Zwischenspeicherung von Maschinencode in CacaoVM

Robert Schuster

Institut für Informatik

FU Berlin

23.04.09

Ziel des Vortrages

- Primär
  - Bearbeitetes Problem
  - Vorstellung des technischen Ansatzes
  - Resultate
- Sekundär
  - Begeisterung für eingebettete Systeme
  - JVM-Entwicklung ist kein Hexenwerk

Hintergrund

Eingebettetes System?

Freie Universität Berlin

- viele Systemfunktionen auf einem Chip
  - zB. CPU, GPU, DSP, USB-, Ethernet- und LCD-Controller
- unteres Ende: Recheneinheit ohne Speicherverwaltung (Mikrocontroller)
- oberes Ende: normale Prozessoren mit der Leistung von Desktop-CPUs von vor ~5 Jahren

Hintergrund

Freie Universität Berlin

- seit 2006 Interesse an Linux-basierten eingebetteten Systemen
- Entwickler im Projekt GNU Classpath
  - Möglichkeit Java auf Nicht-PCs auszuführen

GNU Classpath

GCJ

OpenEmbedded

Jalimo

CacaoVM

IcedTea/OpenJDK

Java **SE** auf eingebetteten Geräten?

Java SE auf eingebetteten Geräten

Freie Universität Berlin

- Java ME – CDC
  - ~ Java SE 1.3
  - kein „richtiges“ Desktop-Java
  - kein Swing, nur minimales AWT
  - mangelnde Kompatibilität mit gängigen Bibliotheken
  - keine Annotationen, Generics usw.

aktuelle eingebettete Systeme

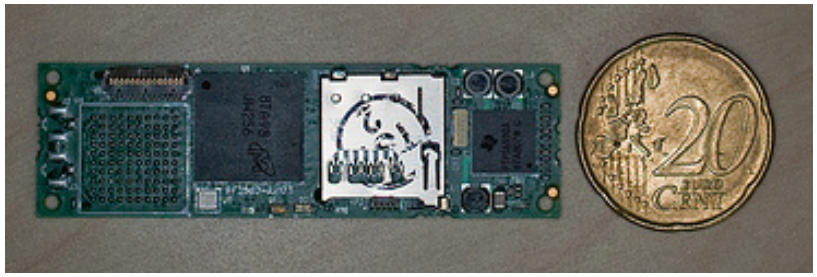
Freie Universität Berlin

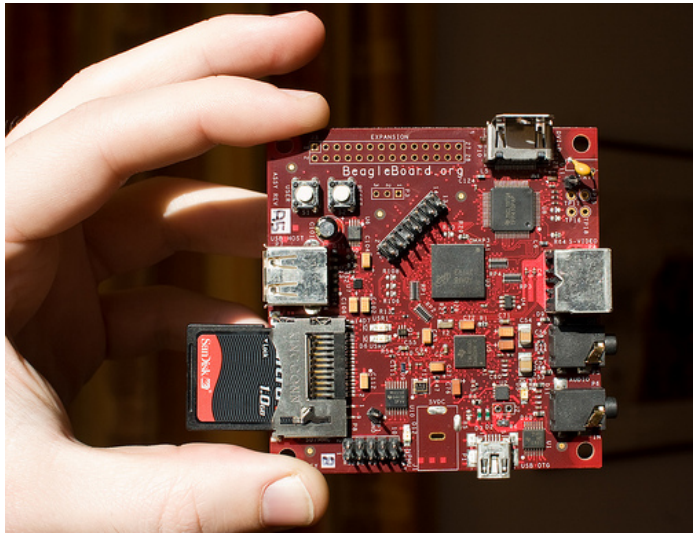
- unteres Ende: OpenMoko Freerunner
  - 400 MHz
  - 128 MiB RAM
  - 128 MiB Flash + SDHC

aktuelle eingebettete Systeme

Freie Universität Berlin

- oberes Ende: Gumstix Overo/Beagleboard
  - 500 MHz
  - 128/256 MiB RAM
  - 128 MiB Flash + SDHC
  - CPU ist drei Generationen weiter (ggü. Freerunner)
  - Basis für Pandora-Spielkonsole, „Touch Book“ und kommendes Nokia Internet Tablet





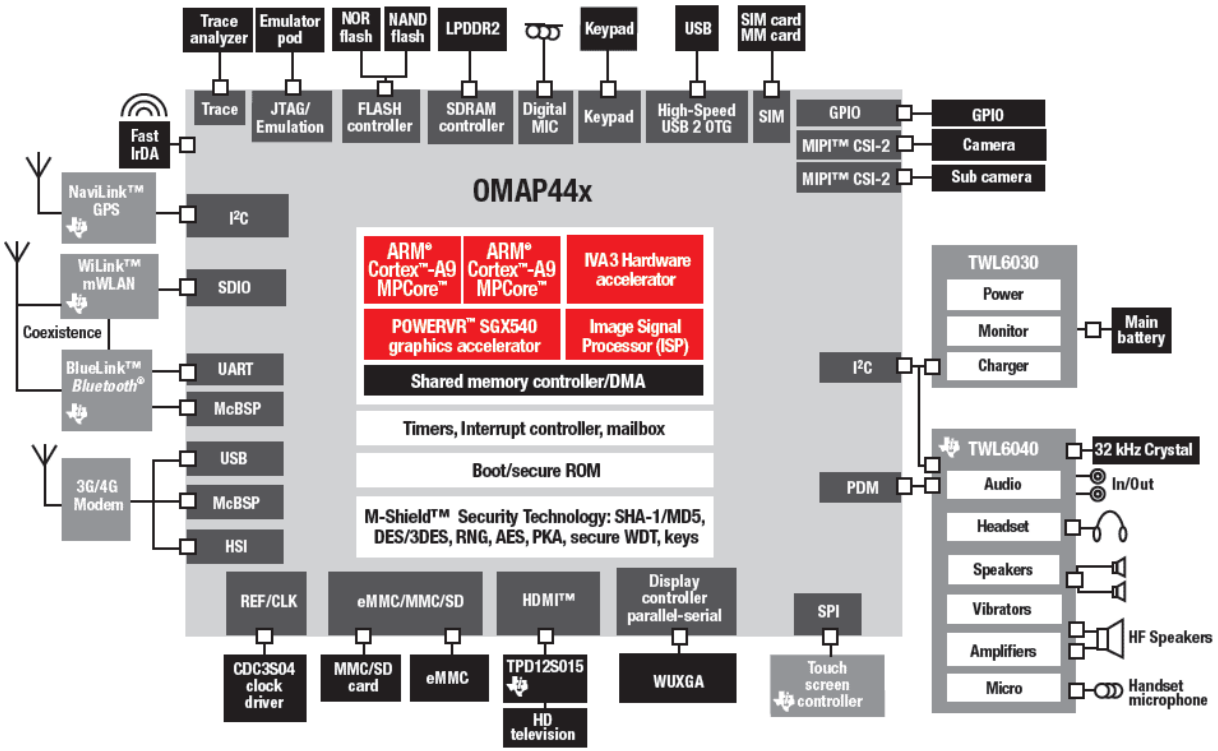
aktuelle eingebettete Systeme

Freie Universität Berlin

- oberes Ende: SheevaPlug
  - 1,2 GHz
  - 512 MiB RAM
  - SDHC
  - Gigabit-Ethernet

nächste Generation

- OMAP4:
  - >1 Ghz
  - zwei Kerne



"We're trying to converge everything to the Java SE specification. Cell phones and TV set-top boxes are growing up, [...]"

(James Gosling, Oktober 2007)

http://news.cnet.com/8301-13580_3-9800679-39.html

Das Problem

Problem

Freie Universität Berlin

CacaoVM + OpenJDK:

```
time java HelloWorld
real      0m 1.78s
user      0m 1.45s
sys       0m 0.25s
```

- 286 Klassen geladen
- 1021 Methoden kompiliert

Startzeitverkürzung wird angestrebt.

Beobachtung

- JIT-Kompilierung bei jedem Lauf
- viele Klassen ändern sich nicht („bootclasspath“)
- somit auch die Ergebnisse des JIT-Compilers

Lösungsvorschlag

Freie Universität Berlin

- erzeugten Maschinencode permanent speichern
- in Folgeläufen der VM gesicherten Maschinencode verwenden (anstelle der JIT-Kompilierung)

Einsatzzweck

eingebettetes System

↓

spezielle Aufgabe

↓

Einzelanwendung

↓

Startoptimierung

CacaoVM – lohnenswertes Ziel

Freie Universität Berlin

- JIT-Compiler für viele Architekturen
- unterstützt Java SE
  - GNU Classpath und OpenJDK
- jede Java-Methode wird vor erster Ausführung JIT-kompiliert
- spezieller Ansatz der Codegeneratoren erleichtert Implementierung der Startoptimierung

grobe Arbeitsweise

- Einhängen in den JIT-Kompilierungsablauf
- Bsp: java.io.PrintWriter.println() soll aufgerufen werden

```
...
call „PrintWriter.println“
...
```

- entweder: Maschinencode der Java-Methode wird ausgeführt
- oder: Maschinencode wird erst erzeugt (und danach ausgeführt)

grobe Arbeitsweise

Freie Universität Berlin

```
jit_compile(methodinfo *m) {
  [...]

  if (jitcache_load(m))
    return;

  /* Maschinencodeerzeugung */
  [...]

  jitcache_store(m);

  [...]
}
```

Hauptschwierigkeit

- Maschinencode enthält Verweise auf Speicherobjekte
- Adressen sind zwischen zwei Programmläufen verschieden
- oder Speicherobjekt existiert noch gar nicht
  - normalerweise während JIT-Kompilierung angelegt

Verweise im Maschinencode

Freie Universität Berlin

- Bsp: java.io.PrintWriter.println()
- Code an Adresse `$0x1234`
- alle Methoden die PrintWriter.println() aufrufen verwenden diesen Wert

Verweise im Maschinencode

- nächster Lauf der VM
- Code an Adresse `$0x8000`
- vom Festspeicher geladener Maschinencode verwendet noch die alte Adresse

- Lösung
  - Korrektur der Verweise nach dem Laden

Grundlage für JIT-Cache

Freie Universität Berlin

- „lazy linking“ - Beispiel

```
if (someCondition)
  new SomeClass();
```

- Verhalten bei Ausführung
- Klassendatei für `SomeClass` nicht vorhanden
- Bedingung falsch -> keine Probleme
- Bedingung wahr -> `NoClassDefFoundError`

Grundlage für JIT-Cache

Freie Universität Berlin

- Maschinencode muss für alle Varianten funktionieren
- soll keine explizite Abfrage nach Existenz enthalten

Lösung in Cacao: Codepatching

Codepatching I

Freie Universität Berlin

- Erzeugung von Blanko-Maschinencode:

PC	Instruktion
`$F00F`	`CALL $0x0`

- Anlegen von Zusatzinformation:
  - Patcher für Codeposition $F00F
  - Methodenaufruf: „new SomeClass“

Codepatching II

Freie Universität Berlin

- Ersatz mit Fallenbefehl:

PC	Instruktion
`$F00F`	`UD2*`

* undefined opcode

Codepatching III

- bei Ausführung
  - Signal „ungültige Instruktion“ (SIGILL)
- Laden der geforderten Information
- Ersatz der Instruktion:

PC	Instruktion
`$F00F`	`CALL $0x3210`

Erkenntnisse für JIT-Cache

Freie Universität Berlin

- Infrastruktur für verzögertes Laden vorhanden
- Ersatz von Maschinenbefehlen ist wesentliche Arbeitsweise von Cacao

JIT-Cacheimplementierung soll ähnliches Verfahren verwenden.

Aufbau JIT-Cache

- Teil 1: Speichern und Wiederherstellen von Maschinencode
- Teil 2: Anlegen von „cached references“
  - während Codeerzeugung
  - beinhaltet Typ, Adresse und Wert

Speicherung/Wiederherstellung

Freie Universität Berlin

- Speichervorgang
  - Maschinencode
  - Patcher- und Cachedref-Exemplare
    - dabei effiziente Wiederherstellung ermöglichen
- Wiederherstellungsvorgang
  - Maschinencode lesen
  - Cachedref-Exemplare lesen und sofort abarbeiten (Ersatz von Adressen)

Schwierigkeiten

Freie Universität Berlin

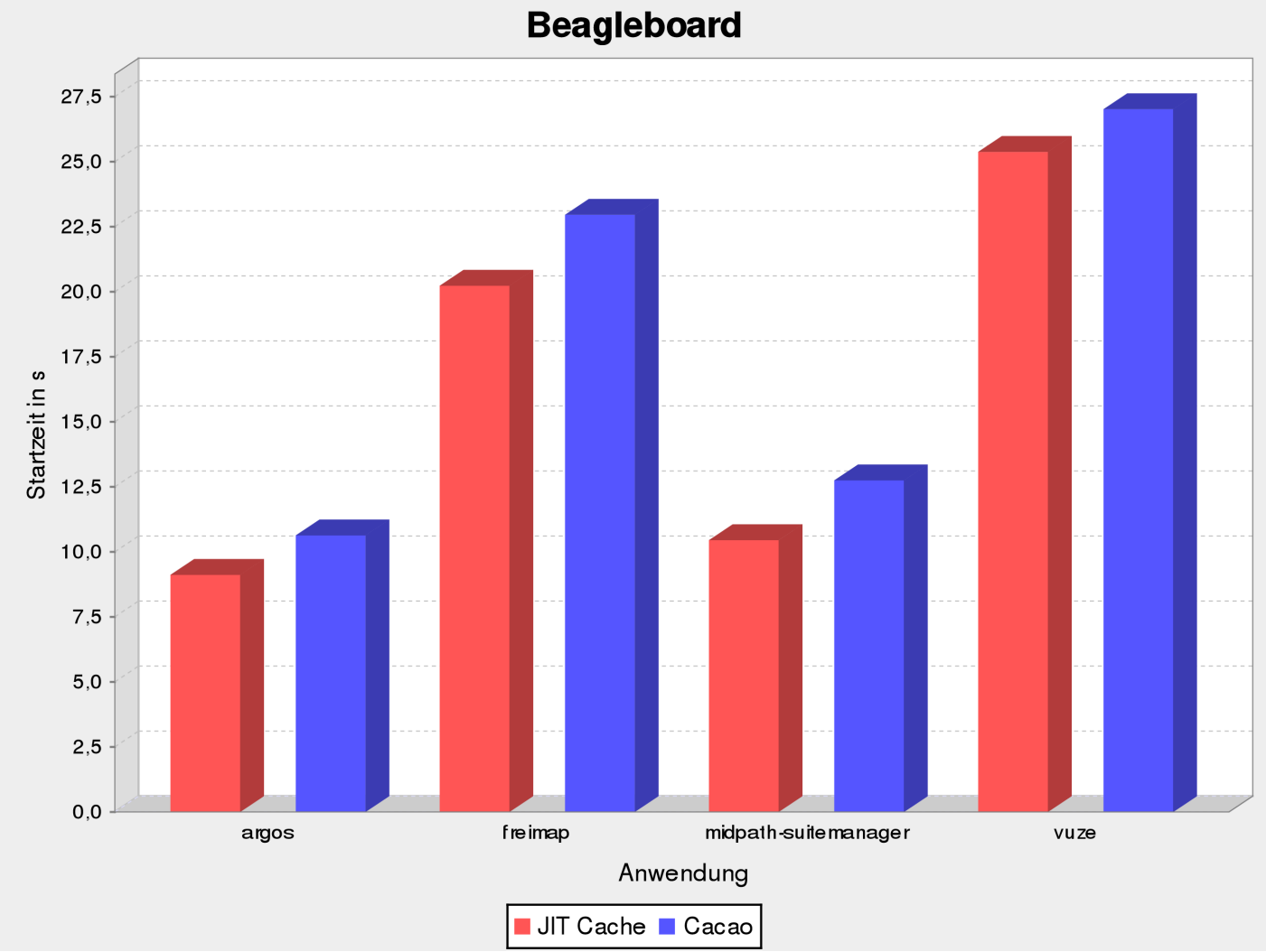
- Patcher- und Cachedref-Exemplare können denselben Arbeitsbereich haben
- feste (geringe) Größe von unmittelbaren Konstanten bei RISC-Architekturen
- Ladevorgang muss **wirklich** effizient durchgeführt werden
  - Speichereinblendung (mmap)
  - Vermeidung von Auflöseoperationen
  - Wiederverwendung von Dateizeigern
- Instruktionscache löschen

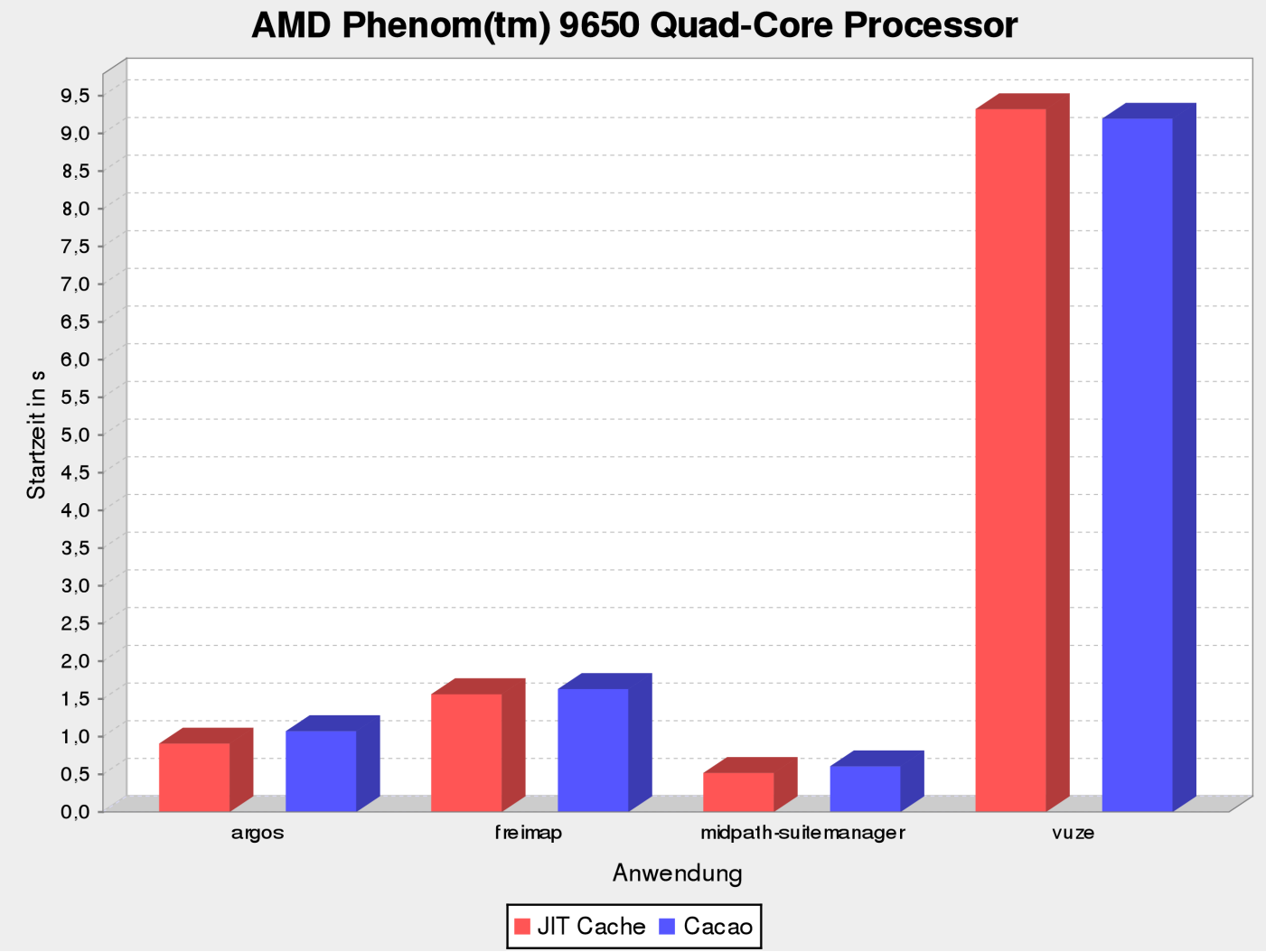
Erreichte Ziele in der Diplomarbeit

Freie Universität Berlin

- Implementierung des JIT-Caches für
  - ARM
  - AMD64
  - i386

Ergebnisse


Beagleboard
27,5
25,0
22,5
20,0
17,5
15,0
12,5
10,0
7,5
5,0
2,5
0,0
Startzeit in s
argos
freimap
midpath-suitemanager
vuze
Anwendung
JIT Cache
Cacao


AMD Phenom(tm) 9650 Quad-Core Processor
Startzeit in s
9,5
9,0
8,5
8,0
7,5
7,0
6,5
6,0
5,5
5,0
4,5
4,0
3,5
3,0
2,5
2,0
1,5
1,0
0,5
0,0
argos
freimap
midpath-suitemanager
vuze
Anwendung
JIT Cache
Cacao

Geschwindigkeitssteigerung

Freie Universität Berlin

- AMD64
  - schlechtester Wert: -1 %
  - bester Wert: 15 %
- ARM
  - schlechtester Wert: 6 %
  - bester Wert: 18 %

Demozeit!

Zusammenfassung

Freie Universität Berlin

- Java SE auf eingebetteten Geräten
  - viele aktuelle Geräte leistungsfähig genug
  - Problem: Startzeit
- JIT-Cache Ansatz
  - verhältnismäßig geringer Aufwand
  - gemessene Geschwindigkeitssteigerung: 6-18%
- verfügbar für
  - ARM
  - AMD64/i386
  - angefangen: PowerPC

Vielen Dank!

Diskussionseinstieg

Freie Universität Berlin

Ist Java SE auf eingebetteten Systemen notwendig?

Wie wurden die Startzeitmessungen bei den interaktiven Programmen durchgeführt?

Diskussionseinstieg

Könnte man das Problem der hohen Anlaufzeit nicht auch mit einem Interpreter/JIT-Compiler-Mischbetrieb lösen?

Diskussionseinstieg

Freie Universität Berlin

Welche anderen Ansätze gibt es die Startzeit zu verbessern?