Portable Ausführung von Altanwendungen durch Laufzeitkompilierung zu Java Bytecode

---

Studienarbeit im Fach Informatik

vorgelegt von

Michael Stilkerich

geb. am 25. April 1982 in Forchheim

Angefertigt am

Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)

Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer: Prof. Dr.-Ing. W. Schröder-Preikschat
Dipl. Inf. Christian Wawersich

Beginn der Arbeit: 15. November 2004
Abgabe der Arbeit: 30. Juni 2005

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

Erlangen, den 30. Juni 2005 _____

Zusammenfassung

Diese Arbeit stellt eine Möglichkeit zur Ausführung von Altanwendungen in einer typsicheren Java Virtual Machine (JVM) vor. Der Binärcode der Gastarchitektur wird zur Laufzeit zu Java Bytecode „hochkompiliert" und dann durch den JIT Übersetzer der JVM wieder zu Binärcode der Hostarchitektur „herunterkompiliert". Die prototypische Anwendung JXEmu ermöglicht also die Ausführung unsicherer Altanwendungen in der sicheren Umgebung einer JVM.

JXEmu unterstützt mehrere Gastarchitekturen, darunter auch die ARM Architektur, deren Modul als Teil dieser Arbeit entwickelt wurde. Anstatt die Gastarchitektur durch die direkte Erzeugung von Java Bytecode zu emulieren, erfolgt die Spezifikation einer Gastarchitektur durch eine Sammlung aus Codestücken, den sog. *Code-Templates*, die in der Java Programmiersprache geschrieben werden, und einer Methode zur Abbildung nativer Anweisungen der Gastarchitektur auf Sequenzen solcher Code-Templates. Die Code-Templates werden durch den regulären Java-Übersetzer zu Java Bytecode übersetzt und dann von einem entsprechenden Codeerzeugungsmodul von JXEmu zu größeren Codeblöcken zusammengesetzt.

Für die Codeerzeugung unterstützt JXEmu verschiedene Strategien, angefangen von der Codeerzeugung auf der Ebene nativer Instruktionen im Interpretermodus bis hin zu der Erzeugung komplexer Superblöcke. Alle diese Strategien sind von derselben Spezifikation hergeleitet und fügen sich daher nahtlos in die Architektur von JXEmu. Die verschiedenen verfügbaren Verfahren werden — zusammen mit ihren Vorteilen und Beschränkungen — in dieser Arbeit ebenfalls vorgestellt und diskutiert.

Auch wenn die Leistung von JXEmu im Moment nicht akzeptabel ist, so lassen die ständige Steigerung der Leistungsfähigkeit der zugrundeliegenden Hardware und die weitere Untersuchung und Verbesserung der Übersetzungsverfahren darauf hoffen, dass in Zukunft die Unterstützung von Altanwendungen auf neuen Architekturen durch deren dynamische Übersetzung zu Java Bytecode bewerkstelligt werden kann. Damit wären sowohl Hardware- als auch Softwarearchitekten von den Einschränkungen der Abwärtskompatibilität befreit.

Abstract

This thesis presents an approach to executing legacy applications on top of a type-safe Java Virtual Machine (JVM). The native code of the *guest architecture* is upcompiled to Java bytecode at runtime and then downcompiled to native code of the host architecure by the JVM's JIT compiler. The prototype application JXEmu therefore allows running unsafe native applications in a safe environment provided by the JVM.

Multiple guest architectures are supported for the legacy applications, beneath this the ARM architecture, whose JXEmu module was developed as a part of this thesis. Rather than emulating the guest architecture by directly emitting Java bytecode, the specification of a guest architecture is provided by a number of code snippets in Java source language and an appropriate method for mapping native instructions to a sequence of those code snippets. The code snippets are compiled to Java bytecode by the regular java compiler and then assembled to larger code blocks by an appropriate code generation module of JXEmu.

Multiple strategies for code generation are available within JXEmu, ranging from simple native instruction based code generation in interpreter mode to the generation of complex superblocks. All of these strategies are derived from the same specification and plug seamlessly into the JXEmu architecture. The various available strategies along with their benefits and limitations are also discussed within this thesis.

Even though the performance is not acceptable at the moment, the steady increase in performance of the underlying hardware as well as the further investigation of compilation strategies may allow to support legacy applications by upcompiling them to Java bytecode, thereby freeing hardware as well as software architects from the constraint of backward compatibility.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Traditionelle Befehlssatz-Architekturen (ISA[1]) werden oft nach Notwendigkeiten der zugrundeliegenden Hardware Implementierung entworfen. Eine wichtige Rolle beim Entwurf spielt oft die Kompatibilität zu älteren (Vorgänger-) ISAs, welche für die Akzeptanz der neuen Architektur bei den Benutzern unabdingbar ist. Bestehende Anwendungen benötigen Binärkompatibilität zu den bestehenden ISAs und schränken die Entwurfsfreiheit entscheidend ein.

Der aktuelle Trend geht hin zu semantisch reicheren ISAs, welche nicht im Hinblick auf eine bestimmte Hardware Implementierung entworfen wurden, sondern auf einer virtuellen Maschine (VM) basieren. Den bekanntesten Vertreter dieser Gattung dürfte Java bilden. Der Code für die virtuelle Maschine wird zur Laufzeit durch einen Just-in-Time Compiler (JIT) zu Code für die ISA der Zielarchitektur, also der tatsächlich ausführenden Hardware übersetzt. Eine Vielzahl existierender Arbeiten haben gezeigt, dass eine solche Übersetzung zur Laufzeit sehr effizient möglich ist.

Neben der erhöhten Entwurfsfreiheit bringen high-level Architekturen weitere Vorteile wie z.B. Typsicherheit mit, welche viele Probleme konventioneller ISAs wie z.B. Pufferüberläufe beheben. Dem Durchbruch dieser Architekturen steht jedoch die Masse an Altanwendungen[2] und die Abhängigkeit der Anwender von diesen etablierten Programmen entgegen. Eine Portierung dieser Anwendungen von einer konventionellen, Hardware-basierten ISA auf eine VM-basierte ISA erfordert eine aufwendige Neuprogrammierung der Anwendung in einer typsicheren Sprache wie Java.

Diese Arbeit stellt einen Lösungsansatz für dieses Problem vor. Der Code der Altanwendung wird dynamisch zu Java Bytecode übersetzt. Dieser wird im Regelfall durch den JIT der Java VM wieder auf eine Hardware-basierte ISA „herunterkompiliert". Die für diesen Zweck entwickelte Anwendung JXEmu ist selbst in Java geschrieben und damit auf allen Zielplattformen lauffähig, für die ein Java Runtime Environment existiert.

JXEmu ist in der Lage, für Linux entwickelte Altanwendungen im ELF-Format auszuführen. Als Gastarchitekturen werden im Moment ARM und PowerPC unterstützt, ein Modul für MIPS ist in Arbeit. Eine Erweiterung auf andere Betriebssystemschnittstellen als Linux wäre ebenfalls denkbar.

[1] ISA: Instruction Set Architecture

[2] für traditionelle, Hardware-basierte ISA entworfene Anwendung

Diese Arbeit ist wie folgt aufgebaut: In Kapitel 2 wird zunächst die Architektur von JXEmu beschrieben. Dabei werden die einzelnen Module der Anwendung, insbesondere die Emulation untypisierten Speichers in der JVM, erläutert und die Interaktion zwischen den einzelnen Modulen beschrieben. Kapitel 3 beschreibt den Mechanismus, der zur Erzeugung von Code in JXEmu verwendet wird. Dabei werden einige Besonderheiten hervorgehoben und Hinweise gegeben, die beim Entwurf von Modulen für neue Gastarchitekturen beachtet werden müssen. In Kapitel 4 wird die ARM Unterstützung von JXEmu näher beschrieben. Dabei werden im Speziellen die Besonderheiten der ARM Architektur betrachtet, da generische Aspekte bereits in Kapitel 3 behandelt wurden. Für die Erzeugung von Code unterstützt JXEmu verschiedene Strategien, die in Kapitel 5 beschrieben werden. Dabei werden insbesondere die Vorteile und Beschränkungen der einzelnen Strategien diskutiert. Die Arbeit wird mit einigen Leistungsmessungen in Kapitel 6 abgeschlossen.

Die einzelnen Kapitel beschreiben die Module umfassend, jedoch nicht bis ins letzte Detail. Insbesondere auf die Darstellung und Erläuterung von Quellcode wurde, bis auf wenige Beispiele, die zum Gesamtverständnis sinnvoll erschienen, verzichtet. Hierfür sei auf die ebenfalls im Laufe dieser Arbeit entstandene Dokumentation des Quellcodes verwiesen. Eine übersichtliche Referenz in verschiedenen Formaten lässt sich aus dem Quellcode mit dem Programm *javadoc* erzeugen.

Kapitel 2

JXEmu Architektur

Zunächst soll ein Überblick über die Architektur von JXEmu gegeben werden. Ein Ziel beim Entwurf des Emulators war die einfache Erweiterbarkeit um zusätzliche Gastarchitekturen und Übersetzungsverfahren. Das daraus resultierende Design der Anwendung ist in Abb. 2.1 dargestellt. Bei einigen der dargestellten Module handelt es sich lediglich um Schnittstellen, für die mehrere Implementierungen verfügbar sind. Diese können beliebig miteinander kombiniert werden, wodurch z.B. für eine neue Gastarchitektur automatisch auch alle Übersetzungsverfahren zur Verfügung stehen. Im Folgenden soll am Beispiel der Initialisierungsphase zunächst jedes der Module kurz beschrieben werden, um einen Gesamteindruck von der Funktionsweise von JXEmu zu vermitteln. Es folgt eine genauere Beschreibung der einzelnen Module in Abschnitt 2.2. Die Ausführungsphase wird in Kapitel 3 zusammen mit dem Gesamtkonzept der Codeerzeugung betrachtet.

2.1 Initialisierungsphase

Kernstück von JXEmu ist der Emulator. Dieser ist für die eigentliche Codeerzeugung zuständig. Hierfür existieren mehrere Strategien, für die jeweils eine eigene Implementierung des Emulators existiert. Die Übersetzung kann beispielsweise instruktionsweise im Interpretermodus erfolgen, oder aber in größerer Granularität wie z.B. Basicblocks[1].

Der Emulator arbeitet eng mit den anderen Komponenten des Systems zusammen. In der Initialisierungsphase verwendet er zunächst den Loader, um die Architektur der vorliegenden Binäranwendung zu bestimmen. Für diesen Zweck bietet der Loader eine statische Methode an.

Mit Kenntnis der Gastarchitektur werden nun Instanzen der architekturabhängigen Teile CPU und JIT erzeugt.

Die CPU Klasse ist — grob betrachtet — eine Ansammlung statischer Variablen. Jede dieser Variablen repräsentiert ein bestimmtes Register der emulierten CPU. Die Inhalte dieser Variablen werden während der Ausführungsphase den Status der CPU zu bestimmten Zeitpunkten enthalten. Diese Zeitpunkte sind von der gewählten Codeerzeugungsstrategie abhängig.

[1] An dieser Stelle sei bemerkt, dass Interpreter und Codegeneratoren, welche tatsächlich Code erzeugen müssen, von derselben Spezifikation abgeleitet sind. Beide Klassen implementieren die Emulator Schnittstelle, und fügen sich so für die restlichen Teile von JXEmu transparent in die Gesamtarchitektur.

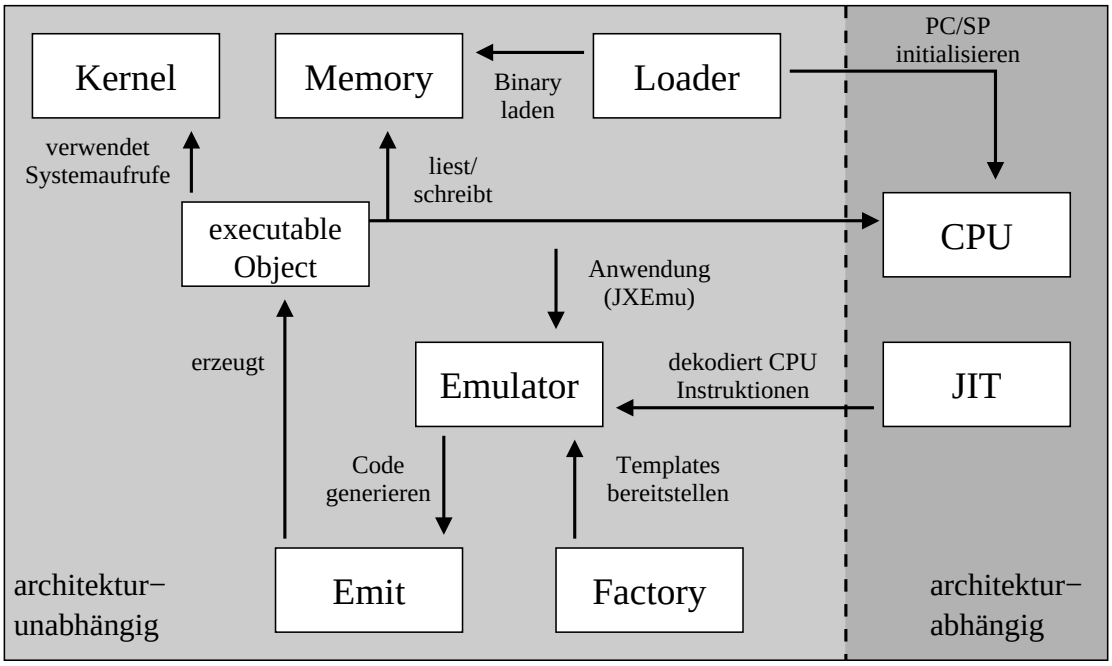

Abbildung 2.1: JXEmu Architektur

Für jede unterstützte Gastarchitektur existiert eine Implementierung des JIT-Moduls. Dem JIT kommen zwei wichtige Aufgaben zu: Zum einen wird er bei der Codegenerierung vom Emulator verwendet, um native Instruktionen der Gastarchitektur zu dekodieren. An dieser Stelle soll der Begriff *Code-Template*, im weiteren Verlauf auch kurz mit *Template* bezeichnet, eingeführt werden. Bei einem Code-Template handelt es sich um eine kleine in Java geschriebene Methode. Ein Beispiel für ein solches Template wäre beispielsweise eine Methode zur Addition von zwei Ganzzahlen, wie in Abb. 3.1 auf S. 22 dargestellt. Die Dekodierung der nativen Instruktion durch den JIT erfolgt in Form einer Abbildung der Instruktion auf eine Folge von Code-Templates, durch deren sequentielle Ausführung exakt die Semantik der nativen Anweisung nachgebildet wird. Eine genaue Beschreibung des Codeerzeugungskonzeptes erfolgt in Kapitel 3. Bis dahin soll die Vorstellung genügen, dass durch Aneinanderreihung von Templates größere Codeblöcke erstellt werden können. Zum anderen müssen alle verwendeten Templates vom jeweiligen JIT implementiert werden.

Nachdem nun Instanzen von JIT und CPU existieren, kann der Loader mit dem eigentlichen Laden der Binärdatei beginnen. Die relevanten Sektionen werden an die entsprechenden virtuellen Adressen im Memory geladen. Außerdem wird der Stack für die Ausführung des Programms vorbereitet, wozu das Environment-Array, Parameter-Array und schließlich die Parameteranzahl in dieser Reihenfolge in den Stack geschrieben werden. Zuletzt wird der CPU Status mit den initialen Werten für Stackzeiger und Befehlszähler initialisiert.

Die Initialisierungsphase ist damit abgeschlossen und der Emulator kann nun mit der eigentlichen Ausführung beginnen. Bevor die Ausführung beschrieben wird, soll jedoch ein genauerer

Blick auf die einzelnen Module erfolgen.

2.2 Module von JXEmu

Die Architekturübersicht in Abb. 2.1 zeigt nur die wichtigsten Module von JXEmu. Neben diesen existieren viele kleine Hilfsklassen, die zum Teil jedoch Kernaufgaben übernehmen. Eine komplette Beschreibung der Hilfsklassen würde den Rahmen dieser Arbeit bei weitem sprengen. Stattdessen werden die Aufgaben der Hilfsklassen in dieser Beschreibung den Modulen zugeordnet, die direkten Gebrauch von den Hilfsklassen machen bzw. konzeptionell am passendsten sind.

2.2.1 Linux Kernelemulation

Die Ausführung von Prozessoranweisungen im Benutzermodus ist für einen Prozess nicht ausreichend. Zum Zugriff auf Peripheriegeräte wie die Festplatte oder auf andere Funktionen des Betriebssystems, wie beispielsweise die Inkarnation neuer Prozesse, wird vom Betriebssystem eine geeignete Schnittstelle bereitgestellt [BC00, Kapitel 8]. Diese zusätzliche Ebene bringt mehrere Vorteile mit sich, insbesondere erhöht es die Sicherheit des Systems, da alle Zugriffe über die Systemschnittstelle der Zugriffskontrolle des Betriebssystems unterliegen.

JXEmu emuliert die Schnittstelle des Linux Kernels [Tor], derzeit aber nur in sehr geringem Umfang. Jedem Systemaufruf ist eine bestimmte Nummer zugeordnet. Der Wechsel in den Systemmodus erfolgt durch das Auslösen eines Software-Interrupts, wofür die CPU i.d.R. eine spezielle Instruktion bereitstellt. Die Aufrufsemantik ist architekturabhängig und für viele Architekturen, darunter auch die ARM-Architektur, kaum dokumentiert. Aufschluß gibt in diesem Fall ein Blick in die Quellen des Linux-Kernels [Tor] oder der C-Bibliothek [BEJ+]. Die Aufrufsemantik für den ARM wird in Kapitel 4 mit der Beschreibung von JXEmus ARM-Unterstützung erfolgen. Dieser Abschnitt beschränkt sich auf die Beschreibung des Kernel Moduls. Nach außen stellt der Kernel die Methode

```
public static int
do_syscall(int cmd, int r0, int r1, int r2,
   int r3, int r4, int r5, int r6);
```

bereit. Diese führt den Systemaufruf mit der in `cmd` übergebenen Nummer mit den in `r0`–`r6` übermittelten Parametern aus und gibt den Rückgabewert des Systemaufrufs zurück. Die Ermittlung der Nummer des Systemaufrufs sowie der Parameter ist, wie bereits erwähnt, architekturabhängig und hat daher im jeweiligen JIT innerhalb eines Code-Templates zu erfolgen, welches auch den Aufruf von `do_syscall()` veranlasst.

Leider ist auch das Kernel-Modul nicht gänzlich von der Architektur unabhängig. Beispielsweise müssen Systemaufrufe, welche den virtuellen Speicher[2] lesen oder schreiben, die Byteorder der vorliegenden Gastarchitektur berücksichtigen. Deshalb existieren Subklassen von Kernel

[2] Der Begriff *virtueller Speicher* wird in dieser Arbeit zur Bezeichnung des durch das Memory Modul emulierten, untypisierten Speichers verwendet. Er ist nicht zu verwechseln mit dem entsprechenden Begriff aus dem Betriebssystembereich.

mit dem Namen Kernel_*ARCH*, welche Methoden des Kernels an den notwendigen Stellen mit architekturspezifischen Varianten überladen.

2.2.2 Speicheremulation

Aus physischer Sicht ist der Speicher lediglich eine Aneinanderreihung von Bytes oder Wörtern, welche nicht typisiert sind. In der JVM hingegen besitzt jede Speicherstelle einen definierten Typ. Der Zugriff auf solche Speicherzellen ist nur unter Verwendung des zugewiesenen Typs möglich. Im Gegensatz dazu könnte im untypisierten Speicher die gleiche Speicherzelle zunächst mit einer Ganzzahl beschrieben werden, um anschließend als Gleitkommazahl wieder gelesen zu werden. Die intuitive Herangehensweise zur Emulation des Hauptspeichers in einer JVM wäre ein Array von Bytes. Das Memory Modul müsste Operationen zur Emulation eines Wortzugriffes bereitstellen, welche diesen auf Byteoperationen abbilden. Dabei würde z.B. beim Schreiben eines vier Byte großen Wortes dieses zunächst in vier einzelne Bytes zerlegt, welche dann einzeln in das Byte-Array eingebracht würden.

Diese Lösungsmöglichkeit ist sehr ineffizient: Beim Zugriff auf ein Arrayelement muss die JVM zunächst prüfen, ob auf einen gültigen Index zugegriffen wird [LY99, S. 39]. Dabei muss der Index sowohl mit der unteren als auch der oberen Grenze überprüft werden, was pro Bytezugriff zwei Indexprüfungen ergibt. Ingesamt entstehen für das Schreiben eines Integer also acht Indexprüfungen sowie 4 Speicherzugriffe für das eigentliche Schreiben der Bytes.

Es wurden deshalb andere Herangehensweisen implementiert. Bevor die zwei vorhandenen Implementierungen beschrieben werden, soll kurz ein Blick auf die Schnittstelle des Memory Moduls geworfen werden.

Schnittstelle des Memory

Der Memory bietet nach außen die oben angesprochenen Methoden zum Schreiben und Lesen der primitiven Datentypen byte, short, int und long an bestimmten Adressen, sowie dem Lesen und Schreiben ganzer Arrays von Bytes. Soweit sinnvoll existieren für jede Methode sowohl little-endian als auch big-endian Varianten. Es ist Aufgabe des jeweiligen JIT, die korrekten Methoden zu verwenden. Außerdem bietet der Memory eine einfache Form von Speicherschutz: Für jede Speicherseite können die Bits *vorhanden*, *nur lesbar* und *ausführbar* vermerkt werden. Beim Zugriff auf den Speicher werden diese Bits überprüft. Auf Speicherzugriffsverletzungen wird mit einer `RuntimeException` reagiert.

SafeMemory

Der Arbeitsspeicher wird in Seiten der Größe 4 kB aufgeteilt. Jede dieser Seiten wird als ein `int[]` realisiert. Eine Seitentabelle enthält die Referenzen auf diese Seiten und ist somit vom Typ `int[][]`. Da aus Arithmetikgründen der Speicher nur bis zu einer maximalen Adresse von 0x7fffffff verwendet wird, genügt eine Seitentabelle mit 512k Einträgen. Damit ergibt sich die Größe der Seitentabelle zu 2MB und die Größe des adressierbaren Speicherbereichs zu 2GB. Die Einträge der Seitentabelle enthalten nur dann gültige Referenzen, wenn die Seite bereits allokiert wurde, was nur dann geschieht, wenn die Seite auch tatsächlich verwendet wird.

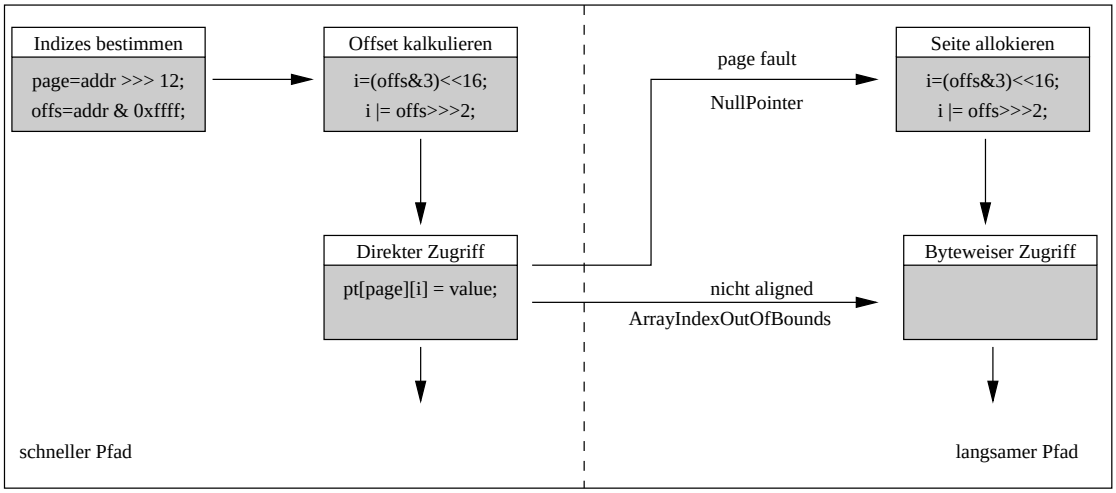

Abbildung 2.2: Speicherzugriff für den Aufruf von `putInt(addr,value)`. Im häufigen Fall des Zugriffs mit Alignment auf bereits allokierte Seiten wird der schnelle Pfad durchlaufen. Falls die Seite noch nicht allokiert war, wird die `NullPointerException` abgefangen und die Seite allokiert. Ist die Adresse nicht auf vier Bytes ausgerichtet, wird ein `offs` größer als 1024 erzeugt, welches bei der Arraygrenzenprüfung der JVM zu einer `ArrayOutOfBoundsException` führt. In diesem Fall wird der Integer in Bytes zerlegt und so in den Speicher eingebracht.

Ein flaches Array wäre zwar schneller, da die Indirektion über die Seitentabelle und die damit verbundenen Kosten entfallen würden. Die JVM initialisiert jedoch jede Variable. Im Falle eines `int[]` werden alle Komponenten mit 0 vorbelegt [LY99, S. 14], was dazu führt, dass das Betriebssystem physischen Speicher für das gesamte Array allokieren muß, in diesem Fall 2GB. SafeMemory ist optimiert für Zugriffe mit korrektem Alignment. Dies ist die weitaus häufigste Zugriffsform, da bei vielen Architekturen ein korrektes Alignment eine für den Wortzugriff notwendige Voraussetzung ist. Auf anderen Architekturen wie der Intel x86 Architektur ist der Wortzugriff auf nicht ausgerichtete Adressen zwar prinzipiell möglich, jedoch langsamer als ein Zugriff mit korrektem Alignment. Die Übersetzer erzeugen daher für korrektes Alignment optimierten Code. Der schematische Ablauf beim Schreiben eines Integer ist in Abb. 2.2 dargestellt. Die Prüfungen zum Speicherschutz wurden zur Vereinfachung weggelassen.

Um unnötige Überprüfungen auf dem schnellen Pfad zu vermeiden werden Ausnahmen der JVM genutzt. Im Falle einer nicht allokierten Seite ist in der Seitentabelle eine `null`-Referenz eingetragen. Die JVM prüft beim Zugriff auf die Seite die Referenz und erzeugt bei einem Seitenfehler eine `NullPointerException`. Für diese muss nur eine Fehlerbehandlung eingerichtet werden, welche die Allokierung der gewünschten Seite veranlasst. Der schnelle Pfad muss außerdem verlassen werden, wenn ein Wortzugriff nicht an einer 4 Byte Grenze stattfindet. Um ein ähnliches Verfahren wie bei der Fehlseitenbehandlung zu verwenden, ist ein kleiner Trick notwendig. Wenn die Adresse nicht auf 4 Byte ausgerichtet ist, so ist wenigstens eines ihrer beiden niederwertigsten Bits von 0 verschieden. Diese beiden Bits werden an die Bitposi-

tionen 16 und 17 des Offsets innerhalb der Seite gebracht, wodurch das Offset garantiert größer als 1024, der oberen Arraygrenze bei einer 4kB Seite, wird. Bei der Arraygrenzenprüfung der JVM wird der ungültige Index erkannt und eine `ArrayIndexOutOfBoundsException` erzeugt. Für diese kann eine Fehlerbehandlung installiert werden, die die Aktionen des langsamen Pfades durchführt. Auf diesem wird das Wort byteweise in den Speicher eingebracht.

DirectMemory

Im Sun JDK bietet die Klasse *sun.misc.Unsafe* Möglichkeiten zum Zugriff auf und zur Allokation von untypisiertem Speicher. Die Unsafe Klasse ist allerdings nur für den internen Gebrauch während des Bootstrappings der JVM bestimmt und kann nicht beliebig instanziert werden. Über eine Hintertür ist es jedoch möglich, an eine während des Bootstrappings erzeugte Instanz der Unsafe Klasse zu gelangen. Dieser Entwurfsfehler scheint in neueren JDK ab Version 1.5.0 behoben zu sein, mit JDK 1.4.2 verhilft er JXEmu aber zu Zugriff auf untypisierten Speicher. Im Gegensatz zum Safe Memory entfällt die Indirektion über die Seitentabelle. Mittels der Unsafe Klasse ist es möglich, einen großen Speicherblock direkt vom Heap zu allokieren, ohne dass dieser initialisiert wird. Das Betriebssystem wird deshalb nur physischen Speicher für tatsächlich verwendete Seiten allokieren.

2.2.3 Virtuelle CPU

Für jede Architektur muss eine Klasse mit dem Namen CPU_*Arch* existieren. Diese repräsentiert den Status der emulierten CPU zu bestimmten Zeitpunkten während der Ausführung. Diese Zeitpunkte sind abhängig von der gewählten Codeerzeugungsstrategie. Bei der Übersetzung von Basicblocks enthält der CPU Status immer die Werte, die nach dem Verlassen des letzten Basicblocks aktuell waren. Hierfür enthält die Klasse für jedes Register der emulierten CPU eine statische Variable vom entsprechenden Typ. Die spezifischen CPU Klassen müssen von der Klasse CPU abgeleitet sein. Damit wird sichergestellt, dass einige Konventionen, welche für das CPU Modul gelten müssen, beachtet werden. Diese existieren entweder aus Optimierungsgründen oder zur Verlagerung architekturspezifischer Merkmale auf die architekturunabhängige Ebene des Codegenerators.

Semantik des Befehlszählers

Die Superklasse CPU definiert einen statischen Integer `PC`. In diesem befindet sich immer die reale Adresse des aktuellen Befehlszählers. Eine eventuelle Pipeline des Prozessors wird nicht berücksichtigt. Zur Ermittlung des Befehlszählerwertes unter Berücksichtigung der Pipeline muss jede CPU-Spezialisierung die Methode `adjustPC()` implementieren. Diese liefert für einen gegebenen Wert des Befehlszählers den angepassten Wert unter Berücksichtigung der Pipeline. Das Offset für die Pipeline kann nicht statisch festgelegt werden, auch nicht innerhalb einer Architektur. Der ARM Prozessor unterstützt beispielsweise zwei Befehlssätze, einen 32 Bit Modus und einen 16 Bit Thumb Modus. Zwischen diesen Modi kann zur Laufzeit umgeschaltet werden. Durch das Pipelining des ARM ist der Befehlszähler der aktuell ausgeführten Instruktion immer um zwei Instruktionen voraus. Das Offset ist dabei, abhängig vom aktuellen

Modus, entweder acht oder vier Bytes. Die Methode `adjustPC()` passt den Befehlszähler unter Berücksichtigung des aktuellen Modus an.

Sprungziel indirekter Sprünge

Für direkte Sprünge muss kein Code zur Berechnung des Sprungziels erzeugt werden, da das Ziel des Sprungs bereits zur Zeit der Übersetzung bekannt ist. Der Übersetzer kann also einfach an dieser Adresse fortfahren. Bei indirekten Sprüngen ist das Sprungziel vorher nicht bekannt, beispielsweise weil das Sprungziel sich in einem Register befindet. Hierfür enthält die CPU Superklasse den Integer `NPC`. Dieser soll nach jedem indirekten Sprung die Adresse der nächsten Anweisung enthalten. Wie schon beim Befehlszähler wird auch hier die Pipeline nicht berücksichtigt. Im Falle eines bedingten indirekten Sprungs, der nicht durchgeführt wurde, ist der Wert in `NPC` nach dem Sprung nicht definiert.

Bedingte Sprünge

Sprungbefehle können, wie oftmals auch andere Anweisungen, an eine gewisse Bedingung geknüpft werden. So eine Bedingung könnte z.B. sein, dass das Zero-Flag des Statusregisters gesetzt sein muss, weil eine vorherige arithmetische Instruktion das Ergebnis 0 errechnet hat. Die CPU sieht hierfür eine boolsche Variable `Z` (Z-Flag) vor. Wenn beim Durchlaufen eines bedingten Sprungs die Bedingung erfüllt war, muss das Z-Flag nach Durchführung der Instruktion gesetzt sein. Ansonsten darf das Z-Flag nicht gesetzt sein. Damit wird es dem Codegenerator möglich zu erkennen, ob ein bedingter Sprung durchgeführt wurde oder nicht.

2.2.4 ELF Loader

Der Loader spielt nur während der Initialisierungsphase eine Rolle, welche in Abschnitt 2.1 beschrieben wurde. Seine Aufgaben sind die Ermittlung der Architektur von ELF Binärdateien[3] [TIS95, ARM01] und das Laden der einzelnen Sektionen in den virtuellen Speicher.

Architekturermittlung

Zur Ermittlung der Architektur einer ELF-Binärdatei stellt der Loader die statische Methode `getArch()` zur Verfügung:

```
public static int getArch(String filename);
```

Diese liest aus der ELF Datei, deren Dateinamen sie als Parameter erhält, das Feld `E_machine` aus dem ELF Header [ARM01, S. 8-9]. Bei `E_machine` handelt es sich um eine vorzeichenlose 16 Bit Ganzzahl. Jeder Architektur ist hier eine bestimmte Zahl zugeordnet, im Falle von ARM/Thumb der Wert 40. Architekturabhängige Module wie die CPU stellen eine statische Methode `instance()` bereit, welche bei Übergabe des Wertes aus `E_machine` eine zur Architektur passende Implementierung instanziert.

[3] In dieser Arbeit wird der Begriff ELF (Binär-)Datei vereinfachend für ausführbare ELF-Binärdateien verwendet. Neben diesen existieren auch dynamische Bibliotheken sowie relokierbare Objektdateien im ELF Format. Diese sind für JXEmu jedoch uninteressant und in dieser Arbeit somit nicht relevant.

Laden der Binärdatei in den virtuellen Speicher

Eine ELF Datei ist aus verschiedenen Sektionen aufgebaut, welche bei der Ausführung den Segmenten des Speicherabbilds des Prozesses entsprechen. Mit Ausnahme von evtl. vorhandenen Kontrollsektionen müssen die Sektionen, darunter fallen insbesondere Text- und Datensegment, also in den virtuellen Speicher geladen werden. Die Anfangsadressen der Sektionen im Speicherabbild werden dem `sh_addr` des jeweiligen Section Header entnommen [ARM01, S. 14]. Während des Ladevorgangs wird auch das anfängliche Ende des Heap, der sog. *brk* ermittelt und im Kernel Modul initialisiert.

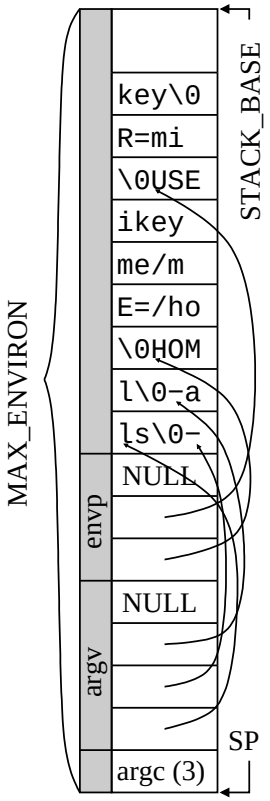

Abbildung 2.3: Stack für die Ausführung von *ls -l -a* vorbereitet

Initialisierung der CPU

Der Status der CPU muss geeignet initialisiert werden, damit mit der Ausführung des Programms begonnen werden kann. Konkret müssen Befehlszähler (PC) und Stackzeiger (SP) initialisiert werden. Der PC wird mit der Anfangsadresse des Textsegments initialisiert. Die Anfangsadresse des SP hängt von der Konfiguration des Loaders ab und berechnet sich als die Differenz aus den konfigurierten Werten `STACK_BASE`, der Startadresse des Stack von der ab dieser zu kleineren Adressen hin wächst, und `MAX_ENVIRON` (siehe Abb. 2.3). Beide Werte können beliebig konfiguriert werden, für STACK_BASE sollte jedoch ein Wert kleiner 2^{31} gewählt werden, um Arithmetik mit vorzeichenbehafteten Ganzzahlen verwenden zu können[4].

Vorbereitung des Stacks

Abb 2.3 zeigt den fertig initialisierten Stack für die Kommandozeile *ls -l -a* sowie ein vereinfachtes Environment, bestehend aus den beiden Variablen *USER* und *HOME*. Direkt über dem anfänglichen SP findet sich die Zahl der Parameter, der Name des auszuführenden Programms inklusive, in diesem Beispiel also 3. Darüber folgen die Zeigerarrays *argv* und *envp*, welche Zeiger auf die einzelnen Argumente bzw. Umgebungsvariablen enthalten und jeweils durch einen NULL-Zeiger abgeschlossen sind.

[4] Java kennt keine Arithmetik mit vorzeichenlosen Ganzzahlen.

2.2.5 Instruktionsdekodierung durch den JIT

Der JIT ist dafür zuständig, native CPU Anweisungen aus dem Memory zu lesen und diese zu dekodieren. Die Dekodierung erfolgt in Form einer Abbildung der nativen Instruktion auf eine Folge von Template Anweisungen, welche das Verhalten der emulierten Instruktion exakt nachbilden. Außerdem enthält der JIT die Implementierungen aller Template Methoden, die er zur Emulierung der nativen Anweisungen benötigt. Das Konzept der Codeerzeugung von JXEmu durch das Verwenden von Template Methoden wird in Kapitel 3 ausführlich erläutert. Die Besonderheiten der ARM Architektur, die beim Entwurf des zugehörigen JIT Moduls beachtet werden mussten, werden in Kapitel 4 beschrieben.

2.2.6 Emulator: der Codegenerator

Das Kernmodul von JXEmu ist der Emulator. Es existieren verschiedene Implementierungen des Emulators, welche in Kapitel 5 einzeln vorgestellt werden. Jede dieser Implementierungen verfolgt eine bestimmte Strategie bei der Erzeugung, Zwischenspeicherung und Ausführung von Codefragmenten. Die Granularität dieser Codefragmente beginnt auf der Ebene einzelner Templates und geht hin bis zu komplexen Superblöcken.

2.2.7 Templatelieferant: die Factory

Die Factory und ihre Hilfsklassen dienen dem Emulator als Lieferant von Code Templates. Diese werden aus dem übersetzten JIT gelesen und so angepasst, dass sie im generierten Code des Generators verwendet werden können. Die notwendigen Anpassungen werden in Kapitel 3 erläutert.

2.2.8 Emit

Mit Hilfe des Emit Moduls ist es möglich, zur Laufzeit neue Java Klassen zu generieren und Instanzen dieser Klassen zu erzeugen. Der Emulator erzeugt mit Hilfe von Emit für die generierten Code-Fragmente jeweils eine eigene Klasse und eine Instanz dieser Klasse. Jede von Emit generierte Klasse enthält eine Methode mit dem Namen `execute()`. Ein Aufruf dieser Methode bewirkt die Ausführung des generierten Codes.

Kapitel 3

Codeerzeugung in JXEmu

Da JXEmu Modi unterstützt, die mit größerer Granularität als der Instruktionsebene arbeiten, ist es erforderlich, Java Bytecode zu erzeugen [1]. Die inituitive Herangehensweise, für jede native Instruktion direkt die Abbildung auf Bytecode durchzuführen, ist nicht nur sehr aufwendig, sondern auch fehleranfällig. Speziell für komplexere CPU Instruktionen steigt sehr schnell die Komplexität des zu erzeugenden Bytecodes an. Der Entwurf von JIT-Modulen für neue Architekturen wäre damit alles andere als trivial und würde fortgeschrittene Kenntnis der Java VM erfordern. Stattdessen wird der Entwurf der JIT Module auf einer höheren Ebene durchgeführt: der Java Programmiersprache.

3.1 Code-Templates

Ein Code-Template ist eine in Java implementierte Methode, die eine Maschineninstruktion oder einen Teil einer solchen emuliert. Die Implementierung dieser Code-Templates findet direkt im jeweiligen JIT-Modul statt. Ein Template wird als solches erkennbar gemacht, indem der entsprechenden Methode das Präfix *jit_* im Namen vorangestellt wird. Die Transformation der Template Methoden zu Java Bytecode erfolgt durch den Java Übersetzer *javac* im Zuge der normalen Übersetzung der Quelldatei des JIT. Für den Entwickler eines JIT sind damit keinerlei Kenntnisse des Java Bytecode erforderlich. Bei *javac* handelt es sich zudem um ein erprobtes Werkzeug, von dessen Zuverlässigkeit bei der Übersetzung ausgegangen werden kann.

```
public static void
jit_add(int Rd, int op1, int op2) {
   Rd = op1 + op2;
}
```

Abbildung 3.1: Template für die Additionsoperation zweier Ganzzahlen

Ein Beispiel eines solchen Templates ist in Abb. 3.1 zu sehen. Das Template dient der Addition von zwei Ganzzahlen. Es fällt auf, dass die Template Methode als solche keinen Sinn macht, und in der Tat von einem findigen Java-Compiler bei der Übersetzung wegoptimiert werden könnte. Es sei deshalb an dieser Stelle darauf hingewiesen, dass der Templatemechanismus mit zukünftigen JDK Implementierungen u.U. nicht mehr funktio-

[1] Im reinen Interpretermodus wäre eine Erzeugung von Code nicht notwendig. Stattdessen könnte für jede native Instruktion eine Java Methode zur Verfügung gestellt werden, welche beim Auftreten der Instruktion direkt von JXEmu aufgerufen wird.

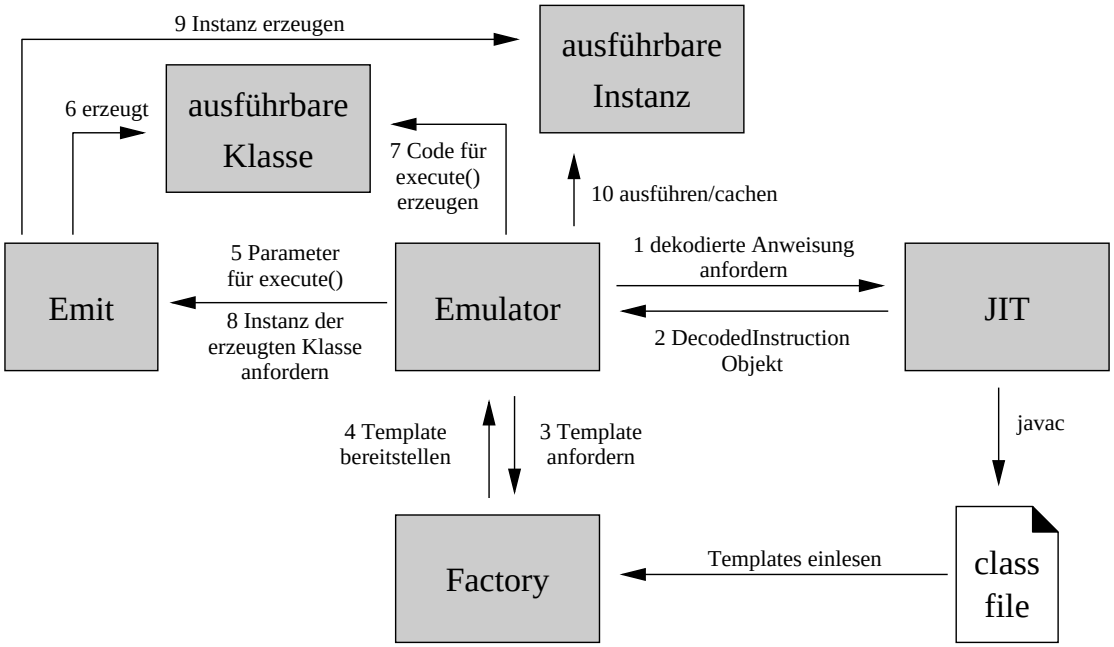

Abbildung 3.2: Die Ausführungsphase von JXEmu. Im Kern steht der Emulator, welcher die anderen Module zur Instruktionsdekodierung, Templatemodifizierung und -bereitstellung und der dynamischen Generierung von Klassen verwendet. Die Synthese des Codes findet im Emulator selbst statt. Er ist auch für das Zwischenspeichern und Ausführen der erstellten Objekte zuständig.

nieren wird, zumindest nicht in der vorliegenden Form. Der verwendete Java Übersetzer[2] erzeugte aus solchen Template Methoden jedoch problemlos den gewünschten Code.

3.2 Ausführungsphase

Abb. 3.2 zeigt die Ausführungsphase von JXEmu, in der auch die dynamische Erzeugung des Codes stattfindet. Am Ende eines jeden Laufes steht die Instanz einer erzeugten Klasse. Diese wurden in Abb. 3.2 als *ausführbare Klasse* bzw. *ausführbare Instanz* bezeichnet. Die mit Hilfe von `Emit` generierten Klassen enthalten immer eine `execute()` Methode, daher auch die Bezeichnung. Das Aufrufen dieser Methode bewirkt die Emulation der Ausführung eines bestimmtes Codestückes, beispielsweise der eines Basicblocks. Nach Ausführung der `execute()` Methode sind Status der virtuellen CPU und Memory entsprechend den emulierten nativen Instruktionen verändert. Außerdem wurden Auswirkungen eventueller Systemaufrufe im nativen Code durch die Kernelemulation nachgebildet. Im folgenden sollen die einzelnen Schritte genauer beschrieben werden.

[2] zum Übersetzen von JXEmu wurde u.a. javac aus dem Sun JDK 1.5.0 verwendet

```
public final class DecodedInstruction {
  // Laenge der Templatesequenz
  public int template_count;

  public String[] signatures;
  public String[] templates;
  public int[][] contexts;

  // Typ des Sprungs, falls zutreffend
  public int branchType;

  // Verfuegbare Flags fr branchType
  public static final int DIRECT = 1;
  public static final int CONDITIONAL = 2;
  public static final int BRANCH = 4;
  public static final int DELAY_SLOT = 8;
  public static final int DELAY_SLOT_ONLY_IF_TAKEN = 16;

  // Adresse der naechsten Anweisung, falls direkter
  // bedingter Sprung nicht erfolgt
  public int nextPC;

  // Sprungziel, falls direkter Sprung erfolgt
  public int targetPC;

  // Handelt es sich um eine Sprunganweisung?
  public final boolean isBranch() { ... }
}
```

Abbildung 3.3: `DecodedInstruction` Objekte werden verwendet, um das Ergebnis der Dekodierung einer nativen CPU Instruktion zum Emulator zu transportieren. In ihnen wird die erzeugte Template Sequenz mit Metainformationen wie dem Sprungtyp der Anweisung gekapselt. Damit werden die architekturspezifischen Merkmale nativer Instruktionen auf die architekturunabhängige Ebene des Emulators befördert.

3.2.1 Dekodierung von Instruktionen

Die Dekodierung von CPU Instruktionen der Gastarchitektur erfolgt in Form einer Abbildung auf eine Sequenz von Templates durch den JIT, welcher hierfür die folgende Methode bereitstellt:

```
public final void decode(DecodedInstruction i);
```

Der Ergebnis der Dekodierung wird in ein `DecodedInstruction` Objekt verpackt, welches der Emulator dafür bereitstellt. Die `DecodedInstruction` Klasse ist in Abb. 3.3 dargestellt. Sie ist von besonderem Interesse, da die in ihr gekapselte Information das gesamte Wissen des Emulators über die native Anweisung widerspiegelt. Daraus folgt insbesondere, dass eine Implementierung komplexerer Codegeneratoren u.U. nicht ohne eine Erweiterung der

Sig	Kontext	Bedeutung
I	Wert	Immediate. Wert wird direkt dem Kontext entnommen.
P	-	PC. Befehlszähler wird direkt als Wert eingesetzt.
N	-	NPC. Zieladresse nach indirekten Sprüngen.
Z	-	Z-Flag. Ausführungkriterium für bedingte Sprünge.
R, G	Nummer des Registers	Registerbänke für 32 Bit Integer Register.
F, J	Nummer des Registers	Registerbänke für 64 Bit Gleitkommazahl Register.
andere	-	Einzelne 32 Bit Ganzzahl Register.

Tabelle 3.1: Bedeutung der Buchstaben in einer Template-Signatur

`DecodedInstruction` Klasse möglich ist. Beispielsweise wäre für einen Interpreter die Kenntnis der Template Sequenz bereits ausreichend, um den Code Anweisung für Anweisung zu emulieren. Ein Codegenerator für Basicblocks muss, um das Ende eines Basicblocks erkennen zu können, hingegen bereits wissen, ob es sich bei der Anweisung um eine Sprunganweisung handelt oder nicht. Komplexere Emulatoren benötigen weitergehendes Wissen über die Anweisung, wie in Abb. 3.3 bereits zu erkennen ist.

Die meisten Mitglieder der `DecodedInstruction` Klasse erklären sich von selbst oder sind durch die kurzen Kommentare in Abb. 3.3 bereits hinreichend beschrieben. Die Namen der einzelnen Templates sind in `templates` sequentiell abgelegt. Da `DecodedInstruction` Objekte wiederverwendet werden und die Länge der enthaltenen Arrays fest und ausreichend groß gewählt ist, wird die Variable `template_count` benötigt, um die tatsächliche Länge der für die aktuell dekodierte Instruktion erzeugten Template-Sequenz feststellen zu können. Die Mitglieder `signatures` und `contexts` bedürfen jedoch für das Verständnis des Template Mechanismus einer weitergehenden Erklärung.

Signatur eine Templates

Viele CPU Instruktionen existieren in verschiedenen Varianten. Beispielsweise existieren für die Addition zweier Ganzzahlen i.d.R. Varianten, die ihre Operanden entweder aus Registern entnehmen oder einen der Operanden unmittelbar in den Befehl kodiert haben. Auf der anderen Seite können auch Code-Templates in sehr unterschiedlichem Kontext verwendet werden, da diese nur als Bausteine bei der Nachbildung einer nativen Anweisung verwendet werden. Die Signatur eines Templates verrät dem Emulator die Herkunft der Parameter einer Template Methode. Die Signatur ist ein String, der für jeden Parameter des Code-Template einen Buchstaben enthält, welcher Auskunft über den Parameter in dieser konkreten Verwendung des Templates gibt. Tab. 3.1 zeigt die Bedeutung der einzelnen Buchstaben in einer Signatur.

Kontext eines Code-Templates

Die Kenntnis der Signatur alleine genügt nicht bei allen Typen. Bei Verwendung einer Registerbank beispielsweise wird zusätzlich die Zahl des Registers in der Bank benötigt. Liegt ein Immediate Operand vor, muss der einzusetzende Wert bekannt sein. Diese Information ist im `Kontext` eines Code-Template enthalten. Der Kontext ist ein Array von Integern, dessen Länge

abhängig von der Signatur ist. Tab. 3.1 zeigt, für welche Buchstaben ein Kontext benötigt wird und welche Bedeutung dieser hat. Da der Kontext aus Ganzzahlen besteht, sind insbesondere keine Gleitkommazahlen als Immediate-Operanden möglich.

Beispiel für die Verwendung eines Template in unterschiedlichem Kontext

Die Verwendung des gleichen Templates zu unterschiedlichen Zwecken illustriert Abb. 3.4.

```
add r2, r1, #5        add r2, r1, r2        b r0

emit("RRI",           emit("RRR",           emit("NPR",
     "jit_add",            "jit_add",            "jit_add",
     2, 1, 5);             2, 1, 2);             0);
```

Abbildung 3.4: Beispiele für die Verwendung des *jit_add* Template

Zu sehen ist jeweils eine ARM-Assembler Instruktion und eine mögliche Emulation durch das in Abb. 3.1 auf S. 22 vorgestellte Template zur Addition zweier Ganzzahlen. Die `emit()` Methode dient im JIT dem Anhängen eines Templates in die Template Sequenz. Sie erhält als Parameter Signatur, Namen des Code-Template und, falls notwendig, den Kontext.

Im ersten Beispiel wird der Wert 5 zum Inhalt des Registers `r1` addiert und das Ergebnis in Register `r2` gespeichert. Die Register `r0`–`r14` liegen auf einer Registerbank der emulierten ARM CPU für die der Buchstabe `R` gewählt wurde. Die Signatur enthält daher für Ziel und Operand 1 ein `R`. Operand 2 ist direkt im Befehlswort der nativen Anweisung kodiert und wird als Direktwert eingesetzt. Die Signatur enthält daher ein `I` an der Position für den zweiten Operanden. Jeder Teil der Signatur benötigt eine weitere Beschreibung im Kontext. Für die Register finden sich dort die entsprechenden Nummern 2 bzw. 1. Für den Immediate Operanden befindet sich der Wert 5 im Kontext, welcher anschließend direkt im Template als Wert verwendet wird.

Das zweite Beispiel demonstriert die Addition mit zwei Registeroperanden. Operand 2 wird bei dieser Addition durch das Ergebnis überschrieben.

Das dritte Beispiel zeigt die Verwendung von *jit_add* in einem völlig anderen Kontext. Bei der nativen Anweisung handelt es sich um einen Vorwärtssprung. Das Offset für den Sprung befindet sich im Register `r0`. Der Sprung ist letztlich nichts anderes als eine Addition auf den Befehlszähler[3]. Die Signatur stellt sicher, dass das Ergebnis der Addition in `NPC` abgelegt wird, welches als Zielregister für indirekte Sprünge in der virtuellen CPU verwendet wird. Als Operanden werden der Befehlszähler und das Register `r0` benötigt. Lediglich für letzteres wird Kontextinformation benötigt.

[3]Der JIT muss allerdings in diesem Fall dafür sorgen, dass der `branchType` in `DecodedInstruction` korrekt belegt wird.

3.2.2 Bereitstellung und Anpassung der Templates

Der erzeugte Bytecode kann nicht unverändert als Bestandteil von generierten Codeblöcken verwendet werden. Um die notwendigen Modifikationen verstehen zu können, sind Kenntnisse des Java Classfile Formats sowie dem Befehlssatz der JVM erforderlich. Eine Beschreibung findet sich in [LY99]. Dieses Wissen wird für das Verständnis dieses Abschnitts vorausgesetzt.

Beim Erzeugen der Factory Instanz liest diese alle durch das Präfix *jit_* gekennzeichneten Template Methoden ein und speichert diese als Template Objekte in einer Hashtabelle. Mit der Methode `get()` kann sich der Emulator ein solches Template Objekt für ein durch seinen Namen bekanntes Template liefern lassen.

In Abb. 3.5 ist der Bytecode des *jit_add* Template zu sehen. Die Parameter einer Java Methode zählen zu ihren lokalen Variablen und werden von 0 ab aufwärts nummeriert[4].

In diesem Falle sind die Indizes (*local variable index: LVI*) 0–2 für die drei Parameter des Templates verwendet worden. Um diesen Code in der `execute()` Methode der erzeugten Klasse verwenden zu können, müssen einige Vorbereitungen getroffen werden. Da die Templates auf lokalen Variablen arbeiten[5], muss der Emulator einen Prolog und einen Epilog erzeugen. Im Prolog werden die benötigten Teile des CPU Status sowie Immediate Werte in lokale Variablen der `execute()` Methode geladen, im Epilog werden die veränderten Werte zurück in den CPU Status geschrieben. Zwischen Prolog und Epilog können die Templates mit einigen Anpassungen verwendet werden. Alle Anpassungen werden in der Methode `imprint()` der Klasse Template durchgeführt. Diese Methode wird verwendet, um ein Template an einen Bytecode anzuhängen.

```
0:   iload_1
1:   iload_2
2:   iadd
3:   istore_0
4:   return
```

Abbildung 3.5: Bytecode des *jit_add* Templates

Anpassen der Indizes ins Array von lokalen Variablen

Die Codeerzeugung im Emulator findet in zwei Läufen statt. Im ersten Lauf wird ermittelt, welche Teile des CPU Status von den verwendeten Templates gebraucht werden. Jedem Register wird dabei ein LVI zugewiesen. Jedem verwendeten Immediate Operanden wird ebenfalls ein LVI zugeteilt. Diese zugewiesenen Indizes stimmen i.d.R. nicht mit den Indizes in der jeweiligen Template Methode überein. Daher werden bei allen Bytecode Befehlen im Template, die auf einer lokalen Variable arbeiten, die dort vorhandenen Indizes durch die tatsächlich verwendeten ersetzt.

Ausdehnen des Bytecodes

Für einige JVM Instruktionen existieren spezielle Varianten, die durch Kodierung ihres Parameter in den Befehl nur ein Byte benötigen. Ein Beispiel dafür ist die auch in Abb. 3.5 verwendete `iload` Anweisung. Diese legt den Inhalt einer lokalen Variable auf den Operandenstack. Der Index der lokalen Variablen wird normalerweise in einem gesonderten Byte im Bytecode kodiert. Für die Indizes 0–3 existieren jedoch spezielle Formen, die auch in Abb. 3.5 verwendet werden. Diese haben jeweils einen eigenen Bytecode, der den Index der lokalen Variable implizit enthält.

[4] Genau genommen handelt es sich hierbei um Indizes in das Array mit lokalen Variablen. Jedes Element in diesem Array ist 32 Bit groß. Da einige Datentypen zwei dieser Elemente belegen, werden für manche lokale Variablen mehrere Indizes verbraucht.

[5] Alternativ hätte man in den Templates auch direkt auf die statischen Variablen der CPU Klasse zugreifen können. Solche Zugriffe sind aber viel teurer als Zugriffe auf lokale Variablen. Mit wachsender Größe der erzeugten Codestücke wächst auch der Leistungsgewinn durch das Verwenden von lokalen Variablen an Stelle des direkten Zugriffs auf statische Variablen.

Ähnliche Varianten existieren auch für andere JVM Instruktionen. Bei der Anpassung der LVI kann es vorkommen, dass bei einer solchen Anweisung der Index durch einen solchen ersetzt wird, für den es keine spezielle Variante der Instruktion gibt. Deshalb müssen die spezialisierten Varianten durch die allgemeinen Instruktionen ersetzt werden. Da hierbei für den Parameter ein zusätzliches Byte benötigt wird, dehnt sich der Bytecode aus. Die Ausdehnung des Bytecode erfolgt einmalig bereits bei der Instanzierung eines Template Objekts.

Aufbau des Constant Pool

Für die generierte Klasse muss ein Constant Pool aufgebaut werden. In diesem befinden sich konstante Werte aller Art. Beispielsweise muss für Immediate Werte, die nicht aus dem Bereich -1–5 stammen, ein Eintrag im Constant Pool erzeugt werden. Auch symbolische Methodenreferenzen, die z.B. beim Aufruf einer statischen Methode benötigt werden, finden sich im Constant Pool.

Es sei an dieser Stelle noch erwähnt, dass nicht alle JVM Instruktionen in Template Methoden unterstützt werden. Hierzu zählen die `lookupswitch` und `tableswitch` Instruktionen, wodurch bei der Programmierung von Templates inbesondere auf switch Blöcke verzichtet werden muss.

3.2.3 Dynamische Klassengenerierung mit dem Emit Modul

Mit Hilfe der `Emit` Klasse ist es möglich, dynamisch Klassen zu generieren und Instanzen dieser Klassen zu erzeugen. Die erzeugten Klassen haben neben einem leeren Konstruktur nur eine einzige Methode, die `execute()` Methode. Eine neue Klasse wird mit der Methode

```
public static Code lambda(String base, String signature);
```

generiert. Der Methode wird der Name der Superklasse im Parameter `base` und die Signatur der `execute()` Methode mitgegeben. Die Signatur ist an dieser Stelle nicht zu verwechseln mit der Signatur beim Templatemechanismus. Bei der hier verwendeten Signatur handelt es sich um einen Java Methodendeskriptor [LY99, S.102]. Dieser beschreibt Parameter und Rückgabewert der Methode. Als Rückgabe liefert `lambda()` ein Objekt der Klasse `Code`, welches den Bytecode der `execute()` Methode repräsentiert. Die `Code` Klasse bietet Unterstützung bei der Generierung von Java Bytecode. Für nahezu jede JVM Instruktion existiert eine Methode, mit deren Hilfe diese Instruktion dem Bytecode hinzugefügt werden kann. Dinge wie das Anlegen von Einträgen im Constant Pool erledigen diese Methoden bei Bedarf automatisch. Mächtigere Unterstützungsmechanismen beinhalten das Konzept von `Labels`. Mit ihnen ist es möglich, dem Bytecode Sprungbefehle anzuhängen, deren Zielposition zum Zeitpunkt des Anfügens noch unbekannt ist. Beim Setzen der Position eines Labels werden die entsprechenden Offsets nachträglich automatisch eingefügt. Für ein weitergehendes Verständnis empfiehlt sich ein Blick in den gut dokumentierten Quellcode der Klasse. Mit Hilfe der `imprint()` Methode ist es möglich, Templates an ein Code Objekt anzuhängen.

3.3 Regeln für den Entwurf von Templatemethoden

In diesem Abschnitt sollen einige Besonderheiten behandelt werden, die beim Entwurf von Template Methoden zu beachten sind.

3.3.1 Allgemeines

Ein Code-Template muss immer nach dem Muster

```
public static void jit_<Name>(...) { ... }
```

deklariert sein. Das Präfix *jit_* macht das Template für die Factory erkennbar. Der Name des Code-Templates muss eindeutig sein, das Überladen von Template Methoden wird nicht unterstützt. Ein Code-Template darf keinen Wert zurückgeben. Dies würde keinen Sinn machen, da der Code eines Code-Templates als Bestandteil eines größeren Codeblocks verwendet wird. Bei der Anpassung des Templates wird die `return` Anweisung aus dem Bytecode entfernt, da diese zum Verlassen des Codeblocks führen würde. Dabei wird nur die void Variante der JVM `return` Instruktion unterstützt. Dies hat nebenbei zur Folge, dass in einem Template kein return Statement verwendet werden darf, um das Template vorzeitig zu verlassen.

```
public static void
jit_branch(int NPC, int PC, int offset, boolean cond) {
  if(!cond) {
    NPC = PC + 4;
    return;
  }
  NPC = PC + offset;
}
```

Das Template für einen bedingten Sprung in diesem Beispiel funktioniert nicht wie erwartet. Bei der Anpassung des Templates für die Verwendung in der `execute()` Methode wird die `return` Instruktion aus dem Bytecode entfernt. Der Sprung würde unabhängig von der Bedingung immer ausgeführt[6].

3.3.2 Initialisierung der lokalen Variablen auf allen Pfaden

Der Java Bytecode Verifier führt beim Laden einer Klasse einige Prüfungen zur Gültigkeit der Klasse durch [LY99, S.140]. Mitunter dürfen lokale Variablen erst dann gelesen werden, wenn sie vorher auch geschrieben wurden. Der Verifizierungsprozess schlägt also fehl, wenn eine uninitialisierte lokale Variable im Bytecode gelesen wird. Für den Entwurf von Code-Templates bedeutet dies, dass jede Variable, die in einem Template geschrieben wird, auf *allen* möglichen Pfaden im Code-Template geschrieben werden muss. Anschaulich soll das an folgendem Beispiel erklärt werden.

[6] Selbst wenn die `return` Instruktion nicht aus dem Code entfernt werden würde, würde sich das Template nicht verhalten wie gewünscht. Durch die `return` Instruktion würde der Codeblock verlassen ohne den Epilog auszuführen. Die Änderung würde nicht zurück in den CPU Status geschrieben und das Template damit wirkungslos.

```
public static void
jit_add(int Rd, int op1, int op2, boolean cond) {
  if (cond) {
    Rd = op1 + op2;
  }
}
```

Der Emulator generiert Prolog und Epilog in den Regel[7] mit einer gewissen Intelligenz. Nur die Register des CPU Status, die in einem Template lesend verwendet werden, werden im Prolog aus dem CPU Status geladen. Umgekehrt werden nur die Register, welche in einem Template geschrieben werden, wieder in den CPU Status eingebracht. Obiges Beispieltemplate führt eine Addition nur dann durch, wenn eine bestimmte Bedingung erfüllt ist. Unter der Voraussetzung, dass die beiden Operanden vom Zielregister verschieden sind, werden im Prolog also nur die beiden Operanden geladen. Im Epilog wird lediglich das Zielregister in den CPU Status zurückgeschrieben, da die beiden Operanden nicht verändert wurden. Dies führt zu einem lesenden Zugriff auf `Rd` im Epilog. `Rd` wurde aber nur initialisiert, falls die Bedingung erfüllt war. Im Epilog würde somit u.U. ein Zugriff auf eine nicht initialisierte lokale Variable erfolgen, was vom Java Bytecode Verifier erkannt und mit einem `VerifyError` quittiert wird. In solchen Fällen muß also sichergestellt werden, dass die Variable `Rd` initialisiert wird, wie in folgendem Beispiel.

```
public static void
jit_add(int Rd, int op1, int op2, boolean cond) {
    Rd = cond?op1 + op2:Rd;
}
```

[7] d.h. in allen bisher vorliegenden Implementierungen

Kapitel 4

Unterstützung der ARM Architektur

In diesem Kapitel soll das JIT Modul für die ARM Architektur beschrieben werden. Dieses Modul ermöglicht die Ausführung von Altanwendungen für den ARM7TDMI [Atm99] in JXEmu. Es wird außerdem eine FPA (floating-point accelerator für ARM CPUs) emuliert, ein Ko-Prozessor zur Ausführung von Gleitkommaoperationen auf ARM Prozessoren [Adv96, Kap. 8].

Zunächst erfolgt eine kurze Einführung, welche einen Überblick über den ARM7TDMI vermitteln soll. Anschließend werden einige Besonderheiten der ARM Architektur behandelt und deren Realisierung im ARM JIT Modul vorgestellt.

Alle Ausführungen zur ARM Architektur beziehen sich auf den Prozessor ARM7TDMI, welcher aber zu einer Vielzahl anderer Prozessoren der ARM Familie kompatibel ist.

4.1 Einführung in die ARM Architektur

Beim ARM7TDMI handelt es sich um ein Mitglied der ARM Familie von 32 Bit RISC (Reduced Instruction Set Computer) Allgemeinzweck-Prozessoren. Im Vergleich zu CISC (Complex Instruction Set Computer) Architekturen ist der Befehlssatz eines RISC Prozessors und auch deren Dekodierung wesentlich einfacher. Die ARM Architektur basiert auf einer von Neumann load/store Architektur. Der Zugriff auf den Hauptspeicher ist nur mit Hilfe von speziellen load/store und swap Befehlen möglich. Die restlichen Instruktionen operieren ausschließlich auf den Registern des Prozessors.

4.1.1 Befehlssatz

Der ARM7TDMI implementiert die ARMv4T Befehlssatzarchitektur, welche eine Erweiterung des ARMv4 Befehlssatzes um den Thumb Befehlssatz darstellt. Es werden also zwei vollständige Befehlssätze, der ARM32 Standard- und der 16 Bit Thumb Befehlssatz unterstützt. Der Thumb Befehlssatz ermöglicht eine wesentlich höhere Codedichte, da dessen Instruktionen nur eine Länge von 16 Bit haben. Die Thumb Instruktionen arbeiten jedoch auf den selben 32 Bit Registern wie die ARM32 Instruktionen, womit der Leistungsvorteil gegenüber einem traditionellen 16 Bit Prozessor mit 16 Bit Registern größtenteils erhalten bleibt. Zudem ist ein Wechsel

zwischen den beiden Befehlssätzen innerhalb eines Programms mit einer speziellen Instruktion möglich. Damit kann in einigen Teilen des Programms auf die volle Leistung des ARM32 Befehlssatzes zurückgegriffen werden, während in anderen Teilen die Codedichte durch Verwendung des Thumb Befehlssatzes optimiert wird.

Ausführungsmodi des ARM7TDMI

Aus Sicht des Programmierers kann sich der ARM7TDMI in einem von zwei Ausführungsmodi[1] befinden, dem *ARM Modus* und dem *Thumb Modus*. Die Instruktionen müssen an 4 Byte bzw. 2 Byte Grenzen im Speicher für den ARM Modus bzw. den Thumb Modus ausgerichtet sein. Dies bedeutet insbesondere, dass beim Lesen von Instruktionen aus dem SafeMemory (s. Abs. 2.2.2) immer der schnelle Pfad durchlaufen wird. Beim Wechsel zwischen diesen beiden Modi bleibt der Prozessorstatus und der Arbeitsmodus erhalten.

4.1.2 Registersatz

Der Registersatz des ARM7TDMI umfasst insgesamt 37 Register, davon sind 31 Allgemeinzweckregister und sechs Statusregister. Von diesen sind jedoch niemals alle auf einmal sichtbar. Die sichtbaren Register hängen vom Arbeitsmodus und dem Ausführungsmodus ab. Für JXEmu sind nur die im Benutzermodus sichtbaren Register interessant, da die Funktion des Betriebssystems durch das Kernel Modul direkt in Java nachgebildet wird. Damit bleiben die 15 Allgemeinzweckregister `r0`–`r14` sowie der Befehlszähler `r15` und das Statusregister `CPSR` für die Ausführung im ARM Modus. Im Thumb Modus ist nur eine Untermenge dieser Register sichtbar, die Allgemeinzweckregister `r0`–`r7`, der Stackpointer, das Linkregister und der Befehlszähler. Abb. 4.1 zeigt die jeweiligen Registersätze und deren Abbildung aufeinander.

Das Link Register

Neben der Verwendung als Allgemeinzweckregister besitzt das Register `r14` die Funktion des Link Registers. Die `BL` (*Branch and Link*) Anweisung speichert die Adresse der nächsten Anweisung im Link Register und führt anschließend einen Sprung durch. Dies kann zum Aufruf von Unterprogrammen verwendet werden, aus denen dann durch Transfer des Link Registers in den Befehlszähler zurückgekehrt werden kann.

Der Befehlszähler

Da die Adressen von Instruktionen immer ausgerichtet sein müssen, sind die beiden niederwertigsten Bits des Befehlszählers im ARM Modus immer 0. Im Thumb Modus ist das niederwertigste Bit immer 0.

[1] Dies ist nicht zu verwechseln mit den Arbeitsmodi des Prozessors, die u.a. Benutzermodus und Systemmodus enthalten. Der ARM Prozessor unterstützt sieben solcher Arbeitsmodi. Für JXEmu ist jedoch nur der Benutzermodus von Interesse.

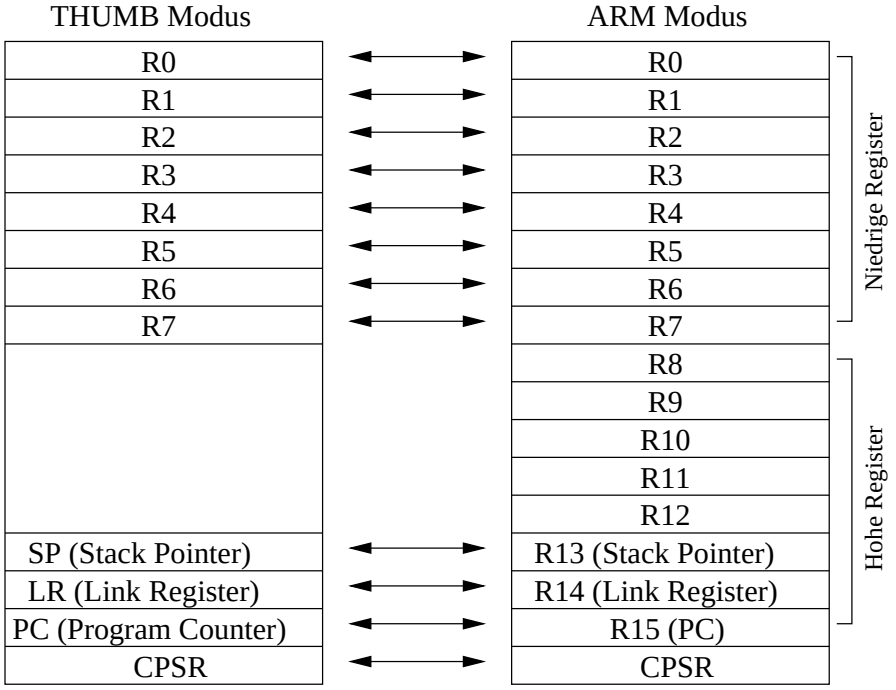

Abbildung 4.1: Sichtbare Register und deren Abbildung aufeinander in ARM und Thumb Modus. Im Thumb Modus sind die Register `R8`–`R15` nicht direkt zugreifbar. Es existieren jedoch spezielle Varianten der `MOV`, `ADD` und `CMP` Instruktionen, mit denen Werte aus bzw. in hohe Register übertragen, hohe Register zu niedrigen addiert und hohe mit niedrigen Register verglichen werden können.

Das Statusregister CPSR

Das CPSR enthält die condition code Flags und einige Bits, die den aktuellen Modus des Prozessors beinhalten. Für JXEmu interessant sind das Negative-Flag, das Zero-Flag, das Carry-Flag und das Overflow-Flag in den Bits 31–28 in dieser Reihenfolge im CPSR sowie Bit 5, welches gesetzt ist, wenn sich der Prozessor im Thumb Modus befindet. Bits 8–27 sind reserviert, die übrigen Bits haben für JXEmu keine Bedeutung.

Die virtuelle CPU für den ARM

Die virtuelle CPU für den ARM ist in der Klasse `CPU_ARM` zu finden. Neben den von der Superklasse `CPU` geerbten Registern, welche in Abs. 2.2.3 beschrieben wurden, enthält sie die Register `r0`–`r14`, die Pseudoregister `shiftedOp2` und `carry` sowie das Statusregister `CPSR`. Der Befehlszähler wird durch das in Abs. 2.2.3 beschriebene Verfahren emuliert. Die beiden Pseudoregister dienen dem Transport von Zwischenwerten zwischen verschiedenen Templates in einer Template Sequenz, ihre Benennung stammt von dem Zweck zu dem sie ursprünglich eingeführt wurden. In der weiteren Entwicklung des JIT wurden sie aber auch für andere Zwecke

Sig	Kontext	Bedeutung
I	Wert	Immediate. Wert wird direkt dem Kontext entnommen.
P	-	PC. Befehlszähler wird direkt als Wert eingesetzt.
N	-	NPC. Zieladresse nach indirekten Sprüngen.
Z	-	Z-Flag. Ausführungkriterium für bedingte Anweisungen.
R	Index: 0–14	Allgemeinzweckregister des ARM `R0`–`R14`.
F	Index: 0–8	Register der FPA `F0`–`F7` und Hilfsregister `F8`.
C	-	CPSR. Statusregister der ARM CPU.
Q	-	FPSR. Statusregister der FPA.
U	-	Carry. Pseudoregister.
V	-	shiftedOp2. Pseudoregister.

Tabelle 4.1: Template-Signatur Buchstaben für den ARM-JIT

verwendet. Beispielsweise erhalten viele ARM Instruktionen ein Register als zweiten Operanden, dessen Bits vor der eigentlichen Operation noch verschoben oder rotiert werden. Der ARM Prozessor stellt hierfür einen Barrel Shifter bereit. Das carry Flag wird bei logischen Operationen dann entsprechend dem carry Flag des Barrel Shifters gesetzt. Die Berechnung des zweiten Operanden erfolgt für viele Instruktionen also immer auf dieselbe Art und Weise, weshalb es sinnvoll ist, hierfür eigene Templates bereitzustellen. Die errechneten Zwischenwerte müssen für das nächste Template zwischengespeichert werden, wofür die Pseudovariablen eingeführt wurden.

Tab. 4.1 zeigt die möglichen Buchstaben für eine Template Signatur im ARM JIT. Dort sind auch die Register der FPA zu finden. Die Emulation der FPA wird in einem eigenen Abschnitt 4.6 beschrieben.

4.1.3 Speichersicht

Der ARM7TDMI sieht den Speicher als eine lineare Sammlung von Bytes die von 0 beginnend nummeriert sind. Die Bytes 0–3 enthalten das erste Wort, Bytes 4–7 das zweite Wort usw. Der ARM7TDMI unterstützt sowohl Little Endian als auch Big Endian Byteorder. Das JIT Modul für ARM von JXEmu unterstützt im Moment allerdings nur Little Endian. Eine Erweiterung zur Unterstützung der Big Endian Architektur wäre leicht realisierbar, da JXEmu mit dem PowerPC bereits eine Big Endian Gastarchitektur unterstützt und das Memory Modul an seiner Schnittstelle Funktionen für den Big Endian Speicherzugriff bereitstellt. Eine Implementierung von Big Endian im ARM-JIT erschien jedoch nicht notwendig, da eine Anwendung nur eine Byteorder verwendet und für Little Endian übersetzt werden kann. Little Endian ist zudem die Standard Byteorder für ARM Linux.

4.2 Behandlung des Befehlszählers und Pipelining

Zur Steigerung der Leistung implementiert der ARM eine 3-stufige Pipeline, die aus den Stufen *Fetch* (Instruktion aus dem Speicher holen), *Decode* (Anweisung dekodieren) und *Execute*

(Instruktion ausführen) besteht. Diese drei Stufen werden parallel durchgeführt, zum Zeitpunkt der Ausführung einer Instruktion wird also die folgende Instruktion bereits dekodiert und die übernächste Instruktion aus dem Speicher geholt. Wird ein Sprung durchgeführt sind die Inhalte der Pipeline ungültig und sie muss von neuem gefüllt werden, was die Kosten einer Sprunganweisung erhöht.

Aus diesem Grund ist der Befehlszähler der aktuellen Instruktion immer um zwei Anweisungen voraus, also 8 Bytes im ARM Modus und 4 Bytes im Thumb Modus. Übersetzer sind sich dieser Tatsache bewusst und berücksichtigen dieses Wissen bei der Erzeugung von Binärcode für den ARM Prozessor. Zum Beispiel muss bei einem relativen Sprung, bei dem ein gewisses Offset auf den Befehlszähler addiert oder von ihm subtrahiert wird, bei der Berechnung des Offset dieser Pipelineeffekt berücksichtigt werden. Das Überspringen der nächsten Anweisung im ARM Modus würde also mit einem relativen Sprung um das Offset 0 bewerkstelligt.

Bei der Erzeugung von Code wird der Befehlszähler immer als Direktwert in den Code eingefügt. Dies ist ohne weiteres möglich, da der Befehlszähler zu jedem Zeitpunkt einen wohlbekannten Wert enthält. Dadurch entfällt u.a. das dynamische Mitführen des Befehlszählers, was eine Additionsoperation nach jeder emulierten nativen Instruktion einspart und zudem den JIT Entwurf vereinfacht. Dem Emulator ist jedoch immer nur die Adresse der aktuellen Anweisung bekannt, nicht der tatsächliche im Befehlszähler vorhandene Wert, der durch die Pipeline der Adresse der Instruktion voraus ist. Die Methode `adjustPC()` der `CPU_ARM` prüft das Thumb Bit des `CPSR` Statusregisters und addiert entsprechend dem Ergebnis das passende Offset auf die übergebene Adresse. Die angepasste Adresse kann dann vom Emulator in den Code eingesetzt werden.

4.3 Bedingte Ausführung von Befehlen

Im ARM Modus kann jede Instruktion bedingt ausgeführt werden. Die verfügbaren Bedingungen finden sich in [Atm99, S.28]. Damit kann das Leeren der Pipeline vermieden werden, wenn nur einzelne Anweisungen bedingt ausgeführt werden sollen und ein Sprung über diese Anweisung teurer wäre als die Zeit für das (nicht-)Ausführen der Anweisung selbst.

Was sich bei der Entwicklung von ARM Binärcode als äußerst nützlich erweist, ist für den Entwurf eines JIT ein lästiges Problem. Die intuitive Herangehensweise wäre, für jede Bedingung ein Template bereitzustellen, welches das Z-Flag entsprechend der Bedingung und dem aktuellen Inhalt des Statusregisters setzt. Jedes Template, das zur Nachbildung der ARM Anweisung selbst verwendet wird, müsste dann das Z-Flag überprüfen und seinen Dienst nur dann verrichten, wenn dieses gesetzt ist. Die Mehrheit aller ARM Instruktionen werden allerdings mit der Bedingung *always*, also unbedingt, ausgeführt. Für diese stellt die redundante Prüfung des Z-Flags — oft mehrmals für eine native Instruktion — sowie dessen vorheriges Setzen eine signifikante Leistungseinbuße dar. Wenn man bedenkt, dass viele Templates nur winzige Codestücke realisieren, kann der Overhead bei dieser Herangehensweise bis zu 100% betragen.

Deshalb wurde eine aufwendigere Realisierungsmethode gewählt, die sich jedoch speziell bei häufig verwendeten Codeblöcken auszahlt. Für jedes Template wird ein spezielle Variante für die unbedingte Ausführung bereitgestellt, die durch das Suffix *_AL* gekennzeichnet wird. Abb. 4.2 zeigt die beiden Varianten des `jit_add` Template. Die unbedingte Variante unter-

```
public static void
jit_add(int Rd, int op1,
    int op2, boolean cond) {
  Rd=cond?op1+op2:Rd;
}
```

```
public static void
jit_add_AL(int Rd,
    int op1, int op2) {
  Rd=op1+op2;
}
```

Abbildung 4.2: Bedingte und unbedingte Versionen von Templates

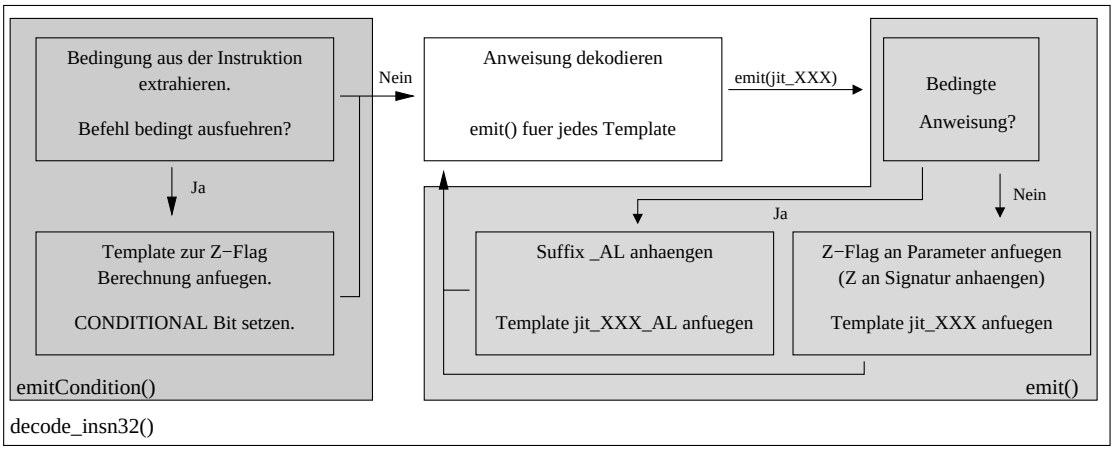

Abbildung 4.3: Dekodierung von ARM32 Instruktionen in `decode_insn32()`

scheidet sich von der bedingten jeweils lediglich durch das Suffix *_AL* sowie das Fehlen der Bedingung in der Parameterliste. Da dies bei allen Templates der Fall ist, die zur Emulation von ARM32 Instruktionen verwendet werden, steigt die Komplexität der Dekodierungsfunktion nur geringfügig an. Abb. 4.3 zeigt den Verlauf der Dekodierung einer ARM32 Instruktion. Zu Beginn wird die Bedingung aus der ARM32 Anweisung gelesen, welche sich immer in den Bits 31–28 befindet. Wird dort nicht die *always* Bedingung gefunden, so wird im `branchType` Mitglied des `DecodedInstruction` Objekts bereits das *CONDITIONAL* Bit gesetzt, womit dieses bereits für alle Sprunganweisungen erledigt ist. Als Beginn der Template Sequenz wird dann ein Template eingefügt, welches die Bedingung am Statusregister prüft und das Z-Flag entsprechend setzt. Bei einer unbedingten Anweisung wird dieser Schritt einfach übersprungen. Anschließend erfolgt der normale Dekodierungsvorgang, bei dem eine Sequenz von Templates erzeugt wird, welche das Verhalten der nativen Anweisung nachbildet. Für jedes an die Sequenz anzuhängende Template wird dabei die `emit()` Methode aufgerufen. Diese prüft das Ergebnis der Bedingungsprüfung am Anfang. Handelt es sich um eine unbedingte Instruktion, so wird dem Namen des Templates das Suffix *_AL* angehängt und das Template in dieser Form in die Anweisung eingefügt. Im anderen Fall erwartet das Template als letzten Parameter das Z-Flag,

welches in der bisherigen Template Signatur nicht berücksichtigt ist. Daher wird der Buchstabe `Z` der Signatur angehängt.

Man muss sich an dieser Stelle bewusst sein, dass zur Modifikation von Template-Namen bzw. -Signatur teure Java Stringoperationen verwendet werden. Beim Kompilieren von Codeblöcken wird jedoch jede Instruktion nur einmalig dekodiert, was die Leistungseinbußen erträglich erscheinen lässt. Der Hauptvorteil ist eine weniger komplexe Dekodierungsfunktion — alternativ hätte man an jeder Stelle in der Dekodierungsfunktion, an der ein Template in die Sequenz eingefügt wird, eine Fallunterscheidung durchführen müssen, um dann die fertigen Strings für Namen und Signatur des Templates direkt als Konstanten einzusetzen.

4.4 Realisierung von Systemaufrufen

Systemaufrufe werden durch das Kernel Modul realisiert, welches den JITs über die Methode `do_syscall()` zugänglich gemacht wird. Diese erwartet als ersten Parameter die Nummer des Systemaufrufs und als weitere Parameter die Parameter des Systemcalls. Die Methode muss aus einem Template aufgerufen werden. Um Nummer und Parameter des Systemcalls bestimmen zu können, ist es notwendig, die ARM-Linux Systemcall Schnittstelle zu kennen.

Der ARM verfügt über die Instruktion `swi` zur Erzeugung eines Software Interrupt. Dieser kann durch das Betriebssystem behandelt werden. Direkt in die Instruktion kann ein 24 Bit langer Wert kodiert werden, welcher vom Prozessor nicht interpretiert wird. Bei ARM Linux wird in diesem Wert die Nummer des Systemaufrufes übergeben.

Für die Systemaufrufschnittstelle von ARM-Linux ist keine Dokumentation verfügbar. Die Information, auf welche Art und Weise die Parameter an den Linux Kernel übermittelt werden, lässt sich daher am besten aus dem relativ gut kommentierten Quellcode der GNU C-Bibliothek entnehmen. Die Schnittstelle sieht vor, dass bis zu fünf Parameter in den Registern `r0`–`r4` übergeben werden können. Einige Systemaufrufe wie `mmap()` erwarten jedoch mehr als fünf Parameter. In diesem Fall wird der Parameterblock auf den Stack gelegt und nur der Stackpointer in Register `r0` übergeben.

4.5 Thumb Befehlssatz und Befehlssatzwechsel

JXEmu unterstützt den 16 Bit Thumb Befehlssatz sowie den Wechsel zwischen ARM32 und Thumb Befehlssatz innerhalb eines Programms, das sog. *Thumb Interworking*. Die Dekodierung der Thumb Befehle ist simpler als die der ARM32 Befehle. Zu fast jedem Thumb Befehl gibt es einen äquivalenten ARM32 Befehl, daher sind praktisch alle nötigen Templates bereits vorhanden.

Im Gegensatz zum ARM Modus können im Thumb Modus nicht beliebige Befehle an eine Bedingung geknüpft werden. Es existiert nur eine einzige Anweisung zur Durchführung eines bedingten Sprungs. Die Methode `decode_insn16()` ist für die Dekodierung von Thumb Befehlen zuständig. In der Hauptdekodierungsfunktion wird das Thumb Bit im Statusregister überprüft, ein entsprechend dem Modus 16 oder 32 Bit langer Befehl aus dem Memory gelesen und die richtige Dekodierungsmethode aufgerufen.

Der Wechsel zwischen den Ausführungsmodi kann ausschließlich durch die BX Instruktion erfolgen. BX führt einen indirekten Sprung durch, dessen Sprungziel einem Register entnommen wird. Da sowohl im ARM als auch im Thumb Modus die Adressen durch zwei teilbar sein müssen, ist das niederwertigste Bit der Adresse einer Instruktion immer 0. Dieses Bit wird beim BX Sprung in das Thumb Bit kopiert, d.h. wenn sich im Operandenregister bei einer BX Anweisung eine ungerade Zahl befindet, wird in den Thumb Modus umgeschaltet, ansonsten in den ARM Modus.

Um die Anweisungen korrekt dekodieren zu können, muss das Thumb Bit beim Aufruf der Dekodierfunktion jeweils korrekt gesetzt sein. Um dies sicherzustellen existieren zwei mögliche Herangehensweisen. Zum einen könnte man verlangen, dass Emulatoren Codeblöcke, die Sprünge enthalten, nur im Interpretermodus erzeugen dürfen. Der Emulator müsste also beim ersten Durchlaufen des Codestücks den Code interpretieren und nebenbei den Codeblock generieren, der dann für zukünftige Ausführungen verwendet werden könnte.

Die andere Möglichkeit wäre eine Erweiterung des `branchType` in der dekodierten Anweisung um ein Bit, welches einen Sprung kennzeichnet, der aus dem Codeblock herausführen muss. Diese Lösung würde die Freiheit des Emulators bei der Codeerzeugung jedoch signifikant einschränken, da der BX Befehl in den meisten Fällen einen regulären indirekten Sprung vollzieht. Deshalb wurde die erste Lösungsform gewählt. Sie stellt keine besondere Einschränkung für den Emulator dar, da dieser bei der Erzeugung von Codeblöcken, die indirekte Sprünge beinhalten, ohnehin eine Form der Interpretierung betreiben muss, um das Sprungziel vorhersagen zu können.

4.6 Emulierung der FPA

Der ARM7TDMI verfügt über keinen integrierten Ko-Prozessor zur Bearbeitung von Gleitkommaoperationen. Er stellt jedoch eine Schnittstelle für Ko-Prozessoren bereit, an die bis zu 16 Ko-Prozessoren zur Erweiterung der Funktionalität des ARM7TDMI Kerns angeschlossen werden können [Atm99, S.135].

Die Ansteuerung der Ko-Prozessoren erfolgt über spezielle Anweisungen zur Ausführung von Operationen auf dem Ko-Prozessor, load/store Operationen für dessen Register sowie Operationen zum direkten Transfer von Registern zwischen ARM7TDMI und Ko-Prozessor.

Bei der Verwendung von Ko-Prozessor Anweisungen an einen nicht angeschlossenen Ko-Prozessor erzeugt der ARM einen Trap. Auf diesen kann das Betriebssystem reagieren und eine Emulation der FPA bereitstellen. Ein Beispiel hierfür ist der FPA Emulator *nwfpe* von NetWinder. Dieser wird auch im Linux Kernel verwendet.

Auch JXEmu emuliert einen Großteil der FPA. Die FPA implementiert vollständig den Standard ANSI/IEEE Std. 754-1985. Sie verfügt über acht 81 Bit breite Gleitkommazahlregister, bestehend aus einer 64 Bit Mantisse, einem 15 Bit Exponenten und einem Vorzeichenbit, sowie dem Statusregister FPSR, in dem verschiedene Flags wie z.B. ein Flag für die Division durch 0 existiert. JXEmu verwendet die Möglichkeiten von Java zur Emulierung der FPA, woraus einige Beschränkungen resultieren. Diese betreffen die verfügbaren Rundungsmodi sowie die zur Verfügung stehende Genauigkeit, sollen hier aber nicht weiter ausgeführt werden.

Kapitel 5

Verfügbare Codeerzeugungsstrategien

In diesem Kapitel werden die verschiedenen Codegeneratoren, die derzeit für JXEmu vorhanden sind, vorgestellt. Die Beschreibung der Codegeneratoren erfolgt in der Reihenfolge der Entwicklung.

Abb. 5.1 zeigt die Klassenhierarchie der verfügbaren Codegeneratoren. Beim *`Emulator`* handelt es sich lediglich um eine abstrakte Klasse, die von allen Emulatoren benötigte Funktionalität enthält und die Schnittstelle des Emulator-Moduls festlegt. Der einfachste Codegenerator `EmulatorByInstruction` interpretiert den nativen Code Anweisung für Anweisung. Dabei werden Codeblöcke für jedes verwendete Template erzeugt.

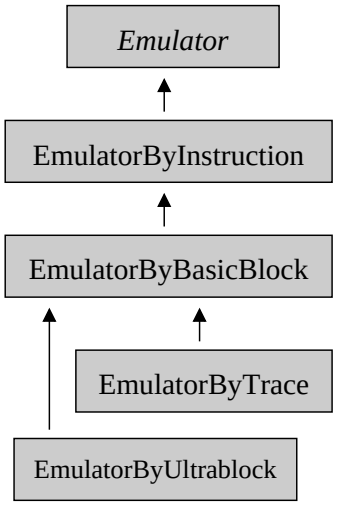

Abbildung 5.1: Übersicht der derzeit vorhandenen Codegeneratoren.

Der `EmulatorByBasicBlock` erzeugt Code, der bereits die Emulation der Anweisungen eines ganzen Basicblocks des nativen Codes nachbildet. Ein Basicblock ist dabei definiert als eine Folge von Anweisungen, die entweder mit dem Programmeintrittspunkt, dem Ziel eines Sprungs oder einer einem Sprung folgenden Anweisung beginnt und mit dem nächsten Sprung endet. Ein Basicblock wird also immer sequentiell durchlaufen und enthält keine inneren Sprünge.

`EmulatorByTrace` versucht, Schleifen im Code zu erkennen, die etliche Male durchlaufen werden. Diese heißen Pfade werden dann zu einem Block kompiliert, der möglichst viele Durchläufe der Schleife an einem Stück ausführt.

Die Strategie der Schleifenerkennung bringt einige schwer lösbare Probleme mit sich. Der nächst komplexere Codegenerator, der `EmulatorByUltrablock`, versucht, diese Probleme zu lösen. Dabei werden populäre Basicblöcke in einem sog. *Ultrablock* zusammengefasst und entsprechender Verknüpfungscode innerhalb der Blocks erzeugt. Damit wird das Ziel verfolgt, möglichst lange erzeugten Code am Stück auszuführen.

Der Rest dieses Kapitels beschreibt die einzelnen Codegeneratoren detailliert und stellt die Stärken und Schwächen der einzelnen Implementierungen heraus. Benchmarks folgen in Kapi-

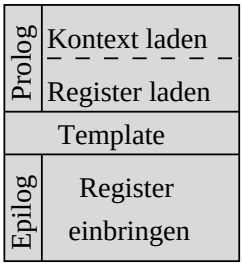

Abbildung 5.2: Aufbau eines `Instruction` Objekts

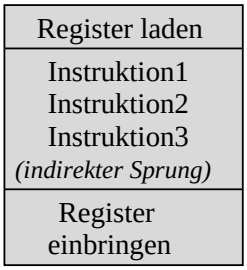

Abbildung 5.3: Aufbau eines Basicblocks

tel 6, weshalb in diesem Kapitel lediglich die allgemeinen Problematiken behandelt werden. Bei allen Optimierungen muss man stets im Hinterkopf behalten, dass die Übersetzung der Codeblöcke zur Laufzeit erfolgt. Jede Optimierung erhöht den Übersetzungsaufwand. Deshalb sind nur einfache Optimierungen sinnvoll, deren Zusatzaufwand bei der Codeerzeugung durch den effizienteren Code wieder ausgeglichen wird.

5.1 Interpreter: Der EmulatorByInstruction

Der `EmulatorByInstruction` (EBI) ist der einfachste verfügbare Codegenerator und realisiert den Interpretermodus von JXEmu. Es wird Instruktion für Instruktion emuliert. Hierzu wird zunächst die Anweisung an der aktuellen Position des Befehlszählers durch den JIT dekodiert, welcher, gekapselt in ein `DecodedInstruction` Objekt, eine Template Sequenz für die Emulation der Anweisung zurückliefert.

Der EBI erzeugt nun für jedes einzelne Template ein `Instruction` Objekt. Die `Instruction` Klasse ist eine vom Emit Modul erzeugte Klasse, deren `execute()` Methode folgende Signatur besitzt:

```
public abstract void execute(int[] context);
```

Sie erhält den Kontext für die Ausführung des Templates als Parameter. Der Aufbau des vom EBI generierten Code ist in Abb. 5.2 dargestellt. Im Prolog werden zunächst die einzelnen Elemente des übergebenen Kontext, die benötigten Teile des CPU Status sowie die verwendeten Immediate Werte in lokale Variablen geladen. Dann folgt der Code für das eigentliche Template, der für die Verwendung der vorliegenden lokalen Variablen angepasst wurde. Im Epilog werden dann die modifizierten Teile des CPU Status in die CPU zurückgeschrieben.

Die erzeugten `Instruction` Objekte sind für gleiche Template/Signatur Kombinationen identisch und können daher sehr oft wiederverwendet werden. Für das gleiche Template muss jedoch für jede verwendete Signatur eine eigene Klasse erzeugt werden, da Prolog und Epilog sowie die Interpretation des Kontext von der Signatur abhängen und sich der erzeugte Code hier unterscheidet. Die bereits erzeugten Objekte werden in einer Hashtabelle zwischengespeichert und bei Bedarf wiederverwendet.

```
0:   aload_1
1:   iconst_0
2:   iaload
3:   istore_2
4:   aload_1
5:   iconst_1
6:   iaload
7:   istore_3
8:   getstatic    #15; //Field arch/arm/CPU_ARM.PC:I
11:  invokestatic #19; //Method arch/arm/CPU_ARM._adjustPC:(I)I
14:  istore  5
16:  iload   5
18:  iload   3
20:  iadd
21:  istore  4
23:  iload_2
24:  iload   4
26:  invokestatic #25; //Method arch/jit/JIT_ARM.emul_store_reg:(II)V
29:  return
```

Abbildung 5.4: Vom EBI erzeugter Code für das Template *jit_add_AL* mit der Signatur *RPI*. Die Referenz auf das als Parameter übergebene Kontext Array befindet sich in der lokalen Variable 1. Die lokalen Variablen 2 und 3 wurden für die einzelnen Elemente des Kontextarrays belegt, also für den Index des Zielregisters in Variable 2 und den Direktwert in Variable 3. Die lokale Variable 4 wurde für den Wert des Zielregisters belegt, in lokale Variable 5 wird der Wert des Befehlszählers geladen.

5.1.1 Overhead im erzeugten Code

Aufgrund der häufigen Zugriffe auf die Hashtabelle und dem übermäßigen Verhältnis von Overhead durch Prolog/Epilog zu Nutzcode des Templates ist eine schlechte Leistung absehbar. Die Zugriffe auf die Hashtabelle erfolgen i.d.R. mehrmals pro nativer Instruktion, da diese meistens auf eine Sequenz aus mehreren Templates abgebildet werden, und sind sehr teuer, wenn man den bei der Emulation erzielten Fortschritt mit dem Aufwand der Zugriffe auf die Hashtabelle vergleicht. Um den Overhead durch Prolog und Epilog zu verdeutlichen, sind in Abb. 5.4 die JVM Instruktionen eines für das bereits bekannte *jit_add_AL* Template mit der Signatur *RPI* erzeugten Codes dargestellt. Der Code addiert einen Direktwert zum Wert des Befehlszählers und speichert das Ergebnis in einem Register. Der Kontext besteht also aus zwei Werten, dem Index des Zielregisters und dem Direktwert, welche in den Zeilen[1] 0–7 in die lokalen Variablen 2 und 3 geladen werden. Die Zeilen 8–14 laden den Wert des Befehlszählers in die lokale Variable 5. Die Zeilen 16–21 enthalten den eigentlichen Nutzcode des Templates, welcher Befehlszähler und Direktwert addiert und das Ergebnis in der für das Zielregister belegten lokalen Variable 4 speichert. In den Zeilen 23–26 wird das veränderte Register in den CPU Status eingebracht, in

[1]Die Zeilennummern richten sich nach der jeweiligen Position im Bytecode und sind daher nicht durchgehend nummeriert, da manche Instruktionen mehr als 1 Byte belegen.

Zeile 29 wird die `execute()` Methode schließlich verlassen.

Das Verhältnis von Overhead:Nutzcode beläuft sich damit auf 14:5 JVM Instruktionen und 22:8 Bytes. Wie man in Abb. 5.4 noch erkennen kann, werden nur die Teile des CPU Status geladen, die auch gelesen werden. In diesem Beispiel wird das Zielregister also nicht geladen. Im Epilog werden nur die Teile eingebracht, die geschrieben wurden, hier also lediglich das Zielregister. Eine weitere Leistungsbremse ist der Dekodiermechanismus, der selbst beim wiederholten Antreffen bereits durchlaufener Instruktionen jedesmal aufs Neue die Abbildung auf die einzelnen Templates vornehmen muss.

5.1.2 Befehlszähler als Operand

Entgegen der Behauptung in Tab. 3.1 auf S. 25 wird der Befehlszähler dynamisch im Code aus dem `PC` Register geladen. In der Tat gilt die Behauptung nur für die restlichen Codegeneratoren, nicht für den EBI. Ein Einsetzen als Direktwert macht beim EBI keinen Sinn, da der Zugriff auf den CPU Status im EBI selbst und nicht im generierten Code stattfinden müsste. Der vom JIT gelieferte Kontext müsste dann im EBI um den Wert des PC erweitert werden, welcher dann im dynamischen Code aus dem Kontext geladen werden würde. Die Anpassung würde beim EBI also einen Leistungsverlust bewirken, weshalb er in dieser Hinsicht eine Sonderbehandlung erfährt.

5.1.3 Behandlung von Sprüngen

Die Behandlung von Sprüngen wurde in Abs. 2.2.3 beschrieben. Für direkte Sprünge wird kein Code erzeugt. Der EBI muss daher den Effekt solcher Sprünge selbst nachbilden und das `PC` Register entsprechend den in der dekodierten Instruktion gekapselten Informationen aktualisieren. Im Falle eines bedingten Sprungs ist das Z-Flag zu prüfen. Bei indirekten Sprüngen muss der Wert aus dem `NPC` Register in das `PC` Register kopiert werden. Auch hier ist im Falle eines bedingten Sprungs vorher das Z-Flag zu prüfen.

5.1.4 Bewertung

Aufgrund des in Abs. 5.1.1 diskutierten sehr hohen Overheads kann mit einer sehr schwachen Leistung gerechnet werden. Die Stärken des EBI liegen in anderen Bereichen. Durch die feine Granularität der Codeerzeugung und Ausführung ist der CPU Status nach der Abarbeitung jeder nativen Instruktion, wenn gewünscht sogar nach der Ausführung jedes Templates, greifbar. Dadurch wird der EBI zu einer sehr wertvollen Hilfe bei der Fehlersuche im JIT Entwurf.

5.2 Übersetzung in kleinen Häppchen: EmulatorByBasicBlock

Der `EmulatorByBasicBlock` (EBBB) setzt bei den Problemen des EBI an. Um das Verhältnis von Overhead und Nutzcode zu verbessern, soll der Anteil des Nutzcodes in einem Codeblock erhöht werden. Als nächste Stufe wurde daher die Granularität von Basicblocks gewählt.

5.2.1 Aufbau des Basicblocks

Jeder Basicblock kann durch seine Startadresse beschrieben werden[2]. Der Basicblock enthält die Anweisungen von seiner Startadresse bis zum nächsten Sprung. Da der Basicblock einen kleinen Ausschnitt der Altanwendung an einer festen Position darstellt, können im Gegensatz zu den `Instruction` Objekten des EBI alle Informationen direkt in den erzeugten Code kodiert werden. Die `execute()` Methode eines Basicblocks erhält daher keine Parameter.

Der Aufbau eines Basicblocks ist in Abb. 5.3 zu sehen. Prolog und Epilog stimmen, bis auf das Laden des Kontext Parameters, das beim Basicblock entfällt, mit denen der `Instruction` Objekte überein. Sie fallen in der Regel jedoch etwas umfangreicher aus, da für mehrere Instruktionen auch mehr Teile der CPU benötigt werden. Der Nutzcode besteht nun nicht mehr nur aus einem Template, sondern aus den gesamten Templates, die zur Nachbildung aller Anweisungen des Basicblocks verwendet wurden.

5.2.2 Behandlung des abschließenden Sprungs

Für den Sprung am Ende des Basicblocks wird nur dann Code erzeugt, wenn es sich um einen indirekten Sprung handelt. In diesem Fall ist nach Ausführung des Sprungs in `NPC` die Adresse der nächsten Anweisung zu finden. Direkte Sprünge werden direkt vom Emulator behandelt. Hierzu werden für jeden Basicblock der Sprungtyp, die Zieladresse und die Adresse der Anweisung nach dem Sprung für den Fall eines nicht ausgeführten bedingten Sprungs gespeichert. Der Emulator führt dann nach Ausführung des Basicblocks abhängig von Sprungtyp und Z-Flag die notwendigen Schritte zur Aktualisierung des Befehlszählers durch.

5.2.3 Wiederverwendung des Basicblocks

Die kompilierten Basicblöcke werden in einer Hashtabelle gespeichert. Wird die Startadresse eines Basicblocks wieder erreicht, kann der bereits vorhandene Basicblock wiederverwendet werden. In diesem Fall entstehen nur die Kosten des Hashtabellenzugriffs. Im Gegensatz zum EBI entfällt der gesamte Dekodierungslauf.

5.2.4 Bewertung

Vom EBBB darf eine wesentlich höhere Leistung als vom EBI erwartet werden. Die Erkennung der Basicblöcke ist sehr einfach und auch die Erstellung des Codeblocks nur unwesentlich teurer als die Erstellung eines Instruction Objekts.

Die Basicblöcke enthalten mehr Nutzcode, wodurch die Hauptschleife des Emulators viel seltener Durchlaufen wird als beim EBI und zahlreiche Zugriffe auf die Hashtabelle eingespart werden. Innerhalb des erzeugten Codes wird der geladene CPU Status für mehrere native Anweisungen verwendet, womit sich das Verhältnis von Overhead zu Nutzcode verbessert.

Der größte Leistungszuwachs dürfte jedoch vom Entfallen der Dekodierungsfunktion erwartet werden. Diese ist im Vergleich zum Bytecode der kleinen erzeugten Codeblöcke um ein

[2] Aufgrund der gelockerten Basicblock Definition kann es zu Code Überschneidungen in verschiedenen Blöcken kommen. Dies ist für JXEmu aber nicht weiter problematisch.

vielfaches komplexer. Der Geschwindigkeitsvorteil greift hier natürlich nur in Programmen, die bestimmte Codeteile mehrmals durchlaufen. Insgesamt darf von einer deutlichen Leistungssteigerung bei nur geringem Zusatzaufwand ausgegangen werden, der dem EBBB bei praktisch allen Anwendungen zu einem großen Vorteil gegenüber dem EBI verhelfen dürfte.

5.3 Pfadfinder: EmulatorByTrace

Der EBBB stellte bereits einen großen Fortschritt gegenüber dem EBI dar. Um die Leistung weiter zu steigern, kann versucht werden, den Overhead weiter zu mindern. Man muss versuchen, solange wie möglich kompilierten Code auszuführen. Dadurch wird der geladene CPU Status weiter effizient genutzt und die Emulatorschleife seltener durchlaufen.

Der `EmulatorByTrace` (EBT) versucht, heiße Pfade in der Ausführung des nativen Codes zu erkennen. Die Herangehensweise basiert auf der Aussage, dass die meisten Programme den Großteil ihrer Ausführungszeit mit dem Durchlaufen von Schleifen verbringen. Der EBT versucht, Schleifen zu erkennen und für diese Schleifen optimierten Code zu erzeugen.

Der EBT hat dabei zwei Teilaufgaben zu lösen. Zum einen muss die Schleife im Code erkannt werden. Anschließend muss der heiße Pfad durch die Schleife zu einem Codeblock kompiliert werden.

5.3.1 Erkennung von Schleifen

Der EBT unterstützt zwei Modi zur Erkennung von Schleifen, welche in diesem Abschnitt beschrieben werden sollen.

Aufzeichnung des Programmpfades

In diesem Modus wird der Programmpfad in einem Ringpuffer aufgezeichnet. Der Parameter `MAX_TRACE_LEN` legt die Größe des Ringpuffers fest und setzt damit effektiv die maximale Größe eines heißen Pfades. Nach jedem Sprung wird geprüft, ob die Zieladresse sich im Puffer befindet. Ist das der Fall, so wird davon ausgegangen, dass der Kopf einer Schleife entdeckt wurde. Der heiße Pfad ist die im Ringpuffer aufgezeichnete Folge von Anweisungen, angefangen beim gefundenen Schleifenkopf bis zu dem gerade durchgeführten Rücksprung. Abb. 5.5 zeigt

```
for (int i=0; i<1000; i++) {
  a();
  for (int j=0; j<1000; j++) b();
}
```

Abbildung 5.5: Codebeispiel: Verschachtelte Schleife

eine verschachtelte Schleife. Die Schleifenerkennung durch Pfadaufzeichnung wird bei solchen Schleifen stets nur die innerste Schleife erkennen, da deren Rücksprung als erstes durchgeführt

wird. Das ist jedoch nicht weiter schlimm, da die innere Schleife mit einer wesentlich höheren Frequenz als die äußere Schleife durchlaufen wird.

Zählen der Ausführhäufigkeit

Das zweite Verfahren zu Erkennung von Schleifenköpfen zeichnet Sprungziele auf und zählt für jedes Sprungziel die Häufigkeit, mit der dieses angesprungen wurde. Die beiden Parameter `COMPILE_FREQ` und `MAX_TRACE_LEN` legen fest, an welcher Schwelle versucht wird, den Codeblock zu erzeugen und wie viele native Instruktionen der Pfad maximal beinhalten darf.

Im Gegensatz zum ersten Verfahren ist hier der heiße Pfad noch nicht bekannt, wenn entschieden wird, den Pfad zu kompilieren. Es wird in einer Art Blindlauf das Programm interpretiert und alle angetroffenen Anweisungen während des Interpretierens in den Codeblock kompiliert — in der Hoffnung vor dem Überschreiten der maximalen Pfadlänge den Eintrittspunkt wieder zu erreichen.

Wie schon Verfahren 1 wird auch hier im Codebeispiel in Abb. 5.5 zunächst die innere Schleife erkannt, da deren Eintrittspunkt mit einer höheren Frequenz angesprungen wird. Im weiteren Verlauf ist es jedoch möglich, dass auch der Eintrittspunkt der äußeren Schleife den Schwellwert überschreitet. Der Vorteil dieses Verfahrens ist, dass erst bei einer bestimmten Häufigkeit versucht wird, einen Codeblock zu kompilieren. Damit kann man das Risiko senken, nicht lohnenswerte Pfade zu kompilieren. Das erste Verfahren kompiliert bereits bei der Erkennung des ersten Rücksprungs den heißen Pfad in einen Codeblock.

Problematisch ist hingegen die Erzeugung des Codeblocks im Blindlauf. Handelt es sich um eine Fehlerkennung oder ist der heiße Pfad länger als die maximale Pfadlänge, so wird der Übersetzungsvorgang erst beim Überschreiten dieser Grenze abgebrochen. In diesem Falle wurde dann viel Zeit vergeudet ohne jeglichen Fortschritt bei der Emulierung der Anwendung zu erzielen. Dieses Problem wird verschärft durch die Erkennung von Köpfen äußerer Schleifen, da dann die innere Schleife ausgerollt wird und damit die maximale Pfadlänge in der Regel überschritten wird. Der limitierende Faktor für die maximale Pfadlänge ist die 64 kB Grenze für Java Methoden.

5.3.2 Erzeugung des Codeblocks

Der Codeblock des EBBB kann für den EBT weiterverwendet werden, da die `execute()` Methode auch hier ohne Parameter auskommt. Auch der Übersetzungsmechanismus selbst kann in vielen Teilen den des EBBB nutzen.

Aufbau des Codeblocks

Abb. 5.6 zeigt den Aufbau des erzeugten Codeblocks. Im Gegensatz zu den Codeblöcken des Basicblock Modus beinhaltet der Code nun mehrere Basicblöcke. Zum Erzeugen von Prolog/Epilog und dem Code für die Basicblöcke selbst wird der EBBB verwendet. Um den korrekten Programmablauf zu gewährleisten, müssen jedoch einige Prüfungen eingesetzt werden.

Nach der Ausführung von jedem Basicblock muss sichergestellt werden, dass der heiße Pfad nicht verlassen wurde. Hierfür wird nach jedem Basicblock ein sog. *guard* eingefügt, der

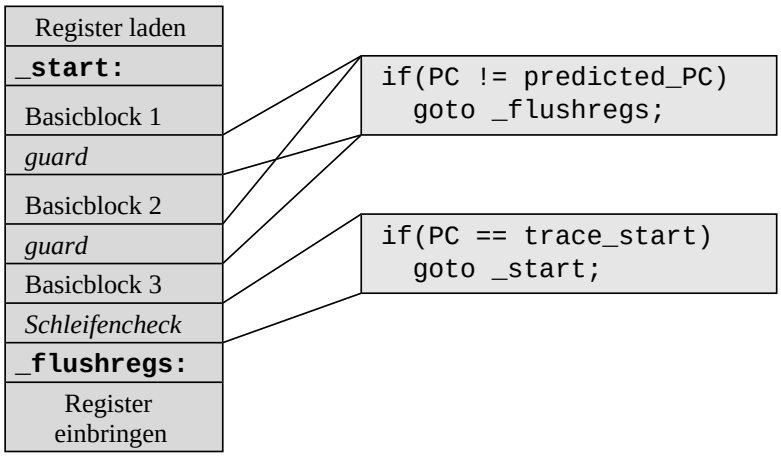

Abbildung 5.6: Aufbau der Codeblöcke beim EBT

prüft, ob der Befehlszähler mit der Startadresse des nächsten auf dem heißen Pfad liegenden Basicblocks übereinstimmt. Ist dies nicht der Fall werden die Register eingebracht und der Codeblock verlassen. Der tatsächliche für den *guard* erzeugte Code ist abhängig vom Typ des Sprungs am Ende des Basicblocks. Für einen

- indirekten unbedingten Sprung wird der Wert in `NPC` mit der vorhergesagten Adresse verglichen.
- indirekten bedingten Sprung wird das Z-Flag geprüft. Ist das Z-Flag nicht gesetzt, wird die Adresse der nächsten Anweisung nach `NPC` geschrieben. Anschließend wird eine Prüfung wie beim unbedingten indirekten Sprung generiert.
- direkten unbedingten Sprung muss kein *guard* erzeugt werden.
- direkten bedingten Sprung wird das Z-Flag geprüft und der Block verlassen, falls das Z-Flag bei der Aufzeichnung des Pfades an dieser Stelle einen anderen Wert hatte.

Nach dem letzten Basicblock wird geprüft, ob — wie erwartet — die Startadresse des ersten Basicblocks des Pfades wieder angesprungen wurde. In diesem Fall wird, ohne den Block zu verlassen, direkt zurück zum Anfang des Pfades gesprungen. Die Prüfung der Adresse erfolgt dabei wie bei den *guards*.

5.3.3 Bewertung und Probleme

Wie die Messungen in Kapitel 6 zeigen werden, erzielt der EBT nicht den gewünschten Leistungsschub. Das vorgestellte Verfahren ist mit mehreren Problemen verbunden. Zum einen ist die Erkennung der heißen Pfade mit vergleichsweise viel Aufwand verbunden. Hinzu kommen die Probleme, die bereits in Abs. 5.3.1 aufgezeigt wurden.

```
for(int i=0; i<1000; i++) {
  if( (i%2) == 0 ) a();
  else b();
}
```

Abbildung 5.7: Codebeispiel: Verschachtelte Schleife mit zwei heißen Pfaden

Zum anderen geht das Verfahren davon aus, dass in einer Schleife nur ein heißer Pfad durchlaufen wird. Dies soll das folgende Codebeispiel illustrieren.

Tritt eine solche Situation im Code auf, wird nur für einen Pfad optimiert. Der andere Pfad wird im sehr langsamen Interpreter Modus durchlaufen. Aufgrund des Zusatzaufwandes für die Schleifenerkennung ist die Ausführung im Interpretermodus noch langsamer als beim EBI.

5.4 EmulatorByUltrablock

Der `EmulatorByUltrablock` (EBUB) versucht, die Defizite des EBT zu beheben. Die Aussage, dass Programme die meiste Zeit ihrer Ausführung in Schleifen verbringen, soll allgemeiner formuliert werden: *Programme bringen die meiste Zeit ihrer Ausführung in besonders populären Basicblocks zu*. Der EBUB versucht also, diese populären Basicblöcke der heißen Bereiche des Programms zu identifizieren und sie zu einem sog. *Ultrablock* zu verschmelzen. Dieser soll dann möglichst lange ausgeführt werden.

5.4.1 Vorgehensweise

Der EBUB startet die Emulation der Altanwendung im Basicblock Modus. Dabei wird die Häufigkeit der Ausführungen eines jeden Basicblocks sowie die Gesamtzahl der Ausführungen aller Basicblöcke mitgezählt. Beim Überschreiten der Grenze `COMPILE_UBLOCK_FREQ` werden die populärsten Basicblöcke zu einem Ultrablock verschmolzen. Welche Basicblöcke in den Ultrablock kompiliert werden, kann mit dem Parameter `ELITE_PERCENTAGE` konfiguriert werden. Dieser legt den Prozentsatz der häufigsten Basicblöcke fest. Ein zweiter Parameter `MAX_UBLOCK_CODESIZE` dient als limitierender Faktor für die Codegröße eines Ultrablocks. Diese darf aufgrund der maximalen Java Methodengröße niemals größer als 64 kB sein.

Zu jedem zwischengespeicherten Basicblock wird vermerkt, ob dieser auch Teil eines Ultrablocks ist. Wann immer die Ausführung auf einen solchen Basicblock trifft wird der Ultrablock anstelle des Basicblocks ausgeführt. Um zu erkennen, wann das heiße Gebiet verlassen wurde, wird mitgezählt, wie oft der Ultrablock verlassen wurde. Der Parameter `BB_FALLBACK_FREQ` setzt schließlich fest, wann der Ultrablock verworfen und in den Basicblock Modus zurückgekehrt wird. Die Zähler werden dann zurückgesetzt und das Verfahren beginnt von neuem.

5.4.2 Aufbau des Ultrablocks

Abb. 5.8 zeigt den Aufbau eines Ultrablocks. Prolog und Epilog laden und speichern wie gehabt die benötigten Teile des CPU Status. Da der Ultrablock über jeden enthaltenen Basicblock

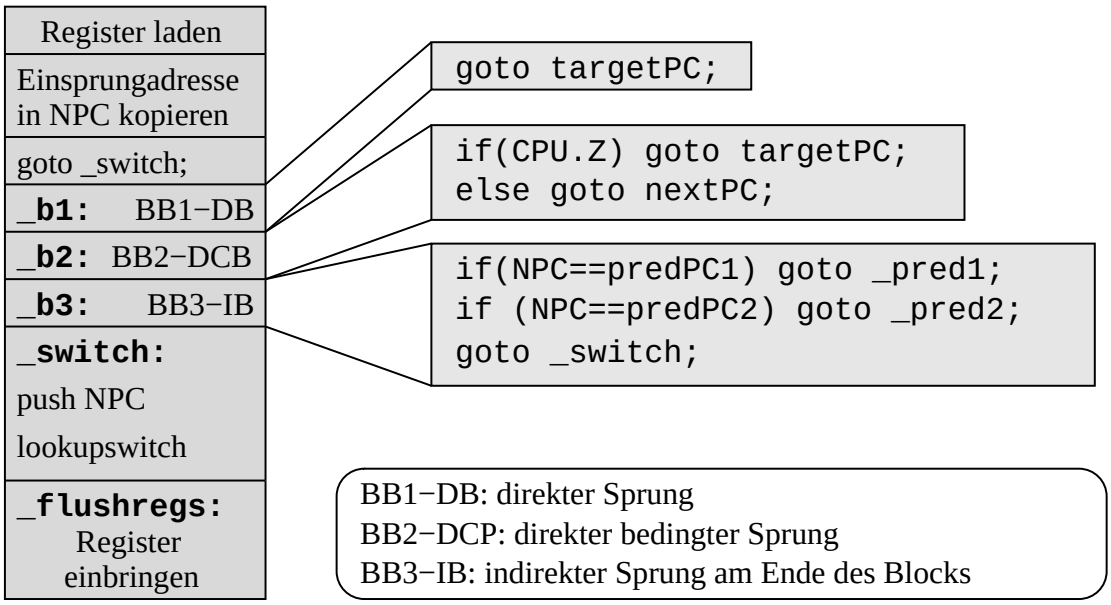

Abbildung 5.8: Aufbau eines Ultrablocks. Nach jedem Basicblock wird, abhängig vom Sprungtyp, nach Möglichkeit ein direkter Sprung zu einem anderen Basicblock im Ultrablock generiert. Bei indirekten Sprüngen wird auf vorhergesagte Ziele geprüft bevor der *lookupswitch* am Ende des Block durchlaufen wird. Dieser vergleicht den Wert in NPC mit den Startadressen aller im Ultrablock vorhandenen Basicblöcke. Ist das Ziel nicht enthalten, wird der Ultrablock verlassen.

betreten werden kann, benötigt die `execute(int entryPC)` Methode nun die Einsprungsadresse in den Ultrablock, also die Startadresse des ersten Basicblocks, als Parameter. Diese wird nach dem Laden der Register in die dem `NPC` Register zugeteilte lokale Variable kopiert. Dann wird an das Ende des Blocks gesprungen, wo sich ein *lookupswitch* befindet. Dieser vergleicht den Wert in `NPC` mit den Startadressen aller im Block vorhandenen Basicblocks. Ist der Basicblock im Ultrablock vorhanden, wird dieser direkt angesprungen. Ansonsten wird der Ultrablock verlassen.

Vor dem *lookupswitch* befinden sich die einzelnen Basicblöcke. Diese werden wie im EBBB eingefügt.

5.4.3 Sprünge innerhalb des Ultrablocks

Die Adressauswertung über den *lookupswitch* ist teuer und sollte daher nicht für die generelle Sprungbehandlung nach jedem Basicblock verwendet werden. Zwar erfordert der *lookupswitch*, dass die Vergleichswerte aufsteigend sortiert vorliegen. Damit kann eine JVM Implementierung günstigere Suchverfahren wie die binäre Suche anwenden. Dennoch ist ein direktes Springen zwischen den einzelnen Blöcken wesentlich günstiger. Nach jedem Basicblock wird daher, abhängig vom Sprungtyp, Code für direkte Sprünge generiert.

Im Fall eines unbedingten direkten Sprungs ist die Zieladresse bereits bekannt. Es kann ohne

weitere Prüfungen direkt zum entsprechenden Zielblock gesprungen werden. Da der Zielblock jeder Ausführung des Basicblocks folgt, besitzt dieser die gleiche Häufigkeit und befindet sich daher auch im Ultrablock.

Für bedingte direkte Sprünge sind die beiden möglichen Sprungziele ebenfalls bekannt. Nach Prüfung des Z-Flags kann ein direkter Sprung zum Zielblock erzeugt werden. Es ist allerdings gut möglich, dass sich mindestens eines der Ziele nicht im Ultrablock befindet. In diesem Fall wird direkt zum Epilog des Blocks gesprungen, wo die Register eingebracht werden und der Ultrablock verlassen wird.

Für indirekte Sprünge ist keine sichere Vorhersage des Zielblocks möglich. Es konnte jedoch beobachtet werden, dass zu bestimmten Phasen der Ausführung mit hoher Wahrscheinlichkeit auch bei indirekten Sprüngen immer die gleichen Ziele angesprungen werden. Während der Ausführung im Basicblock Modus wurden daher einige Folgeadressen nach der Ausführung solcher Blöcke aufgezeichnet. Die Aufzeichnung von zwei Adressen pro Block ist dabei i.d.R. ausreichend. Nach der Ausführung eines Basicblocks, der durch einen indirekten Sprung beendet wird, wird daher im Ultrablock zunächst Code erzeugt, der den `NPC` mit dem vorliegenden Sprungziel vergleicht. Bei Übereinstimmung wird ein direkter Sprung innerhalb des Ultrablock generiert, soweit der Block vorhanden ist. Ist der Block nicht vorhanden wird der Ultrablock direkt ohne Umweg über den *lookupswitch* verlassen. Stimmt das Sprungziel mit keiner der vorhergesagten Adressen überein, wird zum *lookupswitch* gesprungen. Diese Aktion basiert auf der Hoffnung, dass es sich bei dem Folgeblock ebenfalls um einen heißen Block handelt, der im Ultrablock vorhanden ist.

5.4.4 Bewertung

Der EBUB lässt auf eine Lösung der Probleme des EBT hoffen. Durch die Ausführung im Basicblock Modus treten die Nachteile des langsamen Interpreter Modus während der Bewertungsphase für die Basicblöcke nicht mehr auf. Das Verhalten des EBUB wird stark von den gewählten Werten für die Parameter abhängig sein. Je mehr Basicblöcke in den Ultrablock aufgenommen werden, desto geringer ist die Wahrscheinlichkeit, dass der Block verlassen werden muss. Allerdings steigt mit der Zahl der enthaltenen Basicblöcke auch die Größe und damit die Durchlaufdauer des *lookupswitch* und die Zeit, die zur Erstellung des Ultrablocks benötigt wird. Die Erstellung eines Ultrablocks ist ohnehin, verglichen mit dem Aufwand bei der Erstellung von Basicblöcken, nicht unerheblich. Das Erzeugen eines Ultrablocks an einer nicht lohnenden Programmstelle führt daher zu einer Verlangsamung der Emulation. Auch ist die Ausführung eines Ultrablocks teurer als die eines Basicblocks, da ein größerer Kontext geladen werden und der *lookupswitch* durchlaufen werden muss. Die Ausführung eines Ultrablocks lohnt daher nur, wenn eine ausreichend große Zahl von Basicblöcken innerhalb des Blocks durchlaufen wird.

Kapitel 6

Benchmarks und Ausblick

In diesem Kapitel wird die Leistung der einzelnen Codegeneratoren miteinander verglichen. Außerdem wird JXEmu zwei anderen Simulatoren und einem Intel StrongARM als Referenz gegenübergestellt. Abschließend wird ein Ausblick auf die kommenden Arbeiten an JXEmu gegeben.

6.1 Die Testprogramme

Im Moment ist es, hauptsächlich durch die rudimentäre Kernelemulation bedingt, nicht möglich, für reale Anwendungen repräsentative Benchmarks wie die SPEC CINT Benchmarks in JXEmu auszuführen. Deshalb werden an dieser Stelle nur einige kleine Programme verwendet, die i.A. nicht für Leistungsmessungen tauglich sind. Hierbei handelt es sich zum einen um ein Programm, welches eine verschachtelte Schleife beinhaltet und in dieser insgesamt 10^9 Iterationen durchläuft. Das zweite Programm enthält eine verschachtelte Schleife, die, wie in Abb. 5.7 auf S. 47 dargestellt, zwei heiße Ausführungspfade beinhaltet. Das dritte Programm berechnet rekursiv die 35. Fibonacci Zahl. Beim vierten Programm handelt es sich um das Programm *wsort*, einem Programm zum alphabetischen Sortieren von Zeilen, ähnlich wie das Programm *sort* von GNU. Dieses stellt hier den Test mit der größten Nähe zu realen Anwendungen dar.

6.2 Andere ARM Simulatoren

Zum Vergleich mit JXEmu wurden zwei andere ARM Simulatoren herangezogen. Besonders interessant für den Vergleich ist das Programm *QEmu* [Bel05], da dieses — wie auch JXEmu — dynamisch Code generiert. Allerdings wird anstelle von Java Bytecode direkt nativer Code für die Hostarchitektur erzeugt. Zum anderen wird der ARM Simulator *SimIt-ARM* [Qin05] herangezogen. Dieser simuliert den ARM Prozessor instruktionsweise und wurde bei der Entwicklung des ARM Moduls als Debugging Hilfe verwendet.

Kandidat	Geschachtelte Schleife, 10^9 Iterationen	Schleife mit zwei heißen Pfaden	Fibonacci(35) rek.	wsort
QEmu	24.203s	1m21.287s	20.103s	0.340s
SimIt	6m34.705s	15m49.464s	4m9.249s	—
JXEmu-EBI	—	—	48m+	7m19.027s
JXEmu-EBBB	4m6.001s	11m9.882s	2m38.796s	7.066s
JXEmu-EBT	1m22.172s	4m7.646s	1m17.815s	3m6.523s
JXEmu-EBUB	1m25.664s	4m2.551s	1m5.090s	30.410s
StrongARM	1m10.210s	3m10.910s	47.960s	0.770s

Tabelle 6.1: Meßergebnisse

6.3 Testumfeld

Die Messungen erfolgen auf einem AMD Athlon XP Hostsystem mit 2GHz. Als Referenz Prozessor kommt ein Intel StrongARM-110 zum Einsatz. Die gemessenen Zeiten stellen die gesamte Ausführungszeit des Testprogramms dar und beinhalten im Fall von JXEmu die Zeit zum Starten der Java VM sowie die vom JIT Übersetzer beanspruchte Zeit zur Übersetzung des Bytecodes sowohl von JXEmu selbst als auch der erzeugten Codestücke in Maschinencode der Hostarchitektur. Da die beiden Referenzsimulatoren in den traditionellen Programmiersprachen C bzw. C++ geschrieben sind und daher als native Anwendung vorliegen, entfällt bei ihnen dieser Zusatzaufwand.

6.4 Meßergebnisse

Die Ergebnisse der Testläufe sind in Tab. 6.1 dargestellt. Eine korrekte Ausführung von wsort in SimIt war nicht möglich. Ursache hierfür ist vermutlich ein Fehler in SimIt. Die Ausführung des Fibonacci Tests unter JXEmu im EBI Modus wurde nach 48 min. abgebrochen. Es ist bereits an dieser Zahl zu erkennen wie langsam der EBI arbeitet. Auf die Schleifentests wurde aus diesem Grund verzichtet. Diese sind noch zeitintensiver als der Fibonacci Test.

Vergleicht man die einzelnen JXEmu Modi miteinander, so sind einige unerwartete Ergebnisse erkennbar. Entgegen den Erwartungen bringt der EBUB lediglich einen geringen Vorteil gegenüber dem EBT bei der Ausführung des mehrpfadigen Schleifentests. In der geschachtelten Schleife ist der EBUB aufgrund des höheren Overheads durch Erstellung und Ausführung der Ultrablocks sogar ein wenig langsamer als der EBT, der diesen Schleifenfall optimal abdeckt. Beim Test wsort, welcher einer realen Anwendung am nähsten kommt, kann der EBUB gegenüber dem EBT seine Überlegenheit deutlich zeigen. In diesem ist jedoch der EBBB um ein vielfaches schneller. Das Verfahren zur Erkennung der heißen Gebiete beim EBUB und zur Selektion der aufzunehmenden Basicblöcke ist noch nicht ausgereift. Durch die Einbeziehung von Basicblöcken aus anderen heißen Gebieten steigt der Overhead in den Ultrablocks und führt außerdem dazu, dass diese häufig unnütz aufgerufen und sofort wieder verlassen werden. Die Erstellung eines Ultrablocks lohnt nur bei sehr häufigem Durchlaufen des heißen Gebiets, wie

die anderen Test zeigen. Der EBBB arbeitet dagegen durch sein simples Verfahren zur Erkennung der Basicblöcke und deren Übersetzung gerade bei Anwendungen mit weniger Schleifendurchläufen effizienter als die anderen Verfahren.

Gegenüber den anderen Simulatoren ist JXEmu erwartungsgemäß in allen Tests schneller als der im Interpreterverfahren arbeitende SimIt. QEmu ist jedoch in allen Bereichen stark überlegen. Hier kommt — neben den Schwächen der Codeerzeugung in JXEmu — v.a. der Vorteil von QEmu durch das Entfallen des Overheads zum Starten der JVM sowie des zusätzlichen Übersetzungsschrittes durch den Java JIT Compiler zum Tragen.

6.5 Zusammenfassung und Ausblick

Wie die Meßergebnisse zeigen, ist die Leistung von JXEmu für die Verwendung mit realistischen Anwendungen im Moment nicht akzeptabel. Es ist jedoch deutlich erkennbar, dass die Leistung in starkem Maße von der gewählten Strategie zur Codeerzeugung abhängig ist. Die Verbesserung der Codeerzeugung ist daher ein wesentlicher Aspekt bei der weiteren Arbeit an JXEmu. Die Entwicklung von Strategien ist jedoch mit viel Arbeit bei der Analyse der Strukturen der Altanwendungen und der Auswertung des Verhaltens der entwickelten Codegeneratoren verbunden. Aufgrund der Vielfältigkeit der Altanwendungen in ihrer Struktur kann es keine für alle Anwendungen perfekte Strategie geben. Es muss immer ein Kompromiss zwischen Aufwand zur Codeanalyse und Codegenerierung und der Geschwindigkeit bei der Ausführung des erzeugten Codes gemacht werden. Die Erzeugung komplexer Codeblöcke lohnt nur dann, wenn diese auch ausreichend oft verwendet werden.

Ein weiterer begrenzender Faktor ist die Emulation des untypisierten Speichers. Auch die DirectMemory Implementierung bringt keinen signifikanten Leistungsvorteil gegenüber dem SafeMemory, da der Speicherzugriff in der Unsafe Klasse über *Java Native Interface* (JNI) Methoden realisiert ist. Der Aufruf solcher Methoden ist wesentlich teurer als der Aufruf von Java Methoden, da die Parameter und der Rückgabewert entsprechend dem JNI Standard gekapselt werden müssen. Eine effiziente Speicheremulation kann daher nur mit Unterstützung durch die JVM realisiert werden. Die JVM muss hierfür Mechanismen zum Zugriff auf untypisierten Speicher bereitstellen. Die Erweiterung einer bestehenden JVM um solche Mechanismen ist bereits geplant.

Zur Ausführung realistischer Anwendungen in JXEmu ist außerdem eine vollständigere Implementierung der Kernel Emulation notwendig. Alternativ könnte hier auch die Emulation eines vollständigen Systems gewählt werden, welche das Ausführen eines Gastbetriebssystems erlauben würde. Im Moment können nur Anwendungen für den Benutzermodus ausgeführt werden.

Literaturverzeichnis

[Adv96] Advanced RISC Machines Ltd (ARM). *ARM7500FE Datasheet*, b-01 edition, September 1996.

[ARM01] ARM Limited. *ARM ELF*, b-02 edition, June 2001.

[Atm99] Atmel Corporation. *ARM7TDMI^{TM} (Thumb ®) Datasheet*, rev b edition, January 1999.

[BC00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, first edition, October 2000.

[BEJ^{+}] Mark Brown, Paul Eggert, Andreas Jaeger, Jakub Jelinek, Roland McGrath, and Andreas Schwab. GNU C library. Web: `http://www.gnu.org/software/libc/libc.html`.

[Bel05] Fabrice Bellard. Qemu—Generic and Open Source Processor Emulator. `http://fabrice.bellard.free.fr/qemu`, 2005.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley Longman, Inc., second edition, 1999.

[Qin05] Wei Qin. SimIt-ARM. `http://simit-arm.sourceforge.net`, 2005.

[TIS95] TIS Committee. *Tool Interface Standard (TIS): Executable and Linking Format (ELF) Specification*, 1.2 edition, May 1995.

[Tor] Linus Torvalds. Linux kernel. Web: `http://www.kernel.org`.

Abbildungsverzeichnis