sog. “Background Debugging Mode” zugänglich. Als Taktfrequenzen sind 16 bis 20MHz möglich, der Leistungsverbrauch beträgt dabei ca. 400mW bei 16MHz, also ca. 25mW/MHz.

Der im PalmPilot und anderen kompatiblen PDA-Systemen (Handspring Visor, Sony Clié) eingesetzte MC68328 (“Dragonball”) [106] besitzt neben dem 68EC000-Kern unter anderem zwei 16Bit Timer, drei serielle Schnittstellen

(UART und SPI), einen LCD-Controller und eine Logik zur flexiblen Speicher- und Peripherieansteuerung. Die Taktfrequenz wird über eine interne PLL erzeugt und reicht von 0 bis 32MHz.

Aufgrund der geringen Taktfrequenz und des unbeschleunigten Kerns (minimal 4 Taktzyklen pro Befehl) ist die Leistungsfähigkeit des 68328 geringer als die des 68331, zusätzlich fehlt für größere Systeme eine Unterstützung für virtuellen Speicher (MMU).

4.3.3.1 Motorola ColdFire

Motorola stellte 1994 die ColdFire-Architektur (MCF5xxx) vor [107], die, obwohl auf RISC basierend, dem Anwender ein dem 68k sehr ähnliches Programmiermodell zur Verfügung stellt. In vielen Fällen lässt sich existierender 68k-Sourcecode ohne Änderungen neu auf die ColdFire-ISA assemblieren [105].

Durch feste Befehlslängen (eines der Hauptprobleme bei der Weiterentwicklung der 68k-Familie) sind starke Performancesteigerungen möglich. Die Taktfrequenzen liegen zwischen 16 und 220MHz, die Leistungsdaten zwischen 44 und max. 316MIPs. Im Gegensatz zu den vorher besprochenen MC68328 und 68331 besitzen die MCF5xxx zusätzlich zu integrierten Peripheriebausteinen auch RAM (1KB-96KB) auf dem Chip und erleichtern so die Integration in eingebettete Systeme.

4.3.4 MIPS

Aus den Prozessoren für Workstations entwickelte die Firma MIPS Prozessorkerne, die als “Intellectual Property” (IP) in eigenen Schaltkreisen eingesetzt werden können. “Fertige” Prozessoren für den Embedded Bereich sind kaum erhältlich.

Die MIPS-Architektur besitzt mit 32 32Bit-Registern und dem Load/Store-Prinzip typische Eigenschaften eines RISC-Prozessors. Durch das Core-Konzept sind Erweiterungen auf 64Bit, DSP- und Verschlüsselungsfunktionen möglich [100]. Die Leistungsfähigkeit reicht dabei von ca. 100 bis 1000MIPS. Low-Power-Versionen erreichen einen Leistungsverbrauch von ca. 0.5mW/MHz und erlauben eine dem ARM-Thumb-Code ähnliche Code-Kompression [101].

4.3.5 Etrax-100LX von Axis

Ein Exot unter den Prozessoren für eingebettete Systeme ist die CRIS-Architektur (“Code Reduced Instruction Set”) mit dem Chip Etrax-100LX [24, 25] der Firma Axis. Von Beginn an als Prozessor für vernetzte Anwendungen wie Webcams entwickelt, hat der Etrax-100LX eine CPU mit RISC-ähnlicher Architektur, eine MMU, einen 100MBit-Ethernet-Controller und weitere Peripherie (IDE,

UARTs, I^2C, USB-Host und USB-Device) integriert. PDA-typische Hardware (LCD-Controller oder Touchscreencontroller) fehlt hingegen, ebenso eine integrierte FPU.

Der CPU-Kern besitzt 16 32Bit-Register, davon sind vierzehn Allzweckregister verfügbar. Die Befehle sind nur 16Bit breit, haben daher ähnlich der 68k-Familie nur ein Zwei-Adress-Format, und ähneln auch sonst sehr dem 68000.

Gegenüber anderen Systemen ist die Speichernutzung des Etrax eingeschränkt. Durch den externen 25Bit-Adressbus werden maximal 16MB SDRAM und 8MB Flash vom integrierten Speichercontroller unterstützt.

Für die CRIS-Architektur mit Etrax-100LX ist inzwischen eine Linux-Portierung verfügbar.

4.3.6 PowerPC

Die PowerPC-Familie (PPC) [72] entstand aus der POWER-Serie von IBM, die als Multichip-Prozessor für die RS/6000-Großrechner verwendet wird. Der PPC ist dabei eine auf eine Mikroprozessorumgebung angepasste RISC-Architektur, die den Motorola 88K Prozessorbus benutzt.

Der PPC-Kern besteht aus 32 32Bit Allzweckregistern und 32 64Bit Fließkommaregistern, die jedoch optional sind und wie auch die MMU bei eingebetteten PPC-Cores nicht immer vorhanden sind. Für einen RISC-Prozessor ungewöhnlich ist die Verwendung von speziellen, befehlsabhängigen Registern z.B. für Unterprogrammaufrufe.

Im Bereich der eingebetteten Systeme sind PPCs besonders bei intelligenten I/O-Karten z.B. im Telekommunikationsbereich verbreitet [20]. Ein weiterer Einsatzbereich ist die Integration in FPGAs (siehe Abschnitt 4.7).

4.3.7 Hitachi SuperH3/SuperH4

Die Hitachi SH3/SH4-Familie [68, 135] ist eine speziell für eingebettete Systeme entwickelte RISC-Architektur. Aufbauend auf den weniger leistungsfähigen SH1/SH2-Serien, bieten sie 16 32Bit Register, eine Memory Management Unit und verschiedene Peripherie auf einem Chip. Der maximal physikalisch adressierbare RAM-Bereich von 448MB kann bei virtuellem Speicher mit Hardwareunterstützung auf maximal 256 Adressräume verteilt werden.

Die Taktraten reichen von 60MHz (SH3) bis 200 MHz (SH4), bei 60MHz werden ca. 600mW (SH7708) verbraucht.

Die SH3/SH4-Prozessoren wurden in einigen Windows-CE kompatiblen PDAs eingesetzt und sind jetzt hauptsächlich in Multimedia-Geräten im Consumermarkt zu finden.

4.4 Betriebssysteme für eingebettete Systeme

Um den Entwicklungsaufwand bei komplexeren eingebetteten System in Grenzen zu halten, werden zunehmend Standardbetriebssysteme engesetzt, die teilweise speziell für dieses Anwendungsgebiet entwickelt wurden. Dies erlaubt auch modulare und abstrakte Implementierungen der Steuersoftware, die damit portabler und einfacher zu warten wird.

Da die Prozess-Steuerung (und damit ein wichtiges Einsatzgebiet von eingebetteten Systemen) vom Zeitverhalten des Systems abhängt, gibt es sog. Echtzeitbetriebssysteme [32]. Diese können durch ihr Design verschiedende Zeitanforderungen garantieren. Dabei werden bei Echtzeit zwei Anforderungen unterschieden:

- Harte Echtzeit (“Hard Real Time”)

  Wenn ein Prozess ein Ergebnis nicht bis zu einer vorher bestimmten Endzeit (“Strict Deadline”) geliefert hat, kann eine nicht akzeptable Verschlechterung der Systemleistung oder auch ein katastrophales Ereignis eintreten. Verletzungen der Deadline müssen daher auf alle Fälle vermieden werden.

- Weiche Echtzeit (“Soft Real Time”)

  Rechenergebnisse werden so schnell wie möglich geliefert, eine Verletzung bestimmter gewünschter Endzeiten führt nur zu kurzzeitigen annehmbaren Verlusten. Dies kann z.B. bei einer kurzzeitigen Überlastung eines Systems auftreten. Weiche Echtzeit darf nicht für sicherheitsrelevante Teile eines Systems benutzt werden, die definierte Antwortzeiten verlangen.

Abhängig vom Anwendungszweck sind auch komplexere Bedienoberflächen und deren Ein- und Ausgabemedien vom Betriebssystem zu unterstützen. Graphische Oberflächen (z.B. Qt/Embedded [159] oder Zinc [167]) benötigen i.A. mehr Speicher und mehr Unterstützung durch das Betriebsystem (z.B. Linux).

Drei der zur Zeit am verbreitetesten Betriebssysteme für komplexere eingebettete Anwendungen werden im folgenden beschrieben.

4.4.1 Pocket PC/Windows CE

PocketPC/WindowsCE [124] ist ein an die Windows-API angelehntes Betriebsystem von Microsoft für eingebettete Systeme. Als Betriebssystem bietet WindowsCE eine Multitasking-Umgebung mit Erweiterungen für den mobilen Einsatz wie z.B. IRDA.

Um eine weitgehende Architekturunabhängigkeit zu erreichen, werden alle vom Prozessor bzw. der Plattform abhängigen Betriebssystemteile in eine sog.

OEM Abstraction Layer (OAL) verlagert. Windows CE ist für Intel 80486 und kompatible, StrongArm, MIPS, PowerPC und SH3/SH4 verfügbar.

Neben dem Haupteinsatzgebiet als Plattform für PDAs rückt inzwischen auch die Nutzung als leichtgewichtiges Betriebssystem für andere eingebettete Anwendungen im industriellen Umfeld in den Vordergrund [78]. Durch das Systemdesign ist Windows CE allerdings nur für weiche Echtzeitbedingungen geeignet.

4.4.2 Linux

Für alle oben erwähnten 32Bit-Prozessoren mit MMU (bis auf Motorola ColdFire) ist Linux verfügbar, eine eingeschränkte Variante (ucLinux) existiert inzwischen auch für MMU-lose Systeme (z.B. MC68328). Damit unterstützt Linux von allen Betriebsystemen die meisten CPU-Architekturen und erleichtert die Auswahl für eine bestimmte Anwendung.

Ein wesentlicher Vorteil von Linux gegenüber anderen Betriebssystemen ist die individuelle Anpassung auf die jeweilige Zielarchitektur. Alle für ein System wichtigen Funktionen (virtueller Speicher, Prozesse und Prozessinterkommunikation, Netzwerk etc.) sind dabei schon vorhanden, andere (z.B. Festplattenansteuerung über IDE) können je nach Einsatzzweck in den Kernel compiliert werden.

Durch die Einheitlichkeit von Linux auf dem Entwicklungssystem und der Zielplattform wird die Entwicklung stark vereinfacht. Anwendungen können zuerst auf dem Entwicklungsystem einfach getestet werden, bevor sie für die Zielplattform compiliert werden. Auch dort ist durch den Einsatz von Standardentwicklungswerkzeugen (Shells, gdb etc.) eine einfache Fehlersuche möglich.

Linux ist aus seiner Entwicklungsgeschichte heraus zunächst nicht für harte Echtzeitbedingungen entwickelt und geeignet, allerdings gibt es einige Erweiterungen für Echtzeitsysteme (z.B. RTLinux [58]).

4.4.3 VxWorks

VxWorks [169] von WindRiver ist ein schlankes und netzwerkfähiges Echtzeitbetriebssystem, das intern mit einem Microkernel arbeitet. Auf dem Kern mit deterministischem Taskscheduling und verschiedenen Interprozesskommunikationssystemen bauen je nach Anwendung weitere Module für Netzwerk, virtuellen Speicher, Dateisystem etc. auf. Das System ist in weiten Grenzen an die Hardware anpassbar und unterstützt auch SCSI und PCMCIA.

Als mögliche Zielplattformen stehen alle wichtigen Prozessoren für eingebettete Systeme (PPC, ColdFire, MC68k, 80586, Arm, MIPS und SuperH) zur Verfügung. Mit JWorks [168] ist ebenfalls eine Java-Umgebung vorhanden, sie basiert auf Jeode von Insignia (siehe auch Abschnitt 5.3.10. Um Speicherplatz im RAM

zu sparen, ist es mit JWorks möglich, JVM-Bytecode direkt aus dem ROM auszuführen.

4.5 FPGA Grundlagen

4.5.1 FPGA-Architekturen

Field Programmable Gate Arrays (FPGAs) sind Logikbausteine, deren interne Verschaltungen und damit die implementierte Logik bei der Herstellung noch nicht festgelegt sind, sondern erst nachträglich durch den Anwender des FPGAs bestimmt werden. Im Unterschied dazu stehen kundenspezifische Schaltkreise, die entweder aus Standardzellen erzeugt werden ("Application Specific IC", ASIC) oder vollständig selbst entwickelt werden ("Full Custom").

Entstanden sind FPGAs als Weiterentwicklung der 1977 von Monolitic Memories entwickelten "Programmable Array Logic" (PAL). Diese PALs basieren auf einer programmierbaren Schaltmatrix für AND und OR-Terme, damit lassen sie sich als allgemeine technische Umsetzung von disjunktiver oder konjunktiver Normalform (DNF/KNF) mit einer begrenzten Zahl von OR/AND-Termen begreifen. Üblicherweise sind damit 10-16 Eingänge und 8-12 Ausgänge realisierbar. Je nach PAL-Typ ist es möglich, in den Ausgang dieser DNF/KNF-Berechnungen einen Inverter oder ein Flipflop einzuschalten. PALs waren zunächst nur einmal programmierbar, da sie auf Fuse bzw. Anti-Fuse-Technik basierten, d.h. auf internen Verbindungen, die dauerhaft zerstört bzw. kurzgeschlossen werden konnten.

Die Firma Lattice entwickelte aus den PALs die sog. GALs (Generic Array Logic), die einerseits durch EEPROM-Technik wiederprogrammierbar waren, andererseits auch eine wesentlich flexiblere interne Struktur besonders für die Ausgangslogik (Makrozellen) hatten. Somit konnte ein GAL-Chip viele PAL-Varianten ersetzen.

Das PAL/GAL-Grundprinzip der AND/OR-Matrix ist heute immer noch bei den sog. CPLDs im Einsatz, aber mit wesentlich mehr Ein- und Ausgängen und flexibleren Verschaltungen der Matrizen untereinander. Diese CPLDs basieren alle auf EEPROM-Technik, sind also nicht-flüchtig und müssen daher nur einmal programmiert werden, sind aber (auch teilweise **i**m **S**ystem **p**rogrammierbar, **ISP**) zu löschen und neu zu beschreiben. Typische moderne Vertreter sind die isp-Bausteine von Lattice oder die Coolrunner-Serie von Xilinx.

Einsatzgebiet der CPLDs ist üblicherweise der Ersatz von "GlueLogic", also z.B. Adressdekoder oder einfache Zustandsautomaten. Durch die festgelegte interne Struktur sind die Verzögerungszeiten genau vorhersagbar und ermöglichen damit ein einfaches Design. Nachteilig an CPLDs ist die Fixierung auf die Abbildung einer DNF in Hardware und die starke Bindung der Ergebnisse an Ausgangspins.

FPGAs sind im Gegensatz zu CPLDs wesentlich feinkörniger strukturiert. Grundlage sind Logikzellen (Logik Cell, LC bzw. CLB, Configurable Logic

Block) mit relativ wenig Komplexität, je nach Hersteller eine Kombination von AND/OR und Multiplexern oder direkt eine Wertetabelle (**L**ook**u**p**t**able, LUT) mit 3-6 Eingangsgrößen. Jede LC beinhaltet dabei zusätzlich zur kombinatorischen Logik noch ein bis zwei Flipflops oder Latches. Eine typischer Innenaufbau eines CLBs von Xilinx ist in Abb. 4.1 gezeigt.

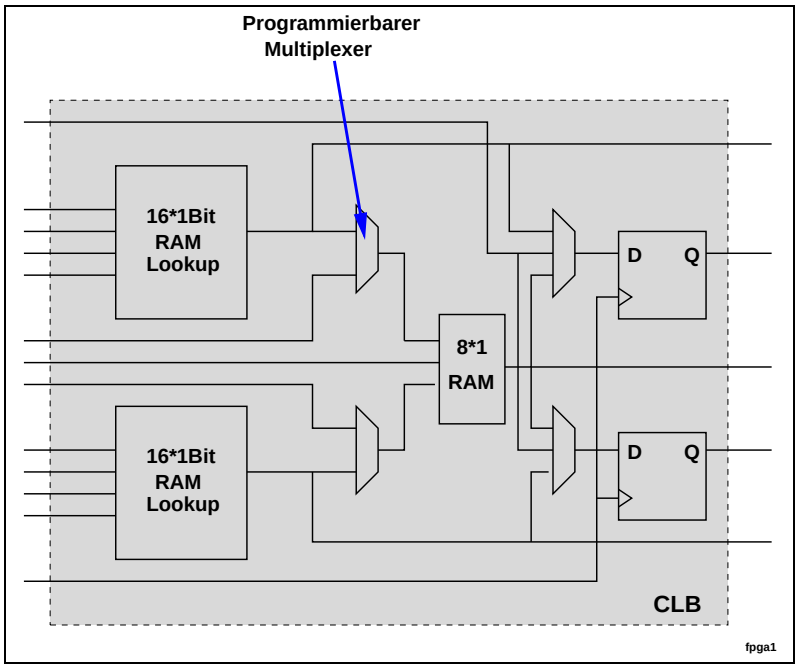

Abbildung 4.1: Typische FPGA-Logikzelle

Die Logikzellen sind im FPGA in großer Anzahl in einer Matrix angeordnet und von spezialisierten IO-Zellen umgeben. Übliche Matrizengrößen reichen von 8*8 (64 Logikblöcke, XC2064, 1984) bis zu 112*104 (46592 Logikblöcke, XC2V8000, 2002). Ein Vergleich der Komplexität mit ASIC-Gatter-Äquivalenten ist dabei schwierig. Die größten zur Zeit erhältlichen FPGAs werden von den Herstellern mit ca. 8 Millionen ASIC-Gattern angegeben.

Die Verschaltung der Logikzellen geschieht über eine Hierarchie von schaltbaren Verbindungen (Interconnects) zwischen den Zellen (siehe Bild 4.2). Die Hierarchie geht von schnellen Verbindungen zwischen Zellen und auch spezieller Carry-Logik bis zu globalen Taktsignalen oder Leitungen für interne Busse.

Zusätzlich zu den flexiblen Logikzellen sind noch andere, feste Bausteine auf dem FPGA integrierbar . Das sind z.B. größere RAM-Blöcke, Multiplizierer, PLLs für die Taktverteilung oder sogar Prozessor-Kerne (siehe Abschnitt 4.7).

Das erste FPGA (XC2064 mit 64 einfachen Logikzellen) mit dem beschriebenen Aufbau wurde 1984 von Xilinx entwickelt und arbeitete mit Systemtakten von max. 15-25MHz.

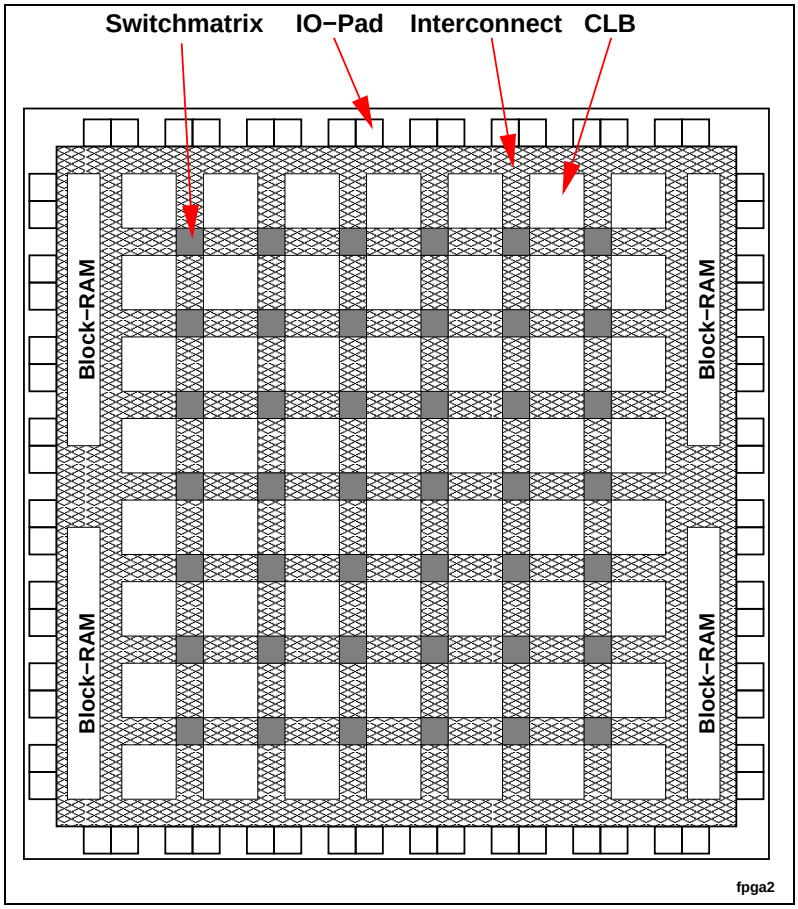

Abbildung 4.2: Aufbau eines FPGAs

Die Programmierung der LUTs und der Verbindungsstrukturen ist auf verschiedene Weise möglich. Die schnellste Technik ist die Verwendung von Anti-Fuse-Zellen zur Steuerung, allerdings sind diese nur einmal programmierbar. Durch den niedrigen Innenwiderstand der Verbindungsstrukturen sind die Verzögerungen relativ gering. Die feste und nicht reversible Programmierung bietet auch den Vorteil, wesentlich weniger anfällig gegen elektromagnetische Störungen zu sein (EMV/EMC) und bietet weniger Angriffsfläche für Reverse-Engineering. Insgesamt sind FPGAs mit Anti-Fuse-Programmierung damit ASICs am nächsten. Hersteller von Anti-Fuse-FPGAs sind z.B. Actel und Quicklogic. Während EEPROM-Speicher bei CPLDs weit verbreitet sind, wird diese Technik bei FPGAs kaum eingesetzt.

Am häufigsten wird als Speichersystem bei FPGAs statisches RAM benutzt. Auf jeder programmierbaren Verbindung bzw. in jeder LUT sind dann eine oder mehrere SRAM-Zellen angeordnet. Nach dem Anlegen der Versorgungspannung

muss das FPGA also erst mit der Logikkonfiguration geladen werden, was somit den Systemstart verzögert. Auf der anderen Seite kann ein SRAM-basiertes FPGA jederzeit neu geladen werden (“Rekonfiguration”) und erlaubt damit besonders einfache Entwicklung und auch Updates im System.

Das Laden der SRAM-FPGAs erfolgt typischerweise über eine bitserielle Kommunikation oder über einen parallelen Datenbus entweder aktiv vom FPGA aus oder passiv durch eine andere Logik (z.B. Mikroprozessor).

4.5.2 Speicherkonzepte in FPGAs

Im Gegensatz zu ASIC- oder Full-Customschaltungen, wo statischer Speicher in relativ beliebigen Größen implementiert werden kann, muss dieser auf FPGAs schon von Anfang an vorgesehen sein. Eine Implementierung von RAM mit den Flipflops der Logikzellen ist sehr ineffizient und nur dann sinnvoll, wenn ungewöhnliche Speicherstrukturen (Mehrwege-Zugriff etc.) benötigt werden.

Ein auch von anderen Herstellern verwendetes Konzept der FPGA-Speicheraufteilung stellt SelectRAM von Xilinx dar, das eine Art Speicherhierarchie mit mehreren Typen von Speicher implementiert.

Die kleinste Einheit ist dabei die LUT (Funktionstabelle) der Logikzelle. Neben der nur lesenden Funktionalität für eine Wertetabelle ist sie auch FPGA-intern beschreibbar und wird somit zu RAM (“Distributed RAM”). Übliche Größen sind dabei 16*1Bit, 16*2Bit, oder 32*1Bit. Als Zusatzfunktionalität stehen synchrones (getaktetes) Schreiben oder die Möglichkeit der Nutzung als Dual-Ported-RAM zur Verfügung. Größere oder breitere Speicher benötigen eine Vordekodierung und Kaskadierung bzw. mehr parallel geschaltete Logikzellen.

Nachdem jede LUT auch RAM sein kann, sind die Wege zwischen Logik und Speicher sehr kurz und es sind Geschwindigkeiten von mehreren 100MBit/s pro Datenleitung zu erreichen. Allerdings benötigen größere Speicherbereiche auch dementsprechend viele Logikzellen und werden durch die größere zu verbindende Fläche und die Kaskadierungen langsamer.

Als Option für größere oder breitere Speicher sind sog. BlockRAMs vorgesehen. Diese haben Kapazitäten zwischen 4Kbit und 16Kbit und sind mit Breiten von 1 bis 18Bit und als Dual-Ported-Ram nutzbar. Je nach Chipgröße sind zwischen 4 (9KByte) und 168 (378KByte) dieser RAM-Blöcke auf einem FPGA integriert. Sowohl Distributed RAM als auch BlockRAM können bei der FPGA-Konfiguration mit Startwerten beschrieben werden. Damit sind sie auch als ROM z.B. für mikroprogrammierte Abläufe einsetzbar.

4.5.3 Vor- und Nachteile von FPGAs

FPGAs sind gut geeignet für die Entwicklung von anwendungsspezifischen Schaltkreisen, da sie durch den unkomplizierten Entwicklungsfluss das “Rapid Prototyping” gut unterstützen. Dabei werden SRAM-basierte FPGAs eingesetzt, da sie beliebig oft programmierbar sind und so teure Herstellungsläufe einsparen.

FPGAs sind in großen Stückzahlen zwar deutlich teurer als ASICs, bieten aber bei kleineren Stückzahlen (bis ca. 10000 Stück) den Vorteil der fehlenden ASIC-Grundkosten (Non Recurring Investment, NRI). FPGAs erlauben in der Anfangsphase eines Produkts die Änderung (Updates) der Schaltungslogik ohne erneute Investitionen, und vermeiden damit aufwendige und teure Fehlerbehebungen.

Im Gegensatz zu ASICs sind höher integrierte SRAM-basierte FPGAs nicht für Low-Power-Anwendungen geeignet, da Leckströme auch in nicht benutzten Gattern und Verschaltungen fließen. Andererseits kann durch die Anpassung der Logik an die benötigte Anwendung und die mögliche Parallelität im FPGA ein großer Geschwindigkeitsgewinn gegenüber Standard-Prozessoren erreicht werden. Diese Vorteile treten besonders in den Bereichen der digitalen Signalverarbeitung auf, wo einzelne, spezielle Lösungen bei vergleichbaren Preisen teilweise 50-100mal schneller sind als die Implementierung auf einem DSP.

4.6 Rekonfigurierbare Systeme mit FPGAs

4.6.1 Grundlagen und Klassifikation

Da die Logikfunktionen und die interne Verschaltung von SRAM-basierten FPGAs allein durch den Inhalt des FPGA-internen-Konfigurationsspeichers bestimmt wird, ist es möglich, durch Änderungen dieses Speichers die Funktionalität des FPGAs neu zu definieren. Dieser Vorgang wird als “Rekonfiguration” bezeichnet. Je nach FPGA-Architektur sind dabei folgende Arten der Rekonfiguration anwendbar:

- Totale Rekonfiguration

  In diesem Fall wird der gesamte Konfigurationspeicher ausgetauscht. Üblicherweise muss dazu der Speicher erst gelöscht werden, bevor er neu beschrieben werden kann. Dies führt zu einem Totalausfall der FPGA-Funktion während der Rekonfiguration. Je nach Größe und Art der Konfigurationsschnittstelle liegt diese Zeit zwischen einigen Millisekunden und einigen Sekunden. Dieser Ausfall muss vom System toleriert werden. Dadurch darf das FPGA keine systemrelevanten Funktionen beinhalten, wie z.B. Speichercontroller.

Sind solche Funktionen im FPGA integriert, darf so eine Rekonfiguration typischerweise nur zu Startzeiten des Systems erfolgen.

- Partielle Rekonfiguration

  Einige FPGAs erlauben das teilweise Beschreiben des Konfigurationsspeichers, während die normalen Logikfunktionen unverändert weiter laufen. Die Granulariät der partiellen Rekonfiguration hängt von der FPGA-Architektur ab. In der Xilinx 6K-Serie ist jede Speicherzelle individuell adressierbar. In den Bausteinen der Virtex-Serie ist es dagegen nur möglich, eine ganze Spalte umzukonfigurieren.

  Mit der partiellen Rekonfiguration kann die Grundfunktionalität des FPGA erhalten bleiben, während die Rekonfiguration abläuft. Problematisch an diesem Modell ist allerdings die immer noch fehlende Integration der Rekonfigurationsmöglichkeiten in den normalen FPGA-Entwicklungsfluss.

Diese beiden grundsätzlichen Arten der Rekonfiguration können zusammen mit dem Aufbau und der Verschaltung mehrerer FPGAs in einem Gesamtsystem benutzt werden. Diese Systeme lassen sich ebenfalls in unterschiedliche Klassen einordnen. Eine mögliche Klassifizierungsart (Olymp [130]) bezieht dabei die Art und Flexibilität der FPGA-Verbindungen und die Granularität der FPGA-Funktionen in das Schema mit ein.

Ein Hauptproblem der Rekonfiguration ist ein effizientes Management der verschiedenen Rekonfigurationsteile sowohl während der Logikerzeugung und Entwicklung als auch während der Laufzeit. Gerade letzteres bedingt zusätzliche Logik im FPGA selbst oder im konfigurierenden System, um die zeitaufwendige Rekonfiguration möglichst selten und ohne Beeinträchtigung des Systems durchzuführen. Die dafür nötigen Strukturen sind ähnlich denen von Caches oder virtuellem Speicher, lassen sich aber nicht ohne weiteres übertragen. In [151] werden beispielsweise einige Strategien vorgestellt, um diese Schwierigkeiten bei dynamisch wechselnden Konfigurationen zu umgehen. Wie auch bei herkömmlichen Cache-Ersetzungsalgorithmen sind die Ergebnisse stark von der eingesetzten Anwendung abhängig, eine allgemeingültige Lösung des Problems ist daher nicht möglich.

4.6.2 Anwendungen

Haupteinsatzgebiet von rekonfigurierbaren Architekturen sind adaptive, rechenzeitintensive Anwendungen [162], die von der speziell für den Anwendungszweck entwickelten Logik profitieren. Durch die Flexibilität wird allerdings auch eine Änderung des ansonsten im Vergleich zu Software relativ starren Hardwareentwurfs verlangt [66].

Eine der ersten dokumentierten Anwendungen der totalen Rekonfiguration von SRAM-basierten FPGAs ist in [152] beschrieben. Das FPGA dient dabei zur visuellen Flugsteuerung einer taktischen Lenkwaffe. Der Teil der Bilderkennung, die im FPGA abläuft, wird dabei je nach Bodenbeschaffenheit (Wasser bzw. Land) und Missionsauftrag vor dem Start rekonfiguriert.

Das Firefly-Projekt [103] besteht aus FPGAs, die einen zellulären Automaten simulieren, der Einsatzbereich ist die Forschung im Bereich von genetischen Algorithmen. Durch die erzielbare Parallelität der FPGA-Berechnungen arbeitet diese Simulation wesentlich schneller als entsprechende Algorithmen auf einem PC.

Ähnliche FPGA-Systeme im Bereich der neuronalen Netze und genetischen Algorithmen sind FAST [123] und DECPeRLe-1 [53].

4.7 FPGAs mit integrierten Mikroprozessoren

Neben der Möglichkeit, eine vollständig selbst entwickelte Prozessoren auf einem FPGA zu integrieren, bieten mehrere FPGA-Hersteller inzwischen fertige “Soft-Cores” bzw. fest in die FPGA-Hardware implementierte Prozessor-Kerne (“CPU-Cores”) an.

Altera verfügt mit dem Soft-Core Nios [11] eine in das FPGA einbindbare CPU, die je nach Anforderung 32Bit oder 16Bit besitzt. Die CPU-Architektur ist proprietär und basiert auf einem RISC-Ansatz mit einer fünf-stufigen Pipeline. Da die CPU aus den FPGA-Logik-Ressourcen erstellt ist, ist es möglich, über eine Erweiterung der ALU eigene, hardwarebeschleunigte Funktionen zu integrieren (“Custom Instruction Logic”). Die maximale CPU-Taktfrequenz ist vom Place&Route-Zyklus abhängig und ist typischerweise größer als 125MHz.

Die Virtex2Pro-Serie von Xilinx [175] hat je nach Chip-Größe einen bis vier PowerPC-Kerne direkt integriert, die je 16+16KB Instruktions- und Datencache und eine MMU (aber keine FPU) besitzen. Die maximale Taktfrequenz beträgt 400MHz. Diese PPC-Kerne können aus dem FPGA-internen RAM betrieben werden und benötigen damit keinerlei externe Beschaltung. Durch die freie Verdrahtbarkeit ist es aber auch möglich, die CPUs über einen Speichercontroller im FPGA an externen Speicher anzubinden. Diese FPGA-Architektur ist damit ideal für Steueraufgaben im Embeddedbereich, da sie eine externe CPU überflüssig macht.

4.8 Vergleich der Rekonfigurierbarkeit von FPGAs

4.8.1 Motivation

Da das in dieser Arbeit entwickelte JVM-Beschleunigerkonzept in einem FPGA ablaufen soll, besteht die Möglichkeit, die Rekonfigurationseigenschaft im System auszunutzen. Inwieweit dies grundsätzlich sinnvoll ist, soll im folgenden untersucht werden.

Wird die Eigenschaft der Rekonfigurierbarkeit von FPGAs in einer Anwendung benötigt, so stellt dies gegenüber der normalen FPGA-Anwendung erheblich erweiterte Anforderungen an das FPGA. In vielen Anwendungen ist die partielle Rekonfiguration beispielsweise nicht notwendig, es reicht das Neuladen des gesamten Chips aus.

Je nach Anwendungsbereich kann die benötigte Zeit zur partiellen oder totalen Rekonfiguration besonders wichtig werden. Als Beispiele seien hier aufgeführt:

- FPGA steuert dynamisches RAM (DRAM)

  DRAM benötigt in regelmässigen Abständen einen sog. Refresh, um den Inhalt beizubehalten. Dieser wird üblicherweise vom Memorycontroller ausgelöst. Übernimmt eine FPGA diese Aufgabe, so fällt bei einer totalen Rekonfiguration dieser Refresh aus und der DRAM-Inhalt kann korrumpiert werden. Bei einer partiellen Rekonfiguration tritt dieses Problem abhängig von der Funktion ebenfalls auf.

- FPGA ist Verbindungslogik (Gluelogic)

  Eines der typischen Anwendungsgebiete von FPGAs ist die Integration verschiedenster, verstreuter Logik im System, die ansonsten über niedrig integrierte Bauteile (z.B. 74er-Serie, GALs) gelöst werden muss. Ein Ausbleiben dieser Funktionen betrifft das ganze System und schränkt es und seine Benutzbarkeit während der Rekonfigurationszeit stark ein.

- FPGA zur Signalverarbeitung (DSP)

  In diesem Fall wird das FPGA zur beschleunigten Abarbeitung von Signalströmen benutzt, also als ein spezialisierter DSP. In bestimmten Anwendungsgebieten (z.B. Wireless Networks) vermindert ein Ausfall auch die gesamte Kommunikationsfähigkeit. Je nach Protokoll wird bei längeren Ausfällen eine aufwendige Neusynchronisation notwendig.

Die Wichtigkeit um das Wissen der Rekonfigurationsdauer ist nun klar, allerdings fällt es auf, dass es im Gegensatz zur Effizienz der FPGA-Logik mit dem Vorschlag des PREP-Benchmarks[127] von den FPGA-Herstellern keine

vergleichbaren Angaben dazu gibt. Damit ist es sehr schwer, für eine gegebene FPGA-Anwendung, die auf Rekonfiguration aufsetzt, den passenden Baustein auszuwählen, bzw. abzuschätzen, wie stark eine Rekonfiguration das System beeinträchtigt.

Es ist offensichtlich, dass ein exakter Vergleich durch die verschiedenen Architekturen nicht möglich ist. Trotzdem stellt sich die Frage, ob und wie Abschätzungen für ein Maß der "Rekonfigurierbarkeit" zu erreichen sind.

Im folgenden wird ein System erarbeitet, dass zumindest eine grobe Metrik erstellt, die einen Vergleich auch von partiell rekonfigurierbaren FPGAs mit nur total rekonfigurierbaren FPGAs erlaubt.

Dabei wird als Grundlage für das Maß der Rekonfigurierbarkeit die Geschwindigkeit der Rekonfiguration selbst benutzt. Als bester Fall wird eine Rekonfiguration angenommen, die genauso schnell erfolgt, wie die Ablaufgeschwindigkeit in einem FPGA. Würde das FPGA also mit 10MHz arbeiten, wäre das Optimum eine vollständige Rekonfiguration mit ebenfalls 10MHz. Offensichtlich ist das kaum realisierbar, dies ist nur mit mehreren Rekonfigurationscaches[4] auf dem FPGA zu erreichen und ist ebenfalls Gegenstand der Forschung. Ansätze dazu sind z.B. in [31] beschrieben. Dennoch stellt dieser Zustand (dessen Erreichbarkeit natürlich von der gewünschten Taktfrequenz abhängt) die obere Grenze dar. Eine schnellere Rekonfiguration ist nicht sinnvoll, da sie zwischen zwei Takten keinen Einfluss hat.

Die Geschwindigkeit der Rekonfiguration wird dann von der Größe des "Programms" bestimmt, das in das FPGA geladen werden muss. Die Ladegeschwindigkeit (zunächst eine Schreibgeschwindigkeit wie Bit/s) muss dabei für einen sinnvollen Vergleich auf die Ressourcen des FPGAs umgerechnet werden.

4.8.2 Vergleichsprinzip

Die erhältlichen FPGAs besitzen sehr verschiedene Architekturen, variierende Komplexität von Logikzellen und Verdrahtungsmöglichkeiten. Beispielsweise enthält eine Logikzelle eines XC6200-FPGAs nur wenige Multiplexer und ein Flipflop, die Logikzelle eines XC4000-FPGAs dagegen drei Lookup-Tabellen und zwei Flipflops. Zusätzlich können die Tabellen als RAM benutzt werden.

Ohne eine spezielle Anwendung und deren architekturspezifische Abbildung auf die Logikzellen kann also nur ein sehr grober Vergleich von benötigten Logik-Ressourcen erfolgen. Aus demselben Grund ist es auch nicht durchführbar, die ebenfalls benötigten Verdrahtungsressourcen (Interconnect) zu vergleichen. Da-

[4] Bei diesem Ansatz sind mehrere Konfigurationsspeicherzellen umschaltbar auf eine FPGA-Funktionseinheit gelegt

mit ist es also insgesamt nicht möglich, die benötigte Speicherkapazität (in Bits) für die Rekonfigurationsdaten zu vergleichen.

Um dennoch einen Anhaltspunkt zu erhalten, wird die individuelle Verdrahtung in der Betrachtung ignoriert und nur eine Logikzelle mit durchschnittlicher Verdrahtung als Berechnungsgrundlage benutzt. Wie später ersichtlich wird, sind die Ergebnisse für verschiedene FPGAs so unterschiedlich, dass eine Anpassung der verschiedenen Komplexitäten der Logikzellen gar nicht benötigt wird.

4.8.3 Zeit für Rekonfiguration einer Logikzelle

Als Ausgangsdaten für die Zeitdauer der Rekonfiguration **einer** Logikzelle werden folgende Werte benötigt:

- Die gesamte Menge an Rekonfigurationsspeicher im FPGA (n_{mem})
- Die Breite des Konfigurationsdatenbusses (w_{bus})
- Die Anzahl der Logikzellen (n_{lc})
- Die höchste Schreibfrequenz auf dem Konfigurationsdatenbus (f_{reconf})

Unter der Annahme, dass n_{mem} nur interne Logik und Interconnect beschreibt (also die Anzahl von IO-Zellen $<<$ n_{lc}), ergibt sich die durchschnittliche Anzahl von Schreibzugriffen auf den Konfigurationsbus für die Konfiguration **einer** Logikzelle $n_{lc.reconf}$ wie folgt:

$$n_{lc.reconf} = \frac{n_{mem}}{w_{bus} \times n_{lc}}$$

Unter Zuhilfenahme dieser Zahl und der Schreibfrequenz für den Konfigurationsbus ergibt sich die mittlere Zeit für die Rekonfiguration einer Logikzelle ($t_{lc.reconf}$):

$$t_{lc.reconf} = \frac{n_{lc.reconf}}{f_{reconf}} = \frac{n_{mem}}{w_{bus} \times n_{lc} \times f_{reconf}}$$

Diese ermittelte Zeit wird nun für den Vergleich der verschiedenen FPGAs benutzt.

4.8.4 Vergleich der Rekonfigurationsgeschwindigkeit

Die Interpretation von $t_{lc.reconf}$ ist nur dann sinnvoll, wenn sie (wie bereits angedeutet) mit der Ausführungsgeschwindigkeit (Systemtakt) der Schaltung in Verbindung gebracht werden kann. Allerdings hängt der Systemtakt wieder stark

von der Anwendung und ihrer Implementierung ab. Zwischen dem maximalen Takt eines FPGAs in verschiedenen Konfigurationen können Größenordnungen liegen. Daher ist eine allgemeinere Referenz nötig, die aber leicht aus den FPGA-Eigenschaften ableitbar sein sollte.

Ein möglicher Wert ist die höchste Frequenz der Flipflops einer Logikzelle (Togglerate f_{max}. Da hierbei keine Verbindungs- und Logikverzögerungszeiten eine Rolle spielen, ist f_{max} wesentlich höher als der höchste mögliche Systemtakt (f_{system}). Für normale Anwendungen ist der Wert f_{max} in keinster Weise ein geeigneter Wert für Geschwindigkeitsbetrachtungen [5]. Allerdings ist f_{max} ein Leistungsindikator, der bei allen FPGA-Architekturen entweder direkt oder indirekt aus der minimalen Taktzeiten errechnet werden kann.

Ein einfacher Weg um f_{system} aus f_{max} abzuschätzen, ist die Einführung eines Korrekturfaktors r_{system}. Unter der Annahme, dass Interconnect- und Logikverzögerungen für eine Logikebene in den meisten FPGAs ungefähr der Zykluszeit von f_{max} entspricht, wäre ein Korrekturfaktor für 4 Logikebenen $r_{system} = \frac{1}{4}$.

Die Wahl von 4 Logikebenen ist willkürlich, und damit auch r_{system}, trotzdem passt dieser Faktor aus eigener Erfahrung bei vielen Anwendungen relativ gut. Wie ebenfalls später noch sichtbar wird, beeinflusst die Wahl von r_{system} die gewonnenen Werte nicht wesentlich, daher ist die Wahl relativ unkritisch.

Wenn nun $t_{lc.reconf}$ in Beziehung zu f_{system} gesetzt wird, ist das gewonnene Verhältnis ($r_{lc.reconf}$) eine Abschätzung dafür, wie schnell die dynamische Rekonfiguration im Vergleich zur Geschwindigkeit der implementierten Logik selbst vollzogen werden kann.

$$r_{lc.reconf} = \frac{1}{t_{lc.reconf} \times f_{system}} = \frac{1}{t_{lc.reconf} \times r_{system} \times f_{max}}$$

Ist $r_{lc.reconf} = 1$, ist mit jedem Systemtakt eine Logikzelle rekonfigurierbar. Für $r_{lc.reconf} < 1$, sind mehrere Schreibzugriffe nötig, um eine Zelle neu zu konfigurieren. Je größer $r_{lc.reconf}$ also ist, umso besser ist die Rekonfigurierbarkeit des FPGAs.

4.8.5 Vergleich verschiedener FPGAs

Als Beispiel für die Anwendung obiger Formel wird die Rechnung mit dem Baustein XC6264-2 [173] [6] von Xilinx durchgeführt. Dieser besitzt 16384 Logikzellen, die insgesamt 192KByte Konfigurationsdaten entsprechen. Der höchste Re-

[5] Xilinx spezifizierte f_{max} in den ersten FPGA-Serien XC2000/XC3000 als Geschwindigkeitsangabe, wechselte später aber zur Verzögerung einer Logikzelle (t_{pd}) für eine bessere Unterscheidung. Bei SpartanII bzw. Virtex-FPGAs ist f_{max} aus rein ausfuhrrechtlichen Gründen mit 250MHz angegeben.

[6] Der Chip ist inzwischen nicht mehr lieferbar

	Altera EPF10K20-3[1]	Atmel AT40K20-2[1]	Vantis VF1020-3[1]
Konfiguration	Byte parallel	2 Byte parallel	Bit serial
Partiell rekonfigurierbar?	nein	ja	nein
n_{lc}	1152	1024	1296
n_{mem} (Byte)	28941	19279	≈37000
w_{bus} (Byte)	1	2	0.125
f_{reconf}	6MHz	40MHz	10MHz
f_{max}	125MHz	125MHz[2]	167MHz
f_{system}	31MHz	31MHz	42MHz
$n_{lc.reconf}$	25	9.4	228
$t_{lc.reconf}$	4.2μs	235ns	22.8μs
$r_{lc.reconf}$	0.008	0.13	0.001

[1] Daten, die nicht in den Datenblättern ersichtlich waren ([10], [19], [160]) wurden vom technischen Support dieser Firmen erfragt.
[2] geschätzt.

Tabelle 4.2: Vergleich verschiedender FPGA-Architekturen von Altera, Atmel und Vantis

konfigurationstakt ist 33MHz bei einer Busbreite von 32Bit. Die Togglerate wird von Xilinx mit 33MHz spezifiziert, mit einem gewählten $r_{system} = \frac{1}{4}$ ergibt sich dann ein Systemtakt von $f_{system} = 55$MHz.

Damit ergeben sich $t_{lc.reconf} = 90$ns und $r_{lc.reconf} = \frac{1}{5}$, d.h. bei einem Systemtakt von 55MHz sind 5 Takte notwendig, um **eine** Logikzelle zu rekonfigurieren. Selbst mit einem langsamen Takt von 10MHz, kann nur eine Logikzelle in jedem Takt geändert werden. Mit diesem $r_{lc.reconf}$ ist es zum Beispiel unmöglich, die gesamte Logik eines Datenpfadknotens (für 32Bit mindestens 32 Logikzellen) auszuwechseln, um eine kontinuierliche Adaption mit jedem einzelnen Takt zu erreichen.

Die Werte weiterer FPGAs sind in den Tabellen 4.2 und 4.3 aufgeführt.

Die sich ergebenden Werte für $r_{lc.reconf}$ zeigen, dass keines der FPGAs auch nur annähernd die “magische” Grenze von einer Logikzelle pro Systemtakt erreicht.

Werden die erhaltenen Daten als Graph $r_{lc.reconf}(f_{system})$ gegen f_{system} aufgetragen, wird in Abb. 4.3 sichtbar, dass fast alle FPGAs (außer XC62xx und AT40K) eine sehr langsame Rekonfigurationsrate haben. Um eine Logikzelle pro Takt zu rekonfigurieren, sind nur Systemtakte zwischen 20kHz und 200kHz möglich.

Die beiden partiell rekonfigurierbaren FPGAs (XC62xx und AT40K) schneiden in dieser Beziehung am Besten ab, da sie wesentlich breitere Datenbusse und kleinere Logikzellen haben. Auf der anderen Seite ist die XC4000E-Serie das langsamste FPGA in der Rekonfiguration, da es eine langsame Rekonfigurationsmethode[7] und eine sehr komplexe Logikzelle besitzt.

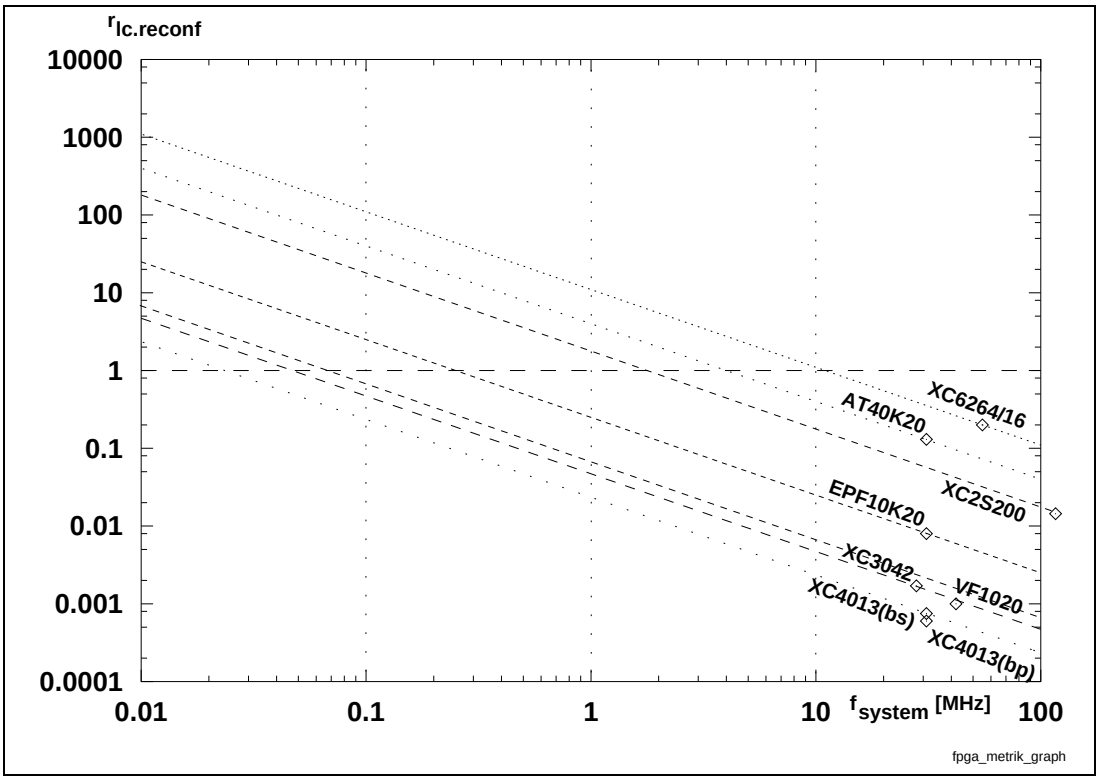

Abbildung 4.3: Vergleich von $r_{lc.reconf}$ bezogen auf den Systemtakt

4.8.6 Schlussfolgerungen

Der obige Vergleich belegt eindeutig und **quantitativ**, wie ineffizient Systeme mit häufiger Rekonfiguration auf der Grundlage heutiger FPGAs arbeiten.

Dieses Ergebnis hat Auswirkungen auf das grundsätzliche Konzept von JIFFY: Während zunächst ein dynamisch rekonfigurierbarer Datenflussprozessor im FPGA zur Ausführung der JVM-Befehle in Erwägung gezogen wurde (ähnlich den Methoden wie in [35] und [85] beschrieben), ergab sich mit der gefundenen Bewertungsmethode eine negative Bewertung.

[7] Interessant ist aber, dass “byte parallel” langsamer als “bit serial” ist.

Da das Datenflusssystem je nach Methodenschachtelung einige hundert bis tausendmal pro Sekunde rekonfiguriert werden muss, wäre selbst bei optimistischen Annahmen durch die Rekonfigurationsdauer nur eine Taktfrequenz von wenigen Kilohertz zu erreichen. Dieses Ergebnis und die zusätzlichen Probleme der Modulerstellung für die partielle Rekonfiguration führten schließlich dazu, dass dieser prinzipiell interessante Ansatz verworfen wurde.

	Xilinx XC6262-2	Xilinx XC6216-2	Xilinx XC3042A-7	Xilinx XC4013E-3	Xilinx XC4013E-3	Xilinx XC2S200
Konfiguration	32bit Bus	32bit Bus	Bit serial	Bit serial	Byte parallel	Byte parallel
Partiell rekonfigurierbar?	ja	ja	nein	nein	nein	ja [1)]
n_{lc}	16384	4096	144	576	576	5292
n_{mem} (Byte)	196608	49152	3848	30990	30990	167000
w_{bus} (Byte)	4	4	0.125	0.125	1	1
f_{reconf}	33MHz	33MHz	10MHz	10MHz	1MHz	50MHz
f_{max}	220MHz	220MHz	113MHz	125MHz	125MHz	$>$250MHz
f_{system}	55MHz	55MHz	28MHz	31MHz	31MHz	120MHz
$n_{lc.reconf}$	3	3	213	430	53.8	31.6
$t_{lc.reconf}$	90ns	90ns	21μs	43μs	53μs	630ns
$r_{lc.reconf}$	0.2	0.2	0.0017	0.00075	0.0006	0.013

Daten basieren vollständig auf Angaben in Datenblättern [171, 172, 173, 174].
[1)] Nur ganze Spalten in der LC-Matrix veränderbar (“SelectMAP”)

Tabelle 4.3: Vergleich verschiedender FPGA-Architekturen von Xilinx

5

Stand der Technik bei JVM-Implementierungen in SW/HW

I've seen things you people wouldn't believe.
Philip. K. Dick, Blade Runner

In diesem Kapitel werden eine Reihe von konkreten Ausführungs- und Optimierungsmöglichkeiten für ein Java-System beschrieben.

Zu Beginn dieser Arbeit waren davon nur wenige bekannt, insbesondere keine speziellen Java-Architekturen in Hardware. Ausser der picoJava-CPU wurden die meisten kommerziellen Hardwarelösungen, also Java-Prozessoren bzw. Java-Coprozessoren erst Anfang 2000 vorgestellt. Im gleichen Zeitrahmen liegen die Softwareumgebungen zur Entwicklung und Ausführung von Java auf eingebetteten Systemen.

5.1 Grundsätzliche Optimierungstechniken

Die Optimierungstechniken, die auf dem Bytecode oder Zwischencode angewendet werden können und strukturelle Änderungen durchführen, bestehen zum größten Teil aus allgemeinen Methoden [6] und einigen Java-spezifischen Vorgehensweisen. Zu den allgemeinen Methoden gehören unter anderem:

– Loopunrolling
– Ersetzung gemeinsamer Ausdrücke (Common Subexpression Elimination, CSE)
– Inlining von Methoden

Diese Methoden sind Standard in allen Übersetzungssystemen und unabhängig von der zu übersetzenden Sprache. Sie benötigen jedoch Informationen über den Datenfluss und globale Daten.

Als Java-spezifische Optimierungen [30] werden vor allem benutzt:

– De-Virtualisierung von virtuellen Methoden (direktes Aufrufen der Methoden bei bekannten Klassenhierarchien)

– Erzeugung von temporären Objekten auf dem Stack

– Eliminierung der Überprüfung der Arraygrenzen

Beide Arten der Optimierungen werden üblicherweise nicht direkt auf dem Bytecode, sondern auf einer oder mehreren Zwischenrepräsentationen durchgeführt.

5.2 Java Optimierungen vor der Ausführung

Um die Java-Ausführung zu beschleunigen, bietet es sich an, schon vor dem Laufzeitsystem Optimierungen vorzunehmen, entweder durch zusätzliche Java-Klassen, die nativ effizienter benutzbar sind oder durch Code-Optimierungen bereits auf Sourcecode-Ebene. Dazu sind auch Abschätzungen der Ausführungszeit nur aufgrund des JVM-Codes, wie z.B. in [28] erläutert, nützlich.

[33] beschreibt statische interprozedurale Optimierungen im Objektkontext, die unter anderem Code-Spezialisierung und Objekt-Inlining umfassen.

[18] nutzt einige Optimierungen auf Sourcecode-Ebene, kann Schleifen in effizientere Ablaufformen transformieren und unterstützt dabei auch Parallelverarbeitung auf Multiprozessorsystemen.

NINJA [102] ist ein System zum effizienten Ausführen von Java-Code für numerische Anwendungen, Ziel ist eine äquivalente Rechenleistung mit Fortran und C-Programmen. Die Optimierungen erfolgen hier bereits auf Sourcecode-Ebene und erzeugen direkt ausführbaren Code, damit ist NINJA also ein Ahead-of-Time-Compiler. Neben Ergänzungen zur Erzeugung rechteckiger Felder werden auch komplexe Zahlen direkt unterstützt und Optimierungen auf alias-freien Feldern vorgenommen.

Gegenüber JDK 1.1.16 (ohne JIT) erzielt NINJA bei numerischen Anwendungen einen Geschwindigkeitsfaktor von 35 bis 75 und erreicht damit ca. 55 bis 85% der Geschwindigkeit von Fortran-Code.

Quicksilver [137] ist ein “quasi-statischer” Compiler für Java, nutzt also Methoden von statischer und dynamischer Compilierung, und behält damit die Kompatibilität zum Java Standard. Als Optimierungen werden hier Standardverfahren wie z.B. Inlining benutzt.

5.3 JVM-Implementierungen in Software

5.3.1 Java Runtime Environment (JRE) von Sun

Das JRE von Sun war die erste Ausführungsplattform für Java und ist die Referenz für alle anderen Systeme.

Die erste Versione (JRE 1.0) basierte auf einem Interpreter, der zur Geschwindigkeitssteigerung den Bytecode beim Ablauf modifiziert hat. Befehle, deren Konstantenpoolreferenz bereits aufgelöst wurde, werden durch sogenannte "Quick-Opcodes" ersetzt, die auf die referenzierten Objekte direkt zugreifen und somit ein erneutes Resolving einsparen. Die Geschwindigkeitssteigerung beträgt einige Prozent in normalen Programmen, verkompliziert aber die Interpretierung.

Ab Version 1.1 wurde für x86 unter Windows und für Sparc (Solaris) ein Just-In-Time-Compiler (sunwjit) mitgeliefert, der Geschwindigkeitssteigerungen um den Faktor 2-10 ermöglichte (siehe auch [139]. Ab Version 1.2 wurde das Hot-Spot-System integriert, das im folgenden Abschnitt besprochen wird.

Für eingebettete Systeme mit ihren Resourceneinschränkungen existiert die Java Micro Edition [142][96], die auf zwei auswählbare VMs zurückgreift:

Die KVM basiert auf einem Interpreter und benötigt nur ca. 160KByte Speicher zur Ausführung. Für leistungsfähigere Systeme gibt es einen JIT-Compiler, der auf der Hotspot-Technologie basiert [141], allerdings nur sehr einfache Optimierungen durchführt.

Weiterhin wurden in der KVM Fließkommaoperationen, die Verifizierung des Bytecodes, sowie bestimmte Methoden zur Laufzeit weggelassen, um den Speicherverbrauch so gering wie möglich zu halten.

5.3.2 HotSpot

Das HotSpot-Konzept [148, 147] nutzt die Vorteile von Interpretierung und JIT-Compilierung. Selten oder nur einmalig ausgeführte Javamethoden werden durch Interpretierung abgearbeitet, da die Interpretierung ohne Zeitverlust anläuft. Stellt das Laufzeitsystem fest, dass bestimmte Methoden häufig benutzt werden ("Hot Spot"), werden nur diese Methoden über einen JIT-Compiler in nativen Maschinencode umgesetzt.

Durch die Informationen über das Laufzeitverhalten kann der JIT-Compiler weitaus effektivere Optimierungen durchführen, z.B. das Methodeninlining. Die gemischte Ausführung führt zu einem geringeren Startoverhead und einem "gleichmäßigeren" Lauf, da die Methodencompilation über die Laufzeit verteilt wird. Da insgesamt weniger Methoden compiliert werden, sinkt zusätzlich der dynamische Speicherverbrauch.

Im Gegensatz zu anderen Systemen wird der Garbage Collector Funktion ebenfalls gleichmässig über die Laufzeit verteilt und arbeitet in Intervallen von max. 10ms. Dadurch entstehen keinen störenden Pausen in der Ausführung der Anwendung.

Der Faktor der Leistungssteigerung im SPECjvm98-Benchmark gegenüber dem Interpreter beträgt im Mittel ca. 5 (HotSpot Client VM) bzw. 6.4 (HotSpot Server VM), das Maximum ca. 21, das Minimum ca. 1.7 [139]. Gegenüber dem "alten" Sun-JIT (sunwjit) ist das HotSpot-System im Mittel ca. 3.2/4-mal schneller, der maximale Faktor beträgt ca. 18, im Minimum (compress-Benchmark) ist keine Geschwindigkeitssteigerung festzustellen.

5.3.3 gcj – GNU Java Compiler

gcj [57] ist ein Frontend für das gcc-Compilersystem, und stellt damit ein AOT-System dar. gcj kanndabei sowohl Java-Code direkt als auch bereits in Bytecode umgesetzte Anwendungen anhand der .class-Dateien compilieren.

Durch das AOT-Prinzip ist das System nicht mehr für alle Anwendungen geeignet, liefert allerdings gute Laufzeitergebnisse. Im SciMark2.0-Benchmark liegt es um den Faktor 2 bis 21 (im Mittel 12) über der Interpretierung. Dabei ist die Ausführungsgeschwindigkeit auf der Basis von Java-Code im Vergleich zur Compilation der Java-Klassen um einige Prozent höher [139]. Diese Werte berücksichtigen aber nicht den den Zeitaufwand der Compilation selbst.

5.3.4 Kaffe

Kaffe [81] ist eine unter der GPL stehende "Clean-Room"-Implementierung einer Java Laufzeitumgebung und besteht dabei aus einer VM und den Java-Klassen. Neben einem Interpreter unterstützt Kaffe auch einen Just-In-Time-Compiler für 80386, Sparc, Alpha, ARM, MIPS und 68k-Systeme.

Der Kaffe-JIT ist portabel geschrieben, da die eigentliche Codeerzeugung aus einem gemeinsamen Übersetzungskern für jede zu übersetzende Aktion einzeln aufgerufen wird. Das Compiler-Backend besteht dazu nur aus Funktionen, die im Normalfall ein bis zwei Assemblerbefehle erzeugen. Die Codeerzeugung basiert auf einer Stacknachbildung auf Slotregistern und vermeidet somit die explizite Stacknutzung. Weitergehende Optimierung außer dem Inlining von statischen Methoden sind nicht implementiert.

Im SPECjvm98-Benchmark ist Kaffe um den Faktor 0.6 bis 15 (im Mittel 2.0) schneller als die Interpretierung im JDK [139].

5.3.5 TYA – JIT für x86-Prozessoren und JDK

TYA entstand als Ersatz für den Sun-JIT für Linux-x86 und ist nicht auf andere Plattformen portierbar. Der Übersetzungskern besteht aus einem großen switch-Kommando, das abhängig vom JVM-Kommando direkt x86-Befehle erzeugt. Durch das einfache Konzept sind nur sehr rudimentäre Optimierungen möglich, die Leistung liegt mit einem Faktor zwischen 0.8 und 4.5 (im Mittel 1.7) gegenüber der JDK-Interpretierung noch unterhalb der von Kaffe [139].

5.3.6 Caffeine

Caffeine [71] ist ein System zur statische Compilation von Bytecode in nativen Assemblercode. Die Übersetzung erfolgt dabei in mehreren Stufen mit 3 Zwischensprachen, auf denen jeweils Analysen und Optimierungen stattfinden. Der Java-Stack wird dabei mit virtuellen Registern nachgebildet und nach einer Nutzungsdauerüberprüfung auf reale Register abgebildet. Weitere Verbesserungen betreffen die Optimierungen von Objektzugriffen und Ausnahmen.

Als erzielbare Geschwindigkeit wird durchschnittlich 68% gegenüber äquivalenten, C-compilierten Methoden angegeben, bzw. eine Beschleunigung von mehr als 20 gegenüber dem JDK1.1-Interpreter.

5.3.7 Jalapeño von IBM

Jalapeño [17] ist eine Forschungs-JVM, selbst in Java geschrieben, die ausschließlich auf Compilation des Bytecodes ausgelegt ist. Eine Interpretation findet nicht statt. Die JIT-Compilation ist aus mehreren Grund- und Optimierungsstufen auswählbar und damit adaptiv. Die Übersetzungsgeschwindigkeit auf einem 333MHz PPC604e beträgt dabei ohne Optimierung (“Baseline”) ca. 274Bytecode Bytes pro Millisekunde, mit Optimierung zwischen 2 und 9 Bytecode Bytes/ms.

Der interne Aufbau des Systems besteht aus mehreren, asynchron kommunizierenden Threads, die Teile des Laufzeitsystems unabhängig voneinander verwalten, aber auf einer gemeinsamen Datenbasis arbeiten. Die adaptive Optimierung bestimmt aufgrund der von Laufzeitmesssystem gelieferten Daten häufig benutzte Methoden, schätzt die Effektivität der geplanten Optimierung und startet bei Bedarf eine erneute Compilation mit stärkerer Optimierung (“Recompilation”). Eine Variante des Systems (“Feedback-Inlining”) erreicht durch direktes Einfügen oft benötigter Funktionen eine noch etwas höhere Geschwindigkeit.

Die adaptive Optimierung beschleunigt vor allem die “Startup”-Zeit und gewinnt gegenüber der Baselineausführung zusätzlich. Gegenüber der Laufzeit in der höchsten Optimierungsstufe ist die adaptive Optimierung nur unwesentlich

langsamer. Das Feedback-Inlining beeinflusst die Startzeit nicht, ergibt aber im Mittel einen Beschleunigung von 11%.

[88] beschreibt eine Erweiterung von Jalapeño auf "Lazy-Compilation", d.h. die Compilation nur der gerade aktuell benötigten Methoden und nicht der gesamten Klasse. Der dadurch entstehende Geschwindigkeitsgewinn liegt insgesamt zwischen 14 und 26%, auch die reine Ausführungszeit verringert sich durch Einsparungen beim dynamischen Linken um 11 bis 13%.

5.3.8 CACAO – JIT für 64Bit-Prozessoren

CACAO [62] ist ein JIT-Compiler für 64Bit-RISC-Prozessoren, in der ersten Implementation wurde eine Anpassung an den DEC Alpha geschrieben. Der Übersetzungalgorithmus arbeitet in mehreren Phasen. Zunächst wird der Bytecode analysiert und die einzelnen Superblöcke extrahiert. Diese werden dann einzeln in einen Zwischencode mit Pseudoregistern umgewandelt. Der Zwischencode wird optimiert, um unnötige Kopieroperationen einzusparen, anschließend werden die Pseudoregister den tatsächlich vorhandenen CPU-Register zugeordnet. In der letzten Phase wird der Zwischencode mit den zugewiesenen CPU-Registern in native Maschinenbefehle übersetzt.

Die Registerallokierung für die Pseudoregister, die aus der Stackumsetzung und lokalen Variablen entstehen, wird in CACAO nicht über eine "Graph Coloring"-Strategie, sondern über einen recht einfachen, linearen Algorithmus ermittelt. Dieser verteilt die benötigten Pseudoregister in der Reihenfolge ihres Auftretens auf die zahlreichen CPU-Register. Werden dabei mehr Register als vorhanden benötigt, werden diese im Stack angelegt ("Register Spilling").

Durch die Einfachheit des Übersetzungsvorganges ist dieser sehr schnell (ca. 500000 Bytecodes/s auf einem 21164/500MHz), er erzeugt aber einen relativ effizienten Code. Gegenüber der Interpretierung ist CACAO um den Faktor 2.3 bis 11 schneller.

5.3.9 LaTTe von IBM

LaTTE [176] ist ein Laufzeitsystem mit JIT-Compiler für SPARC-CPUs. Der JIT-Compiler überführt den Bytecode zunächst in eine RISC-ähnliche Zwischenform. Dieser Zwischencode basiert auf symbolischen Registernamen, die in späteren Übersetzungsphasen auf reale Maschinenregister abgebildet werden. Diese Abbildung erfolgt anhand eines Kontrollflussgraphen (CFG) und mehrerer Läufe vorwärts und rückwärts über den Code. Dieser Algorithmus ("Reconciling Register Allocation") erlaubt es, gemeinsame Register über mehrere Ablaufpfade hinweg zu benutzen und spart damit Stackzugriffe bzw. Kopieraktionen.

Weitere Optimierungen in LaTTe betreffen schnellere Objektzugriffe, und "Light-Weight Monitors", die neben dem JIT zusätzliche Effizienz erbringen. Im Vergleich zu JDK1.1.6 mit JIT ist LaTTe ca. 2.2-2.4mal schneller und damit ungefähr genauso schnell wie das Sun Hotspot-System. Die Übersetzungsoverhead liegt je nach Optimierungsstufe zwischen 60% (Laufzeit insgesamt ca. 0.8s) und 0.1% (Laufzeit ca. 320s) und ist absolut gesehen relativ konstant (0.23-2.19s bzw. mit aufwendigeren Optimierungen 1.13-10.0s).

[34] beschreibt eine Ergänzung von LaTTe, um virtuelle Methodenaufrufe durch Nutzung von monomorphen und polymorphen Inline-Caches statt virtueller Methodentabellen und erreicht damit eine Beschleunigung zwischen 3 und 9%. Diese Optimierung basiert auf dem direkten Einfügen (Inlining) der virtuellen Methoden in den erzeugten Code, als Datenbasis dienen dazu Codefragmente, die aus Übersetzunge der möglichen Ableitungen bzw. Erweiterungen der Methoden in unterschiedlichen Klassen entstanden sind.

5.3.10 jeode von Insignia

Insignia bietet mit jeode [75] eine speziell auf eingebettete Systeme angepasste Laufzeitumgebung an. jeode wird unter anderen in VxWorks [168], aber auch z.B. für den Linux-basierten PDA Zaurus [138] eingesetzt.

Der integrierte Just-In-Time-Compiler DAC (Dynamic Adaptive Compiler) ist in seinem Laufzeitverhalten sehr fein auf das System einstellbar. Unter anderem sind die Parameter für die Umschaltung zwischen Interpreter und Compiler, die Startzeit der Compilation, die Zeitdauer der Compilation und auch das Löschen der bereits compilierten Methoden justierbar, um eine existierende Speichergröße (Memory Footprint) möglichst gut auszunutzen. Damit ist das System dem Hot-Spot-Prinzip sehr ähnlich, allerdings auf minimale Speichernutzung ausgelegt. Insgesamt soll ein Speedup von ca. 6 gegenüber der Interpretierung erreicht werden, wobei nur ca. 25% des Speichers einer "normalen" JIT-Lösung benötigt werden. [128]

Ähnlich flexibel ist der Garbage Collector. Dieser arbeitet in einem separaten Thread und kann in seiner Priorität verändert werden, um eine bestmögliche Anpassung an das erwartete Laufzeitverhalten zu erreichen.

Um Speicherplatz zu sparen, ist es möglich, Klassenbibliotheken in ein ROM-taugliches Format umzuwandeln. Somit laufen dauerhaft für das System benötigte Java-Klassen direkt aus dem ROM ab.

Der ROM-Speicherbedarf der gesamten Laufzeitumgebung mit Klassenbibliotheken und der grafischen AWT-Oberfläche liegt bei ca. 2.5MByte.

5.3.11 Weitere Implementierungen und Optimierungen für die JVM

In [165] werden die mit dem Einsatz von Java auf eingebetteten Systemen entstehenden Probleme umrissen und vereinzelt Lösungsansätze angeboten. Besonders das nicht-deterministische Zeitverhalten durch die Garbage Collection ist noch eines der größten Probleme. Mit einer speziellen API besteht dabei eine Möglichkeit, den einzelnen Verwaltungseinheiten der JVM definierte Zeitbudgets zuzuordnen.

[69] beschreibt die Realisierung eines JVM-Interpreters auf einer VLIW-Architektur (Philips-TriMedia). Durch eine Verschachtelung (Pipelining) der einzelnen der Phasen der Interpretierung (Befehl holen, Dekodieren, Ausführen) konnten Geschwindigkeitsgewinne bis zu 20% erreicht werden.

In [2] wird eine JIT-Ergänzung der Microsoft JVM beschrieben, die auf einem einfachen Übersetzungskonzept ohne Zwischenarchitektur beruht, d.h. die x86-Maschinenbefehle werden direkt aus dem Bytecode erzeugt. Interessant ist hier die Art der Common-Subexpression-Elimination (CSE), die direkt anhand der Bytecodesequenzmuster auf JVM-Ebene die Gemeinsamkeiten findet. Weitere Optimierungen wie Register Allocation, Beschleunigung von Arrayzugriffen und Auslösungen von Exceptions führen zu einem mit dem MS JIT-Compiler vergleichbarem Ergebnis, d.h. der 1-Pass-Ansatz bietet keine unmittelbar deutlichen Vorteile bezüglich der Laufzeit der Übersetzung und des Compilats.

In [83] wurde untersucht, inwieweit überlappende Ausführung und Compilation ohne Eingriff in den Java-Code möglich ist, als Gewinn wurden für verschiedene Benchmarks Werte zwischen 0.2 und 8.9% ermittelt.

Das Excelsior JET-System [54] ist eine Kombination verschiedenster Bausteine zur Compilation und Ausführung von JVM-Bytecode. Es arbeitet mit einem gemischten Modell aus AOT und JIT-Compilierung. Die Compilation nutzt dabei traditionelle Optimierungen (Inlining, Loop Unrolling, Instruction Colouring, etc.), als auch Java-spezifische Optimierungen, wie Eliminierung von Laufzeitüberprüfungen und Vereinfachung von Interface-Aufrufen. Zusätzlich steht ein darauf angepasstes Laufzeitsystem zur Verfügung. Weitergehende Informationen oder Geschwindigkeitsmessungen sind nicht veröffentlicht.

Mit OpenJIT-System [116] wird ein JIT-System beschrieben, dass fast ausschließlich in Java geschrieben ist, und als JIT-Plugin für das Sun JDK entwickelt wurde. Es besteht intern aus zwei Teilen: Ein Front-End, das Optimierungen auf dem Bytecode vornimmt und den erzeugten Zwischencode an ein Back-End weitergibt, das wiederum Optimierungen für die Zielmaschine ausführt und den Maschinencode für die Zielarchitektur erzeugt. Die bislang veröffentlichten Ergebnisse [117] zeigen eine Geschwindigkeitssteigerung bei SPECjvm98 um den Fak-

tor 1 bis 7 gegenüber der Interpretierung. Damit ist OpenJIT vergleichbar mit der Leistung von sunwjit.

MIPSAOUT [140] ist ein Ahead-of-Time-Compiler für MIPS-Prozessoren unter IRIX. Es wird zusammen mit dem JDK verwendet, und arbeitet über die JIT-API des JDK, sodass vorcompilierter Code mit interpretiertem Code während der Ausführen gemischt werden kann ("Mixed Mode").

Intern das System zweigeteilt: Ein Front-End liest eine Klassenbeschreibung ein und wandelt sie in eine interne Repräsentation um, wobei bereits einige Laufzeittests eliminiert werden. Anschließend wird der Zwischencode mit dem optimierenden SGI MIPSPro Compiler Back-End in eine ausführbare Bibliothek umgewandelt, die zur Laufzeit vom JDK dynamisch gelinkt wird. Das Ergebnis dieses optimierten Codes ist überraschenderweise nur in Spezialfällen schneller als ein JIT-Compiler. Als Grund für dieses Ergebnis wird das Java-unspezifische Back-End genannt, das einen zu wenig effizienten Code für die Java-Konstrukte erzeugt.

FastJ [93] ist ein AOT-Compiler, der speziell für den Einsatz von Java-Anwendungen in eingebetteten Systemen entwickelt wurde. Neben dem Einsatz von lokalen und globalen Code-Optimierungen wurden besonders die Java-Bibliotheken neu geschrieben und so gestaltet, dass nicht im normalen JDK durch Abhängigkeiten effektiv fast alle Bibliotheksklassen geladen werden müssen.

5.4 JVM-Implementierungen in Hardware

5.4.1 Forschungssysteme

Bei "Hard-Int" [129] handelt es sich um ein System, das ähnlich JIFFY eine Übersetzung in nativen Code auf Hardwarebasis ausführt. Diese Übersetzung ist allerdings direkt in den Mikroprozessor integriert (z.B. SPARC) und arbeitet einstufig, d.h. ohne Zwischensprache. JVM Bytecodes werden dabei über ein ROM in entsprechende Befehlssequenzen der nativen Ausführungsarchitektur übersetzt, bereits übersetzte Befehle können in einem Cache abgelegt werden und stehen für eine weitere Optimierung zur Verfügung. Vorteilhaft ist die direkte Integration (kurze Latenzen durch Verzicht auf die Zwischenschicht) in den Prozessor. Als Geschwindigkeitsgewinn gegenüber einem Interpretersystem wird ein Faktor von ca. 5.4 genannt, gegenüber dem JDK1.2 JIT ca. 2.5. Nachteilig an diesem System ist die starke Architekturabhängigkeit der Übersetzungshardware und die fehlende Updatemöglichkeit, wenn das System, wie vorgeschlagen, auf einem Übersetzungs-ROM basiert.

Im JAMAICA-Projekt [14, 49] wird eine Hardwarearchitektur entwickelt, die auf einem Single-Chip-Multiprozessorsystem mit Java laufen soll. Dabei unter-

stützt die Architektur unter anderem auch Multimedia-Erweiterungen. Der Prozessor selbst ist Register-basiert, als Compiler wird eine Weiterentwicklung von CACAO benutzt.

[119] beschreibt eine Mikroprozessorarchitektur für eingebettete Anwendungen, die eine Mischung zwischen JVM und RISC darstellt. Dazu wird die eigentliche ALU und Registereinheit von zwei unabhängigen Kontrolleinheiten benutzt, die über Threads getrennt ablaufen. Komplexere JVM-Befehle werden abgefangen und müssen im RISC-Modus emuliert werden. In Simulationen wurden bei 12.5MHz Takt ca. 6-7 MIPs im RISC- und JVM-Modus bestimmt, der Leistungsverbrauch liegt dabei bei ca. 3.5mW.

5.4.2 picojavaI/II von Sun

Der picoJava-Kern von Sun [150] [146] [115] ist ein nativer JVM-Prozessor, er verarbeitet die Bytecodes also direkt. Zur Beschleunigung von Speicherzugriffen sind ein Instruktion- und Datencache integrierbar, als Cachegrößen sind 0 bis 16KByte möglich.

picoJava bildet den JVM-Stack intern auf einen 64 Einträge großen Cache mit zwei Lese- und einem Schreibport ab, der über die Stack-Management-Unit verwaltet wird. Diese erkennt Stackunter- oder Überläufe und greift auf den Hauptspeicher zu.

Im Gegensatz zu vielen anderen JVM-Prozessoren kann der picoJava-Kern mit einer optionalen FPU ausgestattet werden. Diese basiert auf einer Steuerung mit Microcode und ist auf einfache Genauigkeit ausgelegt. Operationen mit doppelter Genauigkeit erfordern zwei bis viermal soviele Taktzyklen.

Die meisten Befehle sind direkt verdrahtet, komplexe Befehle werden im Microcode oder über Traps emuliert. Der picoJava-II-Kern besitzt im Gegensatz zum picoJava-I dazu noch eine 6-stufige “RISC”-Pipeline mit Instruction-Folding, kann also Lade- und Operationsbefehle in einem Takt zusammenfassen und ausführen. Direkt im Bytecode unterstützt werden auch die JVM-Synchronisationsbefehle und Hilfen zur Garbage Collection.

Zur Verwaltung der Chipressourcen sind Erweiterungen des Bytecodes vorgesehen, die nur in einem Supervisormode ausführbar sind. Diese Sonderbefehle erlauben das Beschreiben und Lesen von Speicher und internen Registern.

Der picoJava-II-Kern benötigt ca. 120K Gatter und läuft in einem 0.25μm-CMOS-Prozess mit einer Taktfrequenz von ca. 100MHz. Eine Implementierung des picoJava-II von Fujitsu [59] benötigt bei 66MHz und 2.5V ca. 360mW Leistung, eine 1.7V-Version mit 40MHz ca. 90mW.

5.4.3 MAJC von Sun

Die MAJC-Architektur (Microprocessor Architecture for Java Computing) [158, 144] von Sun beruht auf einem VLIW-Modell und ist kein Java-Prozessor im eigentlichen Sinn. Der MAJC-Kern besteht aus zwei parallel arbeitenden VLIW-Prozessoren mit einem 128Bit-Instruktionsfeld für 4 parallele funktionale Einheiten pro Prozessorkern und u.A. auch einem graphischen Coprozessor (z.B. zur Sortierung von Geometriedaten). Durch die sehr hohe Rechenleistung (ca. 6GFLOPS bei einfacher FPU-Genauigkeit) ist er besonders auch für Multimediaanwendungen geeignet.

Das Konzept des “Space-Time-Computing”, ist im Prinzip eine spekulative Ausführung von Threads auf den beiden Prozessoren mit anschließendem Rollback. Es kann besonders bei Java-Programmen benutzt werden, da hier Daten nur auf dem Heap restauriert werden müssen und die verursachenden Store-Operationen dazu schnell im Bytecode identifizierbar sind. Sun gibt eine Steigerung des SpecJVM-Benchmarkwertes durch dieses spekulative Multithreading um einen Faktor 1.5 an.

Weitere besondere Funktionsgruppen zur beschleunigten JVM-Ausführung sind nicht vorhanden, von daher sind keine sinnvollen Geschwindigkeitsvergleiche zu anderen Systemen möglich.

5.4.4 PSC1000 von Patriot Scientific

Der PSC1000 von Patriot Scientific [122, 136] ist ein stackbasierter 32Bit-Mikroprozessor, d.h. die Befehle haben keine Registeroperanden (0-Adress-Maschine). Zur Zwischenspeicherung von Variablen kann auf ein Registerblock mit 16 Registern zugegriffen werden, Operandenstack und Returnstack sind getrennt und arbeiten als Ausschnitt einen größeren Stacks im Arbeitsspeicher (Stack cache).

Die PSC1000-Befehle sind üblicherweise nur ein Byte lang (Immediate Load und Jumps bis zu 5Byte), sind zunächst auf die Ausführung von Forth ausgelegt und nicht zur JVM kompatibel. Es ist keine Unterstützung für objektorientierte Befehle oder eine FPU vorgesehen. Fast alle der ca. 100 Befehle sind in einem Takt ausführbar.

Trotz der fehlenden Binärkompatibilität lassen sich die JVM-Programme schnell per JIT auf den PSC1000 umsetzen, da keine Stackemulation und nachfolgende Optimierungen darauf erfolgen müssen.

Der PSC1000 besitzt 137500 Transistoren und benötigt 165mW bei 3.3V und 100MHz Taktrate.

Java-Benchmarkdaten für den PSC1000 sind nicht verfügbar.

5.4.5 JVXtreme von inSilicon

JVXtreme von inSilicon [76, 90, 43] ist ein einfacher Java Prozessor, der zusammen mit einer Host-CPU die JVM-Abarbeitung beschleunigt. Eine autonome Funktion oder der Ersatz einer Host-CPU durch den JVXtreme ist nicht möglich. Das JVXtreme-Konzept verzichtet dabei auf Instruktions- und Datencaches, auf eine FPU und die autonome Abarbeitung aller Bytecodes. JVXtreme führt 92 der ca. 200 JVM-Bytecodes direkt in Hardware aus, die verbleibenden Befehle, also ca. 55%, müssen auf der Host-CPU emuliert werden.

Der als IP-Core (Intellectual Property) für ASICs und FPGAs lieferbare Beschleuniger ist fast prozessorunabhängig und arbeitet intern auf einer Stacknachbildung mit 64 Einträgen und einer automatischen Unter- und Überlaufkontrolle. Grundlegende JVM-Befehle (ALU Operationen, Konstanten und Variablenzugriffe) werden in einem Takt abgearbeitet, komplexere Befehle (Arrayzugriffe und Sprünge) benötigen zwei bis drei Takte. Durch eine statische Sprungvorhersage sind nur für falsch vorhergesagte Sprünge vier Wartezyklen notwendig. Einige Befehlskombinationen können mit Instruction-Folding in nur einem Takt abgearbeitet werden.

Der JVXtreme-Kern benötigt nach Angaben von inSilicon ca. 15K bis 35K Gatter und läuft auf einem 0.18μm-Prozess mit ca. 200MHz. Zusammen mit einer 40MHz ARM7-CPU werden Werte im Embedded Caffeine Mark von ca. 200 erreicht.

Eine Anpassung auf andere Prozessoren ist laut InSilicon relativ einfach, da nur das Businterface verändert werden muss.

5.4.6 XPRESSOcore und XC-100 von Zucotto

Der XC-100 [178, 179] von Zucotto Wireless ist ein IP-Core zur Integration in Mobiltelephone, der einen JVM-Prozessor mit Microcode nachbildet. Der Kern besteht dabei aus einer 32Bit-CPU mit einer fünfstufigen Ausführungspipeline und 4KB Instruktionscache und 8KB Datencache. Der Microcode kann dabei entweder in einem eingebetteten RAM (Embedded RAM) oder in einem ROM liegen. Ein Embedded RAM erlaubt Updates des Microcodes auch im schon synthetisierten Zustand.

Der CPU-Kern unterstützt Datentypen mit Größen von 8, 16, 32 und 64Bits, eine FPU-Unterstützung ist aus den Datenblättern nicht erkennbar.

Zucotto gibt eine maximale Taktfrequenz von 55MHz an, der IP-Core braucht dazu (ohne Caches und Microcodespeicher) ca. 70K Gatter. Als Geschwindigkeitsgewinn wird ein Faktor von 20 bis 100 (in KVM Marks) gegenüber herkömmlichen Interpreter-Lösungen für Mobiltelephone angegeben.

5.4.7 Jemcore/aj-100 von aJile

Der IP-Core Jemcore JEM2 [8] bzw. der vollständige Chip aJ-100 [7] von aJile basiert auf der bereits 1997 von Rockwell entwickelten, aber nur intern benutzten JEM1-Architektur [133] und ist eine vollständige und autonome Java-CPU, d.h. die CPU kann Bytecode direkt ausführen und benötigt keinen Zusatzprozessor.

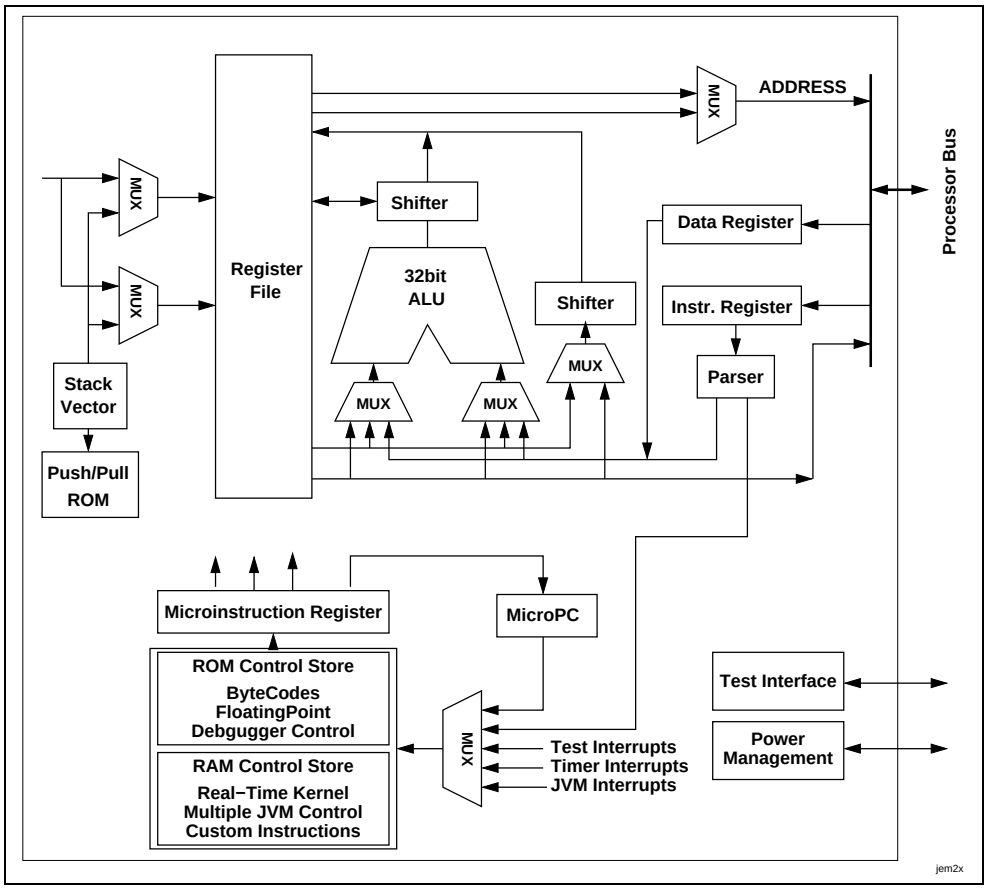

Abbildung 5.1: JEM2 Prozessorkern

Der CPU-Kern (siehe Abb. 5.1) besteht aus einem 24 32Bit-Register umfassenden Datenblock zur Stackemulation und einer Integer und Fließkomma-ALU. Die JVM-Befehle werden über einen Microcode ausgeführt, der in einem 4K*56Bit ROM untergebracht ist. Der Microcode ermöglicht auch zusätzliche Befehle beschleunigt auszuführen, wenn der Classloader bestimmte Methodenaufrufe durch reservierte Bytecodes ersetzt. Dadurch werden bei speziellen Multimediaanwendungen Geschwindigkeitssteigerungen um den Faktor zwischen 5 und 50 gegenüber dem normalen Algorithmus im Bytecode erreicht.

Mit einer optionalen Einheit (“Multiple Java Manager”, MJM) kann der Kern von zwei JVMs benutzt werden, die im Zeitscheiben-Verfahren und unabhängig voneinander laufen. Damit wird es möglich, ein Grundsystem und unabhängige Anwendungen ohne Beeinflussungen parallel zu betreiben.

Laut Ajile benötigt der Jemcore ohne das Microcode-ROM ca. 25K Gatterfunktionen (die MJM zusätzlich 10K), die maximale Taktfrequenz bei 3.3V und einer 0.25μm CMOS Implementierung wird mit ca. 100MHz angegeben.

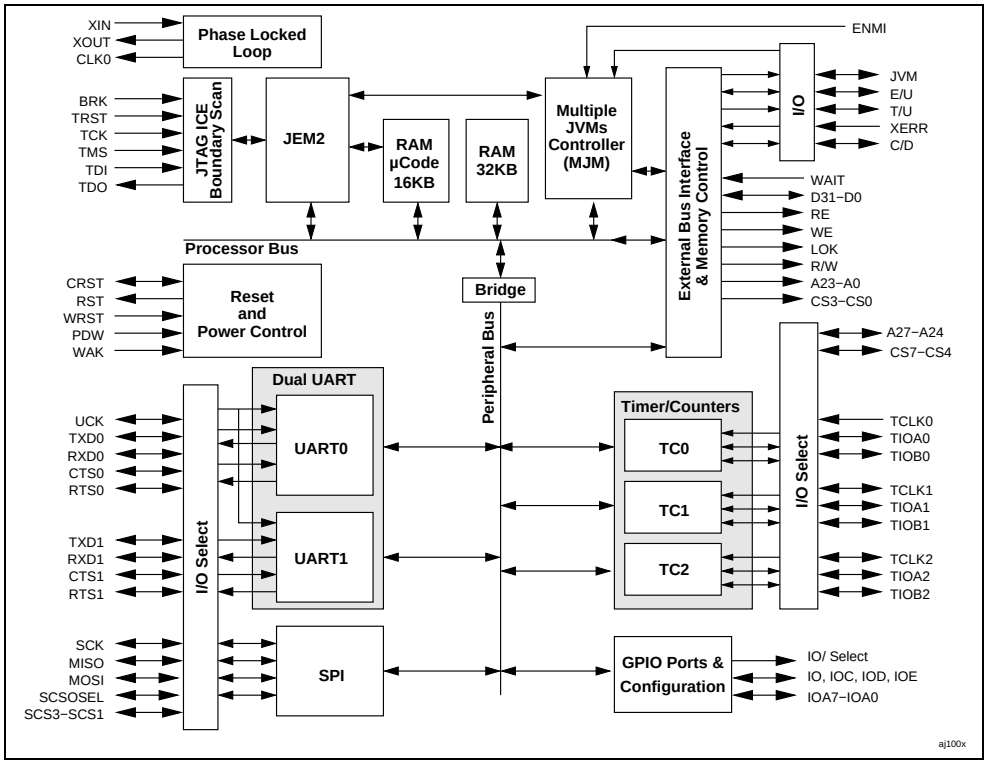

Abbildung 5.2: aJ-100 mit JEM2 Prozessorkern

Im aJ-100 (Übersicht in Abb. 5.2) sind neben der üblichen Peripherie auch noch 48KByte Zero-Waitstate-RAM integriert, von denen 16KB als zusätzlicher Microcodespeicher benutztbar sind, die verbleibenden 32KB stehen dem Betriebssystem zur Verfügung. Über das Businterface (EBI) können bis zu 256MB externes RAM angesprochen werden, allerdings ist dies nur ein generischer Bus. Dynamische Speicher benötigen Zusatzlogik.

Über die Leistungsfähigkeit des aJ-100 sind bislang keine genauen Daten verfügbar.

5.4.8 DeCaf und Espresso von Aurora VLSI

Der IP-Core DeCaf [21, 23] von Aurora-VLSI ist ein autonomer Java-Prozessor, der besonders für portablen Betrieb ausgelegt ist. Er ist in verschiedenen Varianten erhältlich, unter anderem auch als “Bilingual”-Core mit einer sog. “Legacy”-CPU (DeCaf-M Core [22]). Dieser Prozessor beruht dabei auf einem proprietärem Design (32bLOW Processor) und ist nicht kompatibel zu den verbreiteten ARM-Derivaten (siehe 4.3.2).

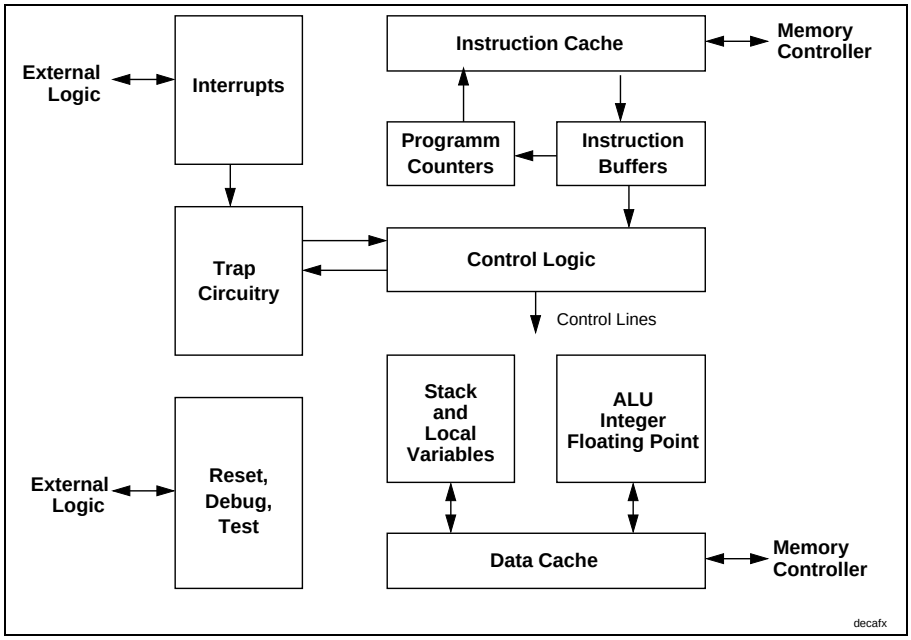

Abbildung 5.3: DeCaf Prozessorkern

Der DeCaf Prozessor ist ähnlich wie der JEM2 aufgebaut, arbeitet allerdings nach dem RISC-Prinzip ohne Microcode mit einer fünfstufigen Ausführungspipeline. Die ALU unterstützt auch Fließkommabefehle direkt. Komplexere Instruktionen (alle objektorientierten Befehle) lösen einen Trap aus und müssen emuliert werden. Das Konzept sieht durch die RISC-Implementierung jedoch insgesamt leistungsfähiger als das des JEM2 aus.

Zur Stacknachbildung sind 64 Register vorhanden. Der Registersatz hat mehrere Zugänge, es kann also mehrfach in einem Zyklus darauf zugegriffen werden. Lokale Variablen sind einem zusätzlichen Registerblock untergebracht, der eine (bei der Core-Generierung wählbare) Größe zwischen 32 und 256 Einträge hat. Reicht dieser Platz während der Ausführung nicht aus, werden die benötigten Werte über den Datencache aus dem Hauptspeicher geladen.

Der JVM-Befehlssatz ist um einige Befehle erweitert (“native Bytecode”), der es erlaubt, Speicher und chipinterne Ressourcen direkt anzusprechen. Diese Be-

fehle stehen dabei nur in einem Supervisormodus zu Verfügung, der für die Systemverwaltung gedacht ist und von normalen JVM-Programmen nicht zu benutzen ist.

Wie auch der Speicher für lokale Variablen sind die Caches für Befehle und Daten parametrisierbar und können Größen zwischen 32Byte und 8KByte annehmen.

In einem 0.25μm-CMOS Standardzellen-Prozess sollen 140-250MHz erreicht werden, allerdings sind diese Angaben noch nicht gesichert. Die geschätzte Leistung im ECM-Benchmark wird vom Hersteller bei 200MHz mit ca. 20000 Punkten angegeben.

5.4.9 Jstar von Nazomi

Jstar [111] von Nazomi ist ein Übersetzungskern zur Erweiterung eines kundenspezifischen CPU-Cores, der am Bus zwischen der Haupt-CPU und dem Speicher plaziert wird. Jstar überwacht dabei die Instruktionszugriffe der CPU auf den Speicher und übernimmt die Rolle des Hauptspeichers, wenn der PC in vorher definierten Grenzen liegt, die dann Bytecode enthalten. Diese Instruktionen werden durch Jstar eingelesen und über einen microcodebasierten Übersetzerkern in native CPU-Befehle übersetzt und an den Prozessor als normaler Befehlstrom weitergeleitet. Optimierungen des erzeugten Codes finden bis auf einige Fälle von Instructionfolding nicht statt.

Dieses Übersetzungskonzept ist fähig, ca. 70% aller JVM-Bytecodes direkt zu übersetzen. Objektorientierte Befehle werden, wie bei anderen JVM-Prozessoren auch, über Subroutinen emuliert.

Durch die Erzeugung von nativem Maschinencode ist der JStar-Core auf den Prozessortyp anzupassen. Bislang sind Adaptationen für ARM und MIPS verfügbar.

Der auf die ARM-Architektur angepasste JStar-Kern benötigt ca. 30K Gatter und ca. 6KByte Microcodespeicher. Nazomi gibt einen Leistungsverbrauch von 0.18mW/MHz in einem 0.18μm-Prozess bei 1.5V Betriebspannung an.

Der Embedded Caffeine Benchmark (EMC) wird durchschnittlich 10mal schneller ausgeführt, der maximale Speedup liegt bei 27 gegenüber einer Interpreterlösung.

5.4.10 Jazelle von ARM

Die Jazelle-Erweiterung von ARM [16] ist ein ähnlich dem Thumb-Modus in den CPU-Kern integrierter Befehlsmodus. Dieser besteht aus einer Übersetzungsschicht in Hardware, die ca. 135 Bytecodes durch Übersetzung in CPU-interne

Steuerbefehle direkt ausführt. Die verbleibenden JVM-Befehle werden mit ARM-Code emuliert.

Der Java-Stack wird im Registerblock mit 4 Register nachgebildet [89]. Stack-über/unterläufe werden automatisch gehandhabt, damit ist die Effizienz der Stacknachbildung vom Speichersubsystem abhängig. Allerdings wird von ARM argumentiert, dass der Java-Stack üblicherweise nicht so intensiv genutzt wird und die benötigte Stacktiefe daher relativ klein belassen werden kann.

Der JVM-Modus wird ähnlich wie der Thumb-Modus als eingebaute Befehlsausführungsart transparent in das System eingebunden, d.h. ARM-Programme können wahlfrei zwischen ARM-Code, Thumb-Code und Bytecode springen. Da der Status der JVM in den gewöhnlichen ARM-Registern liegt, wird dieser wie bei einem normalen Kontextwechsel vom Betriebsystem gerettet und restauriert. Eine besondere Unterstützung in der Hardware ist dafür nicht notwendig.

Alle Bytecodebefehle können im Falle einer externen Unterbrechung neu gestartet werden, die Interruptlatenzzeit bleibt also sehr gering. Die Ausführung der JVM beeinträchtigt somit die Echtzeiteigenschaften des Systems nicht.

ARM gibt den zusätzlichen Aufwand für die Integration von Jazelle in einem ARM-Core mit ca. 12K Gatter an. Weil nur eine neue Befehlsdekodiereinheit zusätzlich benötigt wird und ansonsten die Systemarchitektur nicht verändert wird, ist dieser Wert nachzuvollziehen. In der Gesamtübersicht ist damit Jazelle von den zusätzlich benötigten Gattern am effizientesten.

5.4.11 MachStream von Parthus Technologies

Das MachStream-System [120] (früher “HotShot” von Chicory [37]) ist ein 2001 bekannt gewordenes Konzept zur Integration verschiedener Teilkomponenten zur Beschleunigung vom Multimediafunktionen für mobile Geräte. Dazu werden verschiedene Beschleunigerkerne in einem am Systembus befindlichen Chip integriert.

Eine dieser Komponenten kann eine Javabeschleunigung ausführen. Dazu wird ähnlich dem JIFFY-System eine Hardware-unterstützte Übersetzung von Bytecode in den nativen Maschinencode vorgenommen. Leider waren trotz mehrfacher Nachfrage bei Parthus keine detailierteren Beschreibungen des Übersetzungsvorganges erhältlich.

Soweit aus den vorhandenen “Veröffentlichungen”[1] ersichtlich ist, arbeitet der Übersetzungsvorgang in drei Phasen, wobei zunächst der Bytecode in architekturunabhängige Befehle, sog. Token umgesetzt wird. Anschließend wird die Tokenabfolge optimiert, dabei ist aber nicht ersichtlich, ob und welche der potentiellen Optimierungsmöglichkeiten (Register Allocation, Code Scheduling, Speculative

[1] Es sind nur wenig konkrete White Papers bzw. Werbeinformationen verfügbar.

Code) tatsächlich in der Hardware ausgeführt werden. Als dritte Stufe werden die optimierten Token in einen nativen ARM bzw. MIPS-Befehlsstrom umgesetzt. Andere Zielarchitekturen können über eine veränderte dritte Stufe unterstützt werden.

Wie bei JIFFY besteht der Vorteil dieses Systems darin, dass Verbesserungen in der Zielarchitektur nahezu linear die Leistungsfähigkeit der Java-Ausführung verbessern. Als Geschwindigkeitswerte werden für ein ARM9-System 5.5 CaffeineMarks/MHz mit einer KVM-Laufzeitumgebung angegeben. Verschlüsselungsfunktionen sollen mit MachStream 13-mal schneller als Softwarelösungen ablaufen, wobei allerdings unklar ist, ob diese auf Interpretern oder JIT-Compilern basieren.

Insgesamt ist das MachStream-System ein sehr vielversprechendes Konzept und JIFFY sehr ähnlich (siehe nächstes Kapitel), allerdings sind weitere Vergleiche der Architektur und Ergebnisse aufgrund der fehlenden Information leider nicht möglich.

6

Die JIFFY-Architektur

> It was obvious that it couldn't think. Part of it was clockwork. A lot of it was a giant ant farm (the interface, where the ants rode up and down on a little paternoster that turned a significant cogwheel was a little masterpiece, he thought) and the intricately controlled rushing of the ants through their maze of glass tubing was the most important part of the whole thing.
> But a lot of it had just ... accumulated, like the aquarium and wind chimes which now seemed to be essential. A mouse had built a nest in the middle of it all and had been allowed to become a fixture, since the thing stopped working when they took it out. Nothing in that assemblage could possibly think, except in fairly limited ways about cheese or sugar. Nevertheless ... in the middle of the night, when Hex was working hard, and the tubes rustled with the toiling ants, and things suddenly went 'clonk' for no obvious reason, and the aquarium had been lowered on its davits so that the operator would have some-thing to watch during the long hours ... nevertheless, then a man might begin to speculate about what a brain was and what thought was and whether things that weren't alive could think and whether a brain was just a more complicated version of Hex (or, around 4 a.m., when bits of the clockwork reversed direction suddenly and the mice squeaked, a less complicated version of Hex) and wonder if the whole produced something not apparently inherent in the parts.
>
> *Terry Pratchett, Interesting Times*

6.1 Überlegungen zum Beschleunigerkonzept

Ziel der Arbeit war es, unter Zuhilfenahme von externer Hardware ein effizientes Beschleunigungskonzept für die JVM zu entwickeln. Als Realisierungshardware haben sich FPGAs angeboten, da diese sehr universell benutzbar sind. Dies erlaubt eine flexible Nutzung, erzeugt aber durch die FPGA-Eigenschaften Einschnitte in der gewünschten Funktionalität.

Der Einsatz von FPGAs als Java-Coprozessor bedingt zunächst die Frage nach der Praktikabilität. Ein erster, eher theoretischer Ansatz dazu ist das Gesetz von Amdahl [13]. Es beschreibt die erreichbare Beschleunigung T eines Gesamtsystems, wenn ein Bruchteil α davon um den Faktor s schneller ausgeführt wird:

$$T = \frac{1}{(1-\alpha) + \frac{\alpha}{s}}$$

Damit ist eine obere Grenze der Beschleunigung berechenbar: Wenn der Bruchteil α bekannt ist, kann auch durch hohe Werte von s die Gesamtbeschleunigung nicht mehr verbessert werden.

In [48] wird diese Regel auf einige Anwendungen von FPGA-Coprozessoren übertragen und gezeigt, dass im Normalfall keine Vorteile zu erwarten sind. Als Probleme wurden unter anderem auch die Übertragung der Funktionsparameter identifiziert, die sogar teilweise zu eine Verlangsamung führen. Für fast alle Anwendungszwecke wäre der einfache Einsatz einer schnelleren CPU angebracht.

In diese Bewertung geht aber nicht die mögliche Parallelität von CPU und FPGA ein, ebensowenig Einschränkungen der verwendbaren Ressourcen aus technischen oder finanziellen Gründen. Von daher ist eine allgemeine Betrachtung der Praktikabilität ohne eine Beschreibung der Anforderungen und des Einsatzgebietes kaum möglich.

6.2 Anforderungen und Einsatzgebiet

Das JIFFY[1]-Konzept (bereits 1999 in [1] vorgestellt) beruht auf verschiedenen Anforderungen und Annahmen über Systemumgebung und Einsatzzweck eines nutzbringenden Java-Beschleunigers. Es entstand aus folgenden Überlegungen zu den Anforderungen an einen Java-Beschleuniger:

- Als **Systemumgebung** wird ein eingebettetes System im mittleren Leistungssegment angenommen. Das beinhaltet mindestens einen 32Bit-Prozessor, evtl. noch weitere (auch nicht binärkompatible) Zusatzprozessoren (Protokoll-CPUs, DSPs etc.). Typische Geräte sind hier Mobiltelefone, mobile Terminals oder Prozesssteuerungen im Consumer- und Industriebereich. Wenn mehrere Prozessoren im System vorhanden sind, sollte die Beschleunigungshardware ohne größere Modifikationen für alle zugreifbar sein (z.B. Update eines Protokollstacks auf Java-Basis für den Basisbandprozessor eines Mobiltelefons).

  Auf nur eine CPU zugeschnittene Hardwarelösungen (wie z.B. Jazelle (5.4.10), JStar (5.4.9)) sind damit unbrauchbar. Auch der Einsatz eines expliziten JVM-Prozessors (DeCaf (5.4.8) etc.) erlaubt keine parallele

[1] Akronym für eine bislang nicht näher definierte Wortkombination mit den Wörtern Java und FPGA, angelehnt an den engl. Ausdruck “in a jiffy”=’im Nu”.

Ausnutzung der verfügbaren Rechenleistung und der besonders im Kommunikationsbereich verfügbaren CPU-nahen Spezialhardware (FFT-Cores, Viterbi/Reed-Solomon-Dekoder, etc.).

Nicht geplant oder nicht sinnvoll für das JIFFY-System ist der Einsatz im unteren (Smart-Cards, 8 oder 16Bit Prozessoren) oder im oberen Leistungsbereich (Arbeitsplatz-PCs, Workstations etc.). Smart-Cards erfordern spezielle, auf den Einsatzzweck optimierte CPU-Kerne, Arbeitsplatzrechner besitzen inzwischen genügend Leistungsfähigkeit für reine Software-JIT-Compiler.

- Die **Adaptation** auf neue Prozessorarchitekturen sollte wenig bis keine Modifikationen der Beschleuniger-Hardware nach sich ziehen, alle Änderungen sollten sich (wenn überhaupt) nur auf der Softwareebene abspielen. Diese Anforderung erfüllen nur autonome Java-CPUs zufriedenstellend. Bisherige JIT-Ansätze sind entweder gar nicht flexibel (Jazelle) oder verlangen einen Neuentwurf des Chips (Nazomi, Machstream, siehe 5.4.11).

- Eine **Parallelarbeit** von Hauptprozessor und Java-Beschleuniger sollte möglich sein. Damit kann während der Arbeit des Java-Beschleunigers ein anderer Prozess den Prozessor benutzen und somit die Wartezeit auf die Ausführung der Java-Methode“verstecken”. Bei reinen JVM-Prozessoren oder JVM-Ergänzungen ist dies nicht möglich.

- **Einsatzzweck** soll die gelegentliche bis häufige Java-Ausführung auch für zeitkritische Bereiche sein. Darunter fallen die schon erwähnten portablen Protokollstacks, die eine effiziente Ausführung brauchen. Damit wird für ein JIT-System also eine schnelle bis sehr schnelle Übersetzung gefordert, die trotzdem eine gute Codeerzeugung bietet. Als Abschätzung dafür ist z.B. die Ausführungsgeschwindigkeit des Algorithmus, codiert in C/C++, brauchbar. Ein normaler JVM-Interpreter erreicht dabei im Mittel weniger als 10%.

  Interaktive Anwendungen profitieren von einem JIT-Übersetzer nur bei guter Codequalität und schneller Übersetzung, da die Antwortzeit bei der Bedienung eines “neuen”, noch nicht übersetzten Objekts (inklusive aller Unterobjekte) dann durch die Summe aus Übersetzungszeit und Ausführungszeit bestimmt wird. Wird diese zu lang, wird die Benutzeroberfläche als zu träge und unbedienbar empfunden (“Responsiveness” [143]).

- Die **Echtzeitfähigkeit** für andere, nicht Java-basierte Programme muss gewährleistet bleiben. Interrupts dürfen durch den Einsatz eines Java-Beschleunigers nicht verzögert werden, der Hauptprozessor muss trotz der

JVM-Emulation diese jederzeit unterbrechen und wiederaufnehmen können.

- **Hardwareaufwand**: Das JIFFY-Konzept basiert auf einem rekonfigurierbaren FPGA. Dies ist zunächst ein zusätzlicher Hardwareaufwand. Andererseits ermöglicht das FPGA neben der Java-Beschleunigung auch noch die Übernahme anderer Aufgaben und ergänzt evtl. kundenspezifische Schaltkreise im System. Durch die Rekonfigurabilität sind damit auch spätere Updates dieser Hardware möglich, dies wird von vorverdrahteten Beschleunigern nicht geleistet.

- Der zusätzliche **Speicherverbrauch** eines Hardwarebeschleunigers sollte so gering wie möglich ausfallen. In dieser Hinsicht sind die Instruktionspfadübersetzer (Jazelle, Nazomi) sehr gut, allerdings ermöglichen sie außer einfachem "Instructionfolding" kaum weitergehende Optimierungen während des Ablaufs.

  Der Speicherverbrauch eines JIT-Compilers ist zwar höher, allerdings auch die damit erzielbare Leistung. Ein sehr schneller JIT-Compiler ermöglicht zudem, selten durchlaufene Methoden nach der Übersetzung wieder zu löschen und so Speicherplatz einzusparen. Dies setzt eine zusätzliche Methodencacheverwaltung und eine Instrumentierung des erzeugten Native Codes voraus. Dieses ist aber ohne Probleme in JIFFY zu integrieren, da es auf definierten Schnittstellen aufsetzt.

- Die **Systemeinbindung** eines Java-Beschleunigers soll so flexibel wie möglich sein, das beinhaltet einfache Integration an eine schon existierende Java-Ausführungsplattform als auch (bei Hardware) eine möglichst universelle Anbindung des Beschleunigerkerns bzw. Chips an den Systembus oder andere Schnittstellen.

- Der **Stromverbrauch** sollte so gering wie möglich sein (besonders für mobile Anwendungen) und bei Inaktivität der Java-Beschleunigung vernachlässigbar sein. FPGA-basierte Lösungen sind hier im Nachteil, da sie einen etwas höheren Ruhestrom als vergleichbare Customchips besitzen. Allerdings ist bei neueren Typen (z.B. Virtex2) eine Art Suspendmodus möglich, der die Konfiguration aufrecht erhält, ohne die interne Logik mit Strom zu versorgen.

- Die **Skalierbarkeit** des Beschleunigerkonzepts muss auf absehbare Zeit gegeben sein. Technologieverbesserungen (z.B. Takterhöhungen) in der System-CPU **oder** in der Beschleunigerhardware sollten sich auch möglichst linear auf die erzielbare Geschwindigkeit auswirken. Der Java-

Beschleuniger darf sich nicht zum Nadelöhr entwickeln und muss auch bei schnellerer CPU einen signifikanten Vorteil in der Ausführung bringen.

Dieser Vorteil ist mit dem JIFFY-Konzept gegeben, da es nur in während der Übersetzungsphase im FPGA läuft, der erzeugte Laufzeit-Code profitiert direkt von einer Beschleunigung der Haupt-CPU.

6.3 Wahl des Beschleunigerkonzepts

Nachdem die direkte Integration eines JVM-Beschleunigers in eine CPU (ähnlich Jazelle oder "echten" Java-CPUs) als zu wenig portabel und originell eingeschätzt wurde, wurden die verschiedenen Übersetzungskonzepte auf die Tauglichkeit für den FPGA-Einsatz untersucht.

Meiner Meinung nach bieten keine der vorhanden Übersetzungsmöglichkeiten eine vielversprechende FPGA-Implementierung. Entweder ist der Algorithmus zu komplex für FPGAs (z.B. CACAO) oder die zu erwartende Beschleunigung rechtfertigt nicht den HW-Aufwand.

Die reine Übersetzung in nativen Maschinencode ohne weitere Optimierungen an sich ist trivial und mit relativ wenig Aufwand möglich (siehe TYA), allerdings ist die resultierende Ablaufgeschwindigkeit durch die Stackemulation sehr eingeschränkt. Daher sind weitere Optimierungen notwendig, deren Auswahl durch das FPGA/JIT-Umfeld wieder eingeschränkt wird.

Globale Optimierungen wie z.B. Linearisierung von Rekursionen, Funktions-Inlining oder Blocking von Arrays[2] sind aufgrund der JIT-Struktur nicht möglich. Es wäre zwar denkbar, diese Optimierungen nach jedem Laden von neuen Methode wieder durchzuführen (ähnlich den wiederholten Optimierungsläufen von FX!32), dann müsste das FPGA dazu einen schnellen Zugriff auf alle Strukturen der VM besitzen. Die nötigen Analysealgorithmen erscheinen insgesamt für eine reine HW-Implementierung zu aufwendig. Bei einigen Optimierungen bietet es sich zudem an, sie bereits zu Compilationszeit auf JVM-Ebene durchzuführen, da zu diesem Zeitpunkt wesentlich mehr Informationen über den Code vorhanden sind.

Lokale Optimierungen sind durch den eingeschränkten Sichtbereich auf eine Methode prinzipiell besser für eine Hardware-Optimierung geeignet, allerdings scheinen auch hier effiziente Algorithmen (z.B. Datenflussanalyse, Graph-Coloring, Loop-Unrolling) in fester Verdrahtung sehr aufwendig. Natürlich ist es denkbar, diese Algorithmen in einer Art Mikroprogramm zu implementieren, was aber den Vorteil einer Hardware-Übersetzung (z.B. massiv ausnutzbare Paralle-

[2] Optimierungsart durch Vergrößerung von Feldern, sodass Caches besser ausgenutzt werden können (Vermeidung von Cachetrashing).

lität) zunichte machen würde. Das in CACAO implementierte Registerallokierungskonzept ist zwar relativ einfach, aber nur sinnvoll, wenn eine ausreichende Anzahl von CPU-Registern zur Verfügung steht. Dies ist im 80386 nicht gegeben, ebenso können RISC-basierte Architekturen Registereinschränkungen durch das Betriebssystem aufweisen (z.B. für schnelle Interrupthandler).

Aus diesen Gründen wurde im JIFFY-Konzept zunächst auf ausgefeiltere Optimierungen verzichtet, und der Schwerpunkt auf die effiziente Erzeugung des Codes und die Eliminierung der Stackemulation gelegt, die einen Großteil der Ineffizienz ausmacht. Es wird davon ausgegangen, dass der JVM-Code vom Java-Compiler soweit optimiert wurde, dass im Software-Sinn triviale Optimierungen (z.B. Eliminierung von "totem", d.h. nie ausführbaren Code) bereits durchgeführt wurden.

Daher wurde als Grundlage für die Übersetzung die im Softwarebereich oft benutzte Methode der Transformation in eine Zwischensprache gewählt, auf der auch alle Optimierungen in Peepholetechnik ausgeführt werden. Allerdings bedingt die Umsetzung auf FPGA-Ausführung einige Erweiterungen als auch Einschränkungen dieses Prinzips.

6.3.1 Evaluierungsphasen und Vorgehen

Zur Untersuchung des gewählten Konzepts wurden zunächst einfache Tests, basierend auf der reinen Textersetzung der Ausgabe eines JVM-Disassemblers, erprobt. Da dadurch die gesamte JVM-Laufzeitumgebung nicht mehr vorhanden war, waren nur einfache JVM-Methoden (z.B. sieve-Benchmark) ohne weitere Methodenaufrufe zu testen. Allerdings zeigte bereits diese erste vor-prototypische Evaluierung ohne aufwendige Optimierungen eine erzielbare Geschwindigkeit von ca. 20% bis 50% von C-Compilaten mit identischer Funktion. Gegenüber der JVM-Interpretierung war eine erhebliche Steigerung der Geschwindigkeit festzustellen.

Aufgrund dieser vielversprechenden Ergebnisse wurde dieses Konzept als Ausgangsbasis für ein umfassendes C-Modell der Übersetzung gewählt, das leicht erweiterbar und debugging-freundlich ist. Dabei sollte das Modell dieselben Schnittstellen zur Verfügung stellen wie die spätere Hardware. Auch die internen Abläufe sollten schon relativ hardwarenah dargestellt werden, damit bereits das C-Modell eine grobe Abschätzung der Laufzeit erlaubt, die anhand der zu erwartenden Speicherzugriffe und Takten ermittelt werden.

Um auch komplexere JVM-Codes testen zu können, wurde das C-Modell als JIT-Modul ("Plugin") für das JDK-1.2 entwickelt, wobei die Integration über die Java Native Code API [177] abläuft. Damit wurde es auch möglich, verschiedene Möglichkeiten der Integration von Laufzeitsystem (JRE) und Übersetzer zu testen. Gerade die objektorientierten Merkmale führen hierbei zu Schwierigkeiten, da die Hardware zur Übersetzungszeit nur auf eine sehr beschränkte Informationsbasis

zurückgreift und daher viele Schritte erst zur Laufzeit möglich sind. Hierbei ist eine effiziente Aufteilung der Aufgaben sowohl bezüglich des HW-Aufwands als auch der Laufzeit notwendig.

Während der Erstellung des C-Modells wurden die zugrundeliegenden Datenstrukturen und Algorithmen mehrfach überarbeitet. Besonders die Zwischensprache und die Assemblerstufe erfuhren Änderungen, um auch die Zwischenschichten möglichst einfach und effizient zu gestalten.

Um das Ziel der Architekturunabhängigkeit zu erfüllen, wurde das System exemplarisch für i586 (als CISC-Vertreter) und die Alpha-164-Architektur (als typischer RISC-Vertreter) entwickelt. Dabei haben die CISC-Restriktionen wesentlich mehr Auswirkungen auf die Struktur der Algorithmen als die regelmäßige RISC-Architektur.

Die während dieser Evaluierung und kontinuierlichen Entwicklung des Modells gewonnen Erkenntnisse bezüglich des Übersetzungskonzepts und der benutzen Zwischensprache werden in den folgenden Abschnitten besprochen, die konkreten Abwägungen und Lösungsmöglichkeiten für die Integration in ein JVM-Laufzeitsystem in Kapitel 7.

6.4 JIFFY Einbindung

Eine völlig vom Laufzeitsystem (JRE) unabhängige Einbindung eines JIT ist nicht oder nur mit großen Leistungseinbußen möglich, da gerade die während der Laufzeit benötigten Informationen über Klassen und Methoden schnell verfügbar sein müssen.

Um diese Umwege zu vermeiden und dennoch eine gewisse Unabhängigkeit zu erzielen, wurde die in Abb. 6.1 gezeigte Struktur gewählt. Der darauf aufbauende Datenfluss und die damit verbundene Kommunikation ist in Abb. 6.2 dargestellt.

Ausgehend vom Java-Laufzeitsystem werden JRE-spezifische Anfragen durch eine Zwischenschicht geleitet, die einerseits eine Umsetzung vornimmt, andererseits daraus auch einen eigenen Datenbestand ableitet, auf den die übersetzten Methoden schnell und JRE-unabhängig zugreifen können. Dieser Datenbestand beschränkt sich aber im wesentlichen auf eine komprimierte Version des Konstantenpools, der nur noch Einsprungpunkte für aufgelöste Funktionen bzw. Objekte enthält und durch den Verzicht auf Stringbeschreibungen wesentlich kürzer ist.

Der übersetzte Code benutzt Funktionen in der JIFFY-JRE-Interfaceschicht, um auf Objektinformationen zuzugreifen. Für spezielle, kritische Abschnitte (z.B. Arrayzugriffe oder Parameterübergabe) müssen allerdings auch direkte Zugriffe auf JRE-Strukturen codiert werden. Diese Aufgaben sind aber nicht von der Über-

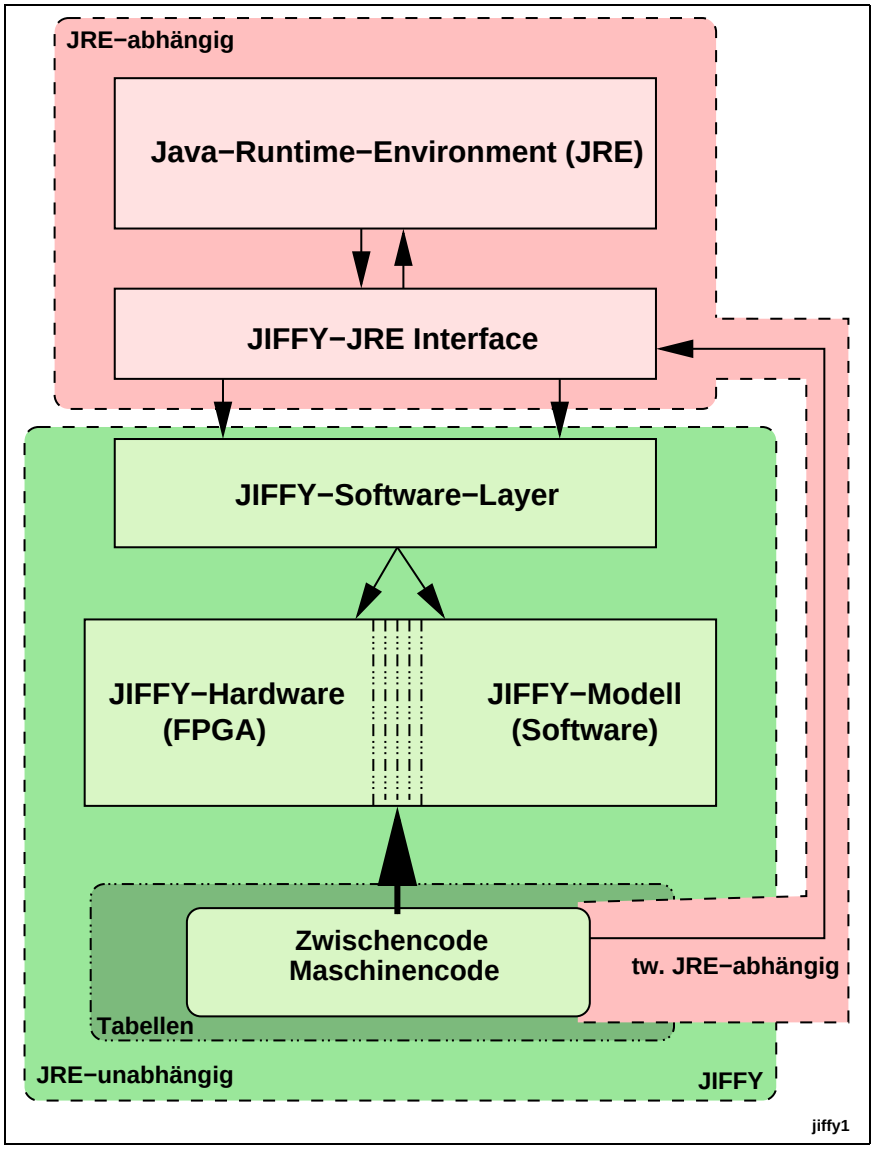

Abbildung 6.1: Einbindung von JIFFY in das Laufzeitsystem

setzungshardware abhängig, und, verglichen mit der übrigen Codebasis, nicht umfangreich.

Die JIFFY-JRE-Interfaceschicht bildet die Anfragen des JRE-Systems (z.B. die Compilation einer Methode) nach der Erzeugung der komprimierten Informationen auf eine wohldefinierte Schnittstelle zum eigentlichen Übersetzungsteil von JIFFY ab. Dieser Abschnitt teilt sich dabei in eine immer vorhandene Software-Schicht zur Initialisierung der Übersetzung und in die konkrete Implementierung

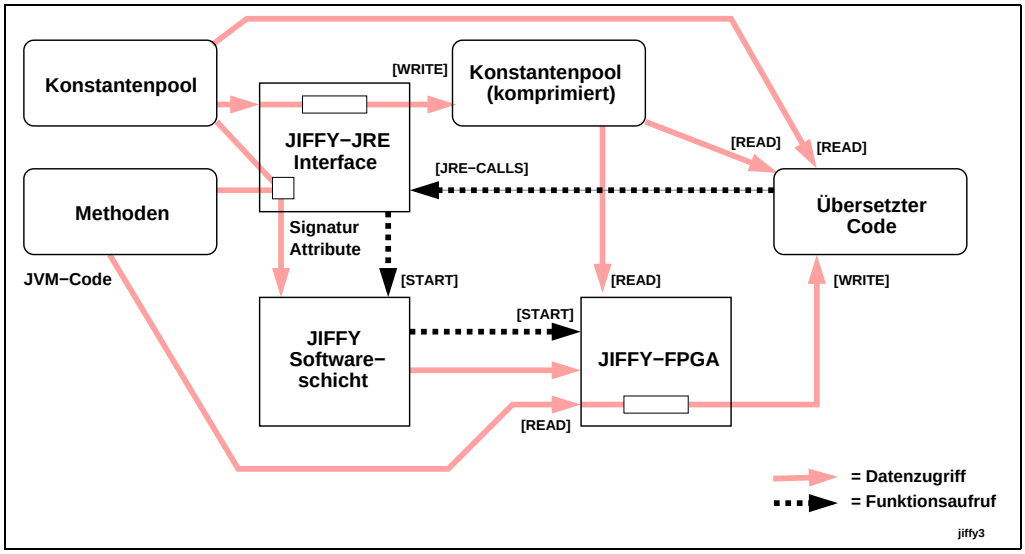

Abbildung 6.2: Datenfluss und Kommunikation

der Übersetzung auf. Diese Übersetzung kann sowohl rein in Software ablaufen, als auch in Hardware implementiert sein.

Zur Erprobung der Konzepte wird der Übersetzungsvorgang zunächst in C modelliert, wobei die Grenze zwischen Software und der späteren Implementierung im FPGA variabel ist, d.h. Teile der Übersetzung können aufgrund definierter Schnittstellen weiterhin in Software verbleiben. Dies vereinfacht die Entwicklung erheblich.

Sowohl Software- als auch Hardwareübersetzung lesen die Daten aus zwei größeren Tabellen (maximal ca. 40kB). Davon ist nur die sog. Paracode-Tabelle, also die Tabelle zur Übersetzung der Zwischensprache in die Zielarchitektur, von der gewählten Zielarchitektur und vom Laufzeitsystem abhängig. Die zweite Tabelle ist für alle Architekturen identisch und hängt indirekt nur von der JVM ab. JVM-Erweiterungen können dabei größtenteils bereits in dieser Tabelle abgefangen werden.

Der Vorteil der beschriebenen Schichtung besteht in der guten Trennung zwischen JRE und der eigentlichen Übersetzungsmaschine und auch der flexiblen Evaluierungs- und Entwicklungsmöglichkeit. Jeder Punkt im Übersetzungsablauf ist überwachbar, jeder Vorgang kann aus der Hardware wieder in Software verlagert und besser instrumentiert werden.

6.5 JIFFY Übersetzungsvorgang

6.5.1 Übersicht

Der Übersetzungsfluss und die geplante Speicherlokalisierung ist in Abb. 6.3 dargestellt. Ausgangspunkt der Übersetzung ist der JVM-Bytecode der zu übersetzenden Methode. Diese Befehlsfolge wird zunächst in einem einfachen JVM-Parser-Durchlauf analysiert. Das wesentliche Ergebnis ist hierbei eine Sprungzieltabelle.

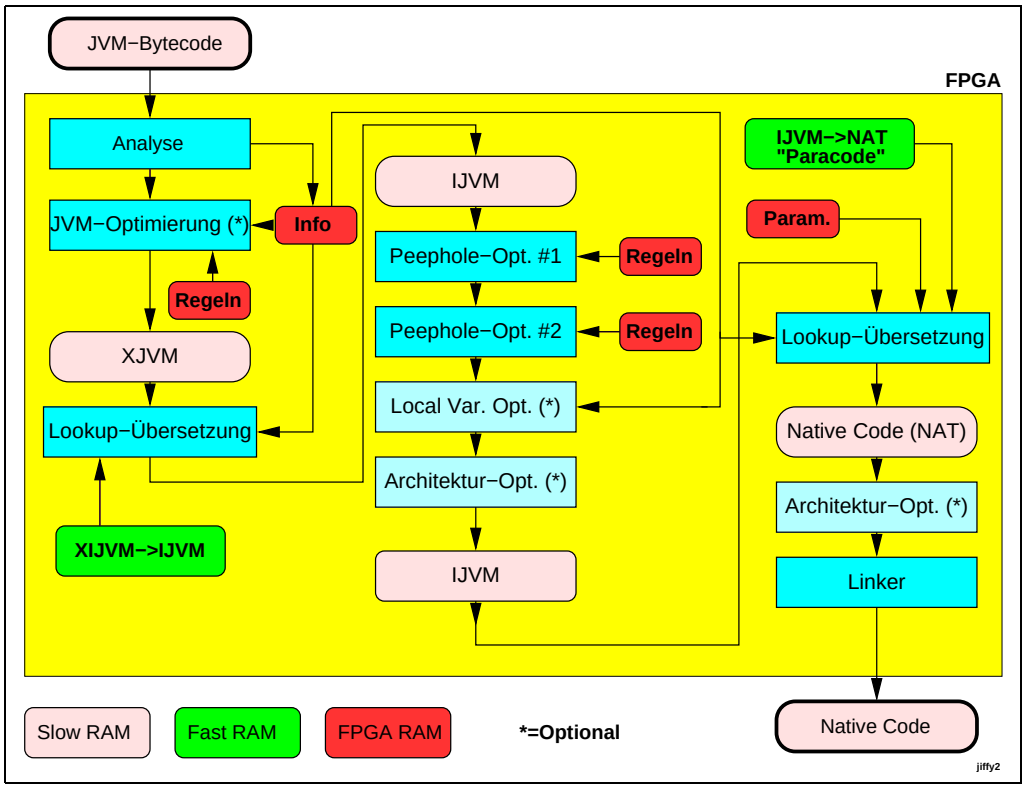

Abbildung 6.3: Ablauf der Übersetzung

Ein optionaler Optimierungsdurchgang auf den JVM-Befehlen ersetzt basierend auf wenigen Regeln gewisse Befehlssequenzen durch Spezialbefehle (“Extended” JVM, XJVM) um eine spätere Optimierung zu vereinfachen. Hierbei ist ein Zugriff auf eine komprimierte Version des Konstantenpools notwendig, um Objekt- und Rückgabetypen einzufügen.

Anschließend wird mit einer tabellengestützten Übersetzung der XJVM-Code in eine interne Zwischensprache umgesetzt (Intermediate JVM, IJVM). Auf dieser IJVM-Ebene finden alle weiteren Optimierungen statt.

Diese Optimierungen bestehen zunächst aus zwei bis drei kaskadierten Peepholeoptimierungen, die auf zwei unterschiedlichen Regelsätzen arbeiten. Als zusätzliche Verbesserungen sind bei registerreichen Zielplattformen auch noch Optimierungen bezüglich der Lage der lokalen Variablen möglich oder auch die Ersetzung bestimmter IJVM-Befehlsfolgen in Platzhalterbefehle für hochoptimierte Assemblersequenzen.

Das Ergebnis dieser Optimierungschritte ist weiterhin die architekturunabhängige IJVM. In der letzten Phase werden diese Befehle mit einem tabellengestützten Algorithmus in nativen Code (**NAT**) umgesetzt. Da diese Stufe einen großen Einfluss auf die Geschwindigkeit hat, wurden die Möglichkeiten der Codegenerierung aber im Vergleich zur JVM→IJVM-Übersetzung wesentlich flexibler gestaltet, sodass sich im Prinzip ein voll ausgestatteter Assembler im FPGA ergibt. Dieser Assembler ist in seiner Struktur aber unabhängig von der Zielarchitektur und basiert allein auf einer relativ kleinen Tabelle (parametrisierbarer Assemblercode, “Paracode”).

Im Anschluss an die eigentliche Übersetzung in den nativen Maschinencode und einer optionalen architekturabhängigen Optimierung müssen die Sprungziele angepasst werden. D.h. ein “Linken” des Codes ist erforderlich. Durch die während der Übersetzung gesammelten Daten und der Nutzung von zwei möglichen Speicherstrukturen ist das Linken mit nahezu linearem Aufwand (bezogen auf die Anzahl der Sprungziele) möglich. Die hierfür nötige Hardware besteht dabei hauptsächlich in der effizienten Speicherverwaltung der Sprungziele, das eigentliche Einfügen der Werte ist wenig aufwendig.

6.5.2 JVM-Analyzer

Vor der eigentlichen Übersetzung steht eine einfache Analysephase des JVM-Codes der zu übersetzenden Methode. Zur Zeit beinhaltet sie lediglich eine Analyse und Speicherung der Sprungziele des JVM-Codes, damit für die anschließende Übersetzung die Basisblöcke definiert werden können, innerhalb derer die Peepholeoptimierung arbeitet.

Die Tabelle der Sprungziele beinhaltet dabei auch die Unterscheidung, ob das Sprungziel von einem oder von mehreren Befehlen angesprungen wird. Dieses Wissen ist bei bestimmten Optimierungen hilfreich, die dadurch über Basisblockgrenzen hinweg arbeiten können.

Für das abschließende Linken ist diese Tabelle nicht mehr wichtig, da nach der Übersetzung in den Zwischencode Sprungziele als spezieller Befehl “in-place” kodiert (Opcode `LABEL`) werden.

Der für die Analysephase notwendige Speicherverbrauch dieser Tabelle liegt bei 16KByte, da für jede der möglichen 65536 Sprungadressen zwei Bit zur Markierung verbraucht werden.

Die Analysephase erlaubt auch weitergehende Analysen, wie z.B. die Nutzungshäufigkeit von lokalen Variablen. Je nach Zielarchitektur ist es dann möglich, diese vom Stack in Register auszulagern.

6.5.3 JVM-Optimierung

6.5.3.1 Verschmelzung von Befehlen, Instructionfolding

Die Untersuchung des erzeugten Zwischencodes nach der Peepholeoptimierung zeigte, dass einige Sequenzen von JVM-Befehlen, die aufgrund der JVM-Struktur recht häufig vom Java-Compiler erzeugt werden, nur sehr ineffizient übersetzt werden können. Diese können für eine bessere Leistung zu einem Befehl miteinander verschmolzen werden.

Als häufigstes Konstrukt fielen z.B. Vergleiche zweier Stackwerte im Long-Format mit anschließendem bedingtem Sprung auf:

Der Java-Code

```
long a,b;
...
if (a>b)
{
  ...
}
```

wird dabei üblicherweise zu folgender JVM-Sequenz:

```
lcmp
ifgt ...
```

Der `lcmp`-Befehl berechnet dabei die Signum-Funktion der Differenz der beiden Stackargumente. Würden beide Befehl einzeln übersetzt, würde also jeder einfache Vergleich zunächst `sgn()` berechnen (und erfordert mindestens einen nicht vorhersagbaren Sprung), und basierend auf dem Ergebnis würde nochmals ein Vergleich mit Null stattfinden, was ebenfalls nicht vorhersagbar ist. Da in der Kombination der Befehle die Berechnung von `sgn(a-b)` aber gar nicht notwendig ist, findet in JIFFY bereits auf JVM-Ebene eine optionale Optimierung (Instructionfolding) statt. Dazu werden diese Kombinationen in nicht standardisierte JVM-Befehle umgewandelt, die später in einen optimierten Maschinencode übersetzt werden. Beispielsweise wird die `lcmp/ifgt`-Sequenz in `if_lcmpeq` umgewandelt. Auf 64-Bit CPUs (z.B. Alpha) ist `if_lcmpeq` in damit zwei statt 4 bis 10 (je nach CPU-Typ) Befehlen abgearbeitet, auf x86 sind es nur noch 4 Befehle statt 9 ohne diese Optimierung.

Neben diesen allgemeingültigen Optimierungen ist es in dieser Phase weiterhin möglich, bestimmte Zielarchitektureigenschaften auszunutzen, wie z.B. Multimediaerweiterungen. Durch die Stackarbeitsweise der JVM sind die bei Formelauswertungen entstehenden Bytecodefolgen relativ starr und sind leicht erkennbar und durch spezielle JVM-Befehle zu ersetzen.

Im Gegensatz zum Instructionfolding bei anderen JVM-Prozessoren erfolgt bei der beschriebenen Vorgehensweise noch keine Verschmelzung von Zugriffen auf lokale Variablen mit anschließender arithmetischen Operation.

6.5.3.2 Typisierung von Methodenaufrufen bzw. CP-Zugriffen

Die JVM-Methodenaufrufe sind für die Rückgabe von Funktionsergebnissen ohne Typ generisch, d.h. ohne einen in den `invoke`-Befehl kodierten Rückgabetyp. Das Ergebnis mit einer vom Rückgabetyp abhängigen Länge wird dabei direkt auf den Stack gelegt. Damit ergibt sich bei der Optimierung der Rückgabe von Funktionsergebnissen das Problem, dass diese Rückgabemethode mit keiner der üblichen, von C stammenden Aufrufstandards übereinstimmt, bei der der Rückgabewert in einem Register steht. Da die Funktionsparameter der aufgerufenen Methode dabei vorher auf den Stack gegeben wurden, muss der Rücksprung nicht nur diese Parameter löschen (ähnlich der frühen PASCAL-Aufrufkonventionen), sondern anschließend auch noch den Rückgabewert auf den Stack schreiben. Tests auf Alpha- und x86-Plattformen ergaben dabei nur sehr ineffiziente Möglichkeiten. Diese Art der Parameterrückgabe zeigte sich als sehr ungünstig und erschwert die Integration in eine C-basierte Laufzeitumgebung stark.

Zusätzlich kommt hinzu, dass die Peepholeoptimierung nur auf konsistenten Typen arbeitet, da der Ablageort in der Zielarchitektur auf IJVM-Ebene nicht definiert ist. Daraus ergibt sich, dass sich die implizite, typenlose Pushoperation der `invoke`-Befehle nicht weiteroptimieren lässt. Dieser Umstand trifft auch die typenlosen Zugriffe auf Konstantenpool, Klassen- und Objektfelder mittels `ldc/getstatic/getfield`, deren Umsetzung in die Zielarchitektur ohne Typisierung ebenfalls nicht einfach möglich ist.

Ähnlich verhält es sich auch mit `putstatic/putfield`, die einen typlosen Wert vom Stack nehmen und abspeichern, hier wird eine einfache Optimierung und spezialisierte Behandlung der Typen verhindert.

Aus diesem Grund erfolgt beim Einlesen der JVM-Codes eine Typisierung aller `invoke/ldc/getstatic/getfield/putstatic/putfield`-Befehle[3] durch eine Übersetzung in nicht-standardisierte, typisierte JVM-Befehle, die entsprechende Gegenstücke in der im folgenden besprochenen “Zwischenarchitektur” IJVM besitzen. Dazu wird bei jedem dieser Befehle der im

[3] Dies sind alles Befehle, die einen Wert zurückliefern, entweder durch direkten Methodenaufruf oder durch Zugriff auf den Konstantenpool

Konstantenpool notierte Typ betrachtet. Als weitere Optimierung ist hierbei eine verkürzte Version des Konstantenpools möglich, in dem nur der Element- bzw. Rückgabetyp verzeichnet ist. Da nur wenig Speicher (max. 4Bits) pro Index notwendig sind, ist diese Tabelle im FPGA selbst oder FPGA-nah unterzubringen und erlaubt einen latenzarmen Zugriff.

6.5.4 Die Zwischenarchitektur IJVM

Die Intermediate Java Virtual Machine (IJVM) ist eine virtuelle Architektur im JIFFY-System, die aus der JVM so abgeleitet wurde, dass sie der JVM sehr ähnlich ist. Allerdings arbeitet sie auf Registern und besitzt einen Universalstack, der für Daten, lokale Variablen und Rücksprungadressen gemeinsam genutzt wird.

Ziel beim Entwurf der IJVM war es, diese Architektur als einfachen Übergang zwischen JVM und nativem Maschinencode zu benutzen. Daher besitzt sie Eigenschaften, die für einen "echten" Prozessor immer noch zu starke Einschränkungen bzw. Komplexitäten bedeuten, andererseits kann sie aber auch als "Superset" von JVM und der Zielarchitektur gesehen werden:

1. Die IJVM besitzt nur drei Universalregister für Integer und Fließkomma, jeweils mit 32 und 64Bit nutzbar. Diese Beschränkung resultiert aus der Eigenschaft der JVM, für Befehle maximal drei Stackoperanden zu benötigen. Der einzige JVM-Befehl mit drei Operanden ist der Arrayschreibzugriff (Array-Referenz, Index und Schreibwert). Damit sind mit nur drei Registern alle JVM Befehle emulierbar.

   Andererseits erlaubt die Reduktion auf drei Register eine sehr einfache Umsetzung auf registerarme CISC-Architekturen, wie dem x86. Würde die IJVM hier intern mit mehr Registern arbeiten, wäre zwingend eine Registernutzungsstrategie notwendig, die die Übersetzungshardware verkomplizieren würde. Prinzipiell ist es im JIFFY-Konzept aber möglich, mehr als diese drei Register auszunutzen.

2. Die Stackbenutzung ist in der IJVM explizit, d.h es gibt PUSH und POP-Befehle. Eine Überprüfung auf Korrektheit der Typen und Zugriffe findet nicht statt, ist aber als "Nebenprodukt" der Optimierung benutzbar (siehe Abschnitt 6.7).

   Funktionsparameter und lokale Variablen liegen auf dem Stack. Um eine Zielarchitekturabhängigkeit beim Zugriff darauf zu vermeiden, kann auf diese Werte aber wie in der JVM nur über spezielle Funktionen zugegriffen werden.

3. Die IJVM besitzt alle funktionellen Befehle der JVM nahezu unverändert, d.h. besonders die objektorientierten Befehle sind bis auf die Registernutzung und die Typisierung (siehe 6.5.3.2) von der Funktionalität identisch.

4. Zusätzlich zu den JVM-verwandten Befehlen sind einige grundlegende Befehle hinzugekommen, die die Schnittstelle zur Zielarchitektur bilden. Darunter fallen explizite Speicherzugriffe, aber auch Befehle zum Zugriff auf Konstantenpoolelemente oder zur Übertragung von Parametern in das Laufzeitsystem. Diese Befehle sind zwar maschinenunabhängig, aber einfach auf jede Zielplattform zu übersetzen. Dies ermöglicht es, architekturspezifische Zugriffsmethoden bis zum Ende des Übersetzungsvorganges allgemeingültig zu halten.

   Diese Unabhängigkeit wird auch für die Erzeugung der Schnittstellen zwischen Interpreter und JIT-Code ausgenutzt, indem die Parameterübertragungen bzw. Hilfsroutinen (“Stubs”) zunächst in allgemeinem IJVM-Code synthetisiert (siehe 7.1.6) und erst anschließend in nativen Code übersetzt werden.

5. Einige Konstrukte in der JVM lassen sich zu einem Makro zusammenfassen, die so optimiert wesentlich effizienter sind als in einer Einzelabfolge von IJVM-Befehlen. Daher gibt es einige IJVM-Befehle, die eine Sequenz von JVM-Befehle ersetzen. Diese Ersetzung kann wie bereits beschrieben, beim Einlesen des JVM-Codes erfolgen.

6.5.5 IJVM-Befehlsaufbau

Um den Aufwand bei der Befehlsdekodierung zu senken, besitzen IJVM-Befehle ein festes, unveränderliches Format (Bild 6.4), in dem alle potentiellen Parameter codiert sind. Dies führt zwar zu einem Mehrverbrauch an Speicher, der aber durch einfachere Hardware und höhere Geschwindigkeit bei der Dekodierphase gerechtfertigt ist.

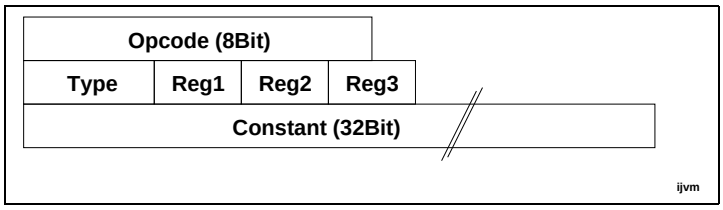

Abbildung 6.4: IJVM-Opcodeformat

Für den eigentlichen Opcode sind 8 Bit reserviert, damit sind also ohne Präfixcodes 256 Befehle möglich. Die JVM selbst hat ca. 200 Befehle, somit sind ca.

50 Zusatzbefehle auch ohne Präfixkommandos codierbar. Nicht alle JVM-Befehle besitzen ein direktes Gegenstück im IJVM-Opcode, da unterschiedliche Parameterlängen, Parametertypen oder fest kodierte Werte durch universelle Befehle ersetzt bzw. im zusätzlichen Typfeld kodiert sind.

Die Beschreibung der IJVM-Befehle in der Übersetzungstabelle (XIJVM) enthält noch weitere Felder, die für das Übertragen von Informationen aus dem JVM-Befehl benutzt werden. Diese werden aber im übersetzten IJVM-Befehl nicht mehr benötigt.

Weitere Felder im IJVM-Format des Softwaremodells dienen zum Monitoring, sind also für die eigentliche Übersetzungsfunktion nicht notwendig und werden in der HW-Implementierung weggelassen.

6.5.5.1 IJVM-Typen

Der Typ, auf dem ein IJVM-Befehl operiert, ist in einem 3 Bit-Feld angegeben und umfasst alle in der JVM vorhandenen Basistypen, allerdings wird bereits in der ersten Übersetzungsphase in die IJVM-Ebene der Referenztyp (A) durch den für die Zielarchitektur passenden Integertyp (I bzw. L, 32 oder 64 Bit) ersetzt.

Je nach Befehlsgruppe sind verschiedene Typen zugelassen, was durch folgende, generische Befehlsuffixe beschrieben wird:

- _X: Integer, Long, Reference, Float, Double
- _Y: Integer, Long
- _Z: Integer, Long, Float, Double

Andere Typen (z.B. Char), die von einigen Befehlen verarbeitet werden können, sind in diesen Suffixen nicht enthalten, da sie nur selten vorkommen und durch fest typisierte Befehle ersetzt werden.

6.5.5.2 IJVM-Befehle

Die wichtigsten von der IJVM verarbeiteten Befehle sind in Tabelle 6.1 aufgeführt. Die Befehle, die in äquivalenter Funktion auch in der JVM vorhanden sind, sind (zuzüglich der Typerweiterung) identisch benannt.

Im Vergleich zur JVM besitzt die IJVM folgende Zusatzbefehle, die architekturabhängige Eigenschaften und Implementierungsdetails verstecken:

- PUSH_X und POP_X

Stackoperationen:
`PUSH_X, POP_X`
Transferoperationen:
`MOVEC_X, MOVE_X, MOVEMR_X, MOVERM_X`
`MOVELVR_X, MOVELRV_X`
Arrayoperationen:
`MOVERA_X, MOVEAR_X`
Bitoperationen:
`AND_Y, OR_Y, XOR_Y, SHL_Y, SHR_Y, SHRU_Y`
Arithmetik:
`ADD_Z, SUB_Z, MUL_Z, DIV_Z, REM_Z, NEG_Z`
Vergleiche:
`CMP_Z, TST_Z, SGN_Z`
Konstantenarithmetik:
`ADDC_I, SUBC_I, CMPC_I`
Konvertierungen:
`CVTI_Z, CVTL_Z, CVTF_Z, CVTD_Z, CVTS_Z, CVTB_Z`
Sprünge:
`BRA, BIcc, BFcc, TRAP, TABLESWITCH, LOOKUPSWITCH`
JVM-Funktionen:
`NEW, NEWARRAY, ANEWARRAY, ARRAYLENGTH, MULTIANEWARRY`
`CHECKCAST, INSTANCEOF, MONITORENTER, MONITOREXIT`
`INVOKEVIRTUAL, INVOKESPECIAL, INVOKESTATIC`
`INVOKEINTERFACE, ATHROW`
Hilfsfunktionen:
`GAFI, GSVI, GFVI, RGAFI, LDC_X`
Funktionsprolog- und Epilog:
`START_FUNC_NORMAL, START_FUNC_NOLV, RETURN_NORMAL`
`RETURN_NOLV, RETDATA_X, RETARG_Z`
Stubs/Funktionen:
`DEFEXC, JITSTUB, CALLJIT, RETJIT, INVIRTSTUB`
Parameterübergabe:
`INITGETI, GETIARGI, GETIARGL, GETIARGA`
`INITGETN, GETNARGI, GETNARGL, GETNARGA`
Assemblerfunktionen:
`LABEL, LABELX, CONST_32, CONST_PC`

Tabelle 6.1: IJVM-Befehle

Explizite Zugriffsmöglichkeit auf den Datenstack. Eine Typüberprüfung erfolgt nicht, es wird vorrausgesetzt, dass diese bereits im JVM-Bytecodeverifier stattgefunden hat[4].

- MOVE_X

  Kopieren von Werten zwischen Registern.

- MOVEMR_X (**M**emory-**R**egister) und MOVERM_X

  Lese- bzw. Schreibzugriffe auf eine Speicheradresse.

- MOVELVR_X (**L**ocal**V**ariable-**R**egister) und MOVELRV_X

  Lese- bzw. Schreibzugriffe auf lokale Variablen anhand ihrer Nummer auf dem Stack.

- MOVEAR_X (**A**rray-**R**egister) und MOVERA_X

  Lese- bzw. Schreibzugriffe auf Arrays anhand Objektadresse und Elementindex.

- GAFI (Get Address from Index), GSVI und GFVI (Get static/field value address from index)

  Zugriffe auf Objektadresse über den Konstantenpool (typunabhängig).

- START_FUNC_* und RETURN_*

  Funktionsprolog- und Epilog zum Sichern von Registern und Stackframe für verschiedene Funktionstypen (keine lokalen Variablen, “Leaf”-Funktion) und zum Methodenrücksprung.

- RETDATA_X

  Systemspezifische Rückgabe des Funktionsergebnisses.

- INITGET* und GET*ARG*

  Transfer von Parametern beim Übergang Interpreter→JIT und JIT→Interpreter (siehe auch 7.1.6).

- LABEL und LABELX

  Markierung für Sprungziel, wird als Trennung der Basicblocks bei der Optimierung verwendet.

[4] Als “Nebeneffekt” der PUSH-POP-Optimierung kann jedoch eine einfache Überprüfung der korrekten Typisierung stattfinden, siehe auch 6.7.

- CONST_32 und CONST_PC

  Einfügen von Konstanten in den erzeugten Code. Dies erlaubt das direkte Übernehmen von Tabellen für `tableswitch` und `lookupswitch`, die erst zur Laufzeit ausgewertet werden. CONST_PC besitzt als Parameter einen JVM-PC-Wert, der im abschließenden Linken durch die entsprechende Ziel-CPU-Adresse ersetzt wird.

6.5.5.3 IJVM-Register

Die Felder für die maximal drei Register pro Befehl sind zunächst mit jeweils 2 Bit benutzt, womit sich also maximal 4 architekturabhängige Register ansprechen lassen, zur Zeit werden allerdings nur drei genutzt (r1, r2, r3). Durch die Erweiterung dieser Felder ist es möglich, in späteren Optimierungsstufen auf mehr Register zurückzugreifen, um Stackzugriffe zu sparen. Als Zielregister der Befehle wird immer das erste Registerfeld benutzt, was Optimierungen vereinfacht.

Ein Register ist dabei unsichtbar in 4 Unterregister (32 und 64 Bit Integer, Single und Double für Fließkomma) aufgeteilt. Welche Funktion bzw. welcher Teil des Registers benutzt wird, hängt von der Typangabe im Befehl ab. Ein Zugriff auf r1 bis r3 mit unterschiedlichen Typen (z.B. Integer bzw. Float) ist nicht definiert und wird in der IJVM nicht ausgenutzt, da auch die Zielarchitekturen üblicherweise keine gemeinsamen Registerbänke besitzen bzw. die architekturspezifische Umsetzung eines 64 Bit-Registers auf zwei 32 Bit-Register in der IJVM noch nicht bekannt ist. Konvertierungen zwischen den verschiedenen Typen nutzen also explizite IJVM-Befehle und können (z.B. für 32→64Bit) nicht implizit über die Register erfolgen.

6.5.5.4 Konstantenfeld

Ein IJVM-Befehl kann maximal **eine** 32Bit-Konstante beinhalten, die entweder statisch aus der JVM→IJVM Übersetzungstabelle gelesen oder als Parameter aus Konstanten der JVM-Befehle übernommen wird. Ob und welcher Parameter aus dem JVM-Befehl eingesetzt wird, ist in der IJVM-Beschreibung eines JVM-Befehls spezifiziert. Die Beschränkung auf ein Konstantenfeld, dass immer im Befehl vorhanden ist, erlaubt einfacheres Parsen (konstante Befehlslänge), stellt andererseits aber keine wesentlich Einschränkung dar. Bezüglich der Aufteilung der JVM in IJVM-Befehle hat sich keine Übersetzung gefunden, die aus Geschwindigkeitsgründen mehr als ein Konstantenfeld benötigen würde.

6.5.6 Transformation JVM→IJVM

Die Übersetzung eines JVM Befehls in die IJVM-Architektur geschieht über Tabellen, in denen zu jedem JVM-Befehl die entsprechende IJVM-Befehlssequenz angegeben ist. Der Bytecode enthält dabei neben dem eigentlichen Befehl auch teilweise folgende Parameter (siehe auch Abb. 6.5):

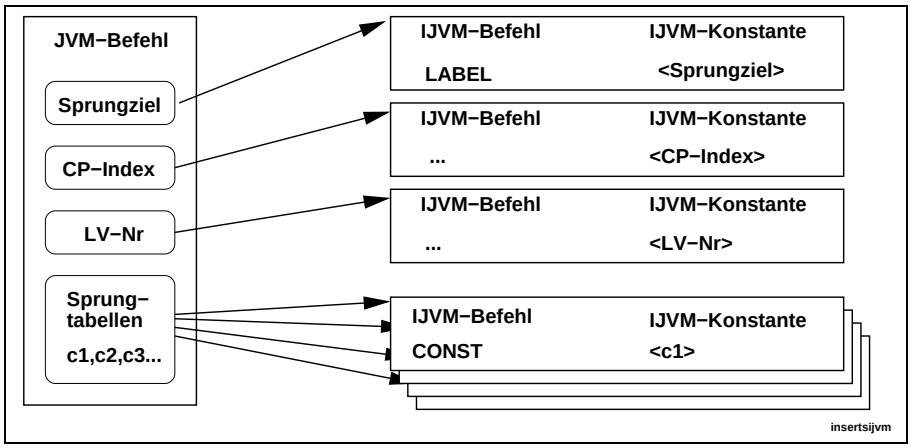

Abbildung 6.5: IJVM-Konstantenquellen

- Sprungziel
- Konstantenpool-Index (CP-Index)
- Nummer einer lokalen Variable (LV-Nr)
- Verzweigungstabellen für `lookupswitch` und `tableswitch`.

Diese Parameter müssen in das IJVM-Konstantenfeld transferiert werden. Dazu ist in der JVM→IJVM-Tabelle für jeden IJVM-Befehl die Quelle des Konstantenfelds angegeben. Im Normalfall wird es unverändert übernommen, als weitere Quellen sind obige Werte, die beim Parsen der JVM-Befehle ohnehin anfallen, möglich.

Neben den JVM-Befehlen wird zusätzlich die in der Analysephase angelegte Sprungtabelle betrachtet und jeder als Sprungziel identifizierbare JVM-PC generiert zusätzlich einen expliziten Marker (`label`-Befehl). In das Konstantenfeld dieses Befehls wird der Original-JVM-PC eingefügt. Dieser einfache Vorgang führt zu folgenden Vorteilen:

- Nach Abschluss der JVM→IJVM-Transformation ist die Sprungtabelle nicht mehr nötig und kann freigegeben werden. Damit wird es auch möglich, einen festen Speicherbereich im FPGA zu nutzen.

- Alle weiteren Optimierungschritte müssen nur noch den IJVM-Code betrachten und erfordern keine weiteren externen Informationen.
- Optimierungen beachten automatisch die Basicblockgrenzen, da durch den eingefügten `label`-Befehl die Regeln nicht mehr treffen.
- Während der Native-Code-Erzeugungsphase wird der zu diesem Ziel gehörende PC automatisch abgespeichert.

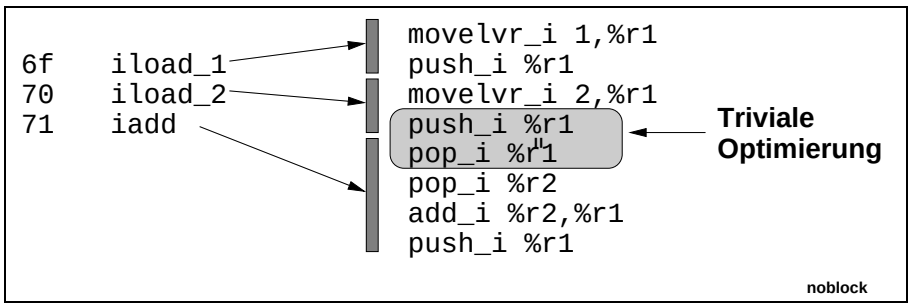

Abbildung 6.6: Triviale Optimierung ohne Label-Blockierung

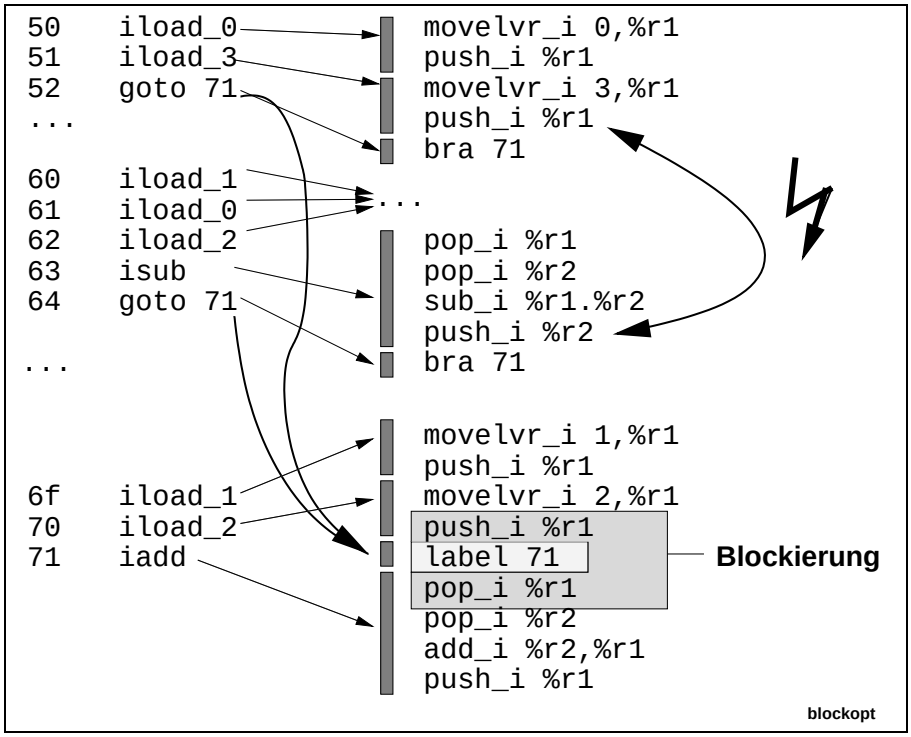

Abbildung 6.7: Blockierung durch Labelmarkierung

Zur Vereinfachung der folgenden Optimierungen wird bereits bei dieser Transformation eine Eliminierung von trivialen PUSH-POP-Paaren vorgenommen. Diese Paare werden ersatzlos gelöscht, wenn ihre Quell- bzw. Zielregister identisch sind und der zu POP gehörende JVM-Befehl nicht auf einem Blockanfang liegt (siehe Abb. 6.6. Der letzte Fall ist automatisch durch die bereits erwähnte Einfügung der `label`-Befehle überprüfbar (siehe Abb. 6.7), der `label`-Befehl verhindert (blockiert) die die Erkennung und damit die Optimierung über einen Basicblock hinweg. Eigene Untersuchungen haben ergeben, dass in typischen JVM-Codes nur ein relativ geringer Anteil ($< 5\%$) der Sprünge (im Prinzip nur vom Typ `goto`) wirklich mit benutztem Stack arbeitet. Aus diesem Grund ist der Verlust der Optimierungsmöglichkeit nicht signifikant.

6.5.7 Peepholeoptimierung auf IJVM-Basis

Die Peepholeoptimierung ist ein weit verbreiteter Ansatz, um Code zu optimieren. Dabei wird über den zunächst erzeugten Code ein Fenster ("Guckloch/Peephole") gelegt, in dem mit Regeln nach Mustern gesucht wird, die optimiert (gestrichen, ersetzt, geändert) werden können. Je größer das Fenster ist, umso effektiver kann die Optimierung werden. Allerdings nimmt der Aufwand, alle möglichen Optimierungsstellen zu finden, ebenfalls stark zu. Aus diesem Grund sind größere Fenster unüblich und es werden andere Methoden zur Vermeidung von redundantem Code benutzt (z.B. Datenflußanalyse).

Eine Implementierung eines Peepholeoptimierers in Hardware besitzt grundsätzlich dieselben Probleme. Da der Inhalt des Fensters immer vorhanden ist und nicht sequentiell durchsucht werden muss, lassen sich allerdings aufgrund der möglichen Parallelisierung in der Hardware wesentlich mehr Regeln gleichzeitig überprüfen. Da sich auch die meisten Regeln auf einfache kombinatorische Tests zurückführen lassen, sind die Vergleiche ebenfalls schneller durchzuführen.

Durch diese Beschleunigungsmöglichkeit und die relativ einfachen Implementierungsmöglichkeiten der Hardware ist die Peepholeoptimierung für diesen Anwendungsfall nicht von Nachteil. Wie gut sich der Vorgang tatsächlich im FPGA umsetzen lässt, wird später beschrieben.

Die in JIFFY benutzte Variante besitzt ein Fenster von maximal 6 IJVM-Befehlen (Slots) und arbeitet mit zwei Regelsätzen von nur 4 bzw. ca. 10 Regeln[5]. Dabei beschreiben die Regeln das zu findende Muster und die bei einem Treffer zu ersetzenden Befehle (sog. Patch). Die Struktur für eine einzelne Regel ist in Abb. 6.8 gezeigt. Pro Befehl und Regel muss im Prinzip eine "Opcodezelle" mit Schieberegister, Komparator und Patchlogik vorhanden sein, allerdings sind Optimierungen für "leere" Regeln möglich.

[5]Dies sind relativ wenig Regeln, der 8051-Compiler sdcc besitzt z.B. über 200

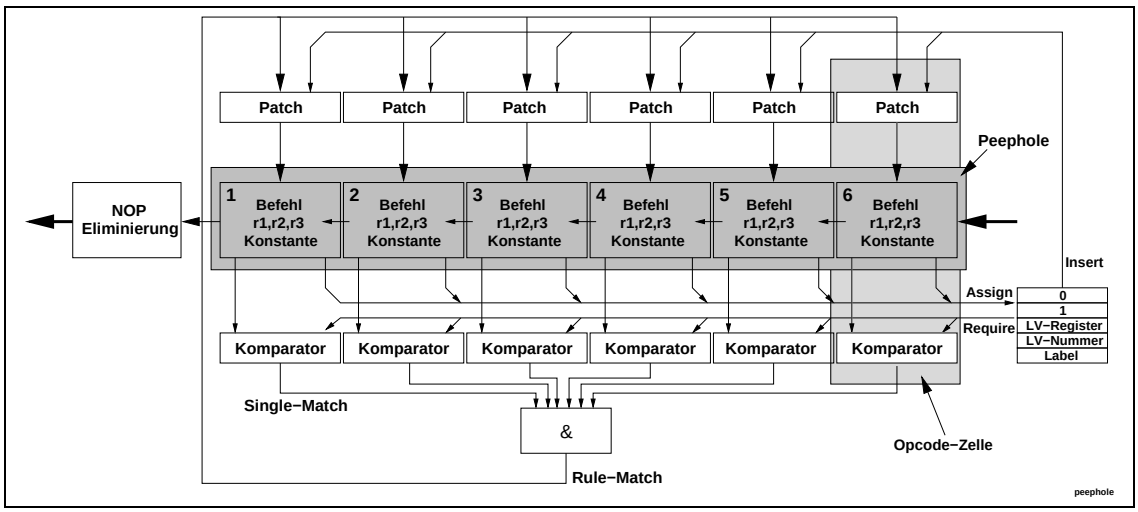

Abbildung 6.8: Block der Peepholeoptimierung für eine Regel

Um einen ganzen Regelsatz zu implementieren, sind zwei prinzipielle Implementierungsmöglichkeiten denkbar (siehe Abb. 6.9). Der gesamte Optimierungsblock arbeitet entweder über eine Priorisierung parallel mit allen Regeln auf dem Schieberegister oder ist für jede Regel hintereinandergeschaltet. Die letztere Variante bietet Vorteile bei der Minimierung der kritischen Datenpfade, erhöht allerdings die Latenz.

Bereits mit nur einer Opcodezelle (d.h. Vergleichs- und Einfügelogik für einen Befehl) ist eine Peepholeoptimierung möglich, wenn auch nicht sinnvoll: Dabei bezieht die Opcodezelle die nötigen Operationen für jeden Befehl jeder Regel aus einem Speicher. Von Nachteil ist hier der hohe Zeitaufwand, 10 Regeln mit je 6 zu überprüfenden Befehlen brauchen im Extremfall 60 Takte.

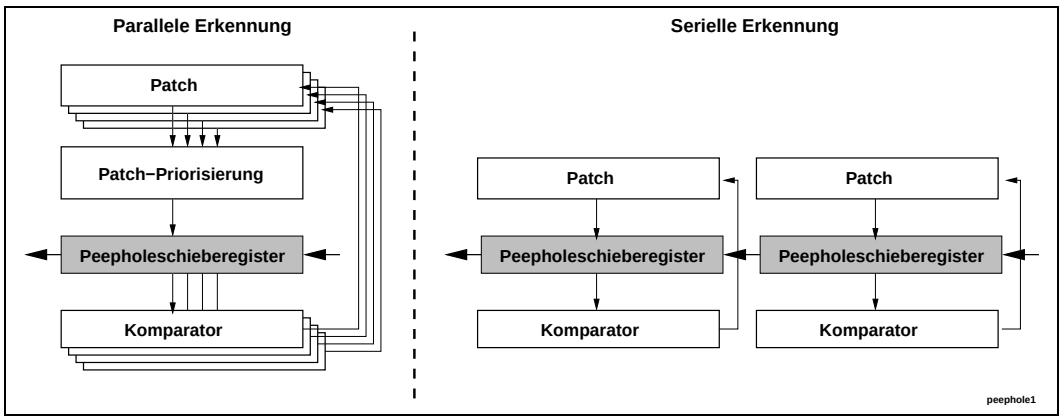

Abbildung 6.9: Peepholevarianten

6.5.7.1 Prinzip der Mustersuche

Als Grundlage der Mustersuche dient der IJVM-Opcode, der mit weiteren Attributen bestimmte Registereigenschaften speichert und überprüft. Folgende Operationen sind dabei möglich:

- Test auf Opcode bzw. Opcodegruppen

  Jeder Slotinhalt kann auf einen bestimmten Opcode oder eine Opcodegruppe verglichen werden. Zusätzlich ist eine Deaktivierung des Vergleichs möglich, somit ist der Inhalt des Slots für den weiteren Ablauf unerheblich.

- Zuweisung der Registernummer auf interne Speicher (**A**ssign)

- Vergleich der gespeicherten Registernummer mit Registernummer (**R**equire)

- Einfügen von Registerwerten aus den internen Speichern (**I**nsert)

- Weitere Vergleichsoperationen für Spezialfälle (Gleichheit von Quell- und Zielregister eines Befehls, Vergleich einer Labeladresse)

Alle bislang vorhandenen Regeln arbeiten mit maximal zwei Speicherzellen für Registerwerte. Die Speicherzelle für den Labelvergleich wird nur in einer Regel benutzt, der Wert ist aber über mehrere Regeln zu speichern. Die Speicherfunktion für Register ist nur für das Einfügen gespeicherter Werte innerhalb der Ausführung einer Ersetzungsregel notwendig. Je nach Implementierung (getaktet/asynchron) des Registervergleichs kann der zweite Vergleichswert um einen Takt verzögert aus dem Registerspeicher bezogen werden oder direkt ohne Verzögerung nach den Multiplexern vor dem Speicher abgegriffen werden.

Als Beispiel für den Einsatz der Operationen sei hier die Regel zum Eliminieren ein PUSH-POP-Paares genannt, die durch ein einfaches MOVE ersetzt wird.

```
S0_1(   !movexr,  push&&A1_1,  pop&&A2_1            \
:       *,        Gnop,        Gmove&&I2_1&&I1_2,*  )
```

Die erste Zeile beschreibt das zu findende Muster. Der erste Befehl darf kein MOVE-nach-Register sein (erlaubt sind aber alle anderen Befehle, die einen Wert in einem Register ablegen), der zweite Befehl ist PUSH, der dritte ein POP. Die gleichzeitig mit dem zu findenden Muster verbundenen Ersetzungsaktionen sind mit && an die Muster gebunden und stellen hier die Zuweisung der Registerwerte aus dem Registerfeld 1 in die Speicherplätze 1 und 2 dar.

Sollte die erste Zeile treffen (“Match”), wird die zweite Zeile ausgeführt, der erste Befehl bleibt unverändert, der zweite Befehl wird als NOP generiert, und der dritte wird in ein MOVE umgewandelt, wobei die vorher gespeicherten Quell- und Zielregister eingestanzt werden (Insertbefehl). Mit “*” werden alle weiteren Befehle unverändert weitergeleitet.

Durch den Einsatz eines Regelinterpreters im C-Modell war es möglich, verschiedene Regelkombinationen einfach auf ihre Effizienz und Korrektheit zu prüfen, die jetzt vorhandenen insgesamt 18 Regeln decken dabei einen Großteil aller auftretenden Fälle ab. Weitere Regeln für Spezialfälle sind zwar möglich, erhöhen allerdings wiederum den potentiellen Hardwareaufwand.

Regelsatz 1 ist nur für einfache Optimierungen gedacht und wird nur einmal angewandt. Regelsatz 2 wird zweimal hintereinander angewandt, wobei dies je nach zulässigem Hardwareaufwand im selben Datenstrom oder in zwei getrennten Durchläufen stattfinden kann.

Regelsatz 2 besitzt Regeln für spezifische Konstrukte, die durch die Stackübersetzung relativ häufig entstehen und eliminiert üblicherweise jegliches Vorkommen von PUSH und POP, solange eine Stacktiefe von 3 nicht überschritten wird. Durch die maximale angenommene Fenstergröße von 6 Befehlen und durch den doppelten Durchlauf ist damit trotz der Einfachheit der Regeln eine gute Codeoptimierung möglich.

Die Fortschaltung des Peephole-Fensters und die Eliminierung der erzeugten NOP-Befehle beim Verlassen des Fensters erfolgt im Software-Modell explizit über weitere Regeln, in der Hardware implizit in der Realisierung des Schieberegisters. Dadurch sind effektiv nur 4 bzw. 10 Regeln in der Hardware zu implementieren.

Wie schon in Abb. 6.7 angedeutet, wird aufgrund der Struktur der übersetzten IJVM-Sequenz eine Voraussetzung für fast alle Regeln des Peepholeoptimizers automatisch erfüllt, nämlich die Operation nur innerhalb eines Basisblocks. Sprungziele sind als `label`-Befehl im IJVM-Befehlsstrom explizit eingefügt. Damit verhindern sie das Erkennen von Regeln an Blockgrenzen, da sie die Muster aufbrechen. Bis auf zwei Regeln, die diese Anforderung absichtlich verletzen (siehe 7.1.7), sind alle Regeln so entworfen, dass sie nicht über Blockgrenzen hinaus arbeiten können.

Ein interessantes Ergebnis während der Erstellung des Peepholeoptimizers war, dass auch die Laufzeit der Gesamtübersetzung vom Optimizer profitiert. Die Taktzyklen, die der Optimizer zusätzlich verbraucht, werden durch eingesparte Takte bei der Übersetzung in Maschinencode aufgewogen. Messungen dazu sind in Abschnitt 8.4.4 angegeben.

6.5.8 Erzeugung von Maschinencode, IJVM→NAT

6.5.8.1 Übersicht

Die Nutzung einer Zwischensprache erlaubt die zwar maschinenunabhängige Optimierung, trotzdem ist der letzte Schritt immer die Umsetzung der Zwischensprache auf den Maschinencode. Dies verlangt ein im Prinzip jeweils speziell angepasstes "Backend". Im JIFFY-Konzept wurde nun versucht, auch diesen Schritt zu universell wie möglich zu gestalten. Das Ergebnis ist ein CPU-unabhängiger, parametrisierbarer Assembler, der ohne Hardwareänderung als Übersetzer für nahezu beliebige Architekturen dient.

Die Optimierung auf IJVM-Ebene erzeugt einen relativ kompakten Code, in dem kaum noch explizite Stackbefehle vorkommen. Dies führt zu einer guten Registerausnutzung und variablem Code, die allerdings eine einfache, nur tabellenbasierte Übersetzung in den Maschinencode verhindert bzw. zu große Tabellen erfordern würde.

Deshalb arbeitet die Übersetzung von IJVM in den Maschinencode (Native Code, NAT) mit einem schablonenorientierten Ansatz. Dieser ist speziell auf die Möglichkeiten der frei verdrahtbaren Hardware zugeschnitten und ist auf normalen Prozessoren relativ ineffizient.

Jeder IJVM-Befehl mit seinen Parametern wird als parametrisierbarer Assemblercode (im folgenden als "**Paracode**" bezeichnet) beschrieben, also als Schablone, in die die verschiedenen Parameter (Register, Konstanten, berechnete Offsets) eingestanzt (gepatcht) werden, um den endgültigen Code zu erzeugen.

Um dem Ziel der Architekturunabhängigkeit so nahe wie möglich zu kommen, wurde ein flexibles Konzept entwickelt, das diese Vorgänge ebenfalls parametrisierbar ausführt. Damit sind für eine neue Zielarchitektur nur die Schablonen und die verschiedenen Patchtypen neu zu entwerfen, der eigentliche Patchalgorithmus (also die ausführende Hardware) muss nicht verändert werden.

6.5.8.2 Parametrisierung

Ausgangspunkt der Übersetzung ist eine Tabelle, die jeden IJVM-Befehl als eine Abfolge von Befehlen der Zielarchitektur beschreibt. Aus dem IJVM-Code werden dabei die variablen Anteile als Platzhalter in die Assemblerbefehle eingesetzt.

Diese Anteile können sein:

- Register

  Die IJVM besitzt nur die drei Register r0, r1 und r2, allerdings universell mit bis zu 64Bit und Integer/Fließkommainhalt. Diese Register werden mit

einer Tabelle auf einige spezifische Register der Zielarchitektur abgebildet. Bei 32Bit-Architekturen ist dabei die Aufteilung eines 64Bit-IJVM-Registers auf zwei Zielregister zu vollziehen.

In einigen Fällen sind nicht alle Register für alle Befehle zugelassen (z.B. Intel x86), dies ist bei der Erstellung der Tabellen bereits zu berücksichtigen.

- Konstanten

  Konstanten können aus JVM-Befehlen (z.B. CP-Indices) oder aus dem IJVM-Code entstehen und sind aufgrund der Quelle als 8Bit oder 32Bit-Konstante erkennbar. Diese Unterscheidung hilft bei der Implementierung besonders für RISC-Prozessoren, da diese meistens keine Befehle zum Laden einer 32Bit Konstante haben. Stattdessen muss diese mit zwei oder mehreren Befehlen kombiniert werden. Ist die Länge bekannt, kann dies ausgenutzt werden.

  Für einige Anwendungen (z.B. Arrayadressierung) ist es von Vorteil, wenn die Konstante bereits bei der Übersetzung mit der Elementgröße multipliziert wird, da dies die Skalierung während der Laufzeit einspart. Letztere ist zwar bei vielen Prozessoren (z.B. MC68020, 80386) möglich, jedoch nicht allgemein vorhanden.

  Eine weitere Quelle von Konstanten ist das Einbringen von methodenabhängigen Konstanten (z.B. Anzahl der Parameter, Start der Methode), die das Laufzeitsystem benötigt. Diese sind bei der Übersetzung in einer Tabelle abgelegt und können anstatt des Konstantenfeldes im IJVM-Code referenziert werden.

- Offsets für lokale Variablen

  Offsets für lokale Variablen sind ähnlich den Konstanten einzufügen, allerdings ist dazu eine Adressberechnung notwendig, da sie entweder Parameter oder echte lokale Variablen sind. Der Unterschied liegt hierbei in der Stackposition, da Parameter im letzten Stackframe hinter der Rücksprungadresse liegen, die restlichen lokalen Variablen jedoch im aktuellen Stackframe. Neben der Skalierung ist also auch noch ein bedingter Offset zu addieren, falls Parameter adressiert werden sollen. Diese Berechnung ist zwar während der Laufzeit möglich, allerdings sehr ineffizient, da sehr viele Zugriffe auf Parameter bzw. lokale Variablen stattfinden. Daher muss diese Berechnung schon während der Übersetzung erfolgen. Aufgrund der unterschiedlichen Anzahl von Parametern der Methoden müssen diese Eigenschaften während der Übersetzung bekannt sein und bei jeder Offsetberechnung berücksichtigt werden.

Besonders CISC-Prozessoren besitzen viele Adressierungsarten und damit auch Opcodeformate. Das Patchsystem muss diese Formate universell handhaben, und Register und Konstanten an verschiedensten Stellen einstanzen.

6.5.8.3 Aufbau des Paracodes

Die Datenstrukturen, die für die Übersetzung der Befehlsschablonen für jede unterstützte Architektur angelegt werden, sind in Abb. 6.10 gezeigt.

Für jeden IJVM-Befehl existiert eine Referenz auf den Start des Maschinencodes in der Paracodetabelle. Ab dieser Position liegen die 1 bis n einzelnen Maschinencodeschablonen mit den für die Zielarchitektur nötigen Information über Art und Ort der einzufügenden Information. Dieser "Lookup"-Zwischenschritt ermöglicht zusätzlich die einfache individuelle Anpassung generischer Befehle an die Eigenschaften der zu übersetzenden Methoden (siehe Abschnitt 7.3) durch Änderung der Referenztabelle.

Als Einschränkung können maximal drei Stellen eines Befehls verändert werden, damit ist also auch die Modifikation der Register einer 3-Adressmaschine möglich. Bislang ist keine Architektur bekannt, bei der dieses nicht ausreicht und im Zusammenhang mit der Umsetzung der JVM-Befehle mehr Patchstellen in einem Befehl erforderlich sind.

Eine einzelne Paracodeinformation besteht damit aus einer maximal 12Byte langen Schablone, einem Startoffset (ein Byte), der Befehlslänge (ein Byte) und drei Patchbeschreibungen (sechs Bytes) und füllt somit 20Byte aus.

Jede der drei Patchbeschreibungen besteht wiederum aus einem Patchtyp, der die genaue Art der Einfügung bestimmt, der genauen Byteposition relativ zum oben beschriebenen Offset und der Zuordnung, welches Register im IJVM-Code eingefügt werden soll.

Die Codeschablone (Opcodetemplate) durchläuft drei einzelne Patchblöcke, die damit drei einzelne Stellen im Befehl verändern können. Diese werden anhand des Patchtyps und den aus der IJVM-Ebene gelieferten Registern bzw. Konstanten errechnet. Die in Abb. 6.11 gezeigte voll parallele Veränderung ist zwar durch das Pipelining die schnellste, da aber mit der bislang benutzen Realisierung nur ca. 25-30% der IJVM-Befehle mehr als einen Patch verlangt, ist sie nicht besonders ressourcensparend. Eine serielle Implementierung mit nur einem oder zwei Patchblöcken ist damit ohne größere Geschwindigkeitseinbußen realisierbar.

Der Patchtyp selbst referenziert eine Tabelle, in dem die Bitposition und Bitanzahl, sowie die Art der Übersetzung von Registernummer oder Konstanten in das einzustanzende Feld beschrieben wird (Patchinfo). Diese Werte werden dann, wie in Abb. 6.12 gezeigt, für die eigentliche Einfügeoperation in einem Patchblock weierverarbeitet.

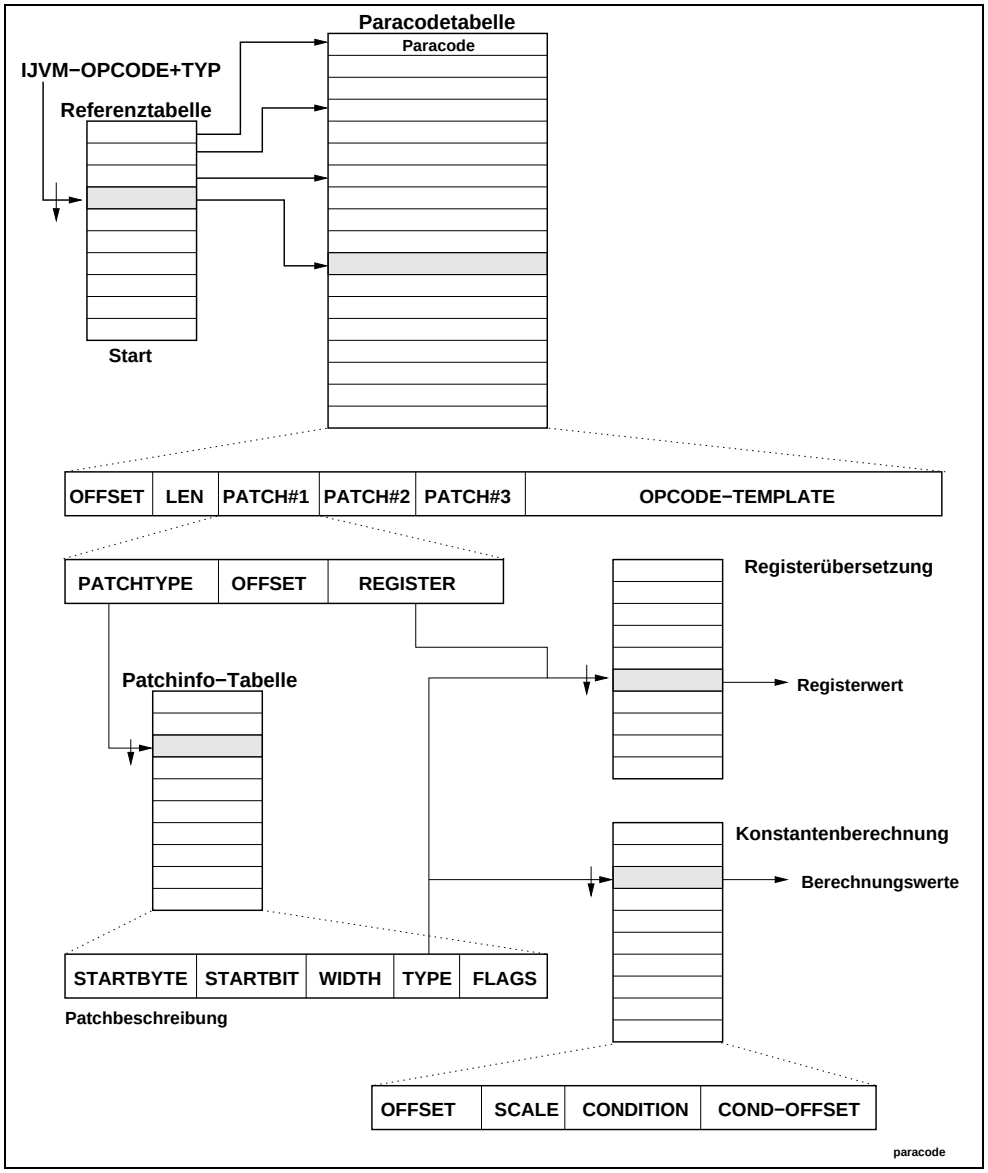

Abbildung 6.10: Aufbau des Paracodes

Die Übersetzungsart referenziert ebenfalls wieder eine Tabelle, aus der die Registerübersetzung für 8Bit, 16Bit, 32Bit und 64Bit-Registerzugriffe bzw. die Kalkulation (Abb. 6.13) der Konstanten mit Offset, Skalierung und evtl. einem zusätzlichen bedingtem Offset (für lokale Variablen) gelesen werden. Da die Funktionsparameter im Gegensatz zu reinen Variablen unabhängig vom Typ nur einen lokalen Variablenslot belegen, wird zusätzlich die lokale Variablennummer über eine Übersetzungstabelle (“Slot” Tabelle) auf die tatsächliche Stackposition über-

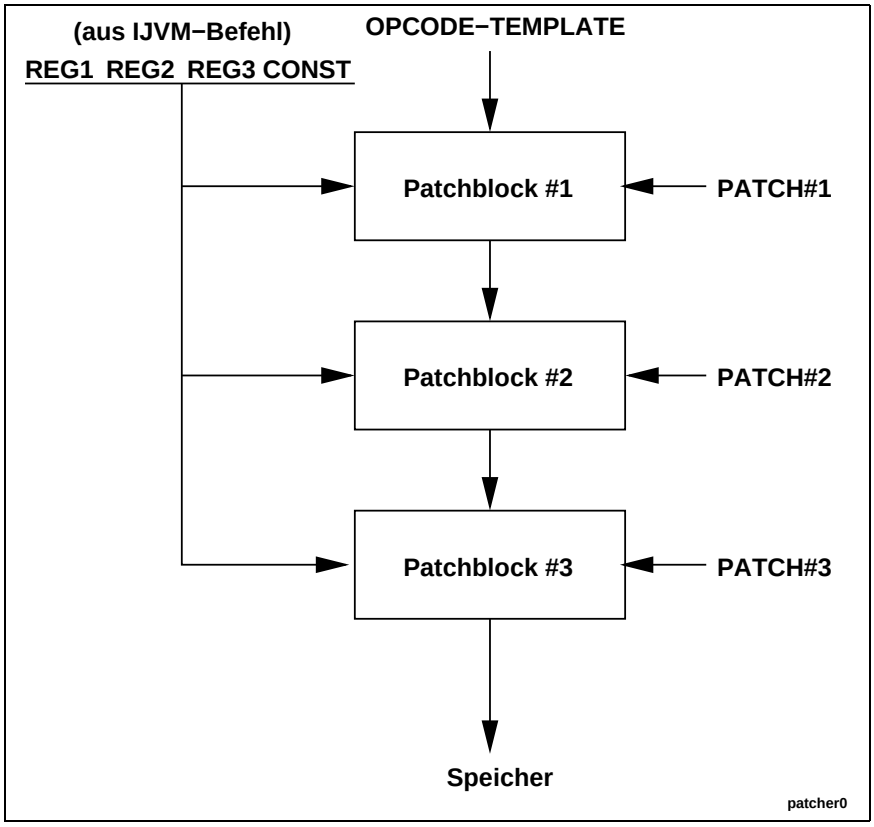

Abbildung 6.11: Ablauf der Opcodegenerierung (voll parallel)

setzt. Dazu sind weitere interne Vergleiche mit der Gesamtzahl der Parameter notwendig.

Da diese Adressierung sehr oft benutzt wird, ergibt sich gegenüber der expliziten Berechnung zur Laufzeit ein starker Gewinn. Da als elementare Operationen nur Additionen und Multiplikationen mit zwei Zweierpotenzen (4 und 8) benutzt werden, ist der Datenpfad auch in Hardware schnell zu durchlaufen und ressourcensparend.

Es ist auch möglich, als weitere Konstantenquelle ein Feld von 12 spezifischen Konstanten auszuwählen, die vor dem Start der Übersetzung passend belegt werden. Dies gestattet es, methodenspezifische Werte (z.B. Startadresse der Methode, Anzahl der Parameter, Sprungziele für spezifische Handlerroutinen) flexibel und ohne Verlassen des HW-Übersetzungswegs einzufügen.

Diese Art der verschachtelten Tabellen erlaubt eine sehr hohe Flexibilität bei der Anpassung an verschiedene Zielarchitekturen und führt gerade bei CISC-Prozessoren zu einer guten Ausnutzung der Adressierungsarten. Der Nachteil der starren Übersetzung, die durch die Auslagerung in HW kaum Informationen über Codeattribute hat, wird damit wieder ausgeglichen.

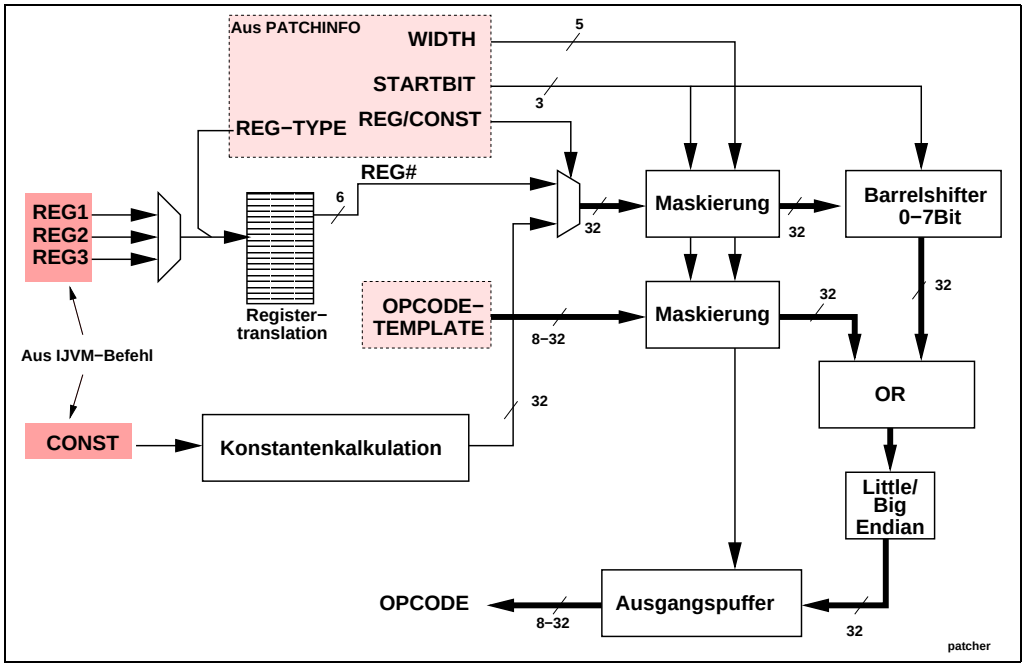

Abbildung 6.12: Aufbau eines Patchblocks

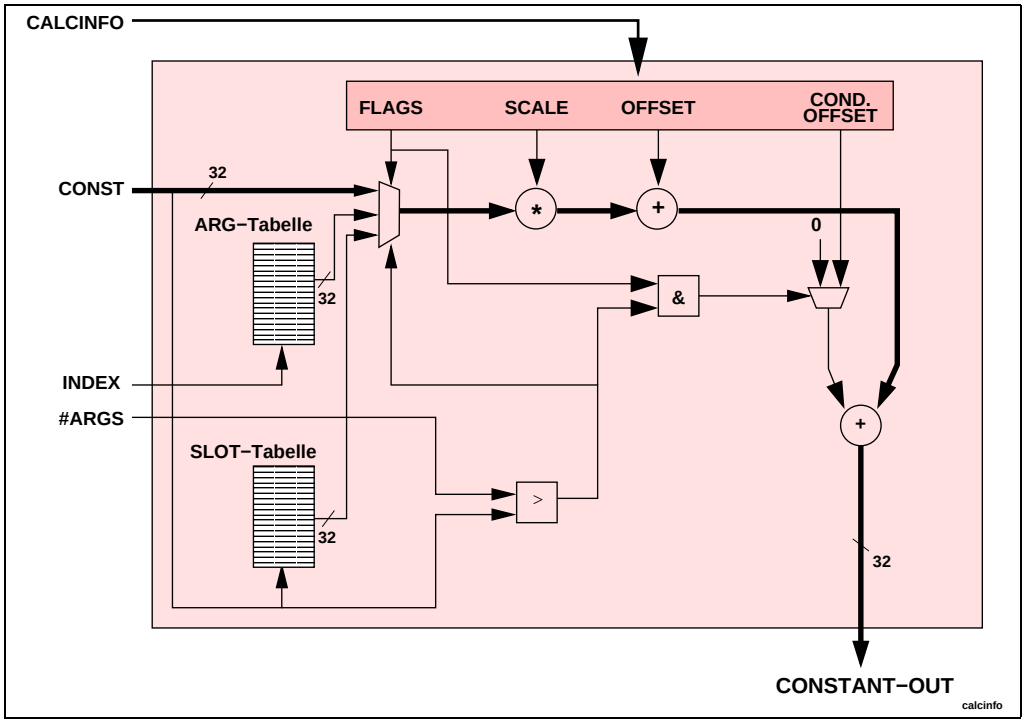

Abbildung 6.13: Aufbau der Konstantenkalkulation

6.5.8.4 Beispiel

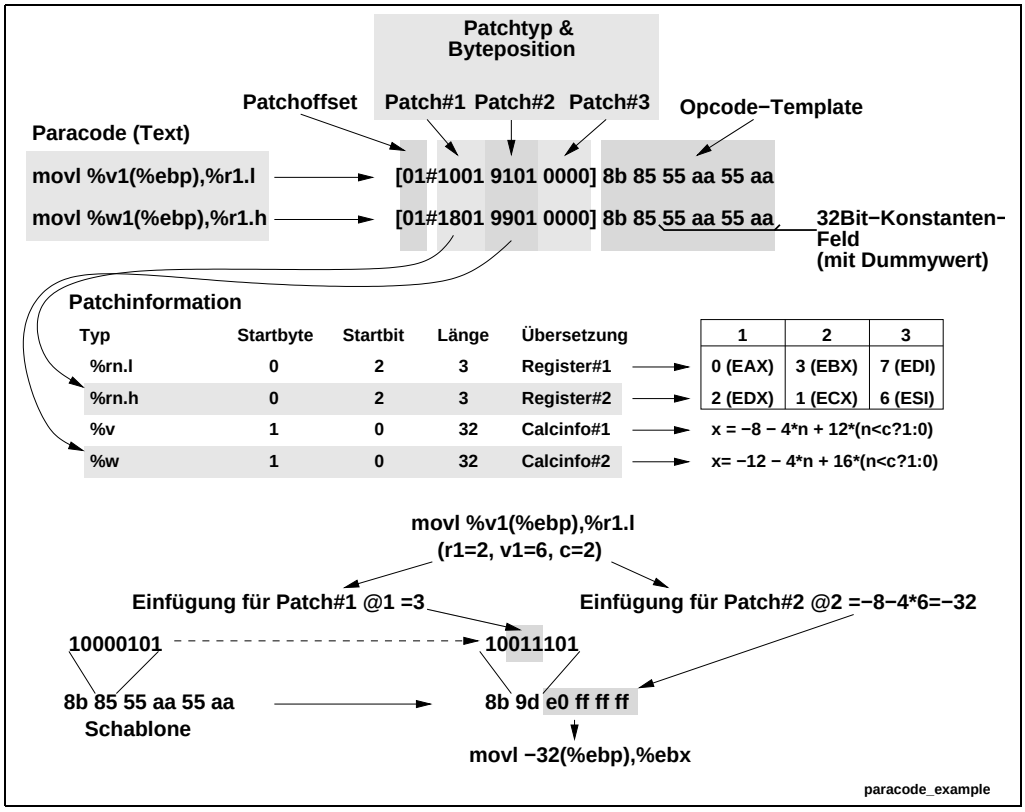

Abbildung 6.14: Beispiel für die Paracodeübersetzung

Als Beispiel für diese Flexibilität sei hier die Übersetzung des IJVM-Befehls “movelvr_l 6,r2” beschrieben. Dieser Befehl lädt den 64Bit-Wert der lokalen Variable 6 in das IJVM-Register r2.

Die Paracode-Umsetzung des IJVM-Befehls in x86 ist folgendermaßen beschrieben:

```
_movelvr_l
        .para %v1
        .para %r1
        movl %v1(%ebp),%r1.l
        movl %w1(%ebp),%r1.h
        .end
```

Dabei wird als erster Parameter des Befehls die Nummer einer lokalen Variable übergeben, deren Wert im IJVM-Konstantenfeld steht. Der zweite Parameter ist eine Registernummer, die dem Registerfeld r1 des IJVM-Befehls entnommen

wird. Durch die Zuordnung der Variablennummer an den Typ “v” wird das spätere Auftreten von %v1 bzw. %w1 als Adressberechnung der Stackoffsets für den nieder- bzw. höherwertigen Teil dieser Variable festgelegt. Dazu wird die Nummer der lokalen Variable mit 4 multipliziert und ein systemspezifischer Offset aufaddiert, der von der Position der Variable (Funktionsparameter bzw. “echte” Variable) abhängt. Diese Berechnung erfolgt erst **während** der Assemblierung im FPGA anhand der Methodeneigenschaften.

Die für den eigentlichen Patchvorgang benötigten Werte sind in Abb. 6.14 dargestellt und anhand des ersten Befehls ausgeführt.

Anhand der obigen Beschreibung wird eine Schablone für die beiden `movl`-Befehle erstellt, die an zwei Stellen veränderbar ist. Die erste Position ist die Einfügung des x86-ModRM-Felds für das Zielregister (r1). Diese erfolgt durch der .l/.h-Suffixe über zwei verschiedene Übersetzungstabellen, die die gegebene IJVM-Registernummern in die entsprechenden x86-Register übersetzen. Der Patch stanzt dabei die drei Bit des Zielregisters ab Bitposition 2 ein, das zu verändernde Byte wird aus dem Startoffset der Paracodebeschreibung und dem Byteoffset des aktuellen Patchtyps berechnet.

Diese Offsetkalkulation erlaubt eine Verschmelzung von unveränderlichen Befehlen mit patchbaren Befehlen innerhalb der Opcode-Schablone von 12Byte Länge. Damit ist innerhalb gewisser Grenzen eine einfache Komprimierung der entstehenden Paracodetabelle möglich[6].

Die zweite einzustanzende Position betrifft die schon beschriebene Adressberechnung von lokalen Variablen, deren 32Bit-Ergebnis in das Offsetfeld des `movl`-Befehls ab der Stelle 2 eingefügt wird. Die Paracodelogik ermöglicht dabei eine Konversion zwischen Little- und Big-Endian-Darstellung, sodass Bitfelder, die über Bytegrenzen hinausgehen, entsprechend der Architektur eingefügt werden.

Nach der Übersetzung des Paracodes und der Ausführung der beiden Patchblöcke entstehen also folgende x86-Befehle (Berechnung nur angedeutet und noch nicht ausgeführt):

```
movl -8-4*6(%ebp),%ebx
movl -12-4*6(%ebp),%ecx
```

Die genaue Syntax des Paracodes ist im Anhang A.1 näher erläutert.

[6] Zur resultierenden Größe der Tabelle siehe Abschnitt 8.2.2

6.5.9 Linker

Der Linkervorgang schreibt die nach dem Übersetzungsvorgang noch fehlenden Sprungziele in den Code. Dazu werden während des Übersetzungsvorgangs IJVM→NAT bereits zwei Tabellen angelegt (siehe Abb. 6.15):

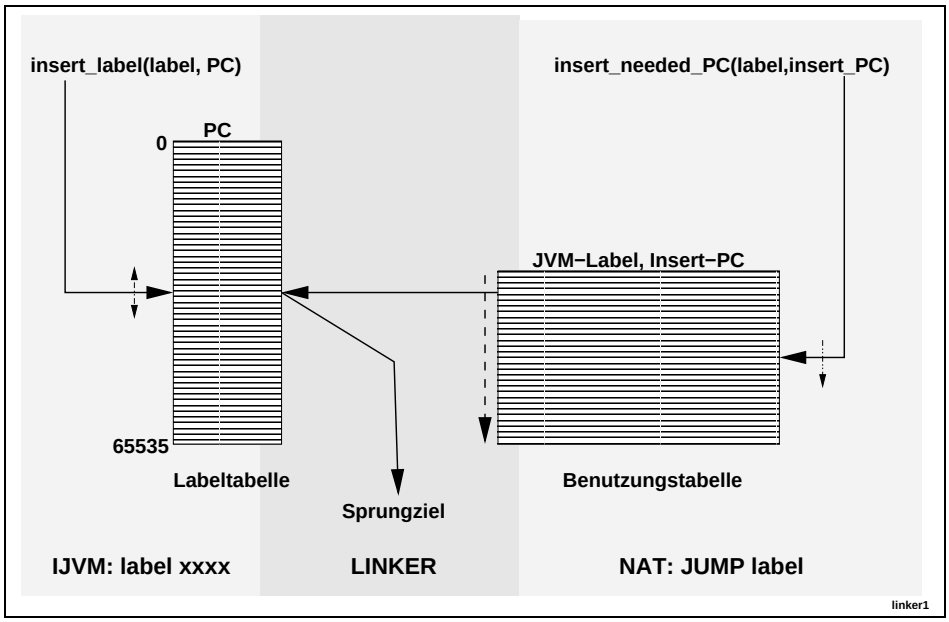

Abbildung 6.15: Linkertabellen (mit O(1) Labelsuche)

1. Labeltabelle

   Für jedes Auftreten des `label`-Befehls im IJVM-Code wird der aktuelle PC-Stand in der Zielarchitektur vermerkt.

2. Tabelle mit benötigten Speicherstellen (Benutzungs/“Usage”-Tabelle)

   Jeder Sprungbefehl fügt das anzuspringende Label und die Position (“Insert-PC”) der einzufügenden Zieladresse[7] in die Usage-Tabelle ein.

Anschließend muss beim Linken nur noch die Benutzungstabelle durchlaufen und der für jede Marke notierte PC in den Code eingefügt werden. Dabei gibt es für das Suchen in der Labeltabelle verschiedene Implementierungsarten, die sich auf die Geschwindigkeit der Labelsuche auswirken:

- **Labelsuche mit O(1) durch ein lineares Array**

[7] Auf x86-Architekturen ist dies ein Sprungoffset

Diese Variante ist die effizienteste bezüglich der Suche, verbraucht aber mindestens 256KByte Speicher und arbeitet nur mit den bislang verwendeten 16Bit langen JVM-Sprungzielen. Dazu wird in einer Tabelle an die durch das JVM-Label spezifizierte Stelle der resultierende PC geschrieben. Die Labelsuche reduziert sich damit auf einen Speicherzugriff.

Der dafür benötigte Speicherbereich ergibt sich aus dem IJVM-Label mit dem höchsten JVM-PC, damit beinhaltet diese Tabelle maximal 65536 Einträge. Für Einträge mit 4Byte ergibt sich also ein maximaler Speicherbereich von 256KByte. Diese Speichermenge ist nicht mehr sinnvoll FPGA-nah zu halten.

• Labelsuche mit C*O(n) durch Hashcodierung

Theoretisch ungünstiger, aber in der Praxis beinahe genauso effizient wie die lineare Methode ist das Abspeichern der Labels über eine Hashfunktion mit linearer Kollisionsauflösung. Der für die Tabelle benötigte Platz ist bereits bei der JVM→IJVM-Übersetzung anhand der Anzahl der `label`-Befehle erkennbar und kann somit bereits vorher bzw. parallel zur Hardwareübersetzung in Software alloziert werden. Der Effizienz dieser Lösung kommt zugute, dass JVM-Methoden typischerweise relativ kurz sind, somit die Anzahl der Sprungziele begrenzt ist und damit der für die Tabelle verbrauchte Speicherplatz weit unter den maximal möglichen 65536 Elementen liegt (siehe auch 3.5.2).

Wenn bei der Speicherallokierung auf die nächste Zweierpotenz aufgerundet wird, ist als einfache Hashfunktion eine Modulo-Rechnung durch Ausmaskieren der oberen Bits der Labelnummer möglich. Da mehrere Labels auf einen Eintrag fallen können, ist neben dem PC auch noch ein Eintrag für den Schlüssel (d.h. die Labelnummer) notwendig. Damit wird der für die Tabelle nötige Platz um mindestens 50% vergrößert[8].

Das Einfügen in die Hashtabelle kann mit aktiver Platzsuche geschehen. Zu Beginn wird die Tabelle gelöscht, dies kann mit Hardwareunterstützung sehr schnell geschehen. Das Einfügen testet zunächst das Vorhandensein eines Eintrags und schaltet den Tabellenindex bei Bedarf weiter. Damit wird der Vorgang durch die nötigen Lesezugriffe auf die Tabelle auch ohne Kollisionen verlangsamt. Effizienter ist es, die Kollisionsüberprüfung über ein Bitfeld abzufragen, das nur ein Bit pro Eintrag enthalten muss und damit typischerweise noch problemlos in einem FPGA-internem RAM abgelegt werden kann. Durch dieses Vorgehen ist es möglich, das Eintragen der Labels stark zu beschleunigen. Die dadurch entstehende Speicherstruktur ist in Abb. 6.16 gezeigt.

Durch die Möglichkeit, die maximale Anzahl der Tabelleneinträge immer etwas größer zu wählen, als tatsächlich `label`-Befehle vorhanden sind, können

[8] Aus Alignmentgründen sind 100% empfehlenswert

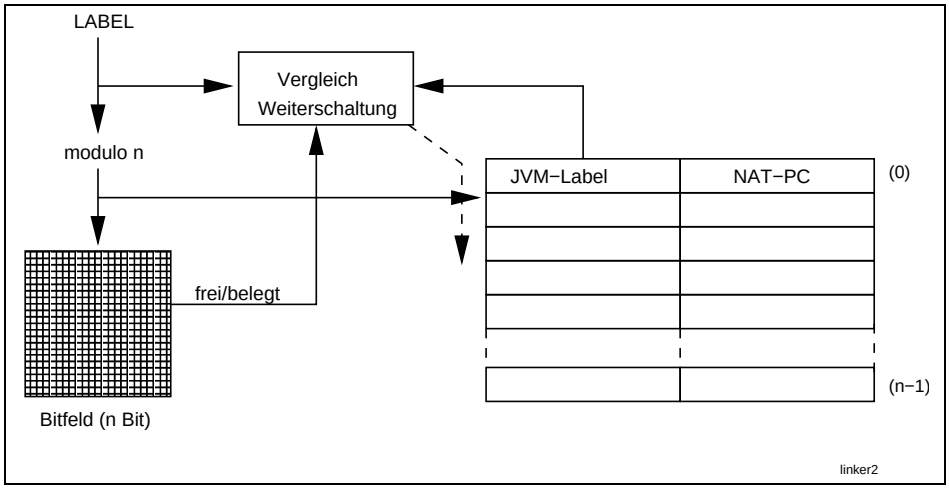

Abbildung 6.16: Zugriff auf Benutzungstabelle mit Hashcodierung

Kollisionen beim Einfügen vermindert werden. Dies wirkt sich direkt auf die Geschwindigkeit der Suche aus.

Betrachtet man die Ergebnisse von 3.5.2, wo die typische Anzahl von Sprünge und Sprungzielen innerhalb einer Methode ermittelt wurde, ist die Realisierung des Linkers durch Hashcodierung wesentlich vorteilhafter. Eine Label- und Benutzungstabelle mit 1024 Einträgen ist für fast alle im JDK vorkommenden Klassen mehr als ausreichend. Der nötige Speicherbedarf liegt dabei im Bereich von ca. 10-12KByte und damit weit unter dem mit linearem Labelarray. Diese Größe ermöglicht auch die Unterbringung der Tabelle im FPGA bzw. in einem FPGA-nahem Speicher. Durch die dünne Belegung einer Hashtabelle dieser Größe sind nahezu konstante Suchzeiten (O(1)) wie mit einem linearen, direkt addressierten Array zu erwarten.

6.6 Speicherhierarchie

Anhand der Beschreibung des Übersetzungsvorganges wird deutlich, dass neben dem eigentlichen Übersetzungsalgorithmus ein schneller Zugriff auf Daten und Tabellen nötig ist, um die Übersetzung nicht zu beeinträchtigen. Da schneller Speicher im geplanten Anwendungsgebiet teuer (Platz, Stromverbrauch, Preis) ist, ist es wichtig, das Übersetzungskonzept so zu entwerfen, dass die notwendigen Speicherschnittstellen klar definiert sind und Abwägungen bezüglich der erzielbaren Übersetzungsgeschwindigkeit möglich werden.

Für das JIFFY-System sind dabei folgende Kategorien für Speicherzugriffe von Bedeutung:

1. Sequentieller Zugriff, “Streaming”

   Das Einlesen und Zurückschreiben von JVM-Befehlen, IJVM-Opcodes und übersetzten Maschinencode läuft linear und damit vorhersehbar ab und kann daher auf Speicher mit höherer Latenzzeit zugreifen, solange dieser ein Pipelining oder Burstzugriffe bzw. Prefetch erlaubt. Als Speichertypen sind damit z.B. SDRAM oder der PCI-Bus möglich, jeder andere DMA-fähige Systembus ist jedoch auch denkbar. Durch die Integration eines SDRAM-Controllers in das FPGA wird es möglich, SDRAM unabhängig vom Restsystem zu nutzen oder ihn sogar als Hauptspeicher in das System direkt einzubinden (Glue-Logic-Aspekt).

2. Nicht vorhersehbarer Zugriff auf große Tabellen

   Die Übersetzungstabellen für JVM→IJVM und IJVM→NAT belegen ohne weitere Kompression max. 50KB Speicher und werden relativ schnell ausgelesen. Dabei sind Latenzen von wenigen Takten tolerierbar, diese beeinflussen allerdings schon die Übersetzungsgeschwindigkeit. Aufgrund der nicht vorhersagbaren Zugriffe müssen diese Datenbereiche in einen FPGA-nahen Speicher gelegt werden, also entweder in externes, schnelles SRAM (Cache-SRAM) oder in FPGA-internes Blockram.

3. Nicht vorhersehbarer Zugriff auf kleine Tabellen

   Die Übersetzungstabellen für die Paracodeerzeugung (z.B. Patchtypen, Regeln der Peepholeoptimierung) werden in jedem Takt gelesen und beeinflussen die erzielbare Leistung sehr stark. Daher müssen diese kleinen Speicherbereiche (einige 100 Byte) im Distributed RAM auf dem FPGA implementiert werden.

Werden die drei Kategorien und die damit möglichen Speichertechnologien auf ihren Stromverbrauch (siehe Abschnitt 4.2.2) untersucht, ergibt sich folgendes Bild:

In Kategorie 1 ist (S)DRAM aufgrund des günstigen Speicherpreises pro Byte und der hohen Burstgeschwindigkeit nicht zu sinnvoll zu ersetzen. In Kategorie 3 hat das FPGA-RAM wenig Einfluss[9] auf den FPGA-Gesamtstromverbrauch und ist durch die direkte Integration und damit wegfallende Leitungstreiber noch sparsamer als externes SRAM.

Eine Optimierung bezüglich des Stromverbrauchs ist nur in Kategorie 2 sinnvoll und möglich. Für die Übersetzungsgeschwindigkeit ist latenzarmes Cache-SRAM am Besten geeignet, allerdings ist der Stromverbrauch relativ hoch.

[9] Diese Einschätzung basiert auf eigenen Erfahrungen.

6.7 Integration des Bytecodeverifiers in die HW

Das JIFFY-System nimmt die Umsetzung der stackbasierten JVM auf eine registerbasierte Architektur vor, daher können Zwischenergebnisse dieser Übersetzung auch zur Prüfung einiger Anforderungen an den Bytecode ausgenutzt werden. Zwar sind diese Anforderungen an den Code teilweise in der FPGA-Logik nicht exakt (d.h. vollständig) nachprüfbar, allerdings vermindern sie zusätzliche Arbeit in einem externen Verifier in Software bzw. erlauben den Wegfall desselben in bekannten Ablaufumgebungen.

6.7.1 Eingeschränkte Typprüfung der Stacknutzung

Da die Peepholeoptimierung für die Entfernung von PUSH/POP-Operationen genutzt wird, ist der Test möglich, ob die beiden Operationen auf denselben Typen operieren. Dies bedeutet wiederum, dass auch auf JVM-Ebene eine konsistente Stacknutzung erfolgt. Durch das begrenzte Fenster der Optimierung sind damit höchstens 3 Stackebenen überprüfbar. Was darüber hinausgeht oder an impliziter Stacknutzung (Parameterübergabe) erfolgt, ist so nicht detektierbar.

6.7.2 Überprüfung auf identische Stacktiefe über verschiedene Ausführungspfade

Diese Überprüfung kann trotz Rückwärtssprüngen durch die bereits erstellte Sprungtabelle in nur in einem Durchgang erfolgen (siehe Abb. 6.17). Dazu muss während der JVM→IJVM-Übersetzung die Stacknutzung durch PUSH/POP und evtl. INVOKE_*-Befehle gezählt werden. Hierbei erhöhen PUSH-Befehle die Stacktiefe um eins (Integer) bzw. zwei (Long, Double), ein POP erniedrigt entsprechend. Bei INVOKE-Befehlen können die benötigten Stackparameter damit verglichen werden, nach einem INVOKE-Befehl ist der Stack um diese Anzahl wieder erniedrigt. Falls Parameter zurückgegeben werden, sind diese wie bei einem PUSH zu zählen. Da vor der JVM→IJVM-Phase ohnehin die Typen der INVOKE-Befehle bekannt sind, ist ein zusätzlicher Eintrag mit einem aus der Methodensignatur abgeleiteten Stackkorrektur zusätzlich möglich.

Bei einem Sprungbefehl wird der aktuelle Stackzustand in einer durch das Sprungziel adressierten Tabelle geschrieben. Ist in diesem Eintrag bereits ein Wert notiert, so muss er mit dem neuen zu schreibenden Wert übereinstimmen.

Wird ein `label`-Befehl eingelesen, wird die bis dahin festgestellte Stacktiefe ebenfalls geschrieben bzw. mit dem bereits für dieses Label gespeicherten Wert verglichen. Ist die errechnete Tiefe mit dem gespeicherten Wert nicht identisch, wird also eine je nach Ausführungspfad unterschiedliche Stacktiefe erkannt. Für

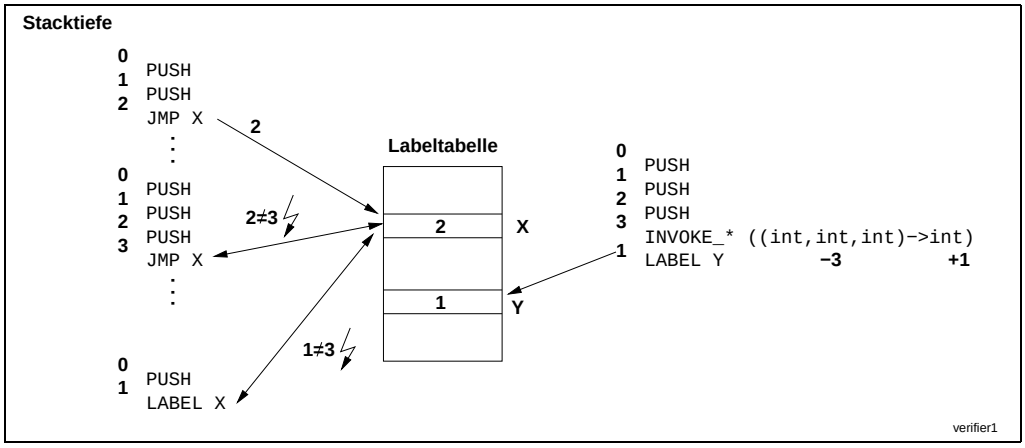

Abbildung 6.17: Überprüfung der Stacktiefe

INVOKE-Befehle ist nur die Stackeffekt zu berechnen, d.h. die Anzahl der nach dem Befehl auf dem Stack liegenden Werte.

Als Speicher für diese Werte kann ein Speicherbereich derselben Größe und Ansteuerungsart benutzt werden, wie für das spätere Linking (siehe 6.5.9), wodurch Chipkomponenten wiederverwendet werden.

Die für diesen Komplex notwendigen HW-Modifikationen sind bis auf den Zugriff auf die aus der Signatur berechnete Stackkorrektur wenig aufwendig, sie beschränken sich auf die Erkennung von `push/pop/invoke` und `label` und einen Akkumulator. Da die aus der Labeltabelle auszulesenden Werte nicht unmittelbar benötigt werden, ist aggressives Pipelining möglich. Wird ein Fehler in den Stacktiefen erkannt, ist das Wissen um die genaue Fehlerposition nach der JVM-Spezifikation auch nicht notwendig.

6.7.3 Gleichzeitige Überprüfung auf Typkonsistenz und Stacktiefe

Ein Kombination der beiden vorangegangenen Tests ist relativ einfach möglich, zusätzlich erlaubt sie ohne Änderung des Speicherkonzepts der Linkertabelle die Überprüfung auf eine Stacktiefe mit bis zu 10 Elementen, wobei hier auch long und double-Typen als ein Element zählen.

Das Prinzip der kombinierten Überprüfung in der Hardware ist in Abb. 6.18 skizziert. Ähnlich der Stacktiefenmessung werden an jedem Einsprungpunkt und Verzweigungspunkt Werte in die durch die Labeladresse spezifizierte Adresse geschrieben bzw. gelesen und verglichen.

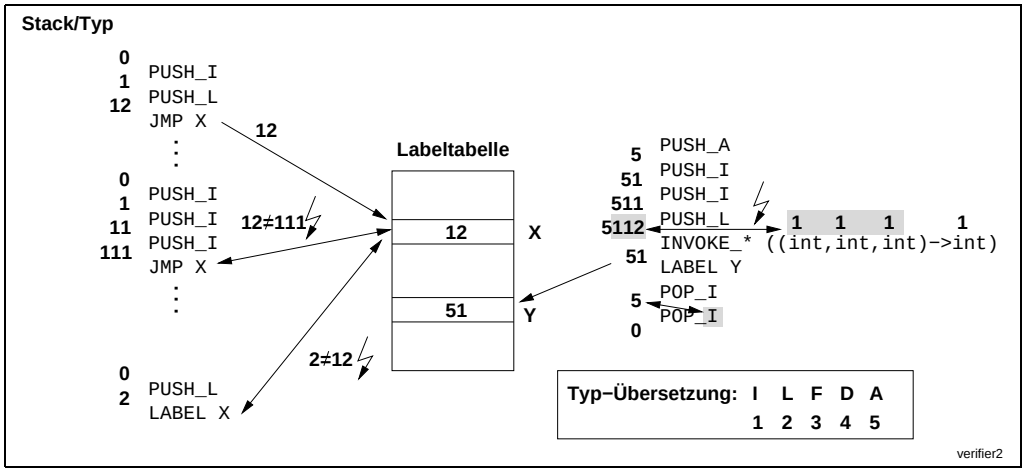

Abbildung 6.18: Modifizierter und kombinierter Typ- und Stacktiefentest

Die Bestimmung der Werte erfolgt diesmal nicht mehr über die reine PUSH/POP-Funktion, sondern bezieht den Typ der Stackoperation folgendermaßen mit ein:

Die Basistypen der Operationen werden über eine Tabelle auf einen 3Bit-Wert übersetzt, der an den bisherigen Stackzustand angehängt wird. Damit ist in einem 32Bit-Wert der Stackfüllstand einschließlich der benutzten Typen mit bis zu 32/3=10 Elementen abspeicherbar.

Dieser Stackzustand kann bei `invoke`-Befehlen zur Typprüfung der n erforderlichen Parameter verwendet werden, in dem die ersten $3 * n$ Bit mit der ebenfalls übersetzten Signatur verglichen werden. Jede POP-Operation entnimmt die 3 untersten Bits des Wertes und vergleicht sie mit dem Typ des Befehls. Bei `label`-Befehlen bzw. Verzweigungspunkten muss der gesamte 30Bit-Wert mit dem für diese Position gespeicherten Stackzustand übereinstimmen.

Die für diese kombinierte Überprüfung notwendigen Modifikationen beschränken sich auf das Typ-Mapping von Befehlen und ein Schieberegister, sind damit also ohne großen Aufwand in HW zu implementieren. Allerdings sind bereits aufbereitete Informationen über die Signaturen aller gelisteten Methoden im Konstantenpool notwendig. Diese Informationen sind bevorzugt FPGA-nah abzulegen. Da die Methodensignaturen im Konstantenpool sehr einfach aufgebaut sind, können die Information evtl. auch im FPGA selbst durch Parsing erzeugt werden. Die Überprüfung kann allerdings nur auf Übereinstimmung von Grundtypen testen, verschiedene Klassenreferenzen werden nicht erkannt.

Wie bereits in Abschnitt 3.5 beschrieben, sind JVM-Methoden mit Stacktiefen > 9 bereits sehr selten, dadurch ist diese HW-basierte einfache Verifikationsmethode nahezu universell einsetzbar. Sollte die maximale Stacktiefe von 10 Elementen doch überschritten werden, ist es möglich, den Test in der Hardware

abzubrechen und diese Überprüfung an die Software zu übergeben. Diese kann anschließend auf denselben Daten wie die Hardwarephase arbeiten.

Durch diese Interoperabilität zwischen der JIFFY-Laufzeitumgebung und der Hardwareübersetzung im FPGA können die Ergebnisse der Sprungzielanalyse für die Software wiederverwendet werden. Dies stellt auch in diesem Ausnahmefall einen Geschwindigkeitsvorsprung gegenüber einer reinen Softwarelösung dar.

6.7.4 Weitere Überprüfungsmöglichkeiten

Korrekte Nutzung von lokalen Variablen bzw. CP-Indices

Während aller Phasen der Übersetzung (auf JVM oder IJVM-Ebene) können Befehle zur Referenzierung von lokalen Variablen oder Konstantenpool-Indices auf den jeweils maximalen Wert der Methode bzw. Klasse überprüft werden. Der zusätzliche Hardwareaufwand ist minimal.

Überprüfung auf korrekte Sprungziele

Eine weitere Anforderung an den Bytecodeverifier ist die Überprüfung der in den JVM-Befehlen angegebenen Sprungziele. Diese müssen immer auf Befehlsgrenzen treffen, damit wird eine alternative Interpretierung des Bytecodes verhindert.

Durch die Erstellung der Sprungtabelle in der ersten JVM-Analysephase wird als Seiteneffekt eine Überprüfung möglich, ob diese Sprungziele auch auf Befehlsgrenzen treffen. Dieser Test kann bei der JVM→IJVM-Übersetzung parallel zum erneuten Einlesen der JVM-Befehle ablaufen und einen Regelverstoss einfach feststellen.

6.7.5 Bewertung der HW-Verifikation

Die beschriebenen, HW-basierten Verifikationsmöglichkeiten des Bytecodes decken bereits eine Reihe der in den JVM-Verifikationsregeln ([94], Section 4.9.2) beschriebenen Tests ab, die in Software **zusätzlich** zur JIT-Übersetzung notwendig wären, in HW aber mit minimalem Aufwand im FPGA parallel zu den einzelnen Phasen stattfinden können.

Alle obigen Überprüfungen führen dazu, dass die Compilation der Methode bereits begonnen wurde, bevor die Nichtübereinstimmung mit den Bytecoderegeln erkannt wird. Dies ist aber laut JVM-Spezifikation erlaubt, da ja im Endeffekt nur die Ausführung einer nicht verifizierbaren Methode verhindert werden muss. Da dieser Fehlerfall auch als Ausnahme anzusehen ist, ist die bereits dafür verbrauchte Übersetzungszeit nicht wesentlich.

Eine Untersuchung, ob die durch die HW-Überprüfung gewonnene Zeit dies möglichweise sogar ausgleichen kann, wäre hier zwar angebracht, ist aber aufgrund der vielfältigen Implementierungsmöglichkeiten des Bytecodeverifiers schwierig.

6.8 Zusammenfassung

Das beschriebende JIFFY-Übersetzungssystem basiert auf einer einfachen algorithmischen Struktur. Damit ist aber die Annahme gerechtfertig, dass sie sich gut in Hardware umsetzen lässt und damit für eine Implementierung in einem FPGA gut geeignet ist. Weiterhin erlaubt der gewählte Aufbau der Übersetzungstabellen eine einfache Anpassung an verschiedene Zielarchitekturen ohne Hardwareänderung.

Eine weitere, überraschende Eigenschaft des Übersetzungssystems ist die Verwertbarkeit der Zwischenergebnisse. Die während der Übersetzung anfallenden Informationen können mit geringem zusätzlichem Aufwand zur Überprüfung einiger wichtiger Anforderungen an den Bytecode benutzt werden und somit den nötigen Bytecodeverifier vereinfachen oder ganz einsparen.

Die aufgrund einfacherer HW-Implementation gegebenen Einschränkungen der “Intelligenz” der Übersetzung werden im folgenden Abschnitt mit ihren Auswirkungen genauer beschrieben.

7

Implementationsaspekte

Things should be made as simple as possible – but no simpler.
Albert Einstein

Dieser Abschnitt beschreibt Aspekte, die bei der Implementierung des C-Modells (siehe Abschnitt 6.3.1) und des Hardware-Prototypen auftraten und durch das gewählte Übersetzungskonzept bedingt sind. Diese Aspekte lassen sich unterteilen in allgemeine Probleme, die für C-Modell und Hardware gleichermaßen gelten und Hardware-spezifische Punkte. Interessant sind in diesem Zusammenhang besonders Abwägungen und Entscheidungen, die die Übersetzungs-Hardware auf Kosten des Laufzeitaufwands vereinfachen können.

Die im folgenden beschriebenen Besonderheiten gelten prinzipiell für alle Laufzeitumgebungen, werden hier aber im Zusammenspiel von JIFFY und des JDK-JRE näher erläutert.

7.1 Allgemeine Aspekte

7.1.1 JIT-Umgebung, Interaktion mit dem Laufzeitsystem

Werden die Auflösung einer Methode oder andere Hilfsfunktionen aufgerufen (z.B. Übergang zur JRE-Umgebung, Auslösung einer Ausnahme), ist es zur korrekten Abwicklung notwendig, dass die gerade abgearbeitete Methode (und damit ihre Umgebung) und die Einsprungpunkte der Hilfsfunktionen bekannt sind.

Zu diesem Zweck sind zwei, in der letzten Übersetzungsstufe einsetzbare Paracode-Konstanten verfügbar (siehe Abb. 7.1), die auf vom Laufzeitsystem aufgebaute Strukturen verweisen:

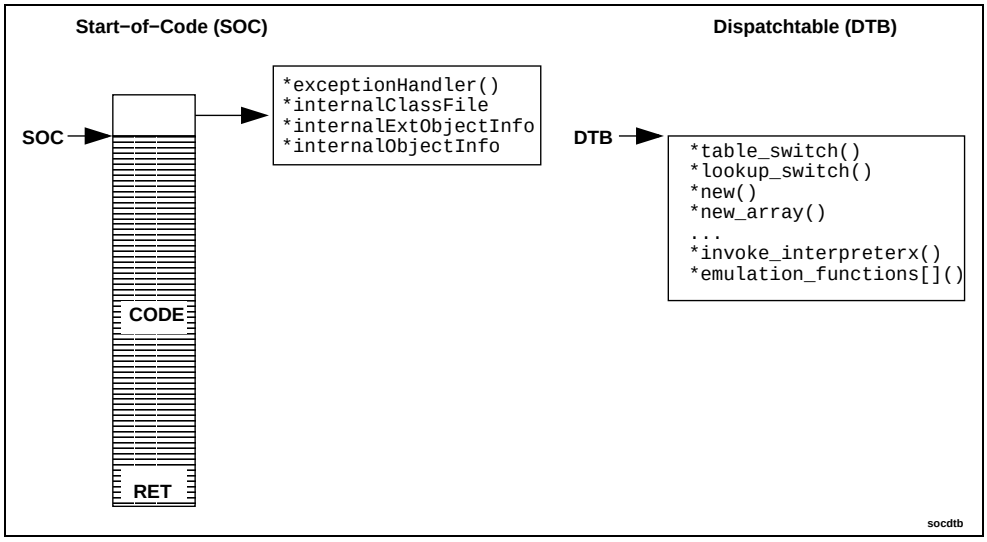

Abbildung 7.1: Strukturen “Start-of-Code” und “DispatchTable”

1. “Start-of-Code”

   Diese Konstante enthält die Startadresse der laufenden Methode. Vor dem Code der Methode befindet sich eine Struktur mit Informationen zu dieser Methode (Klasse, Index, Exceptiontabelle, erweiterte Objektinformationen etc.).

2. “Funktionstabelle”

   Eine weitere Konstante enthält einen Zeiger auf eine global verfügbare Tabelle (“DispatchTable”), die Einsprungpunkte für Hilfsfunktionen enthält. Diese Funktionen stehen jederzeit zur Verfügung und stellen die Verbindung zum Laufzeitsystem dar.

Für diese Werte ist kein weiterer Speicher (Register, Stack) notwendig, wenn diese Zeiger-Konstanten direkt mit dem Paracode-System in den Code eingestanzt werden können. Dies ist z.B. auf dem x86 und anderen CISC-CPUs der Fall.

Die Alpha-Architektur, wie auch viele andere RISC-CPUs erlauben hingegen nur das Einfügen von Konstanten bis zu 16Bit. Größere Werte müssen dann aus diesen 16Bit-Werten zusammengesetzt werden. Aus diesem Grund wird der “Start-of-Code”-Wert der aufzurufenden Methode bereits vor dem Aufruf in ein Register übertragen. Damit wird in der aufgerufenen Methode ein Register zur Speicherung dieses Wertes benutzt und muss daher in Methoden mit weiteren Methodenaufrufen gerettet und restauriert werden.

7.1.2 Methodenaufruf und Methodenrücksprung

Die Parameterübergabe für Methodenaufrufe läuft in der JVM implizit, d.h. die Werte, die auf dem Stack liegen, werden der aufgerufenen Methode als Parameter zur Verfügung gestellt. Der Ort der Speicherung dieser Werte auf dem Stack ist für das einfache JIFFY-System nicht ohne weitere Logik detektierbar, damit können die Parameter nicht mit den erzeugenden Befehlen in Register abgelegt werden. Daher werden die Parameter direkt dem Stack entnommen und nicht in Registern übergeben.

Für virtuelle Methoden besteht dabei zusätzlich das Problem, dass das Objekt, dessen Methode aufgerufen werden soll, als erstes auf den Stack gelegt wurde und somit dem direkten Zugriff entzogen ist. Die dafür erforderlichen weiteren Schritte sind im Abschnitt 7.1.4.2 beschrieben.

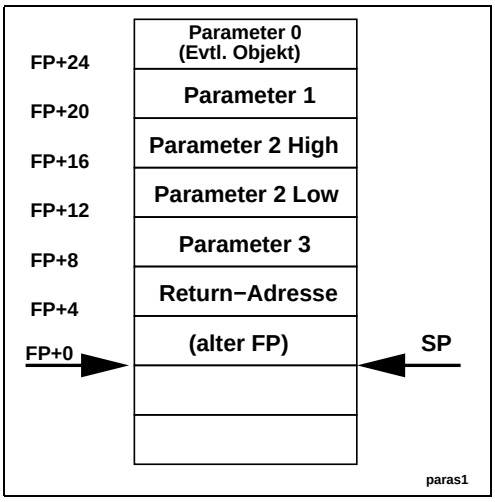

Abbildung 7.2: Stackzustand nach Methodenaufruf (Little Endian)

Nach dem Aufruf der Methode sind die übergebenen Parameter auf dem Stack in der in Abb. 7.2 skizzierten Position. Nach dem Speichern des Stackrahmenzeigers (Basepointer BP bzw. Framepointer FP) ist ein zum üblichen C-Aufrufstandard identischer Stackrahmen vorhanden, über den die Parameter relativ zu BP/FP referenziert werden können.

Ist kein Wissen über die Anzahl der Parameter der aufzurufenden Methode während der Compilation vorhanden, ist eine der Pascal-Konvention ähnliche Vorgehensweise für den Rücksprung zu wählen. Dabei muss die aufgerufene Routine selbst die Parameter vom Stack entfernen und dann auf die zuvor gespeicherte Rückkehradresse springen.

Für Systeme mit explizitem Return-Befehl (typischerweise CISC-CPUs) kann dieser Rücksprung auf zwei Arten geschehen (in Abb. 7.3 für x86 skizziert):

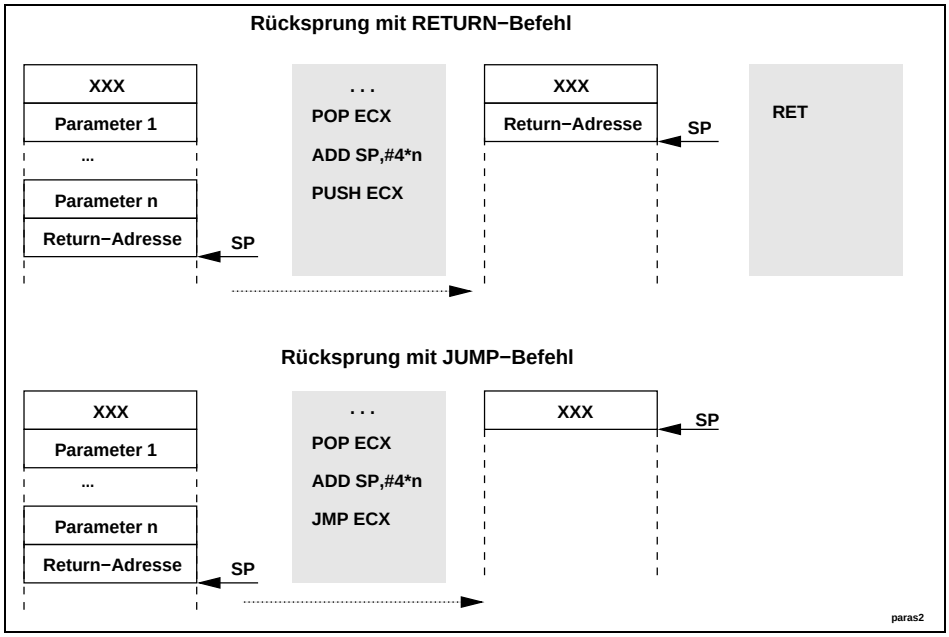

Abbildung 7.3: Varianten des Methodenrücksprungs

1. Die gespeicherte Rückkehradresse wird explizit mit einem PUSH auf den Stack gegeben und anschließend ein Return durchgeführt.

2. Es erfolgt ein expliziter Sprung (JUMP) zur gespeicherten Adresse.

Beide Varianten sind vom Ergebnis äquivalent, zeigen aber abhängig von der Zielarchitektur teilweise stark unterschiedliche Ausführungsgeschwindigkeiten. Messungen dazu sind in Abschnitt 8.3.3.3 aufgeführt.

Sind Informationen über die Methodenparameter verfügbar (wie sie z.B. auch bei einer Bytecode-Verifikation nach dem Verfahren von 6.7.3 vorhanden sind), können die Parameter direkt vom Aufrufer entfernt werden, wie es bei der C-Aufrufkonvention der Fall ist. Dies würde eine erhöhte Ablaufgeschwindigkeit einbringen, allerdings auf Kosten der Einfachheit der Übersetzungshardware. Daher wird auf diese Realisierungsvariante nicht näher eingegangen.

7.1.3 Objektauflösung und Methodencompilation

Während des Übersetzungsvorgangs ist es nötig, dass der Compiler Zugriff auf Informationen des Konstantenpools hat. Da diese Informationen sehr schnell zugreifbar sein müssen, sind sie entweder im FPGA bzw. in einem FPGA-nahen Speicher untergebracht, in den sie vor der Compilation kopiert werden. Um diesen Aufwand zeitlich zu beschränken, müssen nicht alle Informationen kopiert

werden. Das JIFFY-Konzept schränkt die nötigen Informationen daher auf den Rückgabetyp einer Methode bzw. eines Elements ein, damit ist eine einfachere Compilation und Optimierung möglich. Zusätzlich ist der für diese Information benötigte Speicherplatz relativ gering und kann deshalb latenzarm im FPGA untergebracht werden.

Grundsätzlich ist die Implementierung der Auflösung von Methoden einfacher als die der Elementzugriffe. Methodenaufrufe enthalten bereits einen impliziten Sprung, dieser kann gleichzeitig für die erste Auflösung benutzt werden.

7.1.4 Auflösung von Methoden

Ein Auflösen der referenzierten Methoden (Resolving) vor oder während der Compilation ist aus Zeit und Effizienzgründen nicht möglich, schließlich kann die Auflösung zu wiederholtem, rekursiven Laden von Klassen führen. Aufgrund der starren Übersetzung müssen somit einige Aufgaben zur Methodenauflösung von der JIT-Compilezeit in die Laufzeit des eigentlichen Codes verschoben werden.

Diese Umstände müssen bei der Objektauflösung während des Programmlaufs des compilierten Codes berücksichtigt werden. Es gibt hierbei zwei mögliche Varianten, die in der Effizienz und Anwendbarkeit unterschiedlich sind:

- Auflösung ohne selbstmodifizierenden Code

  Jede Klasse besitzt zusätzlich zum Konstantenpool ein Feld mit Zeigern auf übersetzte Methoden und Objektadressen. Ist eine Methode noch nicht aufgelöst, beinhaltet der Eintrag einen Zeiger auf eine universelle Auflösungsfunktion. Übersetzte `invoke`-Befehle springen schließlich auf die durch den Index erreichbare Adresse zur Auflösungsfunktion. Nach der Auflösung einer Funktion und einer eventuellen JIT-Compilierung wird die Startadresse in das Feldelement eingetragen. Damit springen zukünftige Aufrufe direkt die übersetzte Methode an.

  Dieses Vorgehen erlaubt zwar eine einfache Übersetzung von statischen Methoden, geht aber von folgende Voraussetzungen aus:

  1. Die Startadresse der Funktionstabelle muss während des Programmlaufs bekannt sein, d.h. sie muss bereits bei der Assemblierung als Konstante in der `invoke`-Emulation eingefügt sein.
  2. Die Auflösungsfunktion benötigt weitere Informationen zur übersetzenden Methode, die mit übergeben werden müssen. Da vor dem Sprung nicht klar ist, ob die Funktion bereits aufgelöst ist, sind diese Parameter immer zu spezifizieren. In der JIFFY-Implementierung

sind dies u.a. ein Zeiger auf den Methodenanfang (Start-of-Code, siehe 7.1.1). Die dafür notwendigen Befehle werden nur für den ersten Aufruf benötigt, könnten danach also gelöscht werden.

3. In den Konstantenpooleinträgen zu einem Index ist nicht vermerkt, ob es sich um eine statische oder um eine virtuelle Methode handelt. Diese Eigenschaft ist auch nicht zuverlässig aus der Signatur abzuleiten. Der einzige Weg besteht darin, die Klasse der Methode zu laden und die Methodenattribute zu untersuchen. Daher ist es nicht möglich, schon anhand des Index die Tabelle mit spezifischen Auflösungsfunktionen vorzubelegen. Die Unterscheidung zwischen statischen und virtuellen Methoden kann nur anhand des aufrufenden Befehls selbst erfolgen. Dieser muss also einer allgemeinen Auflösungsfunktion noch den Typ der gewünschten Auflösung mitteilen.

Der erste Punkt erfordert die Möglichkeit der Parametrisierung des Übersetzungsvorgangs in Abhängigkeit von der zu bearbeitenden Methode. Dies kann “manuell” nach der maschinellen Übersetzung geschehen, und entweder am Anfang der Funktion auf dem Stack oder in einem Register gespeichert werden oder für jeden Aufruf explizit.

Diese nachträgliche Codebearbeitung widerspricht dem Kompatibilitätsansatz für das Laufzeitsystem und erfordert Informationen über die einzufügenden Stellen. Daher wird in JIFFY diese Parametrisierung automatisch während der Maschinencodeerzeugung vollzogen, da sie durch den parametrisierbaren Assembler ohne größere HW-Änderungen möglich ist. In der Paracode-Spezifikation ist es somit möglich, eine zur Compilationszeit spezifizierbare Konstante in den Code einzufügen (Argumente %a4-%a15, siehe Paracode-Beschreibung in A.1).

Der zweite Punkt resultiert in einem etwas ineffizienteren Ablauf, wäre aber durch die weiter unten besprochene Methode mit selbstmodifizierendem Code zum Teil zu umgehen.

- Auflösung mit selbstmodifizierendem Code

  Diese Methode arbeitet bis zum Zeitpunkt der Auflösung identisch zur ersten, ersetzt allerdings den allgemeinen `invoke`-Code durch einen optimierten Teil. Für statische Methoden wird die gefundene Funktion direkt, ohne Umweg über eine Tabelle angesprungen. Für virtuelle Methoden kann der sonst explizit erzeugte Stub direkt eingefügt werden. Da der übrige Code nicht verändert wird, entstehen (besonders bei den einfachen `invoke_static`-Befehlen) Lücken, die mit neutralen Befehlen (z.B. NOP) ausgefüllt werden.

Diese Methode besitzt drei Nachteile:

1. Da die Veränderung zu einem kurzzeitig inkorrektem Code führt, darf kein anderer Thread diese Methode ausführen, damit ist eine Synchronisation mit parallel laufenden Threads notwendig. Dies ist nur aufwendig sicherzustellen, da normale Methoden keine Synchronisation haben.

2. Eine Löschung des compilierten Codes der aufgerufenen Methode (z.B. im Garbage Collector) bedingt Wiederherstellung aller Codepositionen, die diese Methode aufrufen. Dies ist nur mit zusätzlichem, hohem Verwaltungsaufwand möglich.

3. Bei Prozessoren ohne Instruktionscache-Snooping (z.B. ARM) ist nach dieser Modifikation der gesamte I-Cache zu löschen. Dies führt zu starken Leistungseinbrüchen.

Da diese Nachteile in der Verwaltung wesentlich schwerer wiegen als der Geschwindigkeitsgewinn durch Auslassung überflüssiger Befehle, wird diese Methode in JIFFY nicht benutzt, obwohl sie auch im JIFFY-System prinzipiell einsetzbar wäre.

7.1.4.1 Behandlung statischer Methodenaufrufe

Statische Methoden erlauben eine einfache und kombinierte Behandlung der Auflösung beim ersten Aufruf und direktem Methodenaufruf bei allen weiteren Aufrufen. Basis dafür ist eine Methodentabelle, die parallel zum Konstantenpool angelegt wird und für eine Klasse global gilt. Für jeden Index einer statischen Methode wird dazu eine Funktion eingetragen (siehe Bild 7.4), die die Auflösung und eventuelle JIT-Compilation der Methode ausführt. Nach Abschluss der Auflösung wird die Startadresse der Methode in die Tabelle unter dem Index der Methode eingetragen. Damit kann diese Methode in Zukunft direkt aufgerufen werden.

Als statischer Overhead auch im aufgelösten Fall ist nur die Belegung der Register mit Daten für eine erste Auflösung vorhanden (in Bild 7.4 grau hinterlegt). Diese Werte (CP-Index und Start-of-Code) sind für die Auflösung unbedingt erforderlich. Für x86 sind dies drei zusätzliche `mov` Befehle, die nur unwesentlichen Einfluss auf die Aufrufgeschwindigkeit haben (siehe 8.3.3).

In einem System mit Speicherbeschränkungen erlaubt es das konstante Vorhandensein der Registerbelegung und damit die immer mögliche erneute Auflösung allerdings, zu beliebigen Zeitpunkten eine bereits compilierte Methode wieder zu löschen und somit die dynamische Speicherbelegung zu minimieren.

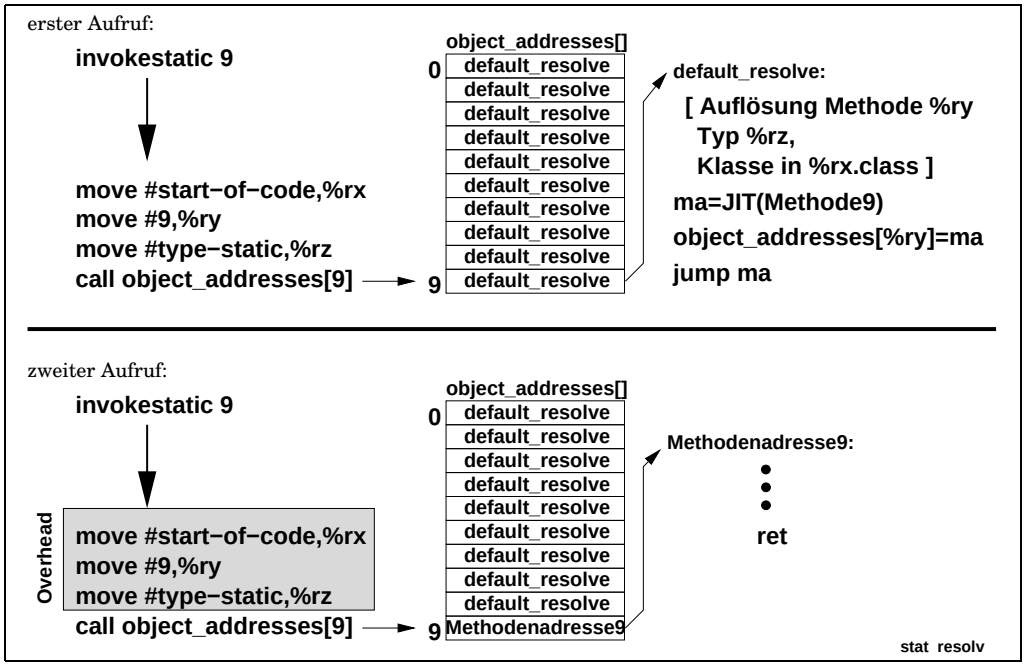

Abbildung 7.4: Auflösung statischer Methoden

7.1.4.2 Behandlung von virtuellen Methoden

Für virtuelle Methoden ist der Ablauf für die Auflösung aufwendiger als für statische Methoden, da die Methode erst anhand des Indices und des übergebenen Objekt gefunden werden muss.

Die dafür notwendige Adressberechnung unterscheidet sich grundlegend von der für eine statische Methode: Die effektive Methodenadresse ist in einer Methodentabelle enthalten, die im Objekt referenziert wird. Allerdings ist in dieser Methodentabelle der Konstantenpoolindex nicht mehr gültig, da das Objekt unabhängig von der aufrufenden Klasse ist. Somit sind vor dem Aufruf die objektspezifische Methodentabelle (und damit ein Zugriff auf das Objekt) und der Index in diese notwendig. Da das Objekt implizit als erstes Argument auf den Stack gelegt wird, ist (ohne Datenflussanalyse der Stackoperationen) zur Referenzierung der Methodentabelle die Anzahl der Argumente in Erfahrung zu bringen. Diese ist jedoch zur JIT-Compilationszeit in der Hardware zunächst nicht vorhanden und wird erst bei der Auflösung bekannt.

Im JIFFY-System wird zur Realisierung dieser virtuellen Methodenaufrufe nach der Auflösung ein zweistufiges Konzept benutzt, das in Abb. 7.5 dargestellt ist.

Die Auflösung selbst erfolgt wie bei statischen Methoden über einen Sprung durch die Methodentabelle, dessen Zielfunktion anhand des Methodenindex be-

stimmt wird. Der Funktion werden neben dem CP-Index auch noch der Stackzeiger mitübergeben, anhand dessen das referenzierte Objekt zu finden ist. Die Auflösungsfunktion befindet sich an derselben Adresse wie die für statische Methoden. Dies ist deshalb notwendig, da anhand des CP-Index nicht erkennbar ist, ob es sich um eine virtuelle oder eine statische Methode handelt. Somit muss der Typ der gewünschten Auflösung mit übergeben werden.

Die Auflösungsfunktion erzeugt dann eine Zwischenroutine (Stub, siehe auch 7.1.6), in der die Parameteranzahl fest eincompiliert ist und somit einen direkten Zugriff auf die Methodentabellen des Objekts erlaubt. Da der Index der gefundenen Methode in der Tabelle der virtuellen Funktionen von der Superklasse abhängt und keinem Zusammenhang mit dem CP-Index aufweist, muss dieser nach der Bestimmung ebenfalls im Stub gespeichert werden. Ist die gefundene Methode noch nicht compiliert, aber im Bytecode vorhanden, wird sie compiliert und anschließend aufgerufen. Zum Abschluss wird die Adresse des erzeugten Stubs in der Methodentabelle vermerkt.

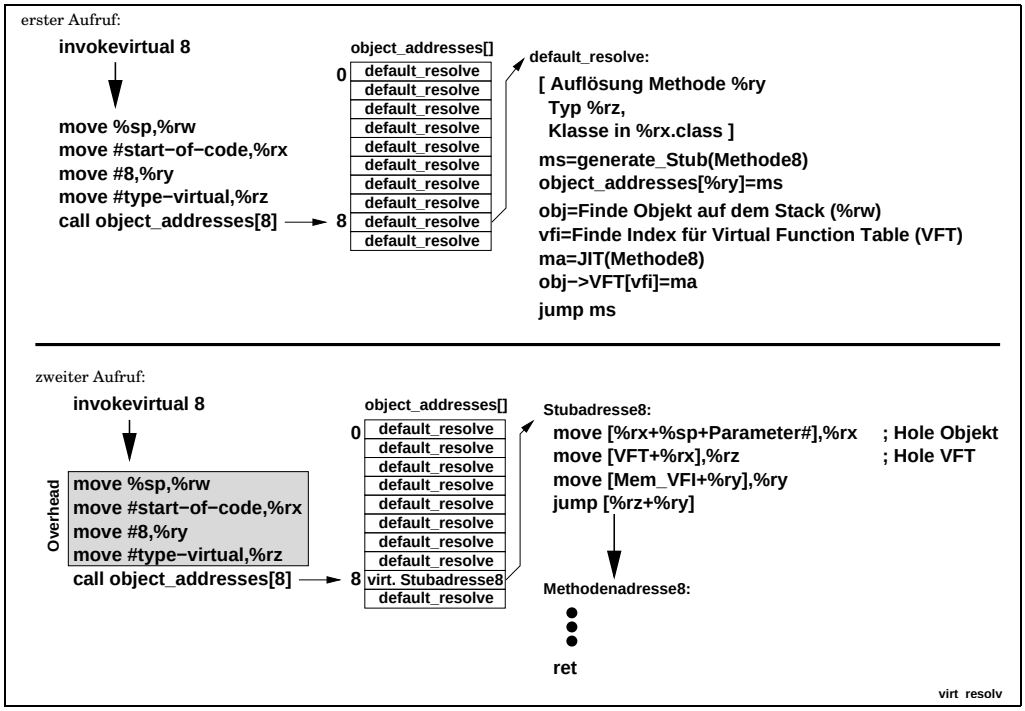

Abbildung 7.5: Auflösung virtueller Methoden

Wird die virtuelle Methode erneut aufgerufen, wird in der ersten Stufe nur anhand des Konstantenpoolindex auf die während der Auflösung speziell für diese Methode erzeugte Zwischenroutine verzweigt. Dieser Stub greift dann auf die Methodentabelle des Objekt zu und springt an den durch den Index spezifizierten Ort des Methodencodes.

Diese Art der Auflösung der virtuellen Funktionen lässt sich bezüglich des Speicherverbrauchs noch optimieren, wenn die Methodennummer nicht direkt in diesen Stub einkompiliert wird, sondern ebenfalls aus einer Tabelle bezogen wird. Damit sind diese Stubs nur noch von der Anzahl der Funktionsparameter abhängig und können bereits im Vorfeld klassenunabhängig erzeugt werden, was Compilationszeit und Speicherplatz spart.

Insgesamt beträgt der zusätzliche, nicht weiter verringerbare Aufwand für einen virtuellen Methodenaufruf nach der ersten Auflösung auf einer x86-CPU ca. 14 Befehle. Darin enthalten sind das Vorbelegen der Register mit Methodeninformation (CP-Index, Start-of-Code, Stackpointer) und der Zugriff auf die Objekteigene Methodentabelle. Eine genauere Auflistung der Befehle wird in 8.3.3 gegeben.

7.1.5 Auflösung von Elementen

Elementzugriffe werden bereits in der Übersetzung JVM→IJVM in zwei IJVM-Befehle aufgeteilt. Der erste Befehl (`gsvi/gfvi`) bestimmt anhand des Objekts und des Index die Adresse des Elements, der zweite Befehl greift dann auf dieses Element typabhängig zu. Damit beschränkt sich die Auflösung auf die Bestimmung der Elementadresse. Ohne selbstmodifizierenden Code, dessen Nachteile bereits angesprochen wurden und hier analog gelten, ist damit für Elementzugriffe (Konstantenpoolelemente, statische oder Objektelemente) immer ein bedingter Sprung notwendig, da der Status des referenzierten Elements erst geprüft werden muss. Je nach Zielarchitektur kann dieser jedoch so gestaltet werden, dass er für den häufigeren Fall (Elementzugriff eines bereits aufgelösten Objekts) effizienter ist.

7.1.6 Portable Erzeugung der Stubs

Neben dem Code, der direkt aus den JVM-Befehlen erzeugt wird, sind weitere maschinenabhängige Programmstücke (Stubs) zur Verbindung von JIT-Code und dem Laufzeitsystem notwendig, die teilweise auch erst zur Laufzeit erzeugt werden:

1. Übergang von interpretierten/nativen JRE-Methoden zu JIT-Methoden
   - Übergabe der Parameter vom Interpreterstack auf den nativen Stack
   - Aufsetzen der JIT-Umgebung (Environment)
   - Rückgabe des Funktionsergebnisses
2. Hilfsfunktionen zur Methodenauflösung im JIT-Code

- Adressierung JIT/JRE-spezifischer Tabellen
- Einfügung von Ergebnissen der Methodenauflösung

3. Übergang von JIT-Code zu interpretierten/nativen Methoden
   - Übergabe der Parameter des nativen Stacks auf den Interpreterstack
   - Sicherung des JIT-Environments (z.B. Register)
   - Rückgabe des Funktionsergebnisses an den JIT-Code

Die Codeerzeugung aller drei Stubtypen ist ebenfalls mit der JIT-Hardware portabel möglich. Dazu sind in der IJVM verschiedene Grundfunktionen spezifiziert, die den Übergang zwischen JRE und JIT-Ebene abstrakt handhaben. Aus diesen Funktionen wird zur Laufzeit der Stub auf IJVM-Ebene an die spezifische Aufgabe angepasst (z.B. abhängig von der Signatur der aufzurufenden JRE-Methode) und dann mit der Paracode-Assembler-HW übersetzt. Damit ist der Zielprozessor nur bei der Erstellung der IJVM→NAT-Tabelle zu berücksichtigen, während der JIT-Laufzeit ist ein spezifisches Wissen darüber nicht mehr notwendig.

7.1.6.1 Beispiel zur Stubgenerierung

Soll aus JIT-Code eine native JRE-Methode mit der Signatur "(int, long)→int" aufgerufen werden, wird bei der Auflösung direkt aus der Signaturangabe folgender IJVM-Stub für diese Methode erstellt:

```
initgetn 2 ; Initialisierung des Parametertransfers
getnargi   ; Transfer integer-Parameter
getnargl   ; Transfer long-parameter
calljdk_i  ; Aufruf des JRE-Systems, Rückgabe eines integer
endfunc    ; Endemarker für Assembler
```

Je nach Nutzung können vor dem Start des Programms bereits mehrere dieser Stubs für häufige Signaturen generisch erzeugt werden, und somit den Overhead während der Compilation auf Kosten des Speicherverbrauchs verringern.

7.1.7 Schleifen-Optimierung

Bei der Analyse des optimierten IJVM-Codes und des entsprechenden nativen Maschinencodes stellte sich eine Befehlssequenz als stark geschwindigkeitssenkend heraus. Diese ist mit den "normalen" Peephole-Regeln nicht optimierbar, da sie eine Optimierung über Basisblöcke hinweg erfordert:

Die Methode, wie Zählschleifen in JVM aus Java übersetzt werden, unterscheidet sich nicht wesentlich von der aus C-Compilern. Üblicherweise ergibt sich mit javac folgende JVM-Befehlsanordnung:

```
                    ; Schleifenzähler: Variable L
        ldc <start_val> ; Startwert des Zählers holen
        istore L        ; Setzen der Zählvariable
        goto test
label_loop:
        <....>          ; Schleifenkörper
        iload L
        ldc <inc_val>   ; Zählerincrement
        iadd
        istore L        ; Schreiben des neuen Zählerwerts
label_test:
        iload L
        ldc <boundary>  ; Grenzwert holen
        if_icmplt loop  ; Wert nicht erreicht

        <....>
```

Interessant ist hier der Bereich um das Label `test`. Der veränderte Schleifenzähler wird zurückgeschrieben, anschließend erneut geladen und mit der Schleifengrenze verglichen. An sich könnte das erneute Laden wegoptimiert werden, da in der registerbasierten IJVM der Wert immer noch in einem Register liegen würde und nicht wie bei der JVM mit `istore` vom Stack gelöscht wurde.

Allerdings springt der Schleifenanfang direkt auf den Ladebefehl und erzeugt damit einen neuen Blockanfang. Daher ist keine regelgerechte Optimierung mehr möglich, da nicht garantiert ist, dass `istore` vor der Sprungquelle den Wert ebenfalls im selben Register gespeichert hat.

Für die beiden untersuchten Architekturen x86 und Alpha ist die direkte Aufeinanderfolge des Schreibens und Lesens aus derselben lokalen Variable eine Serialisierungsoperation. Bei leerem Schleifenkörper ergibt diese eine zusätzliche Ladeoperation auf dem x86 ca. 55% (Duron) bis 67% (Pentium2), auf dem Alpha sogar nur ca. 42% an Geschwindigkeit gegenüber einer Schleife mit nur einem Schreibbefehl.

Weitere Untersuchungen an verschiedenen JVM-Codes mit Schleifen ergaben, dass auch bei komplexeren Schleifenkörpern dieser Effekt noch signifikant messbar ist und daher einer Optimierung bedarf.

In der Ausführung am effizientesten wäre die Verlagerung der lokalen Variable in ein Register, dies ist aber nicht mit allen Prozessoren möglich und verlangt eine Registerallokierungsstrategie. Daher wurde versucht, das Problem architekturunabhängig mit der existierenden Optimierungslogik wie folgt zu lösen:

Es gibt zwei spezielle Optimierungsregeln für diese Schleifenart: Die erste ermittelt das Auftreten der Schleifeninitialisierung (`istore L; goto x`) mit der zugehörigen lokalen Variable und speichert das Sprungziel. Die zweite Regel überprüft, ob das Update des Zählers, das einmalig angesprungene Label und das

Laden der lokalen Variable auftritt. Ist dies der Fall, wird der `iload`-Befehl eliminiert. Da nur eine lokale Variable und ein Sprungziel gleichzeitig gespeichert werden kann, ist es nicht möglich, verschachtelte Schleifen mit Ausnahme der Innersten zu optimieren. Allerdings sind für die äusseren Schleifen auch keine signifikanten Verbesserungen zu erwarten. Durch die Überprüfung auf die Variable und das Sprungziel ist es ausgeschlossen, dass die Optimierung "zufällig" erfolgt. Selbst ohne Schleife ist die Optimierung immer korrekt.

7.2 Benutzte Peepholeregeln

7.2.1 Regeln der ersten Peepholestufe

Für die erste Stufe der Optimierung haben sich die im Anschluss erklärten Regeln als ausreichend herausgestellt. Zusätzlich zur eigentlichen Peepholeoptimierung beschreiben die Regeln auch das Weiterschalten des Fensters, das Eliminieren von NOP-Befehlen und die Beendigung der Optimierung am Ende einer Methode. Diese Regeln wurden zur vereinfachten Evaluierung des Ablaufs eingefügt, sind aber in einer Hardwareimplementierung der Peepholeoptimierung fest verdrahtet und werden hier nicht aufgeführt.

Die Regeln des ersten Regelsatzes sind:

- Laden von Registern aus beliebigen MOVEs mit anschließendem PUSH/POP

```
S0_0(
  movecxr,          push,           pop&&A1_1   \
: I1_1,             Gnop,           Gnop,      *)
```

  Diese Regel ersetzt mit `I1_1`) das Zielregisters des MOVEs durch das Zielregister des POP-Befehls (gespeichert mit `A1_1`) und eliminiert die PUSH/POP-Folge durch die Erzeugung von zwei NOP-Befehlen (Gnop, "Generate NOP").

  Beispiel:
  `movevr_i 5,%r2; push_i %r2; pop_i %r1`
  → `movevr_i 5,%r1`

- PUSH/POP ohne vorhergehenden MOVE-Befehl

```
S0_1(
  !movexr,          push&&A1_1,     pop&&A2_1 \
: *,                Gnop,           Gmove&&I2_1&&I1_2,* )
```

Die PUSH/POP-Folge dient dem Umladen eines Registers, das als Ergebnis einer Rechenoperation entstanden ist. Hierbei wird ein `move`-Befehl eingefügt, der für Quell- und Zielregister die Werte der PUSH/POP-Befehle übernimmt.

Beispiel:
```
add_i %r1,%r2; push_i %r2; pop_i %r1
→ add_i %r1,%r2; move_i %r2,%r1
```

- Schleifenoptimierung, Teil 1

```
S0_2(
  moverv ,bra, * \
: ST_REGLV,              ST_LABEL,              * )
```

Diese Regel speichert bei einem Sprung das angesprungene Label (`ST_LABEL`) ab, wenn der Befehl vor dem Sprung eine lokale Variable geladen hat. Die Nummer dieser Variable und das Zielregister wird ebenfalls abgespeichert (`ST_REGLV`.

Beispiel:
```
moverv_i %r1,5; bra 77
```
→ (LVREG:=r1, LVNUM:=5, LABEL:=77)

- Schleifenoptimierung, Teil 2

```
S0_3(
  moverv&&R_REGLV,label1&&R_LABEL,movevr&&R_REGLV, *\
: *              , *             , Gnop,  *)
```

Tritt der in 7.1.7 beschriebene Fall der Schleifenfortschaltung ein, wird der Ladebefehl der lokalen Variablen eliminiert, falls die Variablennummer, das Zielregister (`R_REGLV`) und das Sprungziel (`R_LABEL`) mit den in Regel S0_2 gespeicherten Werten übereinstimmt.

Beispiel:
(LVREG=r1, LVNUM=5, LABEL=77)
```
moverv_i %r1,5; label1 77; movevr_i 5,%r1
→ moverv_i %r1,5; label1 77
```

7.2.2 Regeln der zweiten Peepholestufe

Die zweite Stufe der Peepholeregeln wird zweimal durchgeführt und besteht aus folgenden Regeln:

- Eliminierung eines PUSH/POP getrennt durch beliebiges Move

```
S1_0(
  movexr,  push,   movecxr,   pop&&A1_1,      *  \
: I1_1,    Gnop,   *,         Gnop,           *  )
```

  Beispiel:
  `movevr_i 5,%r1; push_i %r1; movevr_i 6,%r1; pop_i %r2`
  → `movevr_i 5,%r2; movevr_i 6,%r1;`

- Ersetzung eines Konstanten-MOVEs durch Addition mit Immediate-Wert

```
S1_1(
  movec_i&&A1_1,  add_i&&R1_2&&A2_1  \
: Gaddc&&I2_1,    Gnop              )
```

  Beispiel:
  `movec_i 2,%r2; add_i %r2,%r1`
  → `addc_i 2,%r1`

- Ersetzung eines Konstanten-MOVEs durch Subtraktion mit Immediate-Wert

```
S1_2(
  movec_i&&A1_1,  sub_i&&R1_2&&A2_1  \
: Gsubc&&I2_1,    Gnop    )
```

  Beispiel:
  `movec_i 2,%r2; sub_i %r2,%r1`
  → `subc_i 2,%r1`

- Ersetzung eines Konstanten-MOVEs durch Vergleich mit Immediate-Wert

```
S1_3(
  movec_i&&A1_1,  cmp_i&&R1_2&&A2_1   \
: Gcmpc&&I2_1,    Gnop    )
```

  Beispiel:
  `movec_i 2,%r2; cmp_i %r2,%r1`
  → `cmpc_i 2,%r2`

- Eliminierung des Rückgabekonstrukts, wenn Wert bereits im Standardrückgaberegister r1 steht

```
S1_4(   retdata&&E1_1,  *    \
:       Gnop,           *    )
```

Beispiel:
```
retdata %r1
```
→ {}

- Elimination eines Register-MOVEs mit identischer Quelle und Ziel

```
S1_5(   move&&E12,      *    \
:       Gnop,           *    )
```

Beispiel:
```
move_i %r1,%r1
```
→ {}

- Eliminierung eines PUSH/POP gespeist aus einem MOVE (Wiederholung von S0_0)

```
S1_6(
  movecxr,        push,           pop&&A1_1,      *,   \
: I1_1,           Gnop,           Gnop,           *    )
```

- Eliminierung eines durch MOVE getrennten PUSH/POP mit gemeinsamen Registern

```
S1_7(
  push&&A1_1,     movecxr,        pop&&R1_1, *   \
: Gnop,           *,              Gnop,      *   )
```

Beispiel:
```
push_i %r1; movevr_i 5,%r2; pop_i %r1
```
→ `movevr_i 5,%r2`

- Eliminierung eines durch MOVE getrennten PUSH/POP mit unterschiedlichen Registern

```
S1_8(
  push&&A1_1,        movecxr, pop&&A2_1, *    \
: Gmove&&I2_1&&I1_2, *,       Gnop,      *    )
```

Beispiel:
```
push_i %r2; movevr_i 5,%r2; pop_i %r1
```
→ `move_i $r2,%r1; movevr_i 5,%r2`

- Eliminierung eines durch zwei MOVE getrennten PUSH/POP, gespeist aus einem MOVE

```
S1_9(
  movecxr, push&&A1_1, movecxr, movecxr, pop&&A2_1  \
: I2_1,    Gnop,       *,       *,       Gnop    )
```

Beispiel:
```
movevr_i 5,%r1; push_i %r1; movevr_i 6,%r2;
movevr_i 7,%r3; pop_i %r1
→ movevr_i 5,%r1; movevr_i 6,%r2; movevr_i 7,%r3
```

Die tatsächliche Effizienz der beiden Regelsätze mit ihren Auswirkungen auf zwei Beispielarchitekturen wird in Abschnitt 8.3.2 beschrieben.

7.3 Optimierung durch Methodeneigenschaften

Einige Eigenschaften der zu compilierenden Methode können zur einfachen Optimierung des Codes führen. In JIFFY werden dabei die im Folgenden beschriebenen Optimierungen vorgenommen, die alle durch eine dynamische Modifikation der IJVM→NAT-Lookuptabelle vor der Compilation erreicht werden können:

1. Optimierung der Zugriffe auf lokale Variablen auf dem Stack

   Einige Architekturen besitzen Befehle zur indirekten Adressierung mit unterschiedlich breiten Offsets. Der x86 kann z.B. Offsets von 8 oder 32Bit verarbeiten. Eine Nutzung der 8Bit-Variante spart damit drei Byte pro LV-Zugriff und beschleunigt durch geringere Cachebelegung die Ausführung etwas.

   Eine dynamische Optimierung während der Codeerzeugung wäre zwar möglich, würde aber die Datenpfade bei der JVM→IJVM-Übersetzung verlängern. Aus diesem Grund wird die Entscheidung für 8/32Bit-Offsets statisch pro Methode getroffen. Dazu muss nur vor der Übersetzung der generische LV-Zugriffsbefehl (MOVEVR/MOVERV) in der Tabelle durch eine angepasste Variante (MOVESVR/MOVESRV) ersetzt werden. Die Entscheidung, ob diese Optimierung möglich ist, wird anhand der benutzten Anzahl lokaler Variablen und Aufrufparameter getroffen. Bei einem vorzeichenbehafteten 8Bit-Offset sind somit minimal 32 lokale Variablen möglich (ohne Parameter). Da dieser Wert so gut wie nie überschritten wird, ist diese Optimierung bei minimalem Softwareaufwand sehr effizient.

2. Optimierung für Methoden ohne lokale Variablen

   Methoden, die keine echten lokale Variablen benötigen (aber Parameter haben können) profitieren von modifizierten Enter/Leave-Konstrukten, da im Stackframe kein Platz für lokale Variablen angelegt werden muss.

3. Optimierung für Methoden ohne weitere Methodenaufrufe (RISC)

   RISC-Prozessoren besitzen üblichweise keine speziellen CALL bzw. RET Befehle, die die Rückkehradresse automatisch auf dem Stack abspeichern bzw. von dort laden. Stattdessen wird die Rückkehradresse in einem Register zwischengespeichert, das, wenn weitere Unterroutinen aufgerufen werden, manuell auf dem Stack gesichert werden muss. Ruft also eine Funktion keine weiteren Funktionen auf ("Leaf"-Funktion), kann das Retten und Restaurieren dieser Adresse wie auch der JIFFY-eigenen "Start-of-Code"-Konstante entfallen. Dies erspart auf dem Alpha vier Befehle pro Methode. In Kombination mit Optimierung 2) wird der Funktionsprolog und Epilog von 14 auf 6 Befehle verkürzt.

Die Einsetzbarkeit der beschriebenen Optimierungen lässt sich anhand eines einfachen Tests der Methodeneigenschaften vor der IJVM→NAT-Übersetzungsstufe feststellen. Die Aktivierung lässt sich mit einem einzigen Schreibzugriff auf die IJVM→NAT-Lookuptabelle ausführen, womit das Mapping der IJVM-Befehle auf die verschiedenen Implementationsvarianten verändert werden kann.

7.4 Hardwareaspekte

Das beschriebene JIFFY-System wurde in Teilen bereits auf eine FPGA-Architektur synthetisiert und bezüglich der Implementierungseigenschaften untersucht.

7.4.1 Übersetzung in den IJVM-Zwischencode

Wie in Abb. 7.6 gezeigt, fallen für die Umsetzung auf den IJVM-Zwischencode bis zu 6 Lookupoperationen an. Davon sind aber bis auf den Zugriff auf den Typ eines objektorientierten JVM-Befehls alle auf internem oder FPGA-nahem Speicher durchführbar. Eine Pipelinestruktur mit Handshakesignalen ist durch die sequentielle Struktur ohne größere Probleme einsetzbar und ermöglicht somit eine hohe Taktfrequenz. Mit einem FIFO und einem DMA- oder SDRAM-Bankzugriff kann die Sequentialität der Ein- und Ausgangsdaten effizient ausgenutzt werden.

Für die Komplexität des FPGAs entscheidend ist der Dekodierautomat für einen JVM-Befehl. Durch die variable JVM-Befehlsstruktur sowie der komplexen `tableswitch`/`lookupswitch`-Befehle entstehen relativ viele Zustände. Diese werden entweder direkt in einen Automaten umgesetzt [131] oder über ein Mikroprogramm abgearbeitet.

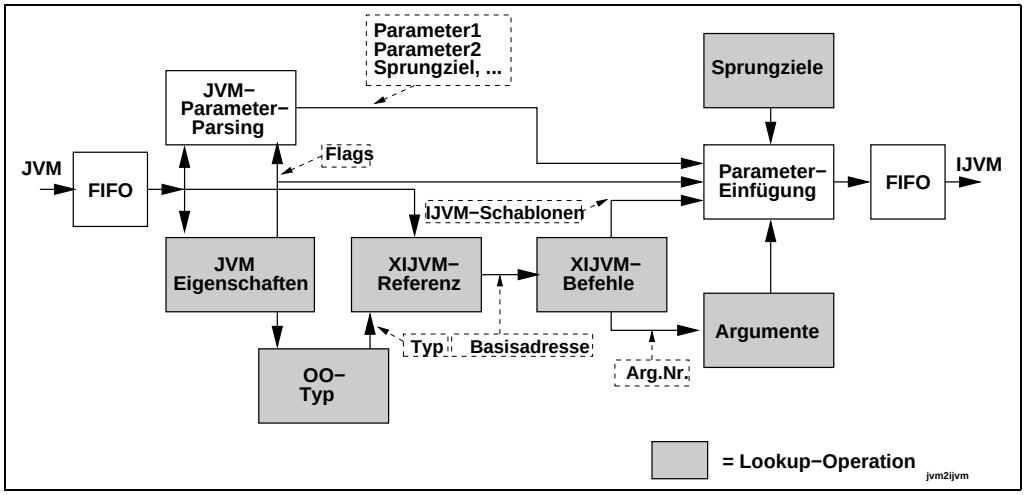

Abbildung 7.6: Aufbau der Übersetzung JVM→IJVM

Die Mikroprogrammlösung ist für ein FPGA über die internen RAM-Blöcke einfacher zu realisieren, erlaubt weniger Verdrahtungsaufwand und ist leichter wartbar und erweiterbar.

Die möglichen Eigenschaften eines JVM-Befehls, die für JIFFY benötigt werden, sind in Tabelle 7.1 zusammengefasst. Die Beschreibung der Eigenschaften lässt sich unkomprimiert mit 16Bit darstellen, mit einer einfachen Vorcodierung ist dies auf max. 5 Bit reduzierbar, da höchstens 4 Eigenschaften gleichzeitig benutzt werden. Nutzt man diese 5Bit als Einsprungadresse in ein 256-Einträge umfassendes Mikroprogramm-ROM, können also minimal 8 einzelne Operationen pro JVM-Befehl ausgeführt werden, dies ist mehr als ausreichend. Die einzelnen dazu nötigen Befehle sind in Tabelle 7.2 aufgeführt, jeder dieser Befehle stößt einen fest verdrahteten, aber relativ einfachen Automaten zur Ausführung der Aktion an.

Da ein großer Teil der Logik zur Verarbeitung des JVM-Befehlstromes für Analyse und JVM→IJVM-Übersetzung identisch sind, bietet es sich an, die Funktionalität beider Phasen zusammenzulegen und somit ca. 40% der insgesamt dazu nötigen FPGA-Ressourcen einzusparen.

7.4.2 Peephole-Optimierung

Ein kritisches Element in der Implementierung des JIFFY-Konzept ist die Peephole-Optimierung, da sie maßgeblich die Geschwindigkeit der Übersetzung bestimmt.

Ein Hauptproblem besteht in der möglichst parallelen Erkennung der Muster und die Durchführung der entsprechenden Ersetzungsregeln. Ist es “nur” mög-

WIDE	Mit WIDE-Opcode erweiterbar
P1_B	Ein Byte-Parameter
P1_W	Ein Wort-Parameter
P2_B	Zwei Byte-Parameter
P2_W	Zwei Word-Parameter
P1_BO	Ein 2Byte-Sprungoffset
P1_L	Ein 4Byte-Sprungoffset
P_TABLE	`tableswitch`-Befehl
P_LOOKUP	`lookupswitch`-Befehl
P_LVAR	Zugriff auf lokale Variable
P_LVAR0/1/2/3	Zugriff auf lokale Variable mit Index im Befehl
P_INVOKE	`invoke*`-Befehl
P_OO	Objektorientierter Befehl
P1_BS	Byte-Parametererweiterung
P1_WS	Word-Parametererweiterung

Tabelle 7.1: Mögliche Eigenschaften eines JVM-Befehls

EXB_1	Extraktion des nächsten Bytes als ersten Parameter
EXW_1	Extraktion des nächsten Words als ersten Parameter
EXB_2	Extraktion des nächsten Bytes als zweiten Parameter
EXW_2	Extraktion des nächsten Words als zweiten Parameter
EXBO	Extraktion des nächsten Words als Sprungoffset
GEN_LABEL	Sprungziel in Labelbefehl umwandeln
GEN_LIDX	Lokale Variable aus Befehlsindex erzeugen
GET_TYPE	OO-Typ suchen
GEN_TYPE	Erweiterten JVM-Befehl aus OO-Typ generieren
GEN_TBSW	Abarbeitung der `tableswitch`-Parameter
GEN_LKSW	Abarbeitung der `lookupswitch`-Parameter
RUN_XFER	Abarbeitung der eigentlichen IJVM-Übersetzung
JVM_END	JVM-Befehl abgearbeitet

Tabelle 7.2: Mikrooperationen bei der JVM→IJVM-Übersetzung

lich, in jedem Takt eine Ersetzungsregel zu überprüfen, wären bei der jetzigen Zusammenstellung der Regeln (siehe Abschnitt 7.2) 4 Takte pro IJVM-Befehl in der ersten Optimierungsstufe und je 10 Takte in der zweiten bzw. dritten Stufe notwendig. Bei einem pessimistisch geschätzten erreichbaren FPGA-Takt von 50MHz sind dies also ca. 2 Millionen IJVM-Befehle pro Sekunde. Diese Berechnung unterlässt aber die Verringerung der Befehle durch die vorhergehenden Stufen, daher ist dieser Wert auch vom zu optimierenden Code abhängig. Ebenso ist es einfacher, das Ausführen der Ersetzungsregel in einem zusätzlichen Takt

durchzuführen, damit würde der Durchsatz je nach Optimierungsrate wiederum verringert. Ergebnisse, wie sich beide Effekte auf den Gesamtdurchsatz auswirken, sind in Abschnitt 8.4.4 näher erläutert.

Könnte pro Takt ein ganzer Regelsatz überprüft und ausgeführt werden, würde sich für drei Durchläufe ein Durchsatz von ca. 16 Millionen IJVM-Befehlen pro Sekunde ergeben. Dieser theoretische Wert ist durch Optimierungen im Datenpfad weiter zu verbessern, wie später noch erklärt wird.

7.4.2.1 Voruntersuchungen

Die Erkennung der Peepholemuster führt zu dem Problem, dass je nach Regel bis zu fünf Befehle (mit je 8Bit) und Befehlsgruppen erkannt werden müssen. Zusätzlich sind Vergleiche auf identische Register und Label notwendig. Insbesondere die Erkennung der Befehlsgruppen ist aufwendig, da eine Gruppe teilweise bis zu sieben verschiedene Befehle umfasst. Eine einfache Erkennung ist hier also notwendig, um FPGA-Ressourcen zu sparen.

In [131] wurde eine Methode entwickelt, die dieses Problem dadurch löst, indem die Befehlsgruppen in einem Hyperwürfel kodiert werden. Mit den verwendeteten Regeln lassen sich die Befehle zu 17 Gruppen zusammenfassen, wobei jede Gruppe eine 5Bit-Kennzeichnung besitzt. Eine Überprüfung auf die passenden Befehle kann dann anhand einer einfachen Lookupübersetzung der Befehle auf die Gruppenzuordnung, einer Maskieroperation und einem Vergleich erfolgen.

[131] beschreibt ebenfalls eine mögliche Implementierung der Peepholeoptimierung mithilfe mehrerer, verschachtelter endlicher Automaten. Diese Implementierung erlaubt zwar eine freie Programmierung der Regeln und bietet einen Durchsatz von einem IJVM-Befehl pro Takt, führt aber bei einer Ersetzungsaktion zu einer Verzögerung von 8 bis 16 Takten und benötigt ebenfalls sehr viel Logik (ca. 2700 Logikzellen/LUTs, siehe auch Abschnitt 8.5). Damit ist der Einsatz der Optimierungseinheit nicht besonders ökonomisch.

Aufgrund dieser ungünstigen Ergebnisse wurde eine verbesserte Realisierung der Peepholeoptimierung über einen strukturierteren Ansatz versucht.

Grundlage der Optimierung war die Funktionalität einer universellen Opcodezelle. Diese bietet zur Erkennung folgende Unterfunktionen:

- Erkennung des Opcodes anhand einer Bitmaske (OPCODE_MASK) und eines Vergleichswertes (OPCODE_MATCH).
- Gemeinsame Speicherung von Registerwerten und Labelzielen in gemeinsamen Speicherplätzen einer Regel.
- Vergleich der gespeicherten Werte.

Wird keine Ersetzung ausgeführt, muss der Befehl unverändert durchgelassen werden. Andernfalls werden die in den Ersetzungsinformationen gegebenen Aktionen ausgeführt. Diese bestehen aus dem Ersetzen des Befehls und dem Einfügen der gespeicherten Werte in die Registerfelder. Die bislang besprochenen Peepholeregeln benötigen maximal je drei Operationen für Erkennung bzw. Ersetzung. Je eine dieser Operationen ist für die Bearbeitung des Opcodes selbst, die beiden anderen sind zum Vergleich bzw. der Speicherung.

Das Blockschaltbild einer universellen Opcodezelle mit Eingängen für variable Erkennungs- und Ersetzungsaktionen ist in Abb. 7.7 gezeigt. Die wichtigsten auszuführenden Operationen sind dabei schwarz hinterlegt.

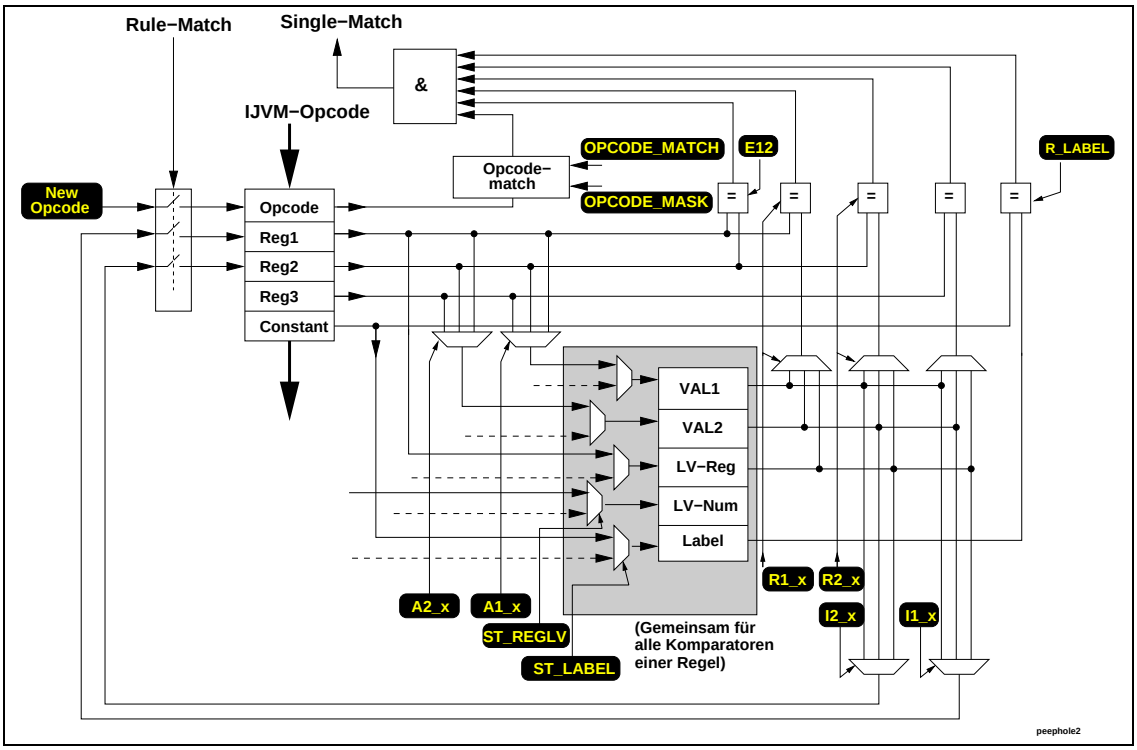

Abbildung 7.7: Grundaufbau einer universellen Opcodezelle

Eine Implementierung dieser universellen Opcodezelle mit Eingängen für variable Erkennungs- und Ersetzungsaktionen in einem Spartan2-FPGA ergab einen Verbrauch von ca. 225 LUTs und 125 Flipflops. Darauf basierend lässt sich also mit zusätzlichem Speicher für die Aktionen eine einfache serielle Optimierung realisieren, wie sie in Abb 7.8 skizziert ist. Dazu sind pro zu vergleichendem Befehl ca. 50Bit abzulegen, der gesamte JIFFY-Regelsatz mit zur Zeit ca. 40 zu testenden Befehlen würde damit ca. 2000Bit (125 RAM-LUTs) verbrauchen. Nachteilig an dieser Realisierung ist die schon erwähnte geringe Geschwindigkeit. Sie

beträgt pro IJVM-Befehl 11 Takte für den ersten Regelsatz und 27 Takte für den zweiten Regelsatz.

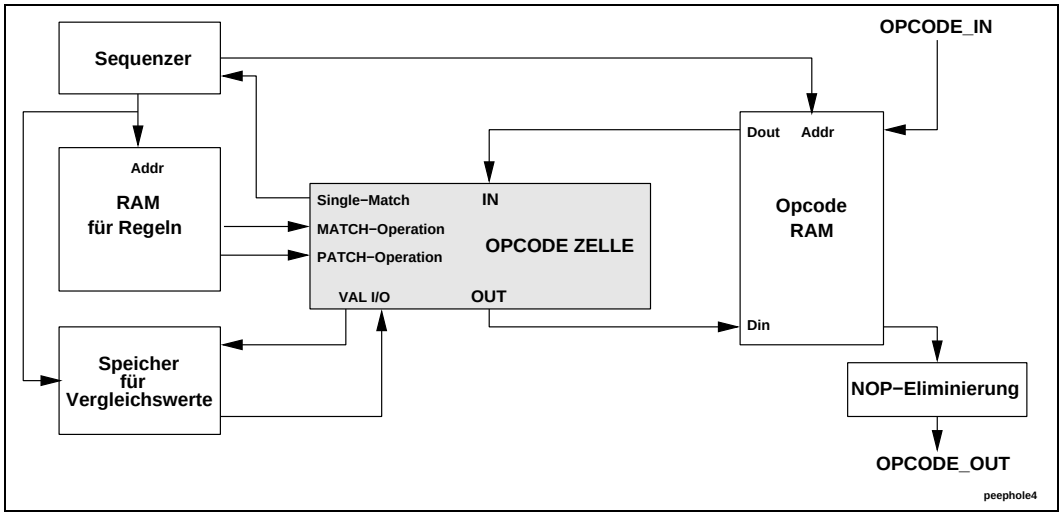

Abbildung 7.8: Serielle Überprüfung mit einer Opcodezelle

Eine Parallelisierung mit mehreren Opcodezellen ist möglich, führt aber wiederum zu einem relativ hohen Logikverbrauch. Beispielsweise wird mit 5 gleichzeitig überprüften Befehlen annähernd schon ein Viertel der Ressourcen des größten Spartan2-FPGAs (XC2S200) verbraucht.

Als Parallelisierungsvarianten sind zwei grundsätzliche Möglichkeiten denkbar, die in Abb. 7.9 dargestellt sind. Die Parallelisierung auf Befehlsebene rotiert dabei auf einem Befehlsfenster die einzelnen Regeln durch und führt bei einem Treffer die Ersetzungsregel aus. Bei der Parallelisierung auf Regelebene wird ein einziger Befehl des Fensters gleichzeitig auf Treffer der verschiedenen Regeln untersucht. Dies bedingt allerdings einen Mehraufwand durch die explizite und adressierbare Speicherung des Befehlsfensters und durch die Auswahl und Speicherung der Vergleichswerte. Andererseits ist mit dieser Methode ein höherer Durchsatz möglich, da bei nicht erfüllten Teilregeln für alle Regeln die weitere Überprüfung abgebrochen werden kann. Für das JIFFY-System kommt noch hinzu, dass viele Regeln auch nicht das ganze Fenster von 5 Befehlen benötigen und somit schneller zu bearbeiten sind. Insgesamt sind damit für die Erkennung im ersten Regelsatz maximal 3 Takte, für den den zweiten Regelsatz 10 Takte pro Befehl erreichbar. Eine durchgeführte Ersetzungsaktion führt zu einem weiteren Takt, da ansonsten die kritischen Pfade von der Erkennung zum Einfügen zu lang werden und die gesetzte Marke von 50MHz nicht erreicht wird.

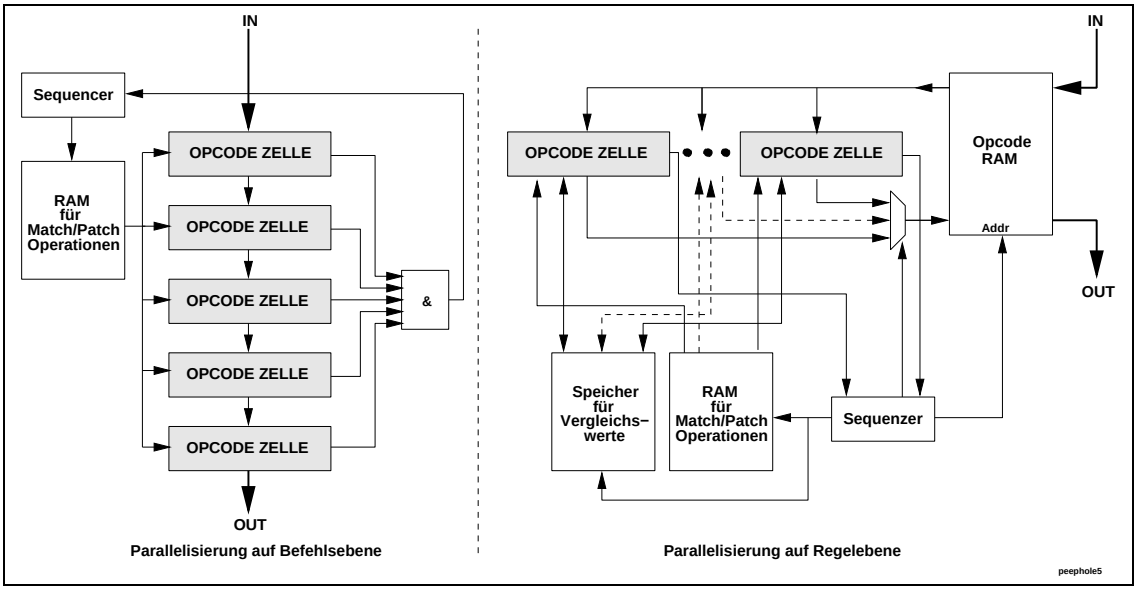

Abbildung 7.9: Parallelisierungsvarianten

7.4.2.2 Effiziente Implementierung

Insgesamt ist der Logikverbrauch als auch die Leistung der verbesserten Versionen immer noch unbefriedigend, während die freie Programmierbarkeit der Regeln durch die Speicherung im RAM ein Vorteil ist. Um den Logikverbrauch weiter zu verringern, muss dieser Vorteil aufgegeben werden. Damit wird es möglich, schon während der VHDL-Synthese die in einer Opcodezelle für eine Regel nicht erforderlichen Datenpfade zur Erkennung wegzulassen.

Aus diesem Ansatz heraus wurde eine sehr effiziente Implementierung entwickelt, deren Grundaufbau in Abb. 7.10 aufgezeigt ist. Im Unterschied zu den vorhergehenden Realisierungen wurde hier die Erkennung und das Einfügen aufgeteilt. Dies ermöglicht eine sehr flexible Realisierung der Regeln.

Der Datenpfad der IJVM-Befehle verläuft über 5 universelle Patchzellen, die die Befehle entweder unverändert weitergeben, oder Ersetzungsaktionen durchführen. Die dazu benötigten Befehle werden aus einem RAM ausgelesen, das mit der gerade zutreffenden Regelnummer adressiert wird. Die für eine Patchzelle möglichen Operationen bestehen aus einer Opcodeersetzung und zwei Befehlskanälen zur Speicherung bzw. Einfügung von Werten. Die Befehle dafür sind in Tabelle 7.3 aufgeführt.

Die 5 Opcodes der Patchzellen werden zu einem Bus zusammengefasst und führen zu den einzelnen Regelerkennungszellen (MATCH CELL), deren Anzahl von der Anzahl der Regeln abhängt. Diese Zellen bestehen intern wiederum aus Modulen (CHECK CELL), die einen einzelnen Befehl einer Regel überprüfen

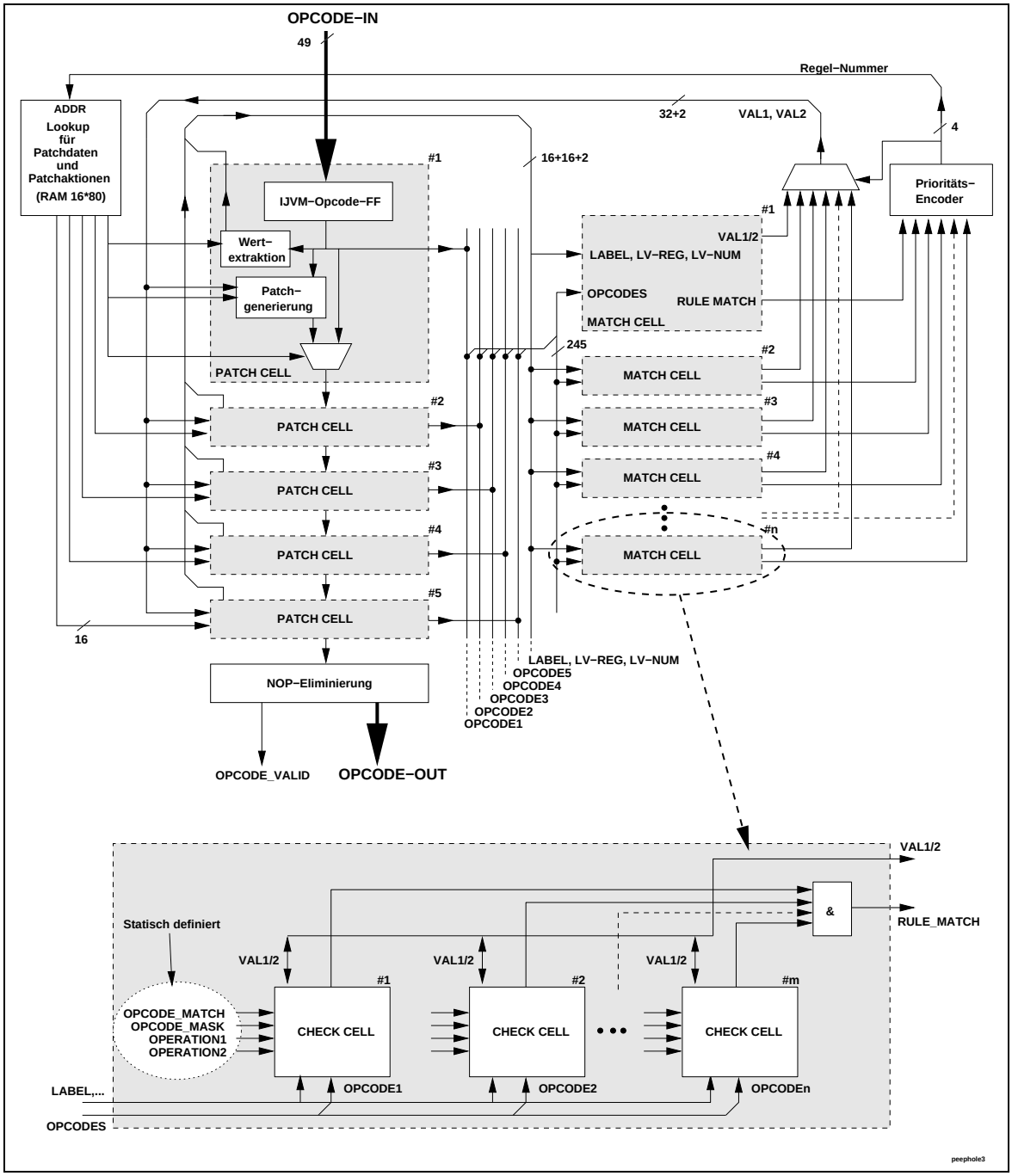

Abbildung 7.10: Effiziente Peepholeimplementierung

und innerhalb einer MATCH CELL die Datenabhängigkeiten (VAL1/2) über Befehle erzeugen und speichern. Die dafür definierten Befehle sind in Tabelle 7.4 aufgelistet.

ACT_NOP	Keine Veränderung
I1_1	Einfügen des Wertes 1 in Register 1
I1_2	Einfügen des Wertes 1 in Register 2
I2_1	Einfügen des Wertes 2 in Register 1
I2_2	Einfügen des Wertes 2 in Register 2
ST_REGLV	Speichern der Konstante in LVNUM und Register 1 in LVREG
ST_LABEL	Speichern der Konstante in LABEL
X0LABEL	Speichern von 0xffff in LABEL

Tabelle 7.3: Befehle einer Patchzelle

OP_NOP	Kein Vergleich, keine Zuweisung
A1_1	Zuweisung von Register 1 an Wert 1
A1_2	Zuweisung von Register 2 an Wert 1
AC_1	Zuweisung von Konstante an Wert 1
A2_1	Zuweisung von Register 1 an Wert 2
A2_2	Zuweisung von Register 2 an Wert 2
R1_1	Vergleich von Register 1 mit Wert 1
R1_1	Vergleich von Register 2 mit Wert 1
R1_C	Vergleich von Konstante mit Wert 1
R2_1	Vergleich von Register 1 mit Wert 2
R2_2	Vergleich von Register 2 mit Wert 2
E12	Vergleich von Register 1 mit Register 2
E1_1	Vergleich von Register 1 mit 1
R_REGLV	Vergleich von Register 1 und Konstante mit LVREG und LVNUM
R_LABEL	Vergleich von Konstante mit LABEL

Tabelle 7.4: Befehle einer Erkennungszelle

Die Ergebnisse der einzelnen Regelerkennungen führen auf einen Prioritätsenkoder, der die erste erkannte Regel kodiert und damit auch die Weitergabe der dazugehörigen Erkennungswerte VAL1/2 an die Patchzellen steuert.

Da fast 40 universelle Befehlserkennungen in einem FPGA nicht möglich sind, werden innerhalb einer Regelerkennungszelle die zu erkennenden Muster statisch auf die Zellen gegeben. Dies führt während der Synthese und der Logikoptimierung zu einer extremen Vereinfachung der nötigen Strukturen. Beispielsweise können Befehlsvergleiche in zwei FPGA-Logikzellen abgearbeitet werden und wie auch andere Tests innerhalb eines Befehls regelübergreifend wiederverwendet werden (Common Subexpression Elimination, CSE).

Die so gestaltete Realisierung der Peepholeoptimierung mit den insgesamt 14 Regeln verbraucht somit nur noch ca. 600 LUTs und ca. 550 Flipflops, ist jetzt aber von den zu realisierenden Regeln abhängig. Die Erkennung und Ausführung

aller Ersetzungsregeln benötigt im Prinzip nur noch einen Takt pro Befehl, allerdings begrenzt der kritische Pfad vom Opcode-Ausgang der Patchzellen über die Erkennung, das Patch-RAM und den Eingang der Patchzellen die Geschwindigkeit. Um dieselbe Taktgeschwindigkeit wie für die anderen Module zu erreichen, wurde eine Zwischenstufe eingebaut, die somit zu einer festen Dauer von zwei Takten pro Befehl führt. Diese konstante Eingaberate der Befehle vereinfacht zusätzlich die Anbindung an das Speicherinterface. Werden die erste und dritte Ausführung der Optimierung in den Datenpfad der JVM→IJVM bzw. IJVM→NAT-Übersetzung gelegt und damit "versteckt", sind mit der zweiten Optimierungsphase bei 50MHz also konstant 25 Millionen IJVM-Befehle pro Sekunde am Eingang verarbeitbar.

Die Auslegung der Ersetzungsregeln in einem RAM erlaubt die dynamische Änderung dieser Regeln. Da die Erkennungsregeln allerdings statisch integriert sind, ist diese Anpassungsmöglichkeit nur in Ausnahmefällen, wie z.B. für eine vorübergehende Deaktivierung einzelner Regeln, nutzbar.

7.4.3 Implementierung des Paracode-Assemblers

Die Übersetzung der IJVM-Befehlen über die Befehlsschablonen (Paracode) in Maschinencode muss über eine frei programmierbare Codeerzeugung verfügen und erlaubt damit keine Optimierung der internen Strukturen bereits bei der Synthese.

Die Realisierung des Paracode-Assemblers lehnt sich daher stark an die in Abschnitt 6.5.8 besprochene Blockstruktur an und beruht hauptsächlich aus programmierbarem Speicher, der in "distributed RAM"-Elementen im FPGA vorliegt. Für die Implementierung wurde allerdings auf drei gleichzeitig arbeitende Patchblöcke verzichtet und dafür eine sequentielle Abarbeitung der Patchbefehle vorgenommen. Andernfalls würden zuviel FPGA-Ressourcen verbraucht.

Ein Aspekt, der in der C-Modellierung relativ unaufwendig erscheint, führt allerdings zu größeren Problemen: Dies ist die Einfügung der aus dem Patch erzeugten Bitmuster mit maximal 32Bit an beliebiger Stelle in die Befehlsschablone und im Endergebnis in den Speicher. Während im Softwaremodell dazu eine einfacher Zeigerzugriff verwendet werden kann, ist dies in HW so nicht einfach möglich. Ein direktes Einfügen in den Befehlsdatenstrom ist ebensowenig möglich, da nicht garantiert ist, dass die Patches nur auf nicht-überlappenden Wörtern arbeiten. Somit muss für den gesamten Patchvorgang auf einem internen Speicher gearbeitet werden.

Ein relativ effizienter Weg, der schließlich für die Realisierung verwendet wurde, ist in Abb. 7.11 dargestellt. Hauptbestandteil ist ein explizit paralleler Speicher für 12 Byte, der die Paracodeschablone und die Veränderungen darin aufnimmt. Basierend auf dem durch den aktuellen Patch definierten Start, werden von ei-

nem Multiplexer bzw. Barrelshifter 4 aufeinanderfolgende Bytes selektiert und im Patchblock entsprechend der Patchregeln verändert. Diese 32Bit werden über einen Demultiplexer wieder an dieselbe Stelle zurückgeschrieben. Sind alle Änderungen durchgeführt, werden die Daten über den Multiplexer auf das Ausgangs-FIFO geschrieben.

Für CISC-Prozessoren mit Byte-Befehlen ist eine Umsetzung auf den Bytebefehlsstrom notwendig. Dazu werden die Befehle über den Multiplexer passend zum Wert des aktuellen PC verschoben und nur die entsprechenden Bytes im FIFO gesetzt.

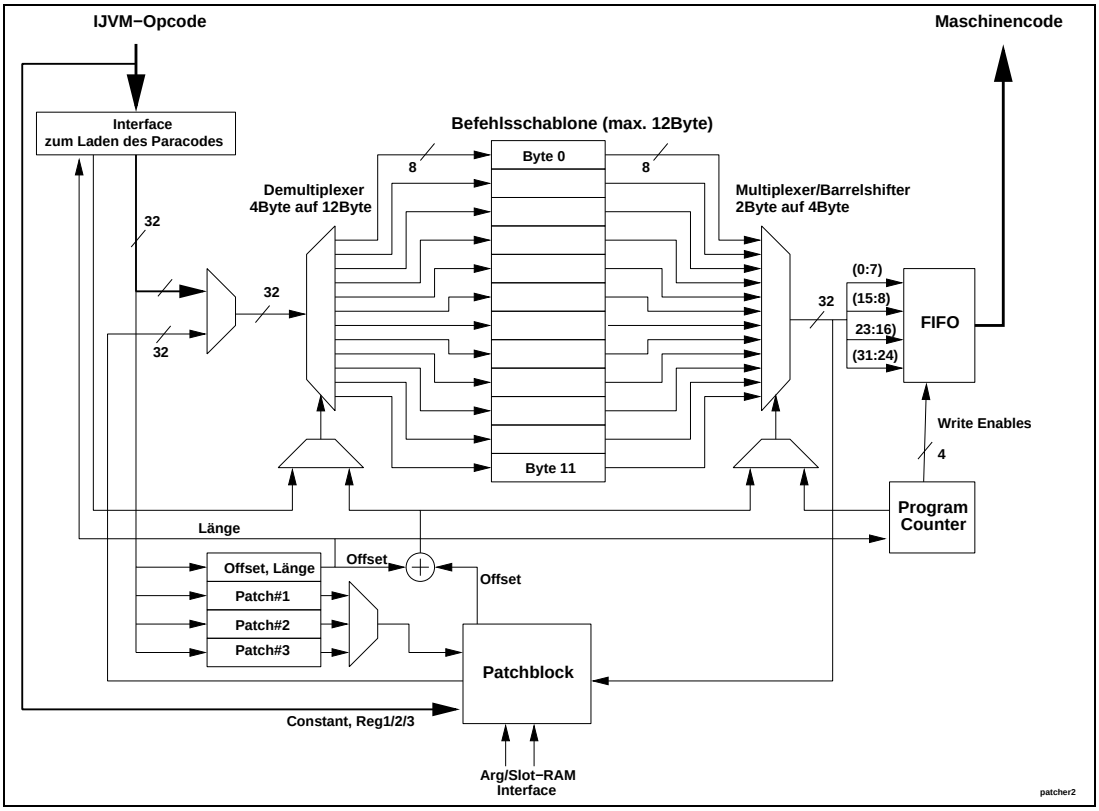

Abbildung 7.11: Realisierung der Paracode-Assemblers

7.5 Werkzeuge zur Erstellung der Laufzeitdaten

Um komfortabel die verschiedenen Laufzeitdateien für die einzelnen Übersetzungs- und Optimierungsstufen zu erzeugen, wurde eine Werkzeugumgebung geschaffen, mit der alle wichtigen Aspekte der Übersetzung in

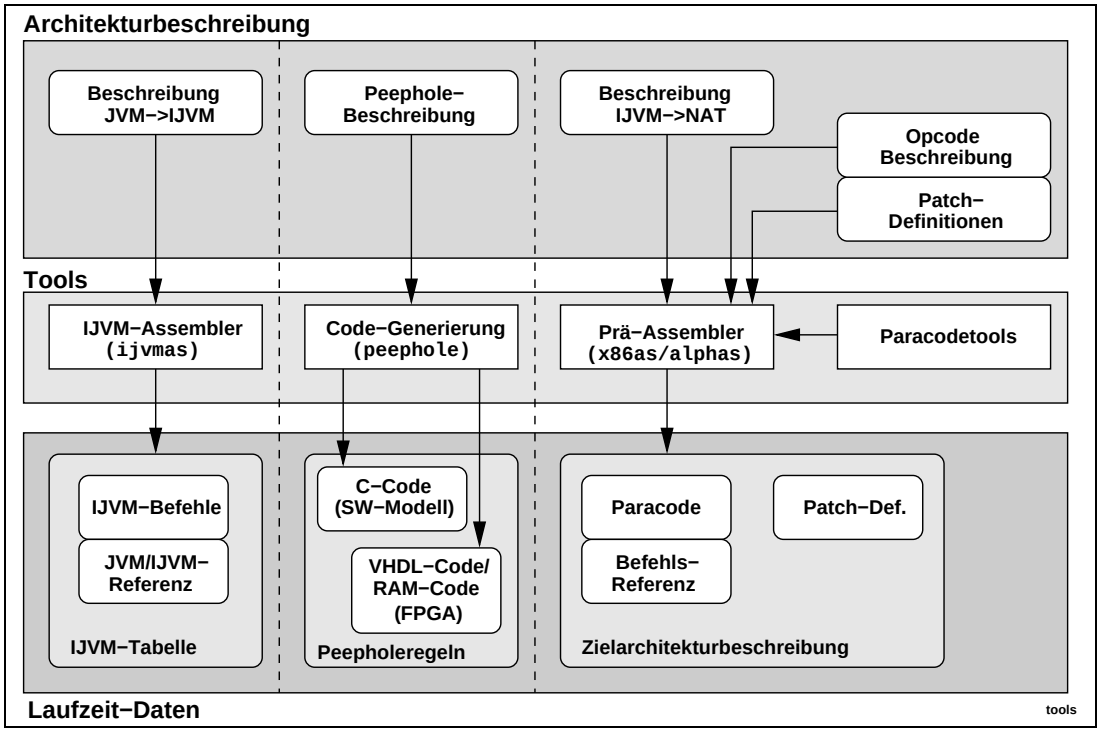

Abbildung 7.12: Datenfluss und Werkzeuge für die Laufzeitdateien

einfacher textueller Form beschreibbar sind. Die Hauptbestandteile der Umgebung sind in Abb. 7.12 gezeigt.

Im einzelnen sind diese Werkzeuge:

- Generierung der JVM→IJVM-Tabelle (IJVM-Assembler)

  Ausgangsdatei ist eine Beschreibung der JVM-Befehle mit IJVM-Befehlen. Durch entsprechende Platzhalter werden die Felder im XIJVM-Format definiert, die bei der Übersetzung die Übernahme von JVM-Parametern in den IJVM-Code auslösen. Das Ergebnis dieser Prä-Coompilierung wird zusammen mit der Referenztabelle als Binardatei abgespeichert, die den Start der XIJVM-Befehle für jeden JVM-Bytecode enthält.

- Generierung der Peephole-Regeln

  Aus der in Abschnitt 7.2 beschriebenen Regelsyntax werden zwei Dateien erzeugt, die diese Regeln in das Laufzeitsystem einbinden. Für das Software-Modell wird dazu ein C-Code generiert, der die Funktion über `if`-Konstrukte und Makroexpansion vornimmt. Für die Realisierung im FPGA werden die Initialisierungskonstanten für die einzelnen Match-Zellen und der RAM-Inhalt für die Patch-Einheit erzeugt. Die Initialisierungskonstan-

ten werden dabei für alle Regeln gesammelt in der obersten Hierarchiestufe als konstantes Feld übergeben. So muss bei Regeländerungen nur eine VHDL-Datei modifiziert werden.

- Generierung der IJVM→NAT-Tabelle (Paracode) mit einen Prä-Assembler (Paracode-Assembler)

  Die Erzeugung der Beschreibungsdateien für den Paracodeassembler im FPGA arbeitet mit drei Ausgangsdateien. Eine definiert die prinzipielle Umsetzung der Mnemonics der Zielarchitektur in der üblicherweise vom Hersteller spezifizierten Syntax.

  Die zweite Datei definiert die verschiedenen Patcharten, die für einzelne Ersetzungen erforderlich sind. Darin werden auch die Registerbelegungen und Konstantenkalkulationen für Variablenzugriffe festgelegt. Diese Daten werden später direkt in die internen RAMs des Patchblocks übertragen.

  Die dritte Datei erzeugt mit einem Prä-Assembler die einzelnen Zielarchitekturbefehle, die für einen IJVM-Befehl benötigt werden. Der Prä-Assembler besitzt dazu neben der Funktionalität eines “normalen” Assemblers und Linkers (für Sprünge innerhalb eines IJVM-Befehls) die Erkennungsmöglichkeit von Platzhaltern.

  Diese Platzhalter in den einzelnen Parametern der Befehle werden mithilfe eines Assembler-unabhängigen Toolkits erkannt und in die entsprechenden Patchbeschreibungen umgewandelt. Mit dem Toolkit können auch wesentliche Teile des Assemblers (z.B. Befehlserkennung, Paracodezusammenstellung und Linker) wiederverwendet werden.

7.6 Zusammenfassung

Die Implementierung des Software-Modells zeigte die wesentlichen Probleme, auf die eine “lineare” Übersetzung in Hardware stößt. An sich einfache, aber nicht vorhersagbare Zugriffe auf Methodeneigenschaften würden bei einer Realisierung in Hardware die erreichbare Übersetzungsleistung wesentlich verschlechtern und den Hardwareaufwand in die Höhe treiben. Um diese Auswirkungen zu vermeiden, sind fast alle dieser Zugriffe in die Laufzeitausführung verlagert, und führen hier ebenfalls zu Leistungsverlusten. Diese sind aber, wie im folgenden Kapitel gezeigt wird, relativ überschaubar.

Die Peepholeoptimierung kann in einem FPGA relativ flexibel implementiert werden. Allerdings ist die Variante mit fest integrierten Erkennungsregeln wesentlich effizienter zu realisieren, und verbraucht nur wenige FPGA-Ressourcen.

Durch die vorgestellte Werkzeugumgebung sind Änderungen bzw. Ergänzungen in allen wichtigen Übersetzungsphasen einfach durchzuführen und ermöglichen eine weitgehend softwarebasierte Entwicklung des JIFFY-Systems.

8

Resultate

30. Wir möchten nicht gern gleich von Anfang unsere Leser durch irgend eine Paradoxie scheu machen, wir können uns aber doch nicht enthalten zu behaupten, daß sich durch Erfahrungen und Versuche eigentlich nichts beweisen läßt.
Johann Wolfgang v. Goethe, Zur Farbenlehre, Polemischer Theil

8.1 Universalität des Paracodeansatzes

Um die Universalität des Paracodekonzepts für CISC und RISC zu testen, wurden die beschreibenden Tabellen für die 80586-Architektur und für die Alpha-Architektur entwickelt.

8.1.1 Paracode für 80586 (CISC)

Der 80586 [77] besitzt durch die CISC-Eigenschaft sehr unregelmäßig "geformte" Befehle, deren Befehlslängen von einem bis 11Byte reichen können. Die in die Befehle einzustanzenden Teile beschränken sich im wesentlichen auf vier Registerfelder (direkt im Befehlsbyte, zwei im ModRM-Wort und ein weiteres im SIB-Wort) und auf verschiedene Arten der Konstantendarstellung (8/32Bit Immediate/Displacement). Insgesamt werden 27 Patchtypen benötigt, um auch die 32/64Bit Registeradressierung zu implementieren.

Zur Umsetzung der textuellen Paracodeschablonen in den eigentlichen Paracode wurde ein Paracode-Assembler (x86as) entwickelt. Aufgrund der komplizierten Befehlsstruktur und der vielfältigen Adressierungsarten des x86 war diese Arbeit zwar etwas aufwendiger, es entstand jedoch als Nebenprodukt das bereits beschriebene Toolkit, mit dem weitere RISC und CISC-CPUs einfacher umgesetzt werden können.

Die Realisierung der IJVM-Befehle ergab unter anderem eine genaue Feststellung des Aufwands von Methodenaufrufen. Beispielsweise werden für einen

statischen Methodenaufruf 5 Befehle benötigt, virtuelle Methoden brauchen mit dem zweistufigen Aufrufkonzept (siehe 7.1.4.2) 7+7 Befehle.

Von den insgesamt 500 erzeugten x86-Befehlen benötigen 250 keinen Patch, weitere 135 einen, 100 zwei und ca. 15 drei Patches.

8.1.1.1 Nutzung des FPU-Stacks des 80586

Die 80586-Architektur besitzt zur FPU-Verarbeitung einen Stack mit einer Tiefe von maximal 7 Elementen. Damit ist eine explizite Adressierung über Register wie bei Integerwerten nicht möglich, wird aber auch nicht benötigt, da sich die Stacknutzung aus der JVM-Ebene übernehmen lässt. Durch eine passende Definition der IJVM→NAT-Übersetzung und der Registernummerzuweisung in der JVM→IJVM-Tabelle wird damit der FPU-Stack automatisch genutzt.

Durch die PUSH/POP-Optimierung sind maximal 3 Werte auf dem FPU-Stack möglich, mehr Elemente werden über PUSH/POP auf dem normalen Systemstack gespeichert. Prinzipiell wäre es möglich, für Methoden mit weniger als 7 Fließkommawerten auf dem Stack PUSH/POP auf FPU-Typen ganz zu eliminieren. Da diese Werte aber damit nicht mehr auf dem normalen Parameterstack für Methodenaufrufe abgelegt werden, müsste während der Übersetzung eine Unterscheidung getroffen werden, ob die Werte für FPU-Befehle oder Methodenaufrufe verwendet werden. Dies ist mit dem einfachen JIFFY-Konzept nicht allgemein möglich, obwohl der Geschwindigkeitsgewinn durchaus bemerkbar ist: Messungen an der später beschriebenen 20x20 Matrizenmultiplikation, die in der innersten Schleife insgesamt 4 `push_d`- bzw. `pop_d`-Befehle ausführt, ergaben eine Beschleunigung um 8% ohne diese Befehle.

8.1.2 Paracode für Alpha 21164 (RISC)

Der Alpha-Prozessor 21164 [45] wird zwar nicht für eingebettete Systeme benutzt, ist aber mit seiner einfachen Struktur ein typischer RISC-Vertreter. Das Testsystem (PC164-Mainboard) besitzt zwar ein dreistufiges Cache-System, die CPU reagiert aber wegen der fehlenden Out-of-Order-Execution sehr empfindlich auf Speicherzugriffe. Das Alphasystem wurde anderen RISC-Systemen (z.B. mit ARM) für den Test vorgezogen, da es durch die vollständige Linux-Entwicklungsumgebung wesentlich einfacher zu benutzen war.

Mithilfe des Toolkits waren Paracodeassembler und die erste IJVM→NAT-Umsetzung für die orthogonale Befehlsstruktur der Alpha-Architektur in relativ kurzer Zeit durchgeführt. Es wurden nur 9 Patchtypen implementiert, da nur 2 Möglichkeiten existieren, wie Konstanten und Offsets in das Befehlswort eingefügt werden können.

Problematisch ist die Einfügung von 64Bit-Konstanten in den Code. Da dies prinzipiell nur für den “Start-of-Code”-Wert erforderlich ist und die Alpha-Calling-Convention [44] für diesen Zweck sogar die Nutzung eines Registers definiert, kann dies auf den Methodenaufruf selbst verlagert werden. Damit sind statische Methoden mit dem zusätzlichen Overhead für eine eventuelle Auflösung mit 6 Befehlen realisierbar, virtuelle Methoden mit dem zweistufigen Konzept nach 7.1.4.2 benötigen 8+10 Befehle.

Von den insgesamt 500 erzeugten Alpha-Befehlen brauchen 230 keinen Patch, weitere 120 einen, 110 zwei und ca. 35 drei Patches.

8.2 Speicherverbrauch in Hard- und Software

Für die Implementierung des JIFFY-Systems in einem eingebetteten System ist der notwendige Speicherverbrauch für die Initialisierung und während der Laufzeit relevant. Dieser Speicher gliedert sich in folgende Bereiche, die im folgenden anhand der im JIFFY-System gefundenen Werte näher erläutert werden:

1. Initialisierungscode für das FPGA
2. Größen der Übersetzungstabellen
3. Speicherverbrauch zur Laufzeit der Übersetzung
4. Speicherverbrauch durch zusätzliche JIFFY-Strukturen
5. Softwareunterstützung für das Laufzeitsystem

8.2.1 FPGA Initialisierung

Der für die FPGA-Initialisierung erforderliche Speicher lässt sich in die eigentlichen Konfigurationsdaten (sog. “Bitstream”) und die Konfigurationsroutinen auf dem Zielprozessor unterteilen.

Je nach Ausnutzung des FPGAs lässt sich der Bitstream mit ZIP oder GZIP [80] auf 20% bis 70%[1] der realen Größen komprimieren. Damit ergeben sich für ein Spartan2-200 mit einer Bitstreamgröße von 164KByte komprimierte Größen von max. 120KByte, was für die typischen Flash-ROM-Bereich der Systeme mit 8-16MByte nur einen kleinen Teil ausmacht. Der Bitstream kann direkt aus dem ROM dekomprimiert werden und benötigt daher kein RAM zur Laufzeit.

Die zur FPGA-Konfiguration benötigten Programmteile bestehen aus der Dekomprimierung (für Java JAR-Archive ohnehin notwendig) und den eigentlichen

[1] Schätzung basiert auf Erfahrungswerten.

Programmieralgorithmen. Der Aufwand dafür liegt typischerweise in Bereichen von wenigen hundert Byte und kann damit in der Betrachtung des Speicherverbrauchs vernachlässigt werden.

8.2.2 Tabellengröße und Speicherlokalisierung

Für die effiziente Speicherung der Übersetzungstabellen JVM→IJVM und IJVM→NAT ist die Größe der dabei benutzen Tabellen wichtig. Die während der JIFFY-Implementierung entwickelten Tabellen haben dabei die in Tabelle 8.2.2 aufgeführten Größen. Zusätzlich zu den realen Größen ist der Speicherbedarf der Tabelle nach den GZIP-Verfahren angegeben. Anhand des Kompressionsverhältnisses ist erkennbar, dass diese Tabellen nur spärlich gefüllt sind, und hauptsächlich Nullwerte enthalten. Somit kann schon eine einfache HW-freundliche Komprimierung (z.B. Lauflängenkomprimierung) bzw. eine nicht auf SW-Alignment ausgelegte Struktur den Tabellenumfang wesentlich vermindern.

Bereits die JVM-Lookupreferenz enthält durch die verwendete Ausrichtung auf 4Byte pro Eintrag 512Byte unnötige Nulldaten, zusätzlich ist ein ungenutzter JVM-Opcodebereich von weiteren 512Byte vorhanden. Die effektive Größe beträgt damit also nur ungefähr 1KByte, was die problemlose Implementierung in FPGA-internen Block-RAMs erlaubt. Dies gilt ebenso für die Opcodereferenztabelle mit 1.5KByte.

Die Patchtypenbeschreibung ist mit 100Byte effektiver Größe so klein, dass sie in das verteilte FPGA-RAM gelegt werden kann, was 25 Logikblöcken eines Spartan2 entspricht. Da auf diese Daten, die sich auf mehrere kleine Tabellen aufteilen, sehr schnell zugegriffen werden muss, wäre eine andere Lokalisierung (z.B. in Block-RAMs) zu ineffizient bezüglich der Geschwindigkeit und der RAM-Ausnutzung.

Sind die eigentlichen Übersetzungstabellen durch ihre Größe nicht in FPGA-internes RAM zu integrieren, ist es auf jeden Fall möglich, sie auf ein FPGA-nahes RAM auszulagern. Vor der Initialisierung des gesamten JIFFY-Systems können diese Tabellen z.B. im GZIP-Format mit einer Gesamtgröße von unter 5KByte im ROM abgelegt sein.

Interessant ist in diesem Zusammenhang, dass sich **der gesamte CPU-architekturabhängige Teil** des JIFFY-Konzepts auf nur **weniger als 3KByte** reduziert. Erlaubt die JIFFY-FPGA-Implementierung ein Umladen der Tabellen während der Laufzeit, sind mit minimalem Zusatzaufwand beliebige CPU-Architekturen unterstützbar und somit auch der Einsatz in einem eingebetteten System mit verschiedenen CPUs.

JVM→IJVM			
	ungepackt/Worst Case	gzipped	effektiv *)
JVM-Lookup	2048Byte	ca. 550Byte	ca. 1000Byte
Befehlstabelle	16384Bytes	1200Byte	ca. 5600Byte
Paracode x86/Alpha			
	ungepackt/Worst Case	gzipped	effektiv *)
Patchtypen	288Byte	ca. 60Byte	ca. 100Byte
Opcodereferenz	ca. 8000Byte	ca. 400Byte	ca. 1500Byte
Opcodetemplates	49192Byte	ca. 2300Byte	ca. 4KByte

Anmerkung: *) effektiv = ohne Nullbytes bzw. nur mit 16Bit-Werten, wo möglich

Tabelle 8.1: Größe der Übersetzungstabellen

8.2.3 Speicherverbrauch zur Laufzeit der Übersetzung

Der Speicherverbrauch zur Laufzeit der Übersetzung einer Methode besteht aus folgenden Teilen:

1. Sprungtabelle (Branchtable)

   Der Speicherverbrauch beträgt, wie in Abschnitt 6.5.2 gezeigt, maximal 16KByte. Dieser Speicher darf FPGA-nah oder auch im Systemspeicher liegen, da die Analysephase Schreibzugriffe darauf puffern kann und der Inhalt zu diesem Zeitpunkt nicht unmittelbar benötigt wird. Das Auslesen während der JVM→IJVM-Übersetzung kann über sequentielles Streaming erfolgen.

2. Temporäre Label- und Usage-Tabelle für Linker und Verifier

   Alle bislang compilierten Klassen hatten signifikant weniger als 1024 Sprünge (siehe 6.5.9), damit ist eine Gesamtgröße von max. 12KByte ausreichend. Dieser Speicherbereich sollte im FPGA oder FPGA-nah ausgeführt sein. Der Bereich kann auch mit der Sprungtabelle identisch sein, da beide Tabellen nie gleichzeitig benutzt werden.

3. Temporärer Zwischenspeicher für IJVM-Code

   Je nach Befehlsstruktur und Optimierungsmöglichkeiten schwankt die benötigte Speichergröße für die IJVM-Befehle. Typischerweise beträgt das Verhältnis der Bytes zwischen IJVM und JVM-Code vor der Optimierung 5.5 bis 6.5, nach der Optimierung liegt es zwischen 4.5 und 5.8. Diese starke Vergrößerung liegt zum Hauptteil im festen IJVM-Befehlsformat begründet. Allein 4 Bytes des Befehls werden für die Konstante verbraucht, selbst

wenn diese nicht erforderlich. Da Konstanten nur bei weniger als der Hälfte des IJVM-Ausgangscodes wirklich benötigt werden, und selbst dann meist nur mit 16Bit-Werten, würde eine komprimierte Variante des IJVM-Codes mit einem optionalen Konstantenfeld diesen Faktor auf 2.5 bis 3.5 reduzieren.

Die reine IJVM-Befehlsanzahl entspricht vor der Optimierung ungefähr der Bytecodelänge der Methode, nach der Optimierung liegt sie bei 60 bis 90% dieses Wertes, größere Methoden erreichen niedrigere Werte.

Im (extrem unwahrscheinlichen) Worst-Case-Fall einer JVM-Methode mit 64KByte Länge wird also ein temporärer Speicher von 400KByte benutzt, mit einfacher IJVM-Komprimierung 200-250KByte. Da die Optimierung linear verläuft und den Code nicht verlängern kann, ist sie "in-place" durchführbar und benötigt keinen zweiten Speicherbereich.

4. Speicher für den übersetzten Maschinencode

   Der für den nativen Maschinencode benötigte Speicher ist ebenfalls stark von den übersetzten Methoden abhängig. Eigene Messungen anhand des durch JIFFY erzeugten Assemblercodes ergaben für die x86-Architektur einen Faktor von 3 bis 6.5 gegenüber der Größe des Bytecodes.

   Ein nicht unbeträchtlicher Anteil wird dabei durch den Overhead zum Aufruf von Methoden verbraucht. Ist dies nicht erwünscht, kann durch den Einsatz von weiteren methodenspezifischen Stubs der Speicherverbrauch um 5 bis 10Bytes pro Methodenaufruf gedrückt werden. Diese Lösung hat allerdings den Nachteil eines etwas erhöhten Verwaltungsaufwands in der Software beim Compilieren einer Methode.

Der Speicher, der temporär durch den Übersetzungsvorgang benötigt wird, beträgt damit ungefähr 16KByte im FPGA-nahen RAM und ohne weitere Kompression maximal 400KByte im Systemspeicher. Dieser Wert ist mit dem Einsatz einer einfachen befehlsabhängigen Kompression auf 200-250KByte zu verkleinern.

8.2.4 Speicherverbrauch durch JIFFY-Strukturen

Zusätzlich zu den von der JVM benutzten Strukturen (z.B. Konstantenpool), die JIFFY mitbenutzt, erfordert das System weitere Strukturen zur Laufzeit. Diese sind im wesentlichen:

1. Stubs

   Es müssen Stubs für den Aufruf von virtuellen Methoden und die Übergänge zwischen dem Laufzeitsystem und JIFFY erstellt werden (siehe 7.1.6).

Stub-Typ	x86	Alpha
Stub für JIFFY-Resolving	29	52
Stub JDK→JIFFY [1)]	24+6*n	20+16*n
Stub für virtuelle Methoden [2)]	30	40
Stub JIFFY→JNI [1)]	54+6*n	72+16*n
Hilfsroutinen [3)]	62	80

Anmerkungen: [1)] Pro Methode, Methoden mit identischer Signatur teilen sich den Stub, n ist die Parameteranzahl.
[2)] Pro Methode, Methoden mit identischer realer Parameteranzahl teilen sich den Stub.
[3)] Hilfroutinen für `tableswitch` und `lookupswitch`.

Tabelle 8.2: Speicherplatz der Stub-Methoden

Aus Portabilitätsgründen werden diese erst zur Laufzeit des Systems aus IJVM-Befehlen mit dem Paracodeassembler in Maschinencode umgesetzt. Der Speicherplatz für die einzelnen Stubs ist in Tabelle 8.2 beschrieben. Der etwas höhere Speicherbedarf des Alphasystems gegenüber dem x86 entsteht neben dem allgemeinen Zuwachs bei RISC hauptsächlich durch die effiziente Nutzung der erweiterten x86-Adressierungsarten für Tabellenzugriffe.

In einem laufenden System werden die Stubs hauptsächlich für virtuelle Methoden und das Aufrufen von JNI-Funktionen benutzt. Die genaue Länge der benötigten Stubs hängt von der Anzahl und Art der aufgerufenen virtuellen Methoden und JNI-Routinen ab und ist damit von der Anwendung abhängig. Da die JVM-Typen intern auf 3 Datentypen abgebildet werden (32Bit, 64Bit und Pointer), sind viele Stubs wiederverwendbar und reduzieren somit die insgesamt benötigte Anzahl weiter.

2. Klassenglobale JIFFY-Objektinformationen

   Diese Informationen benötigen auf der x86-CPU insgesamt 20Byte pro Konstantenpooleintrag einer Klasse. Der Speicherbedarf könnte über eine weitere Indirektion und dynamische Strukturgrößen verringert werden, was allerdings auf Kosten der Geschwindigkeit gehen würde. Eine “Clean-Room”-Implementierung einer Java-VM wäre damit möglich, allerdings ist der Speicherplatz für Konstanten- und Stringeinträge im CP noch nicht berücksichtigt.

   Wird JIFFY als JDK-Plugin benutzt, kommt zusätzlich der Speicherverbrauch der JDK-Strukturen hinzu, der aufgrund dynamischer Strukturen nicht einfach abschätzbar ist.

8.2.5 Softwareunterstützung für das Laufzeitsystem

Die prototypische Implementierung der JIFFY-Übersetzung als C-Modell für das JDK 1.2-Plugin belegt für eine 80586-CPU ungefähr 100KByte Programmcode (ohne Symbole und Debugginginformationen).

Die ungefähre Codelänge der einzelnen Programmkomponenten für ein x86-System wird in Tabelle 8.2.5 gezeigt.

Wenn auch noch nicht alle Anforderungen für die JDK-Einbindung erfüllt sind, ist doch abzusehen, dass der reine Zusatzaufwand für das JIFFY-System weit unterhalb von 100KByte liegen wird und damit auf jeden Fall nicht wesentlich in den Verbrauch des gesamten Java-Laufzeitsystems eingeht. Wird JIFFY statt als JDK-Plugin mit einem eigenen Laufzeitsystem eingesetzt, ist aufgrund der in der Tabelle aufgeführten Programmteile JVM- und JIT-Funktionen eine Größe von 40-50KByte realistisch.

JIFFY:	
JDK-Einbindung	30KByte
JIFFY-interne JVM-Funktionen	20KByte
HW-unabhängige JIT-Funktionen	20KByte
JIT C-Modell	20KByte
Zum Vergleich:	
sunwjit (nur JIT)	160KByte
Hotspot (VM mit JIT)	4.7Mbyte

Tabelle 8.3: Speicheraufteilung von JIFFY (x86) im Vergleich

8.3 Messungen der Leistung

8.3.1 Java-Laufzeitmessungen und Vergleiche

Um die Codequalität des Übersetzungskonzepts zu überprüfen, wurden mehrere Benchmarks in der JDK1.2/1.3-Umgebung durchgeführt, wobei JIFFY als JIT-Plugin benutzt wurde. Als Zielarchitekturen wurden Systeme aus der 80586-Familie und der Alpha 21164 ausgewählt.

Zum Vergleich der Geschwindigkeit wurden folgende Java-Ausführungsplattformen für die 80586-Tests ausgewählt:

- JDK 1.2 mit JVM-Interpreter (Linux)
- JDK 1.2 mit TYA 1.3 (Linux)

- JDK 1.2 mit sunwjit (Linux)
- JDK 1.2 mit JIFFY-Plugin (Linux)
- JDK 1.3 mit Hotspot-Compiler (Client-Version, Linux)
- Kaffe V1.0b4 JIT (Linux)
- Symantec-JIT für Netscape-Communicator 4.51 (Windows)
- JIT-Compiler im Microsoft >Internet Explorer 4.0 (Windows)
- GCJ 2.95.3 mit Option -O3
- GCC 2.95.3 mit Option -O3 (für in C "übersetzte" Java-Programme)

Die Unterschiede zwischen den Interpretern in JDK 1.2 und JDK 1.3 sind kleiner als 5%, daher werden der Ergebnisse mit dem Interpreter von JDK 1.3 nicht explizit aufgeführt. Die JITs von Symantec und Microsoft wurden getestet, da sie typische Laufzeitumgebungen für Browser darstellten, allerdings werden sie inzwischen kaum mehr benutzt.

Für Tests auf der Alpha-Plattform wurden folgende Vergleichsplattformen herangezogen:

- JDK 1.2 mit JVM-Interpreter (Linux)
- JDK 1.2 mit JIFFY-Plugin (Linux)
- JDK 1.3 mit Compaq-JIT (Client-Version, Linux Alpha)
- CACAO
- GCC 2.95.3

Leider war aufgrund von Bibliotheksinkompatibilitäten ein Test von gcj auf dem Alpha nicht möglich.

Da im Bereich der Anbindung von JIFFY an das JDK noch nicht alle Methoden implementiert sind, die das fehlerfreie Ablaufen "großer" Benchmarks ermöglichen, wurde zur Messung auf mehrere Mikrobenchmarks ausgewichen:

- *sieve*

  Dieser Code dient zur Primzahlenberechnung nach dem Algorithmus "Sieb des Erathostenes". Neben Schleifen und bedingten Verzweigungen werden hauptsächlich Arrayzugriffe ausgeführt. Da dieser Test auch teilweise in anderen Benchmarks auftaucht, wird die Geschwindigkeit in Iterationen pro Sekunde angegeben. Dieser Wert ist deshlab besser mit Ergebnissen anderer Benchmarks vergleichbar.

- *s.len*

  Es wird die Länge verschiedener Strings ermittelt, damit werden hauptsächlich Aufrufe virtueller Methoden und Instanzvariablenzugriffe gemessen.

- *fib*

  Die rekursive Berechnung der Fibonaccizahl für den Eingabewert 40 führt neben etwas Stackarithmetik vor allem Zugriffe auf statische Klassenmethoden aus und überprüft damit auch die Aufruf- und Rücksprungkomplexität.

- *sin*

  Dieser Benchmark ruft die statische Methode `java.lang.Math.sin()` aus der Java-Klassenbibliothek auf und testet damit den Aufruf von Methoden, die üblicherweise im Native-Format vorliegen, d.h. sie rufen über eine Hilfsfunktion eine schon im Laufzeitsystem vorhandene Funktion auf.

- *matmult*

  Die Testmethode enthält die Vorbelegung zweier 20×20-Matrizen mit Double-Werten aus dem Konstantenpool und direkten Konstantenbefehlen und anschließend eine Matrixmultiplikation mit drei verschachtelten Schleifen. Dieser Benchmark testet durch das mehrdimensionale Feld für die Matrix Zugriffe auf Arrays bzw. Arrayreferenzen und die Nutzung der Fließkommaeinheit.

Alle Benchmarks überprüfen somit einzelne Bestandteile des Java-Systems. Die Vorteile und Schwächen der einzelnen Ausführungsplattformen und der spezifischen JIFFY-Konzepte treten somit stärker hervor.

Die gewonnenen Messergebnisse (nicht alle Tests konnten auf allen Plattformen durchgeführt werden) sind in den Tabellen 8.4 bis 8.8 aufgeführt, eine grafische Zusammenfassung mit den wichtigsten Vergleichsplattformen in den Abbildungen 8.1 bis 8.5.

Die Messergebnisse auf der x86-Architektur zeigen die durchweg gute Leistung des Hotspot-Systems, die nahe an der von direkter Compilation des C-Algorithmus liegt. Im Vergleich zum Interpreter sind mit JIFFY Geschwindigkeitsgewinne zwischen 10% und 750% möglich. Dies entspricht 20 bis 95% der Geschwindigkeit der direkten C-Compilation.

TYA benutzt Methodeninlining und führt einen Großteil der Methodenauflösung bereits bei der Compilation durch. Daher ist es beim Test *s.len* etwas schneller, während es sonst (wie auch der SunJIT) wesentlich langsamer als JIFFY ist. Hier zeigt sich, dass der Geschwindigkeitsverlust durch das “echte” JIT-Resolving

in JIFFY und der dadurch auch schon bei aufgelösten Funktionen vorhandene Zusatzaufwand schwächer ausfällt, als zunächst erwartet.

Beim *sin*-Test fällt die im Vergleich zu kaffe große Ineffizienz der JNI-Einbindung im JDK1.2 auf. Könnte `sin()` ohne Umweg über den JNI-Wrapper direkt aufgerufen werden, wäre JIFFY 60% schneller. Der sunwjit-Wert, der 40% langsamer(!) als die direkte Interpretierung ist, zeigt dieses Problem besonders deutlich. Daher kann der relativ starke Rückstand gegenüber Hotspot hier nur zum Teil dem einfachen Übersetzungskonzept von JIFFY angelastet werden.

Erstaunlich gegenüber dem Hotspot-Ergebnis ist das Abschneiden von gcj, aufgrund der AOT-Compilation mit dem gesamten Optimierungsteil von gcc wären wesentlich bessere Ergebnisse zu erwarten. Besonders der *fib*-Test zeigt, dass anscheinend die Stackverwaltung extrem ineffizient ist.

Auf dem Alpha sind die Ergebnisse im Vergleich zu gcc insgesamt schlechter und erreichen nur 30 bis 45%, dafür ist der Unterschied zur Interpretierung wesentlich größer, JIFFY ist hier 70% bis 1500% schneller. Auch im Vergleich zum Compaq-JIT ist JIFFY fast immer schneller, teilweise um den Faktor 2.7.

Das schlechte Abschneiden von JIFFY gegenüber gcc oder CACAO liegt neben der einfachen Optimierung auch an den relativ ineffizienten JDK-Strukturen für Objekte, die jedesmal eine zusätzliche Indirektion für den Speicherzugriff benötigen. Dies fällt besonders bei *sieve* (Boolean-Array) und *s.len* (Instanzmethoden- und Variablen) auf. Allerdings sollte sich besonders das *sieve*-Ergebnis verbessern, wenn die beschriebene Verlagerung von lokalen Variablen in Register durchgeführt wird.

Laufzeitsystem	PII-233	Duron-900MHz	Alpha 21164
Interpreter (absolut)[1)]	85/s	231/s	43/s
Speedup			
Interpreter	1.0	1.0	1.0
TYA	3.2	–	–
SunJIT	6.1	8.6	–
Compaq-JIT	–	–	21.3
JIFFY	6.4	9.5	16.3
Hotspot	7.5	8.9	–
Kaffe	7.8	–	–
NS4.51	21.6	–	–
MS	32.0	–	–
cacao	–	–	28.4
gcj	11.0	12.3	–
gcc	32.3	28.5	61.6

[1)] Ergebnis in Iterationen pro Sekunde

Tabelle 8.4: Messergebnisse für den *sieve*-Benchmark

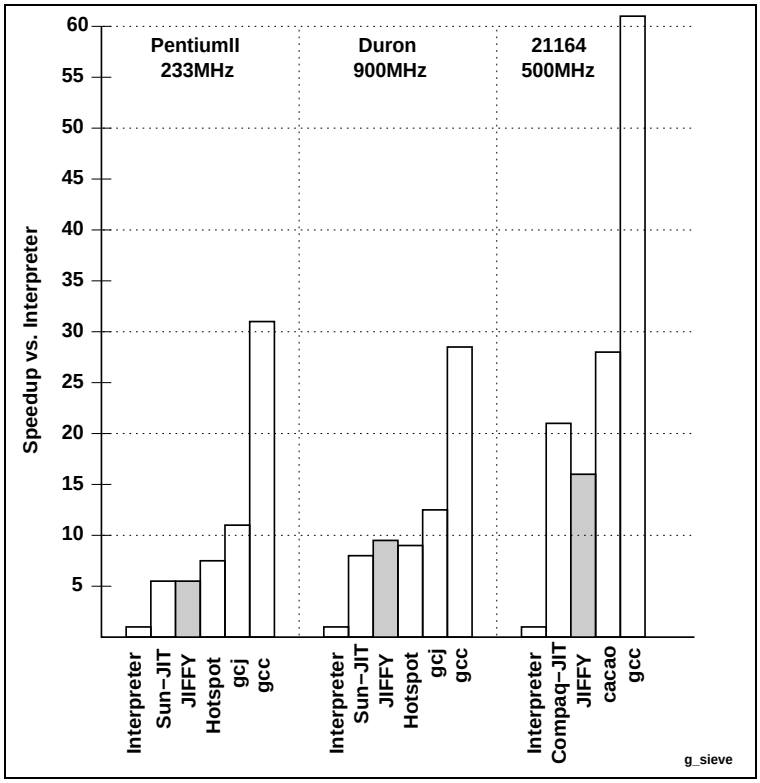

Abbildung 8.1: Zusammenfassung *sieve*-Benchmark

Laufzeitsystem	PII-233	Duron-900MHz	Alpha 21164
Interpreter (absolut)	189s	52s	300s
Speedup			
Interpreter	1.0	1.0	1.0
TYA	1.6	–	–
SunJIT	3.3	4.7	–
Compaq-JIT	–	–	6.1
JIFFY	6.3	8.5	12.0
Hotspot	8.2	10.4	–
Kaffe	1.6	–	–
NS4.51	6.2	–	–
MS	6.7	–	–
cacao	–	–	20.0
gcj	2.2	1.4	–
gcc	6.7	8.6	33.7

Tabelle 8.5: Messergebnisse für den *fib*-Benchmark

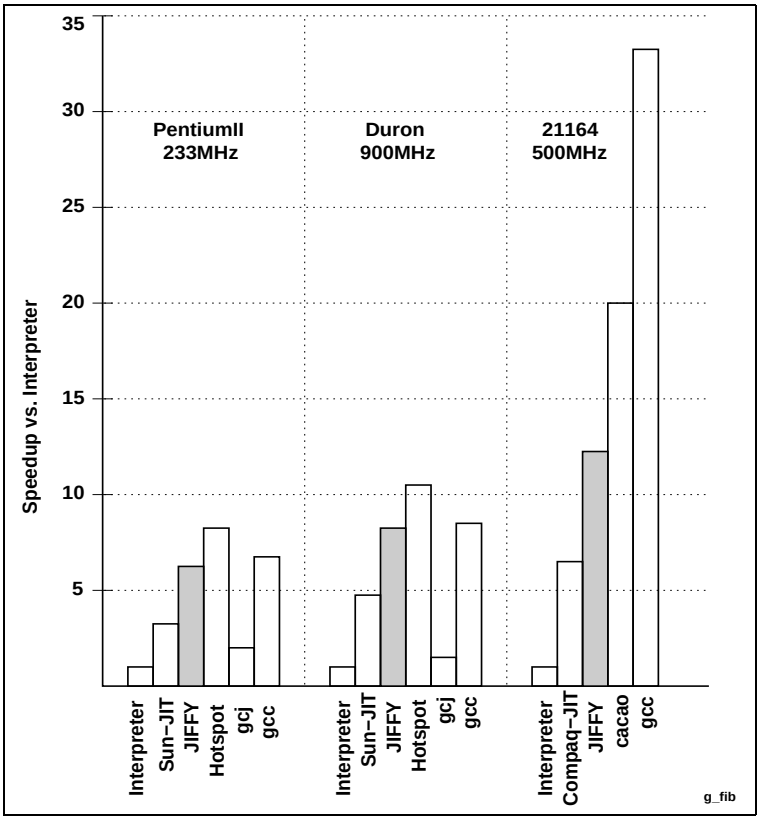

Abbildung 8.2: Zusammenfassung *fib*-Benchmark

Laufzeitsystem	PII-233	Duron-900MHz	Alpha 21164
Interpreter (absolut)	152s	42.3s	1537s
Speedup			
Interpreter	1.0	1.0	1.0
TYA	0.5	–	–
SunJIT	0.6	0.7	–
Compaq-JIT	–	–	1.4
JIFFY [1)]	0.9/1.1	0.8/1.1	–/1.7
Hotspot	3.3	3.0	–
Kaffe	2.1	–	–
NS4.51	(29.5)[2)]	–	–
MS	4.0	–	–
cacao	–	–	2.9
gcj	1.9	1.5	–
gcc	2.8	2.3	3.7

[1)] Wert ohne/mit Optimierung auf Zugriff des Execution Environments (EE)

[2)] Vermutlich Invarianteneliminierung

Tabelle 8.6: Messergebnisse für den *sin*-Benchmark

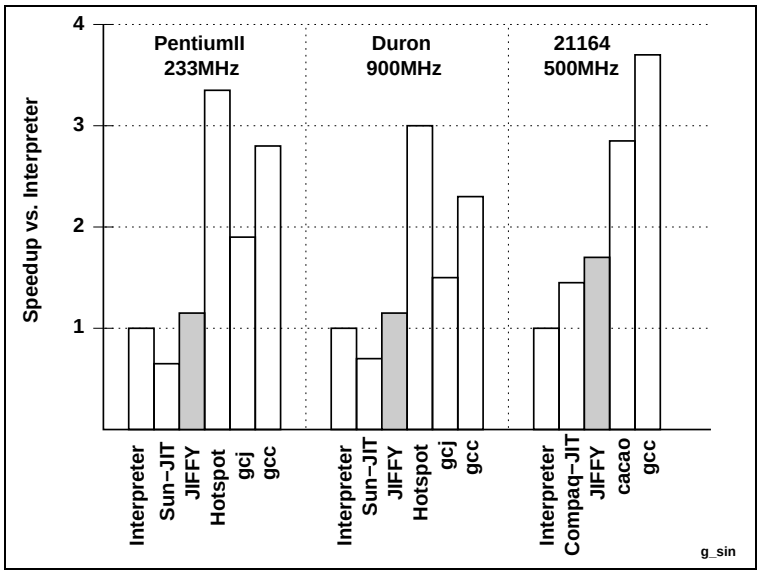

Abbildung 8.3: Zusammenfassung *sin*-Benchmark

Laufzeitsystem	PII-233	Duron-900MHz	Alpha 21164
Interpreter (absolut)	635s	173s	1105s
Speedup			
Interpreter	1.0	1.0	1.0
TYA	7.3	–	–
SunJIT	4.7	4.4	–
Compaq-JIT	–	–	2.7
JIFFY	5.3	5.2	7.4
Hotspot	23.7	22.4	–
Kaffe	2.3	–	–
NS4.51	7.3	–	–
MS	9.0	–	–
cacao	–	–	28.9
gcj	11.8	12.4	–
gcc [1)]	–	–	–

[1)] Messwerte zu stark von Stringklassen-Implementierung abhängig, Speedup lag je nach Implementierung zwischen 10 und 75.

Tabelle 8.7: Messergebnisse für den *s.len*-Benchmark

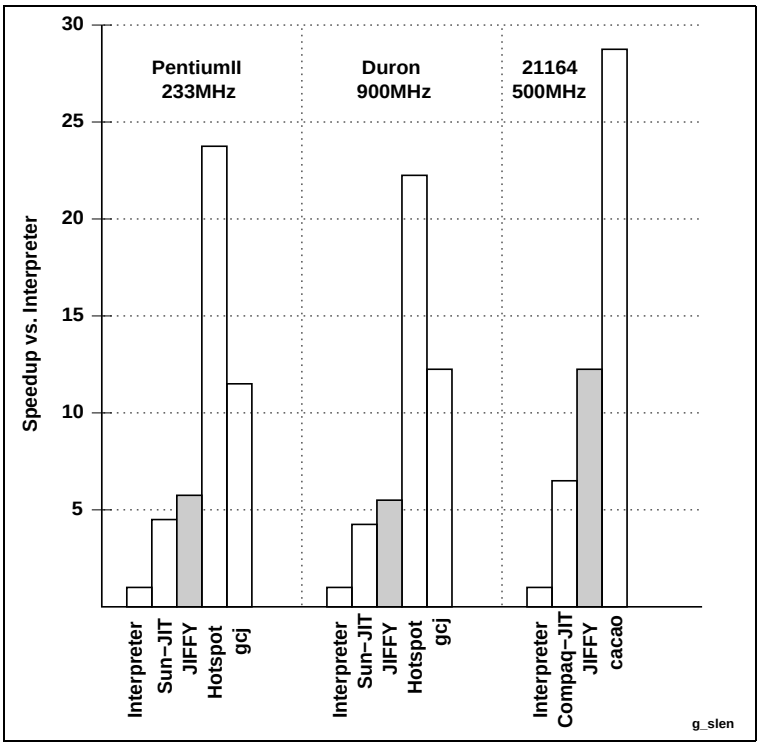

Abbildung 8.4: Zusammenfassung *s.len*-Benchmark

Laufzeitsystem	PII-233	Duron-900MHz	Alpha 21164
Interpreter (absolut)	900s	248s	1998s
Speedup			
Interpreter	1.0	1.0	1.0
SunJIT	5.0	4.4	–
Compaq-JIT	–	–	16.4
JIFFY	5.8	4.7	15.6
Hotspot	8.9	9.7	–
Kaffe	–	–	–
cacao	–	–	25.6
gcj	6.5	5.0	–
gcc	11.3	10.8	59.8

Tabelle 8.8: Messergebnisse für den *matmult*-Benchmark

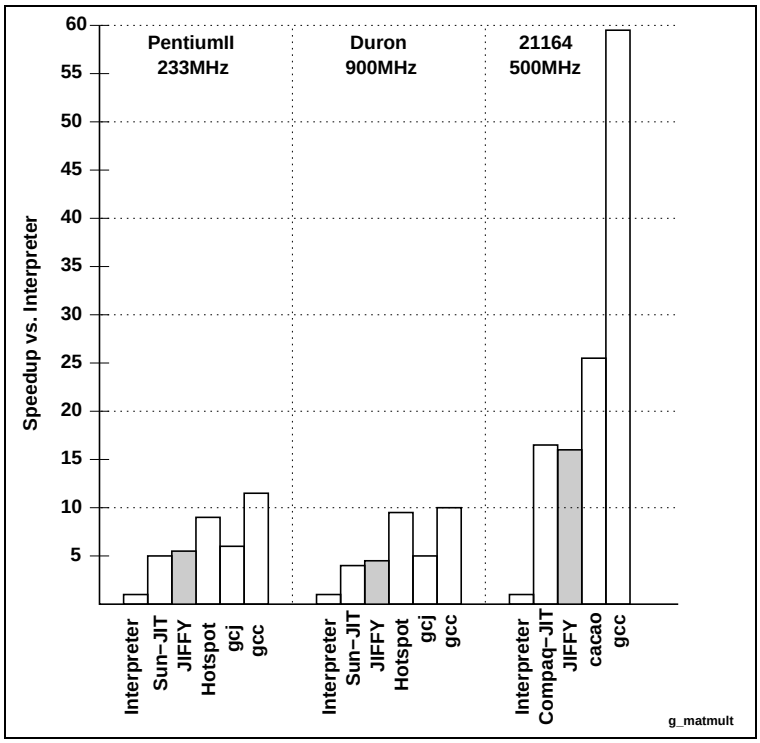

Abbildung 8.5: Zusammenfassung *matmult*-Benchmark

8.3.2 Einfluss der Optimierungen

Um die tatsächliche Effizienz der Peepholeoptimierung zu überprüfen, wurden alle Benchmarks ohne oder nur mit teilweiser Peepholeoptimierung durchgeführt. Als Optimierungsvarianten wurden folgende Varianten durchgeführt:

- Vollständiger Verzicht auf Peepholeoptimierung und triviale PUSH/POP-Eliminierung bei der JVM→IJVM-Übersetzung (NO-OPT). Damit erfolgt die Stackemulation ausschließlich über PUSH und POP.
- Nur triviale PUSH/POP-Eliminierung (OPT-PP), d.h. Eliminierung bei identischem Quell- und Zielregister. Das oberste Stackelement liegt damit meistens in einem Register.
- PUSH/POP-Eliminierung und Peepholeoptimierung mit erstem Regelsatz (OPT-1)
- PUSH/POP-Eliminierung und Peepholeoptimierung mit erstem und zweitem Regelsatzes (OPT-12)
- PUSH/POP-Eliminierung und Peepholeoptimierung mit erstem und zweimaligem Durchlauf des zweiten Regelsatzes (OPT-122)

Die Messergebnisse sind als Absolut- und Prozentwerte für die drei Testplattformen in den Tabellen 8.9 bis 8.11 und zum Vergleich grafisch in den Abbildungen 8.6 bis 8.7 aufgeführt. Die Prozentwerte wurden dabei auf die Ergebnisse mit allen Optimierungen (OPT-122) bezogen.

	sieve	*fib*	*sin*	*s.len*	*matmult*
NO-OPT	253/s, 46.3%	46.0s, 65.2%	144.6s, 93.9%	182s, 66.5%	731s, 21.2%
OPT-PP	346/s, 63.4%	41.8s, 71.8%	144.0s, 94.4%	142s, 85.3%	293s, 52.9%
OPT-1	378/s, 69.2%	38.4s, 78.1%	135.8, 100%	127s, 95.4%	211s, 73.5%
OPT-12	523/s, 95.8%	31.4s, 95.5%	135.8s, 100%	121.2s, 100%	156s, 99.3%
OPT-122	546/s, 100%	30.0s, 100%	135.8s, 100%	121.2s, 100%	155s, 100%

Tabelle 8.9: Einfluss der Peepholeoptimierung (PII/233)

Auf dem x86 erzeugen allein die trivialen Optimierungen eine Beschleunigung zwischen 1.2 und 2.5, die Auswirkungen sind auf dem einfacheren PII stärker als auf dem Duron. Mit allen Optimierungen beträgt der Beschleunigungsfaktor zwischen 1.1 und 4.7.

Auf dem Alpha sind die Einflüsse der einzelnen Optimierung bei allen Tests insgesamt wesentlich deutlicher ausgeprägt. Der mit den Optimierungen erreichbare Speedup beträgt zwischen 1.1 und 3.4.

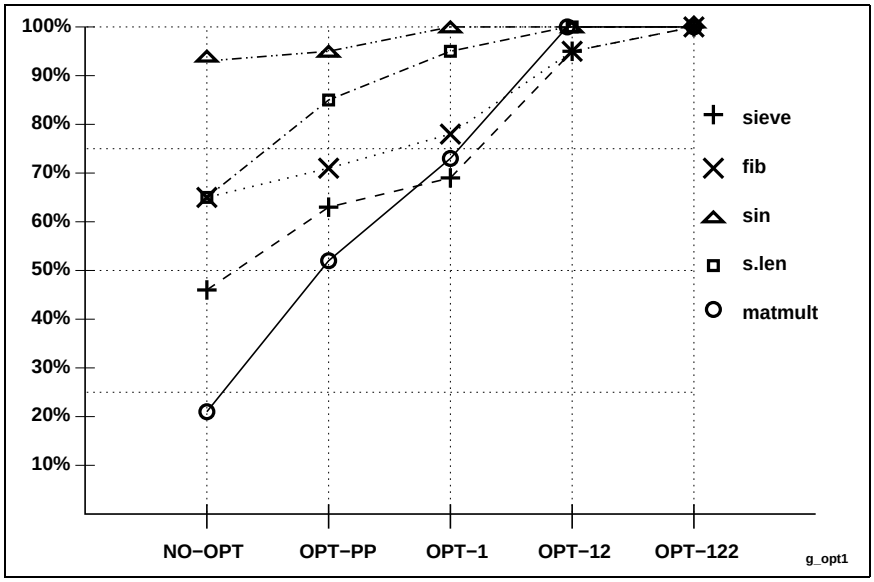

Abbildung 8.6: Einfluss der Peepholeoptimierung (PII/233)

	sieve	*fib*	*sin*	*s.len*	*matmult*
NO-OPT	1041/s, 47%	10.9s, 57.8%	43.5s, 88.7%	51.1s, 64.8%	130s, 40.7%
OPT-PP	1234/s, 56.1%	9.2s, 68.5%	41.6s, 92.8%	39.7s, 83.4%	91.6s, 57.9%
OPT-1	1492/s, 67.8%	8.2s, 76.9%	38.9s, 99.2%	37.3s, 88.7%	88.8s, 59.7%
OPT-12	1815/s, 82.5%	6.4s, 98.4%	38.6s, 100%	33.1s, 100%	54s, 98.1%
OPT-122	2200/s, 100%	6.3s, 100%	38.6s, 100%	33.1s, 100%	53s, 100%

Tabelle 8.10: Einfluss der Peepholeoptimierung (Duron/900)

	sieve	*fib*	*sin*	*s.len*	*matmult*
NO-OPT	205/s, 29.3%	50.5s, 49.5%	994s, 91.5%	266s, 56.0%	379.1s, 32.9%
OPT-PP	250/s, 35.7%	45.6s, 54.8%	977s, 93.1%	203s, 73.4%	272.3s, 45.9%
OPT-1	345/s, 49.4%	40.0s, 62.5%	1608s, 56.6% [1)]	166s, 89.9%	241.2s, 51.8%
OPT-12	598/s, 85.4%	25.1s, 99.6%	910s, 100%	149s, 100%	126.5s, 98.8%
OPT-122	700/s, 100%	25.0s, 100%	910s, 100%	149s, 100%	125s, 100%

Anmerkungen: [1)] “Ausreisser” vermutlich durch Cache-Trashing erzeugt.

Tabelle 8.11: Einfluss der Peepholeoptimierung (21164/500)

Für den `sieve`-Test, der mit den Arrayzugriffen sehr häufig Befehle mit drei Stackparametern ausführt, zeigt sich die Auswirkung des zweiten Durchlaufs des Regelsatzes 2 besonders gut. Dieser Optimierungslauf ermöglicht die Verlagerung auch des dritten Parameters in Register bei Feldzugriffen und erzeugt daher den auffälligen Geschwindigkeitsgewinn.

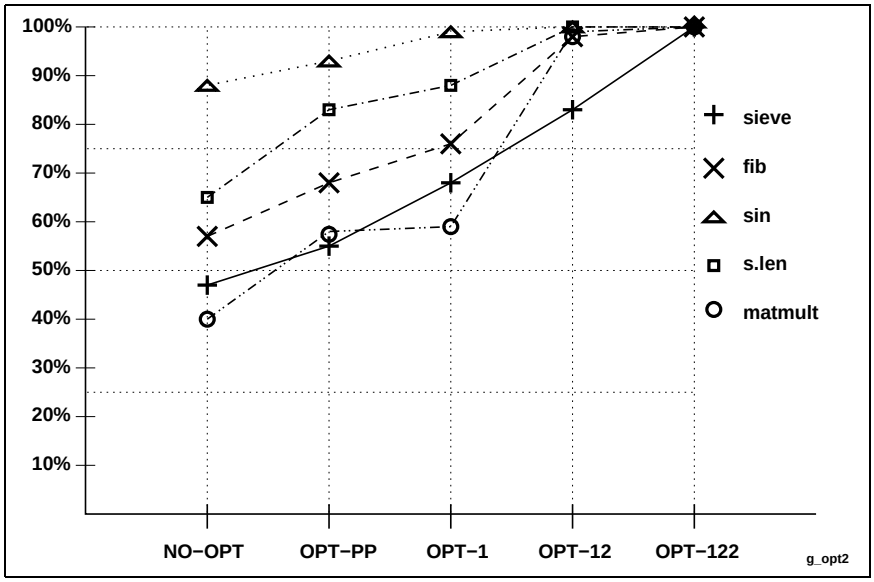

Abbildung 8.7: Einfluss der Peepholeoptimierung (Duron/900)

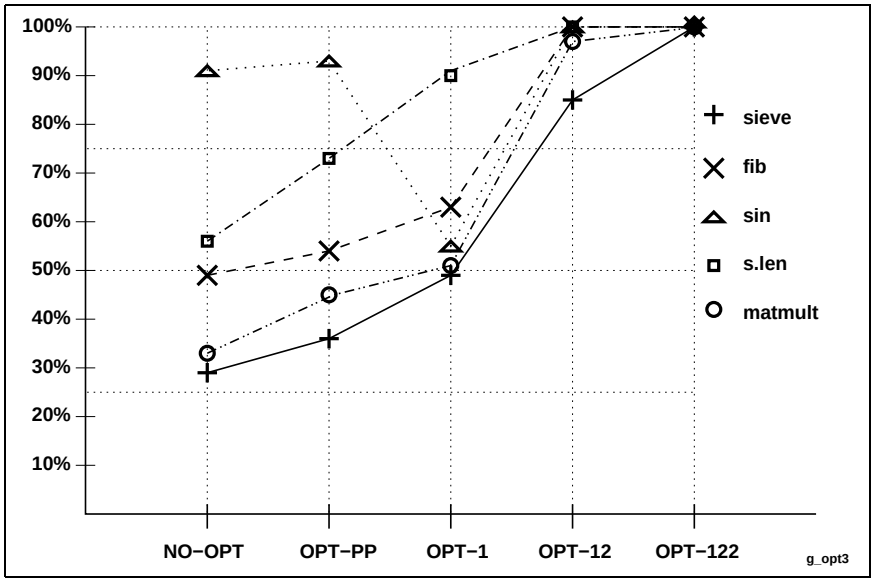

Abbildung 8.8: Einfluss der Peepholeoptimierung (21164/500)

Die Auswirkungen der einzelnen Übersetzungsstufen und Optimierungen auf die Codegröße der getesteten Hauptmethode werden in Tabelle 8.12 dargestellt. Bereits die trivialen PUSH/POP-Optimierungen, die in der Hardware quasi ohne Zeitverlust und nur mit wenig Logikaufwand zu erkennen sind, bringen eine Verkleinerung des resultierenden Zwischencodes auf IJVM-Ebene um 20 bis 30%.

	sieve	*fib*	*sin*	*s.len*	*matmult*
Insg. übersetzte Befehle					
JVM (Befehle)	70	19	28	31	92
Testmethode					
JVM (Bytes)	123	21	48	49	153
IJVM-Opcodes					
NO-OPT	176	43	65	63	251
OPT-PP	131	34	41	45	201
OPT-1	120	31	38	41	184
OPT-12	97	23	34	36	145
OPT-122	93	20	34	36	140
x86					
NO-OPT (Bytes)	468	143	283	244	731
OPT-PP (Bytes)	414	135	243	212	611
OPT-12 (Bytes)	378	121	222	203	547
OPT-122 (Bytes)	374	118	222	203	542
Alpha					
NO-OPT (Bytes)	1148	352	488	476	1692
OPT-PP (Bytes)	796	288	328	300	1292
OPT-12 (Bytes)	524	200	260	248	828
OPT-122 (Bytes)	492	188	260	248	808

Tabelle 8.12: Codegröße während der Übersetzungsstufen (x86/Alpha)

Der resultierende native Code verkleinert sich damit um bis zu 20% auf dem x86, und um bis zu 30% auf dem Alpha.

Alle Optimierungen zusammen (OPT-122) reduzieren die Zwischencodegröße fast auf die Hälfte, die native Codegröße je nach Methode um 20 bis 30% (x86) bzw. 40 bis 55% (Alpha).

Die starke Codereduktion auf dem Alpha entsteht hauptsächlich durch die Eliminierung der PUSH/POP-Befehle, die pro Befehl 8 Byte beanspruchen, auf dem x86 werden dafür nur 2Byte benötigt.

Da jeder IJVM-Befehl in vielen Fällen mehreren nativen Befehlen entspricht, wird damit durch die Reduzierung der IJVM-Größe auch die Erzeugung des Assemblercodes aus dem Paracode stark beschleunigt. Dieser Einfluss wird in Abschnitt 8.4.4 noch näher beschrieben.

Insgesamt gesehen ist das Optimierungssystem trotz seiner Einfachheit relativ effektiv und kann auch flexibel an die Anwendung aufgrund der Abwägung Aufwand gegen Effizienz eingesetzt werden.

8.3.3 Einfluss der Methodenaufrufe

8.3.3.1 Setup-Overhead

Um den Einfluss des Zusatzaufwands bei Methodenaufrufen zu klären, wurde der von JIFFY erzeugte Assemblercode für zwei Tests speziell so verändert, dass die Aufrufe nur mit einem Call ausgeführt werden konnten. Vorbelegung der Register, indirekte Sprünge und Stubs für virtuelle Routinen werden so nicht mehr ausgeführt.

Die Messergebnisse für statische und virtuelle Methoden sind in Tabelle 8.13 aufgeführt. Während auf x86-CPUs die Beschleunigung max. 10% beträgt, ist sie auf dem Alpha wesentlich ausgeprägter, da dort die Speicherzugriffe auf die Objekttabelle stärker ins Gewicht fallen.

Um diesen Effekt zu verifizieren, wurde der Original-Code für den Alpha für diesen Spezialfall so verändert, dass zwar noch die Speicherzugriffe notwendig sind, diese aber nicht mehr voneinander abhängen und auch der Sprung auf eine bereits bekannte Adresse erfolgt. Allein diese Änderung ergab eine Beschleunigung um 16%, allerdings ist diese Modifikation natürlich nicht im normalen Ablauf verwendbar.

	Mit Setup	Ohne Setup	Speedup
fib(40) auf PII/233	27.2s	24.8s	ca. 10%
fib(40) auf Duron/900	5.9s	5.4s	ca. 9%
fib(40) auf Alpha/500	25.0s	18.9s	ca. 32%

Tabelle 8.13: Einfluss des Setup-Overhead auf Methodenaufrufe

8.3.3.2 Overhead der Pascal-Aufrufkonvention

Messungen anhand des aufrufintensiven *fib*-Tests ergaben, dass ein Wechsel zur C-Aufrufkonvention, bei der der Aufrufer alle Stackparameter entfernt, maximal 12% Geschwindigkeitsgewinn erbringt. Die in Tabelle 8.14 gezeigten Messergebnisse wurden ohne die Belegung der Register für Methodenaufrufe durchgeführt.

Auffällig ist, dass auf dem PII der Geschwindigkeitsgewinn wesentlich weniger stark ausfällt als auf dem Duron. Anscheinend ist der Duron mit der Verwaltung des Returncaches wesentlich mehr auf die C-Aufrufkonvention optimiert.

Auf dem Alpha verlangsamt der Wechsel zur C-Aufrufkonvention den Ablauf sogar marginal, da jetzt zwei Stackkorrekturbefehle statt einem ausgeführt werden müssen. Da der Returnbefehl nicht auf dem Stack, sondern auf dem “Returnaddress”-Register (ra) arbeitet, ist auch keine Beschleunigung zu erwarten.

	JIFFY/Pascal-Calling	C-Calling	Speedup
fib(40) auf PII/233	24.8s	23.8s	ca. 4%
fib(40) auf Duron/900	5.4s	4.8s	ca. 12%
fib(40) auf Alpha/500	25s	25.2s	ca. -1%

Tabelle 8.14: Einfluss der Aufrufkonvention

8.3.3.3 Architekturabhängigkeit bei x86-Systemen

Der *fib*-Test wurde im Vergleich zum Ergebnis von `gcc` auch auf weiteren 80586-Systemen ausgeführt, um die Geschwindigkeitsunterschiede bei häufigen Methodenaufrufen und Stackzugriffen zu untersuchen. Dabei wurden hauptsächlich Systeme ausgewählt, die im Vergleich zu heutigen Arbeitsplatzsystemen langsamer sind, aber durch ihren niedrigeren Stromverbrauch für eingebettete Systeme gut geeignet sind. Unter anderem wurde auch ermittelt, welche Auswirkungen die Art des Methodenrücksprungs über `ret` oder eine Return-Emulation mittels `jmp` (siehe 7.1.2) auf die Effizienz hat.

Die Messergebnisse sind in der Tabelle 8.15 aufgeführt. Die Ergebnisse mit dem unkonventionellen `jmp`-Rücksprung sind nur aufgelistet, wo sie einen messbaren Geschwindigkeitsgewinn ergaben.

Es stellte sich bei einigen Systemen eine signifikante Abhängigkeit von der verwendeten Rücksprungmethode heraus, die anscheinend durch die Existenz eines Caches für Returnadressen erzeugt wird. Auf einem Systemen ohne Returncache (AMD K5) ergab die Benutzung des `jmp`-Befehls sogar eine schnellere Ausführung als der mittels gcc compilierte Code. Warum allerdings auf dem K5 bereits der Code mit `ret` im Vergleich mit gcc schneller als alle anderen CPUs ausführt, war nicht nachzuvollziehen.

Insgesamt kann festgestellt werden, dass auch bei einfachen x86-CPUs, die nicht über interne Optimierungen verfügen, JIFFY eine relativ gute Leistung aufweist. Der im Vergleich zu normalen C-Funktionen aufwendigere Code zum Methodenaufruf, Methodenstart und Methodenrücksprung ist zwar feststellbar, wirkt sich aber selbst bei sprungintensiven Anwendungen nicht so deutlich aus wie zunächst angenommen.

Hersteller	Intel	Intel	AMD	AMD	IDT	IDT
Typ	P2	Pentium	K5	K5	C2	C2
MHz	233	133	133	133	240	240
Rücksprung	ret	ret	ret	jmp	ret	jmp
JIFFY	30.0	68.1	65.5	56.1	50.3	48.8
gcc -O3	28.0	58.7	61.5	-	32.5	-
JIFFY vs. gcc	93%	86%	94%	109%	65%	67%

Hersteller	AMD	AMD	Intel	Intel	Intel
Typ	K6	K6	80486DX4	80486DX4	P2 Xeon
MHz	233	333	100	100	450
Rücksprung	ret	ret	ret	jmp	ret
JIFFY	34.3	25.3	189	176	15.5
gcc -O3	19.1	14.6	130	-	13.8
JIFFY vs. gcc	56%	58%	69%	74%	89%

(Laufzeiten in Sekunden)

Tabelle 8.15: *fib(40)*-Benchmark für 80586-Kompatible

8.4 Abschätzungen für die Hardware aufgrund der SW-Simulation

8.4.1 Simulationsannahmen

Die SW-Simulation des JIFFY-Übersetzungsvorgangs wurde auf einer relativ niedrigen Ebene beschrieben, wodurch auch potentielle Schreib- und Lesezugriffe auf internen und externen Speicher, die später im FPGA ablaufen, überwacht und gezählt werden können. Natürlich ist damit nur eine vorläufige Abschätzung der Leistung möglich, da eine reale Implementierung wahrscheinlich teilweise etwas andere Kennzahlen ergeben würde.

Die Simulation wurde dabei mit folgenden, sinnvollen[2] Annahmen für die Architektur des Gesamtsystems instrumentiert:

- Es wird angenommen, dass kontinuierliche Datenflüsse aus zusammenhängenden Speicherbereichen, wie z.B. Bytecodes oder IJVM-Befehle durch den Einsatz von Streaming (z.B. mit DMA-Transfer) und FIFOs nicht als signifikante Wartezeiten ins Gewicht fallen.

[2] Die Werte stimmen fast vollständig auch mit den Zahlen für die spätere HW-Implementierung überein.

- Ein SRAM-Lesezyklus benötigt zwei Taktzyklen, zwei Zyklen mit aufsteigender Adresse (“Cacheline”) drei Taktzyklen.
- Ein SRAM-Schreibzyklus dauert einen Taktzyklus.
- Die Analysephase benötigt pro JVM-Befehl zunächst zwei Takte. Da viele JVM-Befehle nur ein Byte umfassen und ein Streambuffering möglich ist, ist diese Annahme gerechtfertigt.
- Das Einfügen eines Sprungziels in die Sprungtabelle während der Analysephase besteht aus je einem SRAM Lese- und Schreibzugriff. Damit ist auch die Zeit für das Parsen von multiplen Sprungzielen von `lookupswitch` bzw. `tableswitch` erfasst.
- Die JVM→IJVM-Phase ist in folgende Unterteilungen aufgeschlüsselt:
  - Pro JVM-Befehl 1 Takt und 3 SRAM-Lesezyklen für die Lookuptabelle.
  - Pro übersetztem IJVM-Befehl 1 SRAM-Lesezyklus und ein 1 SRAM-Schreibzyklus.
  - Pro `lookupswitch` bzw. `tableswitch`-Sprungziel 1 SRAM-Lesezyklus und ein 1 SRAM-Schreibzyklus.
  - Die Erzeugung von Labelbefehlen anhand der Sprungtabelle benötigt 1 Takt zum Lesen und 4 zum Schreiben in das SRAM.
  - Die Typisierung von von OO-Befehlen, also der Zugriff auf den Konstantenpool benötigt 9 Taktzyklen (z.B. SDRAM-Latenz).
- Die Peepholeoptimierung erfordert pro IJVM-Befehl und Regelsatz einen Taktzyklus (unter der Voraussetzung, dass Streaming möglich ist), für eine ausgeführte Ersetzungsaktion zusätzlich einen weiteren Takt. Damit entspricht diese Annahme einer vollständig parallel arbeitenden Implementierung.
- Die IJVM→NAT-Phase besteht aus folgenden Einzelzeiten:
  - Pro IJVM-Befehl ein Takt und zwei SRAM Lesezyklen.
  - Pro Paracode-Befehl ein Takt und 4 SRAM-Lesezyklen.
  - Pro einzelnem Paracode-Patch zwei Takte.
  - Pro zu linkendem Sprungziel je drei SRAM-Lese/Schreibzyklen.

8.4.2 Simulationsergebnisse

Die unter den beschriebenen Simulationsannahmen gewonnenen Taktzyklen für die Übersetzung der Testmethoden und eventueller Hilfsroutinen (Stubs, Wrapper) sind in Tabelle 8.16 aufgeführt.

	sieve	*fib*	*sin*	*s.len*	*matmult*
x86					
Takte insgesamt	3435	1384	1764	1946	4885
Takte SRAM-Read	1282	595	795	836	1787
Takte SRAM-Write	197	52	65	69	284
Takte/JVM-Befehl	49	73	63	63	53
Alpha					
Takte insgesamt	3515	1397	1691	1950	5168
Takte SRAM-Read	1309	615	730	863	1919
Takte SRAM-Write	198	52	66	70	284
Takte/JVM-Befehl	50	73	60	63	56

Tabelle 8.16: Anzahl der benötigten Takte (simuliert)

Mit der in Tabelle 8.12 angegebenen Anzahl der JVM-Befehle ergibt sich somit ein Verhältnis von 50 bis 70 Takten pro JVM-Befehl und das relativ unabhängig von der Zielarchitektur. Für größere Programme verbessern sich diese Werte etwas, da hier der relative Overhead der Stub/Wrapper-Erzeugung kleiner wird.

Bei einem pessimistisch angenommenen FPGA-Takt von 50MHz können damit also knapp 700000 bis 1 Million JVM-Befehle pro Sekunde übersetzt werden. Im Vergleich dazu erreicht z.B. CACAO auf einem 21164 mit 500MHz, also zehnfach höherem Takt, 500000-700000 JVM-Befehle pro Sekunde.

8.4.3 Anteil der Übersetzungsphasen an der Gesamtzeit

Um einen Anhaltspunkt für weitere Optimierungen im Ablauf zu erhalten, wurden die Anteile der einzelnen Übersetzungsphasen im Gesamtablauf der Übersetzung einer Methode ermittelt. Diese Werte sind in Tabelle 8.17 aufgeführt.

Wie anhand der Zahlenwerte zu erkennen ist, benötigen JVM→IJVM und IJVM→NAT jeweils ungefähr ein Drittel der Gesamttakte, der Anteil variiert nur marginal mit der Komplexität der Methode.

	sieve	*fib*	*sin*	*s.len*	*matmult*
x86					
JVM-Analysephase	423, 13%	128, 14%	279, 18%	279, 18%	498, 10%
JVM→IJVM	1035, 31%	261, 29%	452, 29%	427, 29%	1432, 30%
Peephole-Optimierung	729, 21%	213, 23%	285, 18%	267, 18%	1087, 23%
IJVM→NAT (inkl. Linker)	1174, 35%	303, 34%	539, 35%	516, 35%	1768, 37%

Anmerkungen: Optimierungsart OPT-122, Übersetzung nur der Hauptmethode ohne Stubs

Tabelle 8.17: Aufteilung der Übersetzungsphasen

8.4.4 Einfluss der Optimierungen auf die Übersetzungsgeschwindigkeit

Jeder IJVM-Befehl wird in der letzten Assemblerstufe aus dem Paracode heraus typischerweise in mehrere Assemblerbefehle übersetzt, was bei sequentieller Abarbeitung mindestens einen Takt pro Paracodebefehl benötigt. Hinzu kommen evtl. weitere Takte für den Zugriff auf den Paracodespeicher oder die Berechnung von Konstanten. Mit mehr als einem Drittel der Gesamttaktzahl stellt sich diese letzte Codegenerierungsstufe als Flaschenhals der gesamten Übersetzung an sich dar.

Da die IJVM-Optimierungen die Anzahl der zu übersetzenden Befehle vermindert, stellt sich die Frage, inwieweit zusätzlicher Aufwand für den Optimierungsvorgang die Codegenerierung beschleunigen kann.

Um dies zu untersuchen, wurden mit der im vorhergehenden Abschnitt besprochenen Simulation des Übersetzungsvorganges die Anzahl der benötigten Takte für den gesamten Übersetzungsvorgang der Testmethoden und der Hilfsroutinen gemessen. Die Ergebnisse in Abhängigkeit von der Optimierungsmethode sind in der Tabelle 8.18 (numerisch) und in den Abbildungen 8.9 bzw. 8.10 zu finden.

Vergleicht man die Anzahl der benötigten Taktzyklen ohne (NO-OPT) und mit voller (OPT-122) Optimierung, fällt auf, dass die Gesamtanzahl fast gleich oder mit Optimierung sogar geringer ist, obwohl durch die Peepholeoptimierung mehr Takte benötigt werden.

Wie anhand der einzelnen Optimierungstufen ersichtlich wird, beruht dieser unerwartete Effekt auf dem Einsparen von Takten während der Assemblierung und einem teilweisen Ausgleich dieser Einsparung durch den Mehraufwand der Peepholeoptimierung. Auf dem Alpha liegt die Takteinsparung mit Optimierung durch die etwas höhere Anzahl an Assemblerbefehlen pro IJVM-Befehl sogar auffällig höher, um teilweise bis zu 15%.

Die wenigsten Takte (15% weniger als ohne Peepholeoptimierung) werden beim Einsatz der trivialen Peepholeoptimierung gebraucht, da dieses Verfahren

	sieve	*fib*	*sin*	*s.len*	*matmult*
x86					
NO-OPT	3554	1347	1799	1964	4983
OPT-PP	3086	1263	1577	1685	4416
OPT-12	3269	1336	1678	1840	4643
OPT-122	3435	1384	1764	1946	4885
NO-OPT/OPT-122	103%	97%	101%	101%	102%
Alpha					
NO-OPT	4032	1460	1842	2116	5768
OPT-PP	3350	1336	1532	1713	4993
OPT-12	3367	1349	1603	1842	4926
OPT-122	3515	1397	1691	1950	5168
NO-OPT/OPT-122	115%	105%	109%	109%	112%

Tabelle 8.18: Benötigte Takte (simuliert) in Abhängigkeit von der Optimierung

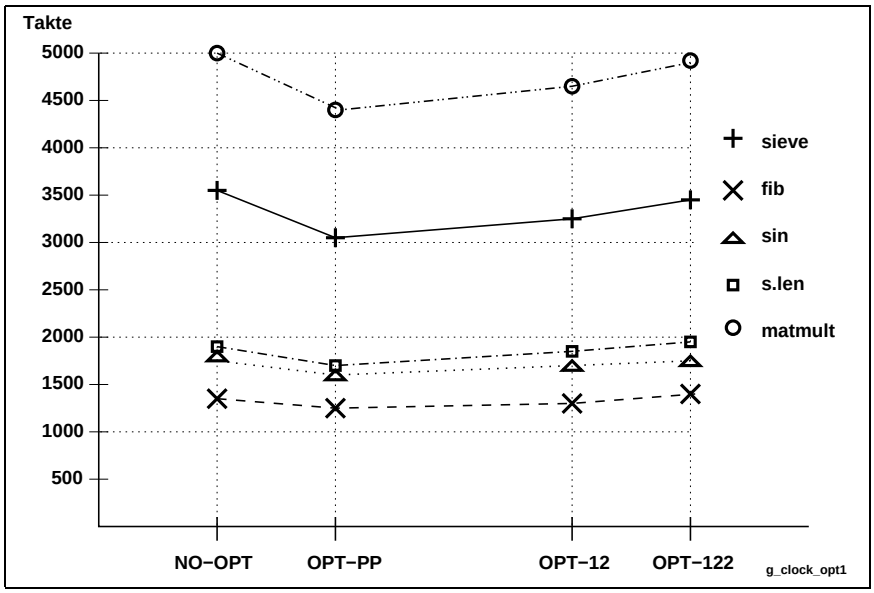

Abbildung 8.9: Einfluss Optimierung auf benötigte Taktzyklen (x86)

ohne zusätzliche Taktverluste implementiert werden kann. Allerdings bringt diese Stufe allein auch nur geringe Optimierungsgewinne (siehe Tabellen 8.9 bis 8.11).

Steht die Peepholeoptimierung in einer schnellen parallelen Variante (zB. 1+1 Takt pro IJVM-Befehl) zur Verfügung, erreicht sie mit 15% mehr Übersetzungslaufzeit eine bis um den Faktor 4 beschleunigte Ausführung des Codes. Von daher sind die Teilstufen der Peepholeoptimierung (OPT-1, OPT-12) zwar benutzbar

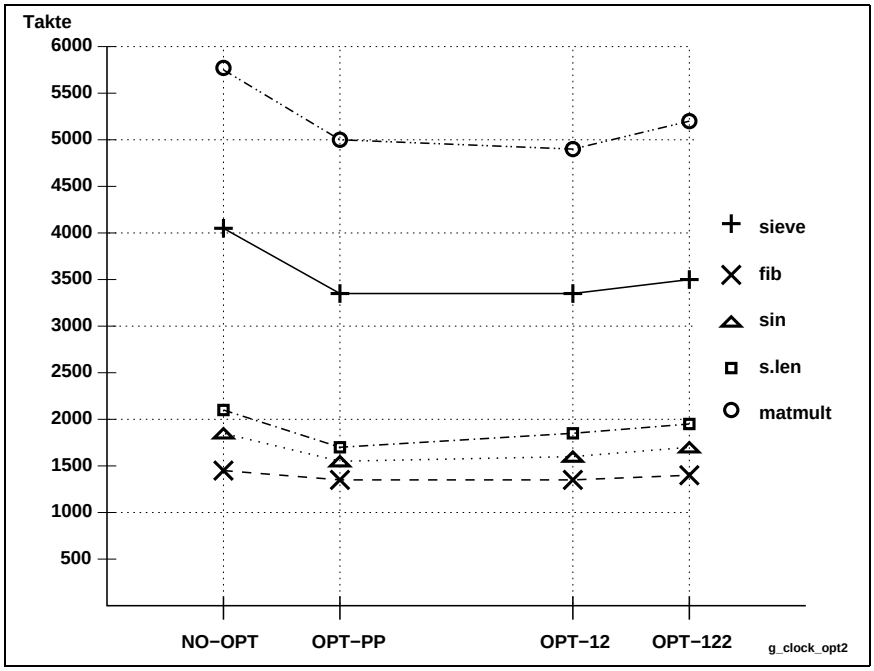

Abbildung 8.10: Einfluss Optimierung auf benötigte Taktzyklen (Alpha)

(wenn FPGA-Logik eingespart werden muss), verschenken aber relativ zu ihrer Laufzeit zuviel Geschwindigkeit.

8.4.5 Optimierung durch Einbettung

Die Ergebnisse für die Laufzeit der Peepholeoptimierung sind noch weiter verbesserbar, wenn eine Verzahnung der einzelnen Phasen der Übersetzung stattfindet, wie in Abb. 8.11 gezeigt. Durch diese Einbettung der Optimierungsphasen wird der effektive Taktverbrauch miniminiert. Die Integration der Optimierung mit Regelsatz 1 in den IJVM-Ausgabestrom der JVM→IJVM-Phase versteckt die für die Optimierung benötigten Takte. Analog ist es möglich, Regelsatz 2 direkt vor der JVM→NAT-Phase im Befehlsstrom zu integrieren. Somit wird nur noch ein Durchlauf von Regelsatz 2 im Speicher benötigt. Die Optimierungsarten OPT-1 bzw. OPT-12 sind dann bis auf die Durchlauflatenz ohne Taktmehraufwand zu erhalten.

Mit der im folgenden Abschnitt besprochenen FPGA-Implementierung der Optimierung und der Einbettung von zwei der drei Optimierungen in den Datenfluss ergeben sich in der Simulation die in Tabelle 8.19 aufgeführten Taktzyklen. Je nach Anzahl der durchgeführten Optimierungen werden dabei Geschwindigkeiten der Varianten zwischen OPT-PP und OPT-12 ohne Verzahnung der Phasen erreicht.

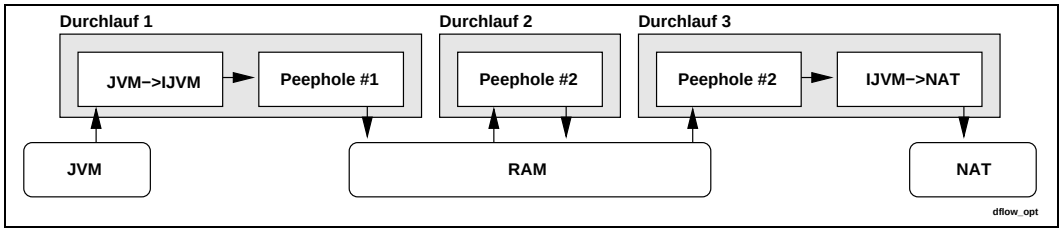

Abbildung 8.11: Einbettung der Optimierung

	sieve	*fib*	*sin*	*s.len*	*matmult*
x86					
OPT-122, nicht eingebettet	3435	1384	1764	1946	4885
OPT-122, eingebettet	3235	1336	1716	1889	4594

Anmerkung: Übersetzung inkl. Stubs und Hilfsroutinen

Tabelle 8.19: Takteinsparung durch Einbettung der Optimierungen

8.5 Ergebnisse erster FPGA-Implementierungen

Die erste, prototypische Umsetzung der einzelnen Übersetzungsstufen in einer Hardwarebeschreibungssprache (VHDL) gibt einen Eindruck von der Hardwarekomplexität.

Die Module wurden ohne besondere Optimierungen in VHDL beschrieben und mit dem Synopsys FPGA-Compiler2 auf die Spartan2-Architektur von Xilinx synthetisiert. Die für die Systemanbindung nötige Logik (Steuerregister, DMA/SDRAM-Ansteuerung, etc.) ist noch nicht miteinbezogen und dürfte erfahrungsgemäß bei 5-10% der Gesamtsumme liegen.

Die durch die Synthese entstandenen FPGA-Ressourcen sind in Tabelle 8.20 angegeben. Basierend auf den Angaben einiger FPGA-Routing-Läufe wurden die erreichbaren Taktzeiten für alle Module mit mehr als 60MHz abgeschätzt.

Aufgrund der Tabelle ist ersichtlich, dass damit noch fast die gesamten Speicherkapazität im Blockram zur Verfügung steht, und so die Integration von Übersetzungstabellen ohne Probleme möglich ist. Damit ist besonders der kritische Pfad der Referenztabellen für die JVM→IJVM und IJVM→NAT Phasen im FPGA zu halten. Dies ergibt einen Geschwindigkeitsgewinn gegenüber einem externen SRAM um zwei bis vier Takte.

Zusätzlich wird es möglich, die am häufigsten benutzten Befehle in der JVM→IJVM- und Paracodetabelle im BlockRAM zu speichern und somit weitere externe Speicherzugriffe auf SRAM oder SDRAM einzusparen.

Damit ist das JIFFY-System gut in einen XC2S200-FPGA zu integrieren, es nimmt in seiner ersten, noch relativ unoptimierten Version knapp 80% des Chips

Phase	Flipflops	LUTs	RAM-Bits
JVM-Analyse und JVM→IJVM	650	1200	
Peepholeoptimizer (seriell, dynamisch)	300	450	inkl. 3200Bit
Peepholeoptimizer (parallel, statisch)	550	600	inkl. 1280Bit
FPGA-RAM zu Peepholeoptimizer			+5KBit
IJVM→NAT	600	1500	inkl. 2600Bit
FPGA-RAM zu IJVM→NAT			+3KBit
Linker	350	500	
Summe	2150	3800	8KBit
Zum Vergleich:			
Spartan2 XC2S200	4704	4704	73KBit+56KBit[1)]

Anmerkung: [1)] distributed/BlockRAM

Tabelle 8.20: Anhaltspunkte für die Komplexität der FPGA-Hardware

in Anspruch. Verbesserungen sind insbesondere noch bei der JVM-Analyse und der IJVM→NAT-Übersetzung zu erwarten. In diesen beiden Phasen werden relativ oft breite Datenbusse über Multiplexer geführt, was zur Zeit noch über generischen VHDL-Code beschrieben wird. Hierbei würde der Einsatz von speziellen, auf die FPGA-Architektur abgestimmte Modulen (“Cores”) für Multiplexer und Barrelshifter wesentlich weniger Logikzellen benötigen.

Eine weitere Optimierung ist mit der Zusammenführung aller Module zu erwarten. Durch die damit mögliche Eliminierung der Hierarchie (“Flattening”) werden ebenfalls die erforderlichen FPGA-Ressourcen etwas verringert.

Besonders interessant ist der geringe Logikverbrauch der schnellen parallelen Peepholeoptimierung, der erst durch die Realisierung mit statischen Erkennungsregeln und dynamischen Ersetzungsregeln möglich geworden ist. Im Vergleich dazu ergibt die serielle Variante nur 30% Logikeinsparung bei einer wesentlichen Verlangsamung, die Regeln sind aber während der Laufzeit dynamisch änderbar. Daher bietet sich diese Version für weitere Entwicklungen bzw. dynamische Anpassungen der Regeln im System an.

Wie bereits gezeigt, ist auch der zusätzliche Zeitbedarf der Peepholestufen in der parallelen Implementierung vernachlässigbar, wodurch diese effektive Art der Codeoptimierung im FPGA sehr günstig zu erhalten ist: Bei 50MHz Takt ist ein Durchsatz der FPGA-Peepholeoptimierung von 25 Millionen IJVM-Befehlen pro Sekunde erreichbar. Eine Implementierung in Software würde für diese Geschwindigkeit dagegen einen CPU-Takt von weit über 500MHz erfordern, hierbei sind auch Probleme mit der in diesem Fall ineffizienten Sprungvorhersage zu erwarten.

8.5.1 Weitere Realisierungsoptionen

Wenn auch das JIFFY-System so entworfen wurde, dass es effizient auf FPGAs implementiert werden kann, ist ein FPGA nicht unbedingt notwendig. Mit bestimmten Einschränkungen ist JIFFY auch in einem ASIC nutzbar, wobei die Realisierungskosten in höheren Stückzahlen sinken. Diese Einschränkungen betreffen hauptsächlich die Möglichkeit, durch die FPGA-Rekonfiguration Änderungen bzw. Verbesserungen am Übersetzungsvorgang vorzunehmen.

FPGAs sind teurer als entsprechende ASICs, da sie komplexer aufgebaut sind. Ein für JIFFY ausreichendes FPGA (XC2S200E) liegt zur Zeit bei einem Einzelstückpreis von 25$, bei hohen Stückzahlen dürfte der Preis auf 12-15$ sinken. Ein ASIC mit vergleichbarer Kapazität würde ungefähr 5$ kosten, allerdings ist der ASIC-Grundkostenaufwand relativ hoch. Werden die Eigenschaften von FPGAs nicht benötigt, sind ASICs also für den Massenmarkt (mehrere 10000 Stück) günstiger.

Andererseits kann ein FPGA zur Java-Übersetzung in der Entwicklung (“Time-to-Market”) und Fertigung billiger sein, als eine schnellere CPU und mehr Speicher für einen Software-JIT.

Ein weiterer Aspekt verbilligt wiederum den Einsatz von JIFFY in einem FPGA: Inzwischen sind auch relativ leistungsfähige Soft-CPU-Kerne für FPGAs verfügbar, wie z.B. der MicroBlaze [170] mit ca. 100MIPS. Dies erlaubt es, ein vollständig frei definierbares System mit CPU, JIT und weiterer Peripherie zu integrieren. Somit wird über die Integration der Gesamtsystempreis wesentlich reduziert.

8.6 Zusammenfassung der Ergebnisse

Die Simulation von JIFFY als JDK-Plugin und erste prototypische FPGA-Implementierungen der Module des JIFFY-Übersetzungsvorgangs zeigen, dass das JIFFY-System trotz seiner Einfachheit relativ schnell durchaus akzeptablen Code für CISC und RISC-Systeme erzeugen kann. Dieser Code kann in vielen Bereichen mit Standard-JIT-Compilern wie dem sunwjit mithalten. Der Ressourcenverbrauch an Speicher liegt dabei noch unter den Werten “normaler” JIT-Compiler wie sunwjit und weit unter denen des Hotspot-Systems.

Auf der anderen Seite übersetzt JIFFY mit einer Geschwindigkeit, die für Software-JIT-Compiler auf Desktop-Prozessoren mit 500MHz oder mehr typisch ist und ist damit von der Übersetzungsgeschwindigkeit nicht mehr das begrenzende Element in einem Java-Laufzeitsystem.

Die internen Abläufe des FPGAs sind in weiten Grenzen auf die Anforderungen und Gegebenheiten der Hardware einstellbar und erlauben noch weitgehende

Optimierungen bzw. Änderungen im Datenfluss. Einige davon, wie z.B. das direkte Einfügen der Peepholeoptimierung in die Übersetzungsphasen, wurden in den vorangegangen Abschnitten bereits angedeutet.

Der Einfluss der Peepholeoptimierungen auf die Geschwindigkeit der Übersetzung und des erzeugten Codes anhand einiger Benchmarks zeigen die große Flexibilität bei der möglichen Anpassung auf das Zielsystem. Zusätzlich ermöglicht die sehr leichte Portierbarkeit des Softwareteils und die CPU-unabhängige FPGA-Architektur die Anpassung auf sonst nicht mit einem JIT unterstützte, “exotische” Systeme.

9

Zusammenfassung

Heisa, juchheia! Dudeldumdei!
Friedrich Schiller, Wallensteins Lager

In den vorangegangenen Kapiteln wurde das JIT-Übersetzungskonzept von JIFFY für den Einsatz in FPGAs entwickelt und seine Vor- und Nachteile anhand eines Simulationsmodells und einer ersten Hardwareimplementierung evaluiert. Die unterschiedlichen Implementierungsmöglichkeiten und deren Komplexität für die beiden Ebenen “Software” und “FPGA” wurden gegenübergestellt und verglichen. Insbesondere wurden die Auswirkungen einer einfachen Übersetzung in Hardware auf das Software-Laufzeitsystem beschrieben. Dabei wurde gezeigt, dass das Konzept in hohem Maße flexibel ist und an einen breiten Bereich von Anforderungen angepasst werden kann. Damit unterscheidet es sich stark von den bereits existierenden JVM-Beschleunigern, die relativ starr auf bestimmte Prozessoren festgelegt sind.

Als Vorteile wurden im Einzelnen dargelegt:

+ JIFFY ist flexibel einsetzbar
    + Das System ist als Plugin oder in einer “Clean-Room”-Implementierung einer Java-VM benutzbar. Die internen Abläufe sind nicht vom Aufbau der VM abhängig.
    + Der HW-Aufwand und die benötigte Übersetzungsgeschwindigkeit können anhand verschiedener Implementierungsparameter in weiten Grenzen auf das geplante Einsatzgebiet angepasst werden.
    + Im Gegensatz zu Java-Prozessoren in HW sind Erweiterungen des JVM-Befehlsumfangs, wie z.B. Multimediabefehle mit SIMD-Semantik, aufgrund von Spezifikationsänderungen oder anwendungsspezifischen “Quick”-Opcodes sehr einfach durchzuführen.

+ Das FPGA ist CPU-Architekturunabhängig durch “Paracode”-Ansatz

  Die Implementierungen für je einen Vertreter von CISC und RISC-Architekturen bestätigen die universelle Verwendbarkeit des Paracode-Assemblers. Dies ermöglicht die Benutzung des JIFFY-FPGAs auch in heterogenen CPU-Systemen durch den Austausch einiger weniger KByte der Zielsystembeschreibung. Weiterhin erlaubt diese Unabhängigkeit den Einsatz in heterogenen Systemen und die JIT-Compilierung für alle integrierten Prozessoren.

+ Geringer Softwareaufwand

  Die in der Software nötigen Vorbereitungen zum Aufsetzen der internen Strukturen und der Integration in eine VM sind überschaubar und erfordern signifikant weniger Codespeicher als übliche JIT-Compiler.

+ Relativ effiziente Codeerzeugung für CISC und RISC

  Die Übersetzungsqualität von JIFFY erreicht oder übersteigt die Qualität einfacher JIT-Compiler und ermöglicht damit ca. 30-60% der Ablaufgeschwindigkeit von komplexen und speicheraufwendigen JIT-Compilern wie dem Hotspot-System. Dies ist sicherlich nicht als “höchste” Leistung zu bezeichnen, ist aber gerade bei RISC-Systemen signifikant besser als eine Interpretierung und kann daher erst den sinnvollen Einsatz einer JVM in einem System ermöglichen.

+ Parallele Übersetzung und CPU-Ausführung

  Da die Übersetzung im FPGA nach dem Start unabhängig von der CPU ist, kann diese in der Wartezeit auf das Ende der Übersetzung in einem anderen Prozess bzw. Java-Thread weiterarbeiten. Diese Parallelausführung steigert die effektive Gesamtleistung des Systems.

+ Gute Skalierung mit CPU- und FPGA-Geschwindigkeit

  Jede Verbesserung in der CPU oder FPGA-Technologie wirken sich unmittelbar auf die Gesamtleistung aus, da dadurch entweder die Geschwindigkeit der Übersetzung oder die der Ausführung direkt profitieren.

+ Eingriffsmöglichkeit/Debugging in allen Phasen möglich

  Durch definierte Schnittstellen ist es möglich, in allen Übersetzungsphasen von der FPGA-Übersetzung wieder in die Übersetzung der C-Simulation zu wechseln.

+ Klare Speicherhierarchie mit definierten Schnittstellen

Das JIFFY-System besitzt eine klare Trennung der verschiedenen Daten und Zugriffsmuster auf die Daten. Damit ist ein mehrstufiges Speicherkonzept möglich, das Speicherlatenzen- und Durchsatz gegenüber den Kosten der Speicherarten definiert abwägen kann.

+ Geringer zusätzlicher, systembedingter Speicherbedarf

  Als temporärer Speicherbedarf für die Übersetzung werden maximal ca. 400KByte benötigt. Mit einer einfachen IJVM-Komprimierung ist dieser Wert auf unter 250KByte zu verringern.

+ Nutzung der FPGA-Ressourcen für andere Anwendungen

  Das FPGA kann durch Rekonfiguration oder direkte Integration weitere Systemaufgaben übernehmen und damit z.B. ein ASIC oder “Gluelogic” einsparen.

+ Updatefähigkeit des Übersetzungsvorganges trotz HW

  Durch eine mögliche Änderung des FPGA-Bitstreams und der Übersetzungstabellen sind trotz einer Implementierung in Hardware alle Möglichkeiten eines “in-System”-Updates zur Fehlerbeseitigung, Geschwindigkeitssteigerung und Implementierung neuer Leistungsmerkmale gegeben.

Die Nachteile können folgendermaßen zusammengefasst werden:

– Keine komplexeren Optimierungen möglich

  Der bei weitem größte Nachteil ist der Verzicht auf weitergehende Optimierungen anhand des Datenflusses, wie sie Stand der Compilertechniken sind. Damit gehen ca. 40-60% der maximal möglichen Leistung verloren. Im FPGA selbst sind solche Algorithmen nur schwer möglich, allerdings sind sie durchaus in SW auf Zwischencode-Ebene nachzurüsten, d.h. das JIFFY-Konzept verhindert weitere Optimierungen nicht.

– Effizienzverluste bei Methodenauflösung

  Durch das weitgehende Fehlen von Informationen über anzuspringende Methoden bei der Compilation entsteht ein permanenter Aufwand, obwohl dieser prinzipiell nur einmal notwendig wäre.

– Je nach FPGA etwas höherer Stromverbrauch

  FPGAs haben im allgemeinen höhere Leckströme bzw. Ruheströme als äquivalente ASICs und schränken daher ohne weitere Vorkehrungen den Einsatz in mobilen Geräten ein. Der dynamische Stromverbrauch ist in etwa mit anderen Schaltkreisen vergleichbar.

– “Teures” FPGA notwendig

  Werden die Eigenschaften eines FPGAs wie die Rekonfiguration nicht verlangt, sind FPGAs zur Logikimplementierung bei höheren Stückzahlen deutlich teuerer als anwendungsspezifische Schaltkreise.

Welche dieser Nachteile den Einsatz von JIFFY ungeeignet erscheinen lassen oder ganz verhindern, hängt letztendlich vom Anwendungsgebiet und den geforderten Kosten der Zusatzhardware ab.

Insgesamt gesehen ist das JIFFY-System also ein offenes JIT-System in FPGA-Hardware, dass die in Abschnitt 6.2 beschriebenen Anforderungen an einen JVM-Beschleuniger für eingebettete Systeme gut erfüllt.

10

Ausblick

> It's hard to predict, especially the future.
> *Niels Bohr*

Das in dieser Arbeit entwickelte JIFFY-Konzept besitzt bereits in der relativ eingeschränkten Simulationsumgebung vielversprechende Eigenschaften. Allerdings konnte es bislang nicht vollständig in realen Anwendungen eingesetzt werden, da hierzu noch einige Bausteine für das JIFFY-Gesamtsystem fehlen.

Zu einer vollständigen Java-VM fehlen dem jetzigen Laufzeitsystem als JDK-Modul noch einige Schnittstellen und die Implementierung weniger Bytecodebefehle. Diese Änderungen betreffen allerdings keine der JIFFY-internen Vorgänge, auch würden sie das Einbindungskonzept nicht verändern.

Weiterhin müssen die zur Zeit nur in getrennten Testumgebungen ablaufenden VHDL-Modelle von JIFFY zusammengeführt und als ein lauffähiges System in ein FPGA synthetisiert werden. Es ist zwar nicht zu erwarten, dass sich die grundsätzlichen, bereits besprochenen Eigenschaften ändern, allerdings dürfte die FPGA-Komplexität durch die Integration weiterer Logik zur Anbindung an Bussysteme und Speicherblöcke etwas verändert werden. Der genaue Einfluss ist noch unbekannt.

Das Ziel ist die Integration und Evaluation von JIFFY in ein reales eingebettetes System, wie z.B. in einen PDA mit ARM-Prozessor. In dieser Umgebung wird es möglich, die Eigenschaften des JIT-Systems weiter zu optimieren und auf den Anwendungszweck anzupassen.

Um die Codeeffizienz zu steigern, ist es denkbar, weitere Optimierungsvorgänge in Software oder im FPGA zu implementieren. Hierbei wären besonders die nahtlose Verzahnung mit dem JIFFY-System und Umsetzungen in effiziente FPGA-Strukturen von Interesse. Ebenso ist zu untersuchen, ob auch mit einem FPGA-JIT sich ein Hotspot-ähnliches System, das also Interpretierung und JIT abwägt, nutzbringend einsetzen lässt.

Die Nebenprodukte, die während der Entwicklung von JIFFY entstanden, sind ebenfalls vielfältig weiterverwendbar, nur ein kleiner Teil davon hängt wirklich von Java und der JVM ab. Damit ist es denkbar, einzelne Teile des JIFFY-Systems auf verwandten oder gänzlich anderen Anwendungen einzusetzen.

Eine mögliche Weiterentwicklung wäre zum Beispiel die Portierung von JIFFY auf C#. Da auch C# auf einer VM basiert und zur Laufzeit ein JIT-System benötigt, sollte es relativ einfach sein, JIFFY auf C# umzusetzen. Dabei ist zu erwarten, dass nur die erste Übersetzungsstufe und einige Peepholeregeln auf C# angepasst werden müssen.

Weiterhin wäre die Nutzung des Paracodeassemblers für Kommunikationsanwendungen zu erforschen. Diese basieren immer mehr auf dem Einsatz von digitalen Signalprozessoren (DSPs) statt ASICs und, damit verbunden, die Verlagerung der Algorithmen von dedizierter Hardware in Software.

Diese “Software-Defined Radio”-Anwendungen (SDR) werden in Zukunft immer weiter zunehmen und irgendwann auch eine Übertragung der eigentlichen Dekodierungs- und Protokollalgorithmen über das Verbindungsmedium bedingen. Um eine “Updatefähigkeit” aller Geräte zu erreichen, wird es unumgänglich sein, die Algorithmen plattformunabhängig zu übertragen. Dies kann über Java oder eine andere, besser auf DSP-Anwendungen angepasste Beschreibung erfolgen (“Radio Virtual Machines”, RVM [65]). Erste Ansätze in diesem Gebiet werden dazu z.B. in der Hardware Abstraction Layer Working Group [67] erarbeitet.

Möglicherweise basieren zukünftige Protokolle sogar auf einer abwechselnden Übertragung von standardisierten Dekodieranweisungen und zu dekodierenden Daten. Für diese Anwendungen wird es notwendig sein, die Algorithmen ähnlich eines JVM-JITs zu übersetzen oder direkt im Befehlsfluss umzusetzen. Da SDR-Anwendungen auch stark auf FPGAs basieren werden, ist der Paracodeassembler aufgrund seiner flexiblen Struktur dafür sehr gut geeignet.

Damit sind die mit JIFFY entworfenen Konzepte trotz der Spezialisierung auf die JVM in weiteren, gerade sich erst in Entwicklung befindlichen Einsatzgebieten wiederzuverwenden.

The best way to predict the future is to invent it.
Alan Kay

A

Anhang

A.1 Paracodebeschreibung

Der Paracode besteht aus eine Reihe von Codeblöcken, die im Assemblerformat der Zielmaschine je eine IJVM-Operation beschreiben. Als Assemblersyntax wurde die AT&T/GNU-Syntax gewählt, die nur bei x86-Assembler größere Unterschiede zur Herstellersyntax aufweist[1].

Die Zuordnung der Platzhalter für Register erfolgt dabei in der Reihenfolge der Registerzuordnung der IJVM-Befehle. Damit ist z.B. bei ALU-Operationen das Pseudoregister %r1 immer das Zielregister.

Die Paracode-spezifischen Syntax in BNF-Notation ist in Tabelle A.1 beschrieben.

Tabelle A.1: Paracode-Syntax in BNF-Notation

Paracode	:=	Kommentar | Codeblock;
Kommentar	:=	**;** ...
Codeblock	:=	IJVM-Label
		{ Registerzuordnung}
		Assemblercode
		.end
IJVM-Label	:=	″__″IJVM-Opcode
Registerzuordnung	:=	**.para** IJVM-Platzhalter
IJVM-Platzhalter	:=	Registerplatzhalter |
		Konstantenplatzhalter |
		Offsetplatzhalter |
		LV-Platzhalter

[1] Dies sind hauptsächlich Vertauschung von Ziel und Quelle und Prozentzeichen vor Registern.

Registerplatzhalter	:=	Integerregister | Floatregister
Integerregister	:=	**%r1** |**%r2** |**%r3**
Floatregister	:=	**%f1** |**%f2** |**%f3**
Konstantenplatzhalter	:=	**%c1**
Offsetplatzhalter	:=	**%o1**
LV-Platzhalter	:=	4Byte-Offset | 4Byte-Offset-High | 1Byte-Offset | 1Byte-Offset-High
4Byte-Offset	:=	**%v1**
4Byte-Offset-High	:=	**%w1**
1Byte-Offset	:=	**%x1**
1Byte-Offset-High	:=	**%y1**
Assemblercode	:=	[Lokales-Label] | Assemblerbefehl
Lokales-Label	:=	″*****″digit
Assemblerbefehl	:=	Opcode [Parameter { **,** Parameter}]
Parameter	:=	Registerplatzhalter[Size] | Lokales-Label | Konstantenplatzhalter | Offsetplatzhalter | LV-Platzhalter | Struktur-Offset | Skalierte-Konstante | Externe-Konstante | Feste-Assemblerparameter
Size	:=	**.l** | **.h** | **.b** | **.w**
Struktur-Offset	:=	″**@**″Struktur″**.**″Element
Skalierte-Konstante	:=	**%s1**
Externe-Konstante	:=	**%a4** |**%a5** |. . . |**%a15**

Als Beispiel für die Paracode-Syntax ist die Umsetzung des IJVM-Befehls `sgn_i` (Integer-Signum) für 80586 und Alpha in Abbildung A.1 dargestellt.

Die Zuordnung der externen Konstanten, die aus der Software übernommen werden, wird in Tabelle A.2 gezeigt.

```
_sgn_i
        .para %r1
        .para %r2
        orl %r2,%r2
        jl *2
        jg *1
        xorl %r1,%r1
        jmp *3
*1
        movl $1,%r1
        jmp *3
*2
        movl $-1,%r1
*3
        .end
```

```
_sgn_i
        .para %r1
        .para %r2
        bis $31,$31,%r1
        addl $31,%r2,%r2
        cmovgt %r2,1,%r1
        cmovlt %r2,255,%r1
        .end
```

Abbildung A.1: Beispiel der Paracode-Syntax

%a4	JIT_NR_LVARS	Anzahl der lokalen Variablen
%a5	JIT_NR_ARGS	Anzahl der Argumente
%a6	JIT_NR_ARGS4	Anzahl der Argumente*4
%a7	JIT_ICP	Zeiger auf internen CP
%a8	JIT_OBV	Zeiger auf interne Objekteigenschaften
%a9	JIT_LVARS4	Anzahl der nicht-Argument LV
%a10	JIT_NR_JVMARGS	Anzahl der Argumentslots
%a11	JIT_SC	Zeiger auf “Start-of-Code”
%a12	JIT_DISP	Zeiger auf Function Dispatcher
%a13	JIT_XOBV	Zeiger auf erweiterte Objekteigenschaften

Tabelle A.2: Externe Paracode-Konstanten

A.2 Beispielablauf der Übersetzung

Im folgenden wird die Übersetzung anhand der rekursiven Fibonaccifunktion erläutert.

Ausgangscode in Java Der Ausgangscode in Java sieht dabei wie folgt aus:

```
public static int fib(int n)
{
    if (n<=1)
        return 1;
    else
        return fib(n-1)+fib(n-2);
}
```

JVM-Bytecode Der Java-Compiler (javac) erzeugt folgenden JVM-Bytecode, der als Grundlage für die weitere Übersetzung dient:

```
 0  iload_0
 1  iconst_1
 2  if_icmpgt 7
 5  iconst_1
 6  ireturn
 7  iload_0
 8  iconst_1
 9  isub
10  invokestatic 4
13  iload_0
14  iconst_2
15  isub
16  invokestatic 4
19  iadd
20  ireturn
```

Der rekursive Aufruf der statischen Methode mittels `invokestatic 4` ist hier gut zu erkennen. Hierbei ist der Parameter 4 der Zeiger (Index) auf die passende Methodenbeschreibung von `fib` (Name und Signatur)im Konstantenpool.

JVM→IJVM Die erste Stufe der JVM→IJVM-Übersetzung erzeugt dabei den folgenden, noch kaum optimierten Code, bei dem zunächst nur triviale PUSH/POP-Paare entfernt wurden. Die invokestatic-Aufrufe sind bereits durch typisierte Aufrufe ersetzt worden.

```
 0 start_func          17 sub_i %r1,%r2
 1 movesvr_i 0,%r1     18 push_i %r2
 2 push_i %r1          19 invokestatic 4
 3 movec_i 1,%r1       20 push_i %r1
 4 push_i %r1          21 movesvr_i 0,%r1
 5 pop_i %r2           22 push_i %r1
 6 pop_i %r1           23 movec_i 2,%r1
 7 cmp_i %r1,%r2       24 pop_i %r2
 8 bigt L7             25 sub_i %r1,%r2
 9 movec_i 1,%r1       26 push_i %r2
10 retdata_x %r1       27 invokestatic 4
11 returnnorm          28 push_i %r1
12 label 7             29 pop_i %r1
13 movesvr_i 0,%r1     30 pop_i %r2
14 push_i %r1          31 add_i %r2,%r1
15 movec_i 1,%r1       32 retdata_x %r1
16 pop_i %r2           33 returnnorm
                       34 END_FUNC
```

Peepholeoptimierung Die Anwendung der Peepholeoptimierung entfernt fast alle PUSH/POP-Operationen. Diejenigen PUSH/POP-Befehle, die noch übrigbleiben, sind in diesem Beispiel aufgrund der rekursiven Struktur des Programms auch nicht zu eliminieren.

```
0 start_func          10 invokestatic 4
1 movesvr_i 0,%r1     11 push_i %r1
2 cmpc_i 1,%r1        12 movesvr_i 0,%r2
3 bigt L7             13 subc_i 2,%r2
4 movec_i 1,%r1       14 push_i %r2
5 returnnorm          15 invokestatic 4
6 label 7             16 pop_i %r2
7 movesvr_i 0,%r2     17 add_i %r2,%r1
8 subc_i 1,%r2        18 returnnorm
9 push_i %r2          19 END_FUNC
```

Die Optimierung zeigt die Entfernung fast aller PUSH-POP-Befehle sowie die Zusammenziehung von arithmetischen Befehlen mit Konstanten (Zeile 1-7/13-17/21-25 im unoptimierten Code, Zeile 1-2/7-8/12-13 im optimierten Code). Der Länge des IJVM-Codes wird dadurch um die Hälfte verkürzt.

Weiterhin zeigt sich der Vorteil der `label`-Pseudobefehle, da Optimierungen zwar die absolute Position der Sprungziele verändern, aber keinerlei Einfluß auf das Vorkommen der Labels haben.

IJVM→NAT Aus dem optimierten IJVM-Code wird in der letzten Phase durch die Paracodeassemblierung 80386-Assemblercode generiert.

```
 0:   55                      push   %ebp
 1:   8b ec                   mov    %esp,%ebp
 3:   68 00 85 12 08          push   $0x8128500
 8:   8b 45 08                mov    0x8(%ebp),%eax
 b:   81 f8 01 00 00 00       cmp    $0x1,%eax
11:   0f 8f 0f 00 00 00       jg     0x26
17:   b8 01 00 00 00          mov    $0x1,%eax
1c:   c9                      leave
1d:   59                      pop    %ecx
1e:   81 c4 04 00 00 00       add    $0x4,%esp
24:   51                      push   %ecx
25:   c3                      ret
26:   8b 5d 08                mov    0x8(%ebp),%ebx
29:   81 eb 01 00 00 00       sub    $0x1,%ebx
2f:   53                      push   %ebx
30:   bb 00 85 12 08          mov    $0x8128500,%ebx
35:   ba a8 27 12 08          mov    $0x81227a8,%edx
3a:   33 c9                   xor    %ecx,%ecx
3c:   b8 04 00 00 00          mov    $0x4,%eax
41:   ff 92 10 00 00 00       call   *0x10(%edx)
47:   50                      push   %eax
49:   8b 5d 08                mov    0x8(%ebp),%ebx
4b:   81 eb 02 00 00 00       sub    $0x2,%ebx
51:   53                      push   %ebx
52:   bb 00 85 12 08          mov    $0x8128500,%ebx
57:   ba a8 27 12 08          mov    $0x81227a8,%edx
5c:   33 c9                   xor    %ecx,%ecx
6e:   b8 04 00 00 00          mov    $0x4,%eax
63:   ff 92 10 00 00 00       call   *0x10(%edx)
69:   5b                      pop    %ebx
6a:   03 c3                   add    %ebx,%eax
6c:   c9                      leave
6d:   59                      pop    %ecx
6f:   81 c4 04 00 00 00       add    $0x4,%esp
74:   51                      push   %ecx
75:   c3                      ret
```

Zu Beginn der Funktion erfolgt das Aufsetzen des Stackrahmens, der zusätzlich neben dem alten Basepointer auch noch den Wert des “Start-of-Code” auf dem Stack sichert, der bei der Rückabwicklung von Exceptions benötigt wird.

Im Endergebnis auffällig sind die Registerbelegungen vor dem `call`-Befehl. Diese entsprechen den in Abschnitt 7.1.4.1 erwähnten Zusatzinformation. Die-

se werden an sich nur für den ersten Methodenauflösung gebraucht, müssen im späteren Verlauf auch weiterhin durchlaufen werden. Register eax entspricht dem CP-Index der Methode (4), ebx ist ein Zeiger auf den “Start-of-Code”, ecx ist der Typ der verlangten Methode (0=statisch), und edx ist die Startadresse der JIFFY-eigenen Objekttabelle. Der `call`-Befehl selbst springt auf die Objektfunktion #4, der Offset wird dazu im Paracodeassembler bereits mit 4 skaliert.

Literaturverzeichnis

[1] Georg Acher. JIFFY: Portierung eines JIT-Compilers auf FPGAs. In C. H. Cap, editor, *Proceedings Java-Informations-Tage 1999*, September 1999.

[2] Ali-Reza Adl-Tabatabai, Michael Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time java compiler.

[3] Adobe Inc. PostScript Language Reference. `http://partners.adobe.com/asn/developer/technotes/postscript.html` .

[4] Advanced Microdevices, Inc. AMD Alchemy Processor Family. `http://www.amd.com/us-en/ConnectivitySolutions/ProductInformation/0,,50_2330_6625,00.html` .

[5] Advanced Microdevices, Inc. Élan SC520 Microcontroller. `http://www.amd.com/epd/processors/4.32bitcont/14.lan5xxfam/24.lansc520/index.html` .

[6] A. Aho, R. Sethi, and J.D. Ullman. *Compilerbau*. Addison Wesley, 1992.

[7] Ajile Inc. aJ-100 TM Real-time Low Power Java TM Processor Processor Datasheet, 2001. `http://www.ajile.com/downloads/aJ100Datasheet_1.3.pdf` .

[8] Ajile Inc. aJile Java Processor Core JEMCore, 2001. `http://www.ajile.com/downloads/jemcoreproductbrief.pdf` .

[9] Alliance Semiconductor Corporation. AS6VA5128, 2.7V to 3.3V 512K x 8 Intelliwatt low-power CMOS SRAM. `http://www.alsc.com/pdf/sram.pdf/lp/AS6VA5128v.1.2.pdf` .

[10] Altera Inc. Datasheet 10k20, 1997. `http://www.altera.com` .

[11] Altera Inc. Nios embedded processor system development, 2001. `http://www.altera.com/products/devices/nios/nio-index.html` .

[12] Erik Altman, David Kaeli, and Yaron Sheffer. Welcome to the opportunities of binary translation. *Computer*, March 2000. `http://citeseer.nj.nec.com/altman00welcome.html` .

[13] G. M. Amdahl. Validity of single-processor-approach to achieve large-scaling computing capability. In *Proceedings of AFIPS 1967*, pages 483–485, 1967.

[14] Amulet Group, University of Manchester. JAMAICA Project. http://www.cs.man.ac.uk/amulet/projects/jamaica/jamaica.html .

[15] Apple Inc. Technical Note PT39: The DR Emulator, February 1995. http://developer.apple.com/technotes/pt/pdf/pt_39.pdf .

[16] ARM Inc. Arm Jazelle Technology, 2001.

[17] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeno JVM. In *Conference on Object-Oriented*, pages 47–65, 2000. http://citeseer.nj.nec.com/arnold00adaptive.html .

[18] Pedro V. Artigas, Manish Gupta, Samuel P. Midkiff, and Jose E. Moreira. Automatic loop transformations and parallelization for java. In *International Conference on Supercomputing*, pages 1–10, 2000. http://citeseer.nj.nec.com/artigas00automatic.html .

[19] Atmel. Datasheet AT40K20, 1997. http://www.atmel.com .

[20] Atmel. PC8260 PowerQUICCII TM Microprocessor Fact Sheet, 2002. http://www.atmel.com/atmel/acrobat/pc8260fs.pdf .

[21] Aurora VLSI Inc. DeCaf - The Wireless Java Core Specification, October 2001. http://vodka.auroravlsi.com/website/product_briefs/DeCaf.pdf .

[22] Aurora VLSI, Inc. DeCaf-M: The Wireless Bilingual Legacy Instruction Set + Java Core Specification, October 2001. http://vodka.auroravlsi.com/website/product_briefs/DeCaf-M.pdf .

[23] Aurora VLSI, Inc. Espresso - The High Performance Java Core Specification, October 2001. http://vodka.auroravlsi.com/website/product_briefs/Espresso.pdf .

[24] Axis Semiconductor. Axis Etrax100LX Programmer's Manual, 2000. http://www.axis.com .

[25] Axis Semiconductor. Axis Etrax100LX Designer's Reference Manual, 2001.

[26] V. Bala, E. Duesterwald, and S. Banerjia. Transparent Dynamic Optimization. Report HPL-1999-77, HP Laboratories Cambridge, June 1999. http://citeseer.nj.nec.com/bala99transparent.html .

[27] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000. http://citeseer.nj.nec.com/bala00dynamo.html .

[28] G. Bernat, A. Burns, and A. Wellings. Portable Worst-Case Execution Time Analysis Using Java Byte Code. In *Proceedings of the 12th EuroMicro Conference on Real-Time Systems, Stockholm.*, June 2000. http://citeseer.nj.nec.com/bernat00portable.html .

[29] Bernd Paysan. GForth Home Page. http://www.jwdt.com/ paysan/gforth.html .

[30] Aart Bik, Milind Girkar, and Mohammad Haghighat. JIT Compilation of Java for the Intel Architecture. Presentation.

[31] Kiran Kumar Bondalapati. *Modelung and Mapping for Dynamically Reconfigurable Hybrid Architectures*. PhD thesis, 2001. http://www.lsc.ic.unicamp.br/people/renon/files/articles/thesis.pdf .

[32] David Brook. Real Time Embedded Operating Systems. In *Proceedings Embedded Intelligence '99*, pages 166–173. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[33] Z. Budimlic and K. Kennedy. Static interprocedural optimizations in java. In *Rice University technical report CRPC-TR98746, 1998.*, 1998. http://citeseer.nj.nec.com/budimlic98static.html .

[34] Junpyo Lee Byung-Sun. Reducing Virtual Call Overheads in a Java VM Just-in-Time Compiler. http://citeseer.nj.nec.com/302904.html .

[35] T. Callahan and J. Wawrzynek. Datapath-oriented FPGA Mapping and Placement for Configurable Computing. *Proceedings of FCCM'97*, 1997.

[36] Cereus Design Corp. HotTEA Whitepaper. http://www.cereus7.com/wpaper1.html .

[37] Chicory Systems. HotShot Java Accelerator. http://www.mips.com/devTools/catalog_2001/catalog_files/silicon/chicor02.pdf .

[38] Yuan C. Chou and John Paul Shen. Instruction path coprocessors. In *ISCA*, pages 270–281, 2000. http://citeseer.nj.nec.com/article/chou00instruction.html .

[39] Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout - a retargetable dynamic binary translation framework. January 2002. http://research.sun.com/techrep/2002/smli_tr-2002-106.pdf .

[40] Compaq Inc. Compaq DIGITAL FX!32 Whitepaper.
http://www.support.compaq.com/amt/fx32/fx-white.html .

[41] Embedded Microprocessor Benchmark Consortium.
http://www.eembc.org/ .

[42] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.specbench.org/osg/jvm98/ .

[43] Electronic Design. Embedded Hardware Directory, Java Series. December 2001. http://www.elecdesign.com/2001/dec0301/p76.pdf .

[44] Digital Equipment Corporation. *DEC OSF/1 Calling Standard for AXP Systems*, February 1994.

[45] Digital Equipment Corporation. *Alpha Architecture Handbook*, October 1996. Version 3.

[46] Paul J. Drongowski, David Hunter, Morteza Fayyazi, and David Kaeli. Studying the performance of the fx!32 binary translation system.

[47] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility, 1996. IBM Research Report RC20538.

[48] M. Edwards. Software Acceleration Using Coprocessors: Is it Worth the Effort? In *Proceedings of the 5th International Workshop on Hardware/Software Co-Design CODES/CASHE'97*. IEEE, 1997.

[49] Ahmed H.M.R. El-Mahdy. *A Vector Architecture for Multimedia Java Applications*. PhD thesis.
http://www.cs.man.ac.uk/~elmahdya/thesis/Title.html .

[50] M. Anton Ertl. A new approach to Forth native code generation. In *Proceedings euroforth92*, pages 73–78, 1992.
http://www.complang.tuwien.ac.at/papers/ertl92.ps.gz .

[51] M. Anton Ertl, David Gregg, Andreas Krall, and Bernd Paysan. vmgen — a generator of efficient virtual machine interpreters. *Software—Practice and Experience*, 32(3):265–294, 2002.
http://www.complang.tuwien.ac.at/papers/ertl+02.ps.gz .

[52] M. Anton Ertl and Christian Pirker. Compilation of stack-based languages (Abschlußbericht). Final report to FWF for research project P11231, Institut für Computersprachen, Technische Universität Wien, 1998.
http://www.complang.tuwien.ac.at/papers/ertl&pirker98.ps.gz .

[53] J. Vuillemin et al. Programmable active memories: Reconfigurable systems come of age. In *IEEE Transactions on VLSI, Vol. 4, No. 1*, March 1996.

[54] Excelsior Inc. Excelsior JET: The Java Performance Solution, Technical Whitepaper. http://www.excelsior-usa.com/doc/jetwp.html .

[55] David Flanagan. *Java in a Nutshell*. O'Reilly & Associates, 1996.

[56] Forth Interest Group. Home Page. http://www.forth.org/ .

[57] Free Software Foundation. The GNU Compiler for the Javatm Programming Language. http://gcc.gnu.org/java/ .

[58] FSMLabs. RTLinux. http://fsmlabs.com/ .

[59] Fujitsu Limited. picoJava-IITM Core Based Chip, 2001. http://edevice.fujitsu.com/fj/MARCOM/javae/javasolu/java10e.html .

[60] Eugene Gluzberg and Stephen Fink. An Evaluation of Java System Services with Microbenchmarks. In *IBM Research Report, RC 21715*, March 2000. http://citeseer.nj.nec.com/gluzberg00evaluation.html .

[61] James Gosling. Oak Intermediate Bytecodes, 1995. http://java.sun.com/people/jag/OakIntermediateRepPaper.ps .

[62] R. Grafl. CACAO - Ein 64bit-JavaVM-Just-In-Time-Compiler, 1997. Diplomarbeit.

[63] D. Gregg and M. Ertl. A fast java interpreter. In Uwe Assmann, editor, *JOSES Workshop at ETAPS'01*, 2001. http://citeseer.nj.nec.com/gregg01fast.html .

[64] David Gregg. Platform Independent Dynamic Java Virtual Machine Analysis: the Java Grande Forum Benchmark Suite, June 2001. http://citeseer.nj.nec.com/503256.html .

[65] Michael Gudaitis and Joseph Mitola. The radio virtual machine, 2000. http://www.sdrforum.org/MTGS/wde_wkshp_11_00/radio_vm_final_11_30_00.pdf .

[66] J. Hadley and B.Hutchings. Design methodologies for partially reconfigured systems, 1995.

[67] HALWG, Hardware Abstraction Layer Working Group. Homepage. http://www.sdrforum.org/tech_comm/halwg.html .

[68] Hitachi Semiconductor. SH7708 Series Hardware Manual, May 1999.

[69] Jan Hoogerbrugge and Lex Augusteijn. Pipelined java virtual machine interpreters. In *Computational Complexity*, pages 35–49, 2000.
http://citeseer.nj.nec.com/hoogerbrugge00pipelined.html .

[70] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: Combining Emulation and Binary Translation. *Digital Technical Journal*, 9, No.1, 1997.
http://research.compaq.com/wrl/DECarchives/DTJ/DTJP01/DTJP01PF.PDF .

[71] C. Hsieh, J. Gyllenhaal, and W. Hwu. Java Bytecode to native Code Translation: The Caffeine Prototype and Preliminary Results, 1996.

[72] IBM. PowerPC Assembler Language Reference.
http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixassem/alangref/architect.htm .

[73] Infineon Technologies. HYB 39S128400/800/160CT(L) 128-MBit Synchronous DRAM.

[74] INMOS/SGS Thomson. IMS T425, 32Bit Transputer.
http://www.classiccmp.org/transputer/documentation/inmos/4257.pdf .

[75] Insignia Solutions. Jeode Platform White Paper, 2001.
http://spacejug.org/resources/Java_VMs/Jeode_VM/Jeode_VM.pdf .

[76] InSilicon Inc. JVXtreme Accelerator.
http://www.insilicon.com/products/images/jvxtreme.pdf .

[77] Intel Corporation. Pentium Pro Users Manual.

[78] Nelly Jacqueson. Windows CE for Industrial Computing. In *Proceedings Embedded Intelligence '99*, pages 532–542. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[79] Udo Jakobza. StrongARM-1100 Board für universelle Anwendungen. In *Proceedings Embedded Intelligence '99*, pages 352–362. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[80] Jean-Loup Gailly. The gzip home page. http://www.gzip.org .

[81] Kaffe, a clean room implementation of the Java virtual machine.
http://www.kaffe.org .

[82] Wolfgang Karl. Mikroprozessoren für eingebettete Systeme.
http://wwwbode.cs.tum.edu/ karlw/Karlsruhe/emp-ws01-04.pdf .

[83] Tarique Kazi. Improving performance on incremental compilation of java bytecodes. http://citeseer.nj.nec.com/189387.html .

[84] Alexander Klaiber. The Technology Behind Crusoe Processors. January 2000.
http://www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf .

[85] A. Koch. Structured Design Implementation - A Strategy for Implementing Regular Datapaths on FPGAs. *Proceedings of FPGA'96*, 1996.

[86] Philip Koopman. *Stack Computers – The New Wave*. Mountain View Press, 1989.

[87] Andreas Krall, Anton Ertl, and Michael Gschwind. JavaVM implementation: Compilers versus hardware. In *Proceedings of Computer Architecture ACAC'98*, pages 101–110, 1998.
http://citeseer.nj.nec.com/248364.html .

[88] Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. Reducing the overhead of dynamic compilation. *Software - Practice and Experience*, 31(8):717–738, 2001. http://citeseer.nj.nec.com/krintz01reducing.html .

[89] Markus Levy. Java to go: Part 1. *Microprocessor Report*, February 12, 2001.

[90] Markus Levy. Java to go: Part 2. *Microprocessor Report*, March, 2001.
http://klamath.stanford.edu/ sundaes/PMCS/microreport.pdf .

[91] Markus Levy and Alan R. Weiss. Opportunities and pitfalls of benchmarking embedded processors. In *Proceedings Embedded Intelligence '99*, pages 638–646. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[92] Tao Li, Lizy Kurian John, Vijaykrishnan Narayanan, and Anand Sivasubramaniam. Branch Behavior of Java Runtime Systems and its Microarchitectural Implications. In *Technical Report TR-000625-01, Department of Electrical and Computer Engineering, University of Texas at Austin*, June 2000. http://citeseer.nj.nec.com/506898.html .

[93] Jan Liband. Compiling Java Code For Embedded Systems: The FastJ Solution. In *Proceedings Embedded Intelligence '99*, pages 647–672. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[94] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[95] Peter Luksch. Rechnerarchitektur, Vorlesungsunterlagen.
http://wwwbode.cs.tum.edu/ luksch/archiv/Skripten/W2001/Rechnerarchitektur/Slides16.pdf .

[96] Qusay Mahmoud. The J2ME Platform.
http://wireless.java.sun.com/midp/articles/api .

[97] J. Meyer. Jasmin Home page. http://mrl.nyu.edu/ meyer/jvm/ .

[98] Microsoft Inc. C# Language Specification, March 2002.

[99] Microsoft Inc. Comparison Between C++ and C#, 2002.
http://msdn.microsoft.com/library/default.asp?url=
/library/en-us/cscon/html/vclrfcomparisonbetweenccsharp.asp .

[100] MIPS Technologies, Inc. MIPS Core Selection Guide.
http://www.mips.com/products/index.html .

[101] MIPS Technologies, Inc. MIPS324KSc Embedded MIPS Processor Core.
http://www.mips.com/products/s2p9.html .

[102] Jose E. Moreira, Samuel P. Midkiff, Manish Gupta, Pedro V. Artigas, Peng Wu, and George Almasi. The NINJA project. *Communications of the ACM*, 44(10):102–109, 2001.
http://citeseer.nj.nec.com/moreira01ninja.html .

[103] Moshe Sipper et al. The Firefly Machine: Online Evolware. In *Proceedings of the ICEC97*, April 1997.

[104] Motorola Inc. MCM6246, 512K x 8 Bit Static Random Access Memory.

[105] Motorola Inc. MCF5407 Integrated ColdFire Microprocessor Product Brief, August 1996.
http://e-www.motorola.com/brdata/PDFDB/docs/MCF5XXXWP.pdf .

[106] Motorola Inc. Motorola MC68328 User's Manual, 1999.

[107] Motorola Inc. Motorola ColdFire VL RISC Processors White Paper, 2001.
http://e-www.motorola.com/brdata/PDFDB/docs/MCF5XXXWP.pdf .

[108] NASA. Space-Related Applications of Forth.
http://forth.gsfc.nasa.gov/ .

[109] National Semiconductor. Geode Products.
http://www.national.com/appinfo/solutions/0,2062,396,00.html .

[110] Nazomi Communications, Inc. Constant pool reference resolution method. US-Patent 6,338,160.

[111] Nazomi Inc. Jstar Nazomi Datasheet, 2001.

[112] Heise Newsticker. Microsofts Java verrechnet sich, 1998.
http://www.heise.de/newsticker/data/ju-28.10.98-00 .

[113] Heise Newsticker. Crusoe: Nicht der Schnellste, aber sparsam, 2000. http://www.heise.de/newsticker/data/jow-11.10.00-000 .

[114] Oberon . A Brief History of Pascal. http://www.oberon.ch/resources/component_pascal/history.html .

[115] J. O'Connor and M. Tremblay. Picojava-1: The java virtual machine in hardware. *IEEE-Micro*, March/April 1997.

[116] Hirotaka Ogawa, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, Kouya Shimura, and Yasunori Kimura. Openjit frontend system: An implementation of the reflective JIT compiler frontend. In *OORaSE*, pages 135–154, 1999. http://citeseer.nj.nec.com/311633.html .

[117] OpenJIT Team. OpenJIT Homepage. http://www.openjit.org/ .

[118] O'Reilly & Associates, Inc. . perl.com – The Source for Perl. http://www.perl.com .

[119] Øyvind Strøm and Einar Aas. A novel microprocessor architecture for executing byte compiled java code. In *Proceedings of ICDA 2000*, 2000.

[120] Parthus Inc. MachStream Technology Backgrounder, 2001. http://www.parthus.com/platforms/parthus_machstream/machstream_technology_backgrounder.pdf .

[121] S. Patel and S. Lumetta. rePLay : A hardware framework for dynamic program optimization. In *Technical Report CRHC-99-16, University of Illinois Technical Report, Dec. 1999.*, 1999. http://citeseer.nj.nec.com/patel99replay.html .

[122] Patriot Scientific. Java Processor PSC1000. *elektronik industrie*, Heft 2:S. 51ff, 1998.

[123] Andres Perez-Uribe and Eduardo Sanchez. Structure-adaptable neurocontrollers: A hardware-friendly approach. In *Proceedings of IWANN97*, June 1997.

[124] Pioneer-Standard Electronics. Windows CE Overview, 1999. http://mypioneer.com/msembedded/ce/overview.asp .

[125] Roldan Pozo and Bruce Miller. Scimark. http://math.nist.gov/scimark2/ .

[126] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, 2000. http://citeseer.nj.nec.com/prechelt00empirical.html .

[127] Programmable Electronics Performance Corp., PREP.
http://www.prep.org .

[128] Mike Quelch. The Evolution of JAVA for Embedded Systems. In *Proceedings Embedded Intelligence '99*, pages 622–637. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[129] Ramesh Radhakrishnan, Ravi Bhargava, and Lizy K. John. Improving Java Performance Using Hardware Translation. pages 427–439, 2001.

[130] Bozidar Radunovic and Veljko Milutonovic. A survey of reconfigurable computing architectures. In *Workshop at FPL'98*, 1998.

[131] Michael Riß. Entwicklung eines Hardwareübersetzers für die Java Virtual Machine (JVM), 2002. Diplomarbeit.

[132] Rockwell Collins, Inc. Real time processor optimized for executing JAVA programs. US-Patent 6,317,872.

[133] Rockwell International Corporation. Rockwell Avionics and Communications Announces Java Milestone, September 1997. Press Release.

[134] Bradford J. Rodriguez and W. F. S. Poehlman. A Survey of Object-Oriented Forths. 31(4), April 1996.
http://www.zetetics.com/bj/papers/oofs.htm .

[135] Manfred Schlett. SH-4: 200MHz Superscalar Embedded Processor. In *Proceedings Embedded Intelligence '99*, pages 249–258. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[136] Patriot Scientific. PSC1000 Microprocessor Datasheet.
http://spacejug.org/resources/Embedded_Java/Patriot_Scientific/PSC1000_Java.pdf .

[137] Mauricio J. Serrano, Rajesh Bordawekar, Samuel P. Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for Java. In *Conference on Object-Oriented*, pages 66–82, 2000.
http://citeseer.nj.nec.com/serrano00quicksilver.html .

[138] Sharp Electronics Corporation. Sharp Zaurus Personal Organizer.
http://www.zaurus.com/ .

[139] Kazuyuki Shudo. Performance comparison of JITs (Jan 2002).
http://www.shudo.net/jit/perf/ .

[140] Todd Smith, Suresh Srinivas, Philipp Tomsich, and Jinpyo Park. Practical experiences with java compilation. In *HiPC*, pages 149–157, 2000. http://citeseer.nj.nec.com/408569.html .

[141] Sun Inc. CLDC HotSpot TM Implementation White Paper. http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf .

[142] Sun Inc. CLDC, The Inner Plumbing of the JavaTM 2 Platform, Micro Edition. http://java.sun.com/products/cldc/ .

[143] Sun Inc. Java Look and Feel Design Guidelines, Chapter 6: Responsiveness. http://java.sun.com/products/jlf/at/book/Responsiveness.html .

[144] Sun Inc. MAJC Architecture Tutorial. http://spacejug.org/resources/Embedded_Java/MAJC_Processor/majctutorial.pdf .

[145] Sun Inc. OpenBoot 3.x Quick Reference. http://www.sun.com/products-n-solutions/hardware/docs/pdf/806-2908-10.pdf .

[146] Sun Inc. picoJava-II Processor. http://spacejug.org/resources/Embedded_Java/picoJava/picoJava-II.pdf .

[147] Sun Inc. Can HotSpot jumpstart your Java applications. *Sun World*, June 1999.

[148] Sun Inc. The Java HotSpot Virtual Machine – Technical White Paper, 2001. http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.pdf .

[149] Sun Microsystems Inc. Optimizing symbol table lookups in platform-independent virtual machines. US-Patent 6,446,084.

[150] Sun Microsystems Inc. Sun Microelectronics' picoJava 1 Posts Outstanding Performance, November 1996. Press Release.

[151] Suman Nath Suraj Sudhir and Seth C. Goldstein. Configuration caching and swapping. In *Proceedings of FPL2001*, August 2001.

[152] Kent Tallyn. Reprogrammable Missile: How an FPGA Adds Flexibility to the Navy's Tomahawk. *Military and Aerospace Electronics*, April 1990. Reprinted in: Xilinx, Programmable Gate Array Data Book 1992.

[153] Micron Technology. MT4C1M16C3 , 1M*16 DPM DRAM Datasheet.

[154] TechTV Inc. Performance Comparison Chart, 2002. http://www.techtv.com/freshgear/products/jump/0,23009,3398105,00.html .

[155] Texas Instruments. TMS320AV7200 Preliminary Specification, February 2002.

[156] The Web Server Times. How the Buffer-Stack Overflow Hacking Technique Compromises Web Security.
http://www.webservertimes.com/network-security-management-3.html .

[157] Lothar Thiele. Unterlagen zur Lehrveranstaltung Technische Informatik 1, ETH Zürich.

[158] Marc Tremblay. MAJC-5200: A VLIW Convergent MPSOC.
http://spacejug.org/resources/Embedded_Java/MAJC_Processor/MPForum99-d.pdf .

[159] Trolltech. The Qt/Embedded Toolkit.
http://www.trolltech.com/products/embedded/index.html .

[160] Vantis. Datasheet Vantis VF1-family, 1997. http://www.vantis.com (now merged with Lattice Semiconductors).

[161] N. Vijaykrishnan and N. Ranganathan. Tuning branch predictors to support virtual method invocation in java. In *Proceedings of the 5th USENIX Conference of Object-Oriented Technologies and Systems, pages 217–228, 1999.*, 1999.

[162] J. Villasenor and W. Mangione-Smith. Configurable computing. *Scientific American*, June 1997.

[163] John Waldron. Analysis of Virtual Machine Stack Frame Usage by Java Methods. In *Internet, Multimedia Systems and Applications*, pages 271–274, 1999. http://citeseer.nj.nec.com/275807.html .

[164] John Waldron and James Power. Comparison of Bytecode and Stack Frame Usage by Eiffel and Java Programs in the Java Virtual Machine. In *Workshop on Computer Science and Information Technologies CSIT'2000, Ufa, Russia*, 2000.
http://citeseer.nj.nec.com/article/waldron00comparison.html .

[165] Terry West. Embedded Java: New Options and Challenges. In *Proceedings Embedded Intelligence '99*, pages 699–721. Design&Elektronik, WEKA Fachzeitschriftenverlag, March 1999.

[166] Winbond USA. W28J800B/T 8M(512kx8,1Mx8) Boot Block Flash Memory.
http://www.winbond-usa.com/products/winbond_products/pdfs/Memory/w28j800bt.pdf .

[167] Wind River Systems, Inc. Zinc – The Cross-Platform GUI Toolkit.
http://www.zinc.com .

[168] WindRiver Inc. JWorks Delivers on the Promises Of Java in Embedded Systems. http://www.windriver.com/products/jworks/jworks_wp.pdf .

[169] WindRiver Inc. VxWorks 5.x Operating Systems. http://www.windriver.com/products/vxworks5/vxworks_54.pdf .

[170] Xilinx Inc. MicroBlaze Soft Processor. http://www.xilinx.com/xlnx/xil_prodcat_product.jsp?title=microblaze .

[171] Xilinx Inc. XC3000 Field Programmable Gate Array Data Sheet, 1997. http://www.xilinx.com .

[172] Xilinx Inc. XC4000 Field Programmable Gate Array Data Sheet, 1997.

[173] Xilinx Inc. XC6200 Field Programmable Gate Array Data Sheet, 1997.

[174] Xilinx Inc. Spartan2 Data Sheet, 2001.

[175] Xilinx Inc. Virtex2Pro Data Sheet, 2001.

[176] Byung-Sun Yang, Soo-Mook Moon, Seongbae Park, Junpyo Lee, SeungIl Lee, Jinpyo Park, Yoo C. Chung, Suhyun Kim, Kemal Ebcioglu, and Erik R. Altman. LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation. In *IEEE PACT*, pages 128–138, 1999. http://citeseer.nj.nec.com/yang99latte.html .

[177] Frank Yellin. The Java Native Code API, 1996.

[178] Zucotto Inc. XPRESSOcore 100 processor (XC-100) Description. http://www.zucotto.com/products/xpresso_100.html .

[179] Zucotto Inc. XPRESSOcore Featuresheet. http://www.zucotto.com/products/XPRESSOcoreFeatureSheet.pdf .

I

Index