Herzlich willkommen

zum Vortrag

SJC – ein schlanker modularer Java Compiler für Forschung und Lehre

am 5. Oktober 2010
an der Hochschule Kempten

Hochschule Kempten
University of Applied Sciences

Inhalt

1. Motivation

1.1 Umfeld

- Motivierendes übergeordnetes Vorhaben:
  - Komplett eigenes Betriebssystem
  - System kann verteilt werden
  - Verständliches Konzept
  - Verständlicher Code
  - Cluster-Dauer-Betrieb
  - Erweiterung zur Laufzeit möglich
  - Homogene und heterogene Hardware
  - Entwicklung: nicht nur unter MS-Windows

1.2 Wunschliste

- Initialer Wunschzettel Kategorie „must have“:
  - Typsicherheit und Objektorientierung
  - Systemprogrammierung möglich
  - Compiler übersetzt sich selbst
  - Memory-Layout anpassbar
- „Nice to have“ Wünsche:
  - Tool-Chain verständlich
  - Sprache weit verbreitet
  - Schlank und modular

1.3 Sprachauswahl

- Welche Sprache?
  - C, C++: nicht typsicher
  - Oberon: nicht verbreitet
  - (diverse andere: skipped)
  - Java: wäre schon toll, aber
    - Selbstübersetzungsfähigkeit des JDK?!?
    - Systemprogrammierung mit JDK?!?
    - Änderung zur Laufzeit mit JDK?!?
    - JDK schlank und modular?!?
- Eigener Java-Compiler!

2. Zielsetzung

1. Motivation
2. Zielsetzung
3. Implementierung
4. Beispielanwendungen
5. Zusammenfassung und Fazit

2.1 Ausgabe-Anforderungen

- Gesucht: Java-Compiler plus
  - Memory-Layout-Anpassbarkeit
  - Fähigkeit zur Selbstübersetzung
  - Erzeugung von nativem Maschinencode
    - von 8-Bit Mikrocontrollern bis 64 Bit Multicores
  - Einschränkbarer Zugriff auf Hardware in Java
  - Abstinenz von Bibliotheken und JNI-Funktionen
  - Grundlage für Implementierung eines Cluster-OS
- Alles bitte schlank, modular, vermittelbar, … ;-)

2.2 Compiler-Anforderungen

- Kompatibilität mit Java-Sprache
  - Compiler selbst vollständig in Java
    - Kompatibilität mit SunJava / Eclipse etc.
  - Einschränkungen bei „Syntactic Sugar“
    - kein Auto-(Un)Boxing, Generics, Enumerations
    - optional ohne implizite Basistyp-Konvertierungen
  - Compiler versteht eigenen Quellcode
- Selbstübersetzbarkeit in Mini-OS
  - Anforderungen an OS minimal
  - Keine externen Bibliotheken

2.3 System-Anforderungen

- Erweiterungen für die Systemprogrammierung
  - Zugänglich für Kernel- und Treiberprogrammierer
  - MAGIC: Hardwarezugriff und mehr „Magisches“
    - direkter Zugriff auf Hauptspeicher und I/O-Raum
    - Zugang zu RTE und Meta-Informationen
  - STRUCT: Speicherstruktur-Abbild
    - keine echten Objekte, nur Layout
  - @SJC: Compilerhinweise
    - z.B. für Profiler-Aktivierung, Interrupt-Handler
  - Direkt für HW-Emulation in Java verwendbar

2.4 Use-Case I

- Use-Case „Videospeicher“: Java-Seite
  - „Normales“ Java für „normale“ Funktionalität

```
public static void main() {
  print("Hallo liebe Kollegen");
}
public static void print(String str) {
  for (int i=0; i<str.length(); i++)
    print(str.charAt(i));
}
private static int vidPos;
public static void print(char c) {
  if (vidPos<0 || vidPos>=2000) vidPos=0;
  vidMem.digit[vidPos].ascii=(byte)c;
  vidMem.digit[vidPos++].color=0x07;
}
```

2.4 Use-Case II

- Use-Case „Videospeicher“: Magische Seite
  - „Magische“ Initialisierung für STRUCT

```
➔ private static VidMem vidMem
    =(VidMem)MAGIC.cast2Struct(0xB8000);
```

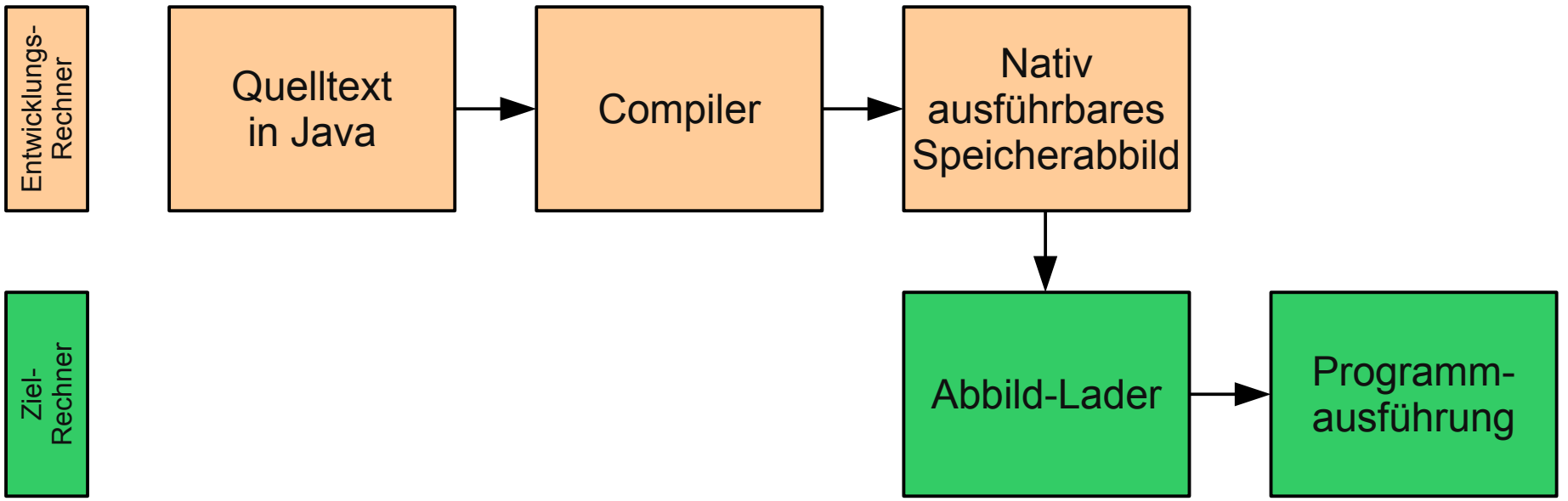
  - Java-konforme Deklaration des Speicherlayouts

```
➔ public class VidChar extends STRUCT {
    byte ascii, color;
  }
➔ public class VidMem extends STRUCT {
    @SJC(count=2000)
    VidChar[] digit;
  }
```

  - Somit typsicherer und Überlauf-sicherer HW-Zugriff

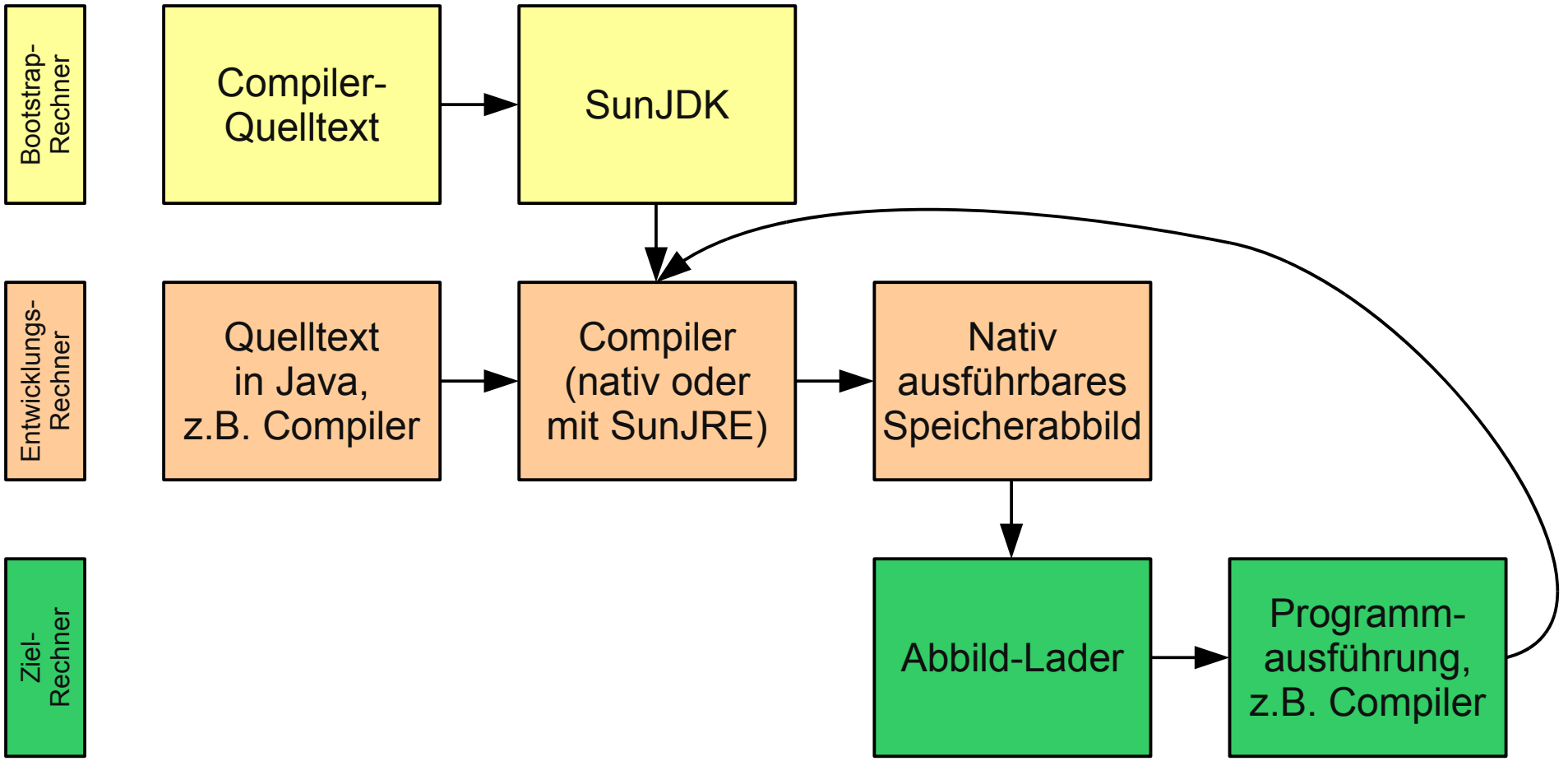
2.5 Cross-/Target-Übersetzung

- Abbild mit vollständigem Zielsystem
  - Speicherabbild vom Compiler erzeugt
  - Abbild-Lader in wenig Assembler
    - initialisiert CPU, Stack etc.



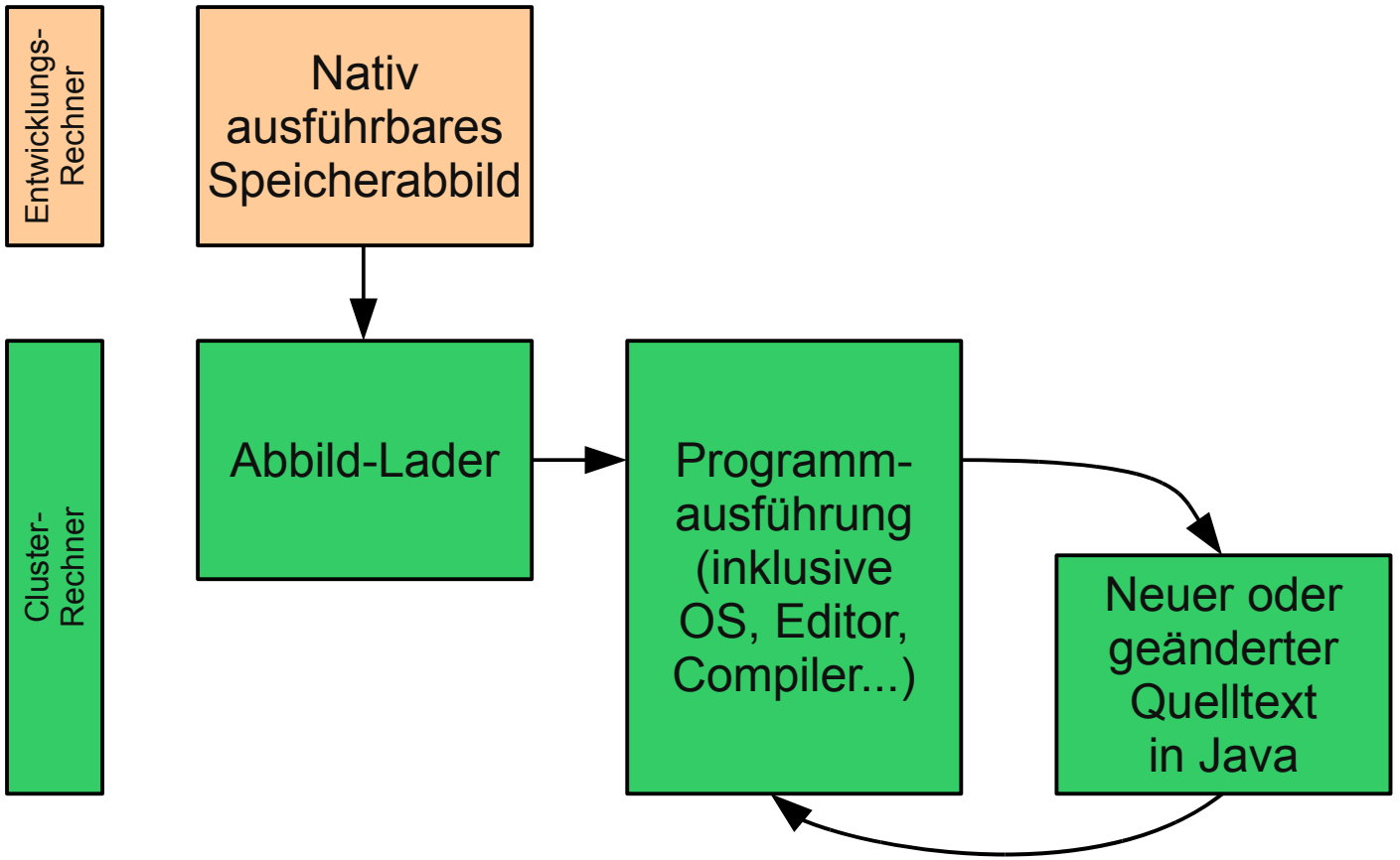
2.6 Bootstrap-Übersetzung

- Compiler-Erst-Übersetzung mit Sun-JDK



2.7 In-System-Übersetzung

- In-System-Übersetzung im Cluster



3. Implementierung

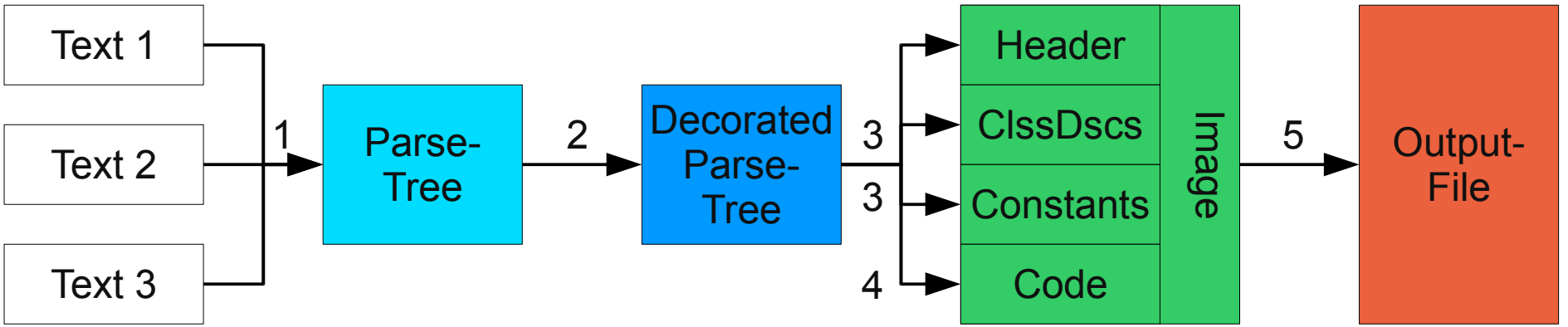
1. Motivation
2. Zielsetzung
3. Implementierung
4. Beispielanwendungen
5. Zusammenfassung und Fazit

3.1 Modularisierung

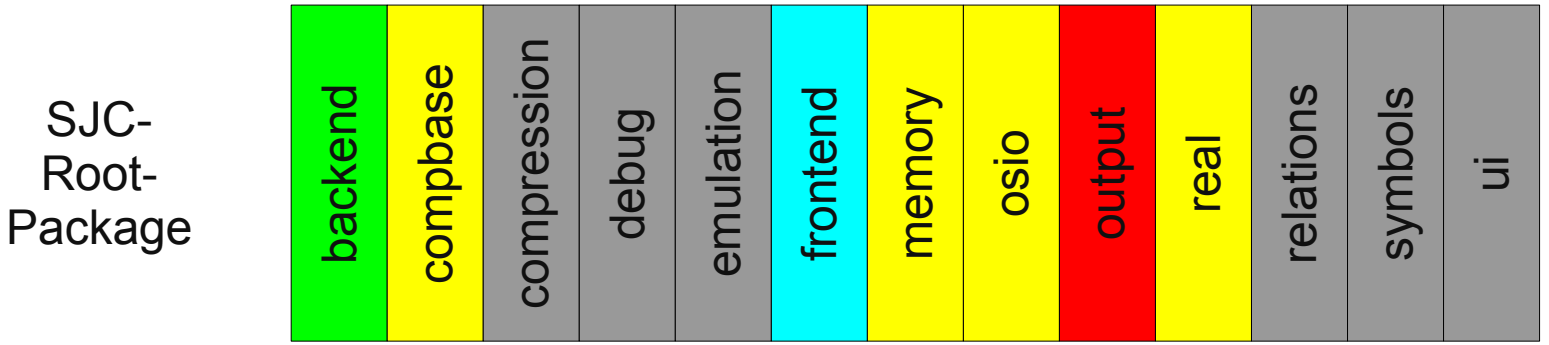
- Modularisierungsregeln für den Compiler
  - Sub-Packages implementieren je ***eine*** Schnittstelle
    - Schnittstelle ist im zugehörigen Root-Package definiert
  - Kein Zugriff auf Klassen fremder Sub-Packages
  - Minimierung von Klassen in Root-Packages
- Klassische Modulbenennung
  - Frontend: Quelltext → Parse-Tree
  - Backend: Parse-Tree → Maschinen-Code
  - Hilfsmodule: OS-Zugriff, Abbild-Erzeugung etc.

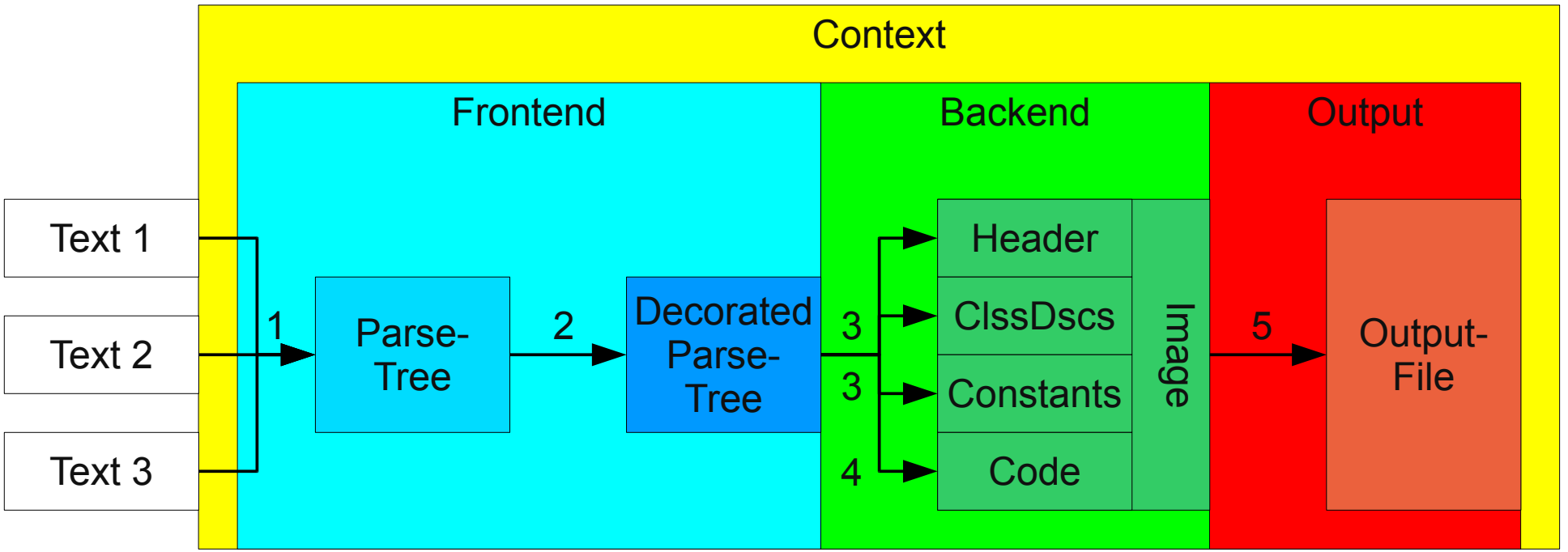
3.2 Arbeitsschritte

1. Scannen und Parsen
2. Namensauflösung
3. Deskriptorenerzeugung
4. Codierung und Optimierung
5. Ausgabe des Speicherabbilds

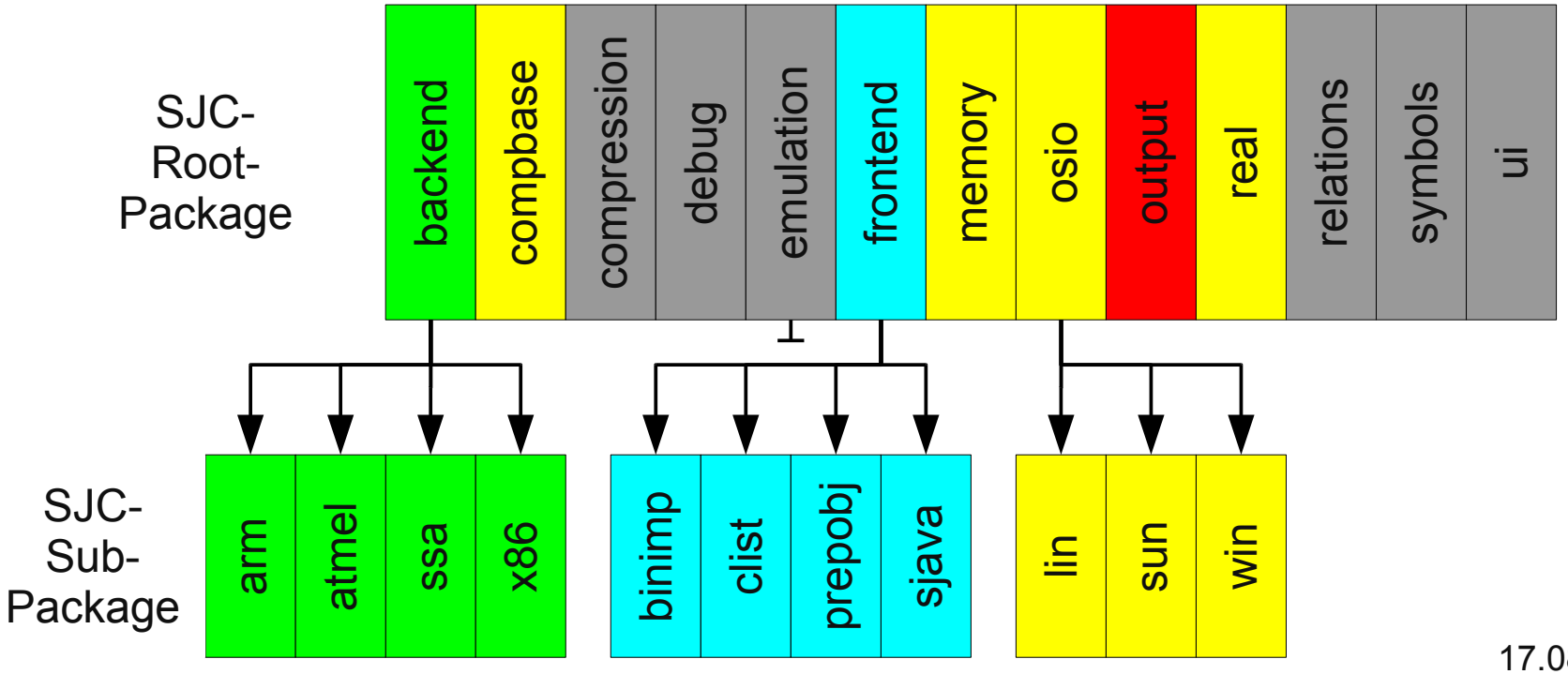


3.3 Paketisierung





3.4 Größenvergleich



Stand:
17.08.2010

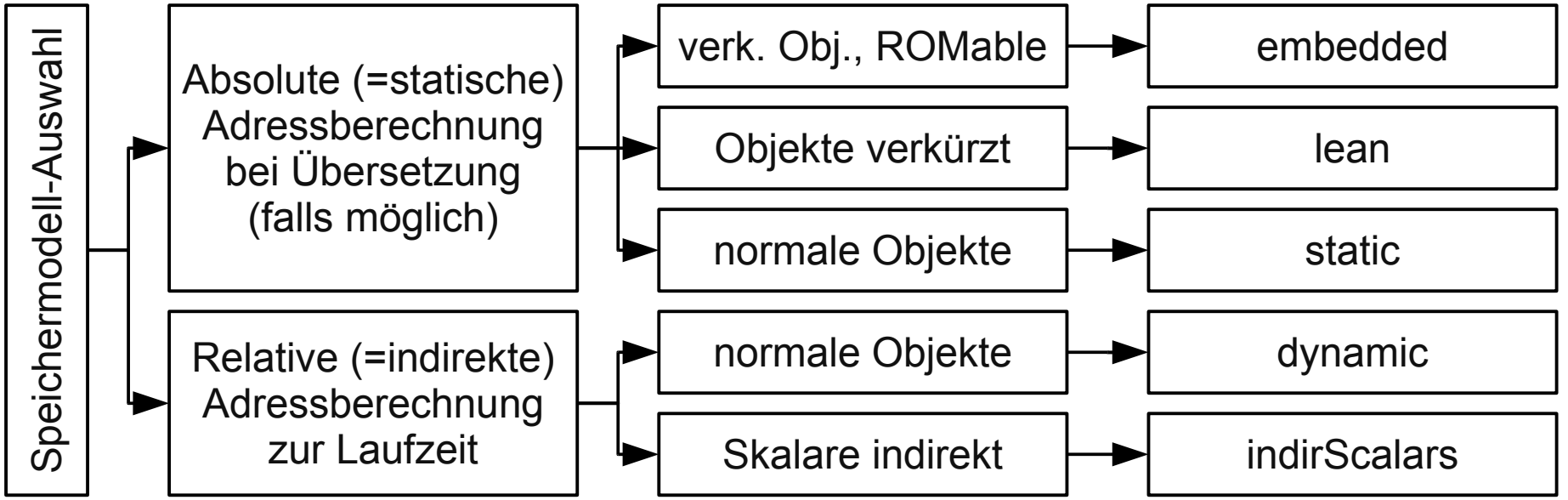
Größenvergleich	SunJDK	JikesRVM	SJC
Dateien	26505	2292	172
Textzeilen	7.60 Mio	0.48 Mio	0.05 Mio
Textzeichen	337.83 Mio	18.25 Mio	1.99 Mio

3.5 Objekt- und Code-Design

- Speicherlayout mit zweiköpfigen Objekten
  - Trennung von Referenzen und Skalaren
    - Objektanalyse ohne zus. Feldinformation
  - Optionale Indirektion für Skalare
    - flexible Konsistenzmodellwechsel
- Standard-Stack für Methoden
  - Für Parameter und lokale Variablen
- Registerbasierte Code-Erzeugung
  - Stack nur einbezogen, falls keine Register frei

3.6 Speicherlayout-Design

- Wahlmöglichkeiten für Gesamt-Speicherlayout
  - Objekt-Verschiebbarkeit bei ROM-Image unsinnig
  - Code-Verschiebbarkeit für Cluster erforderlich
  - Wenig Zusatz-Code für verschiedene Modi



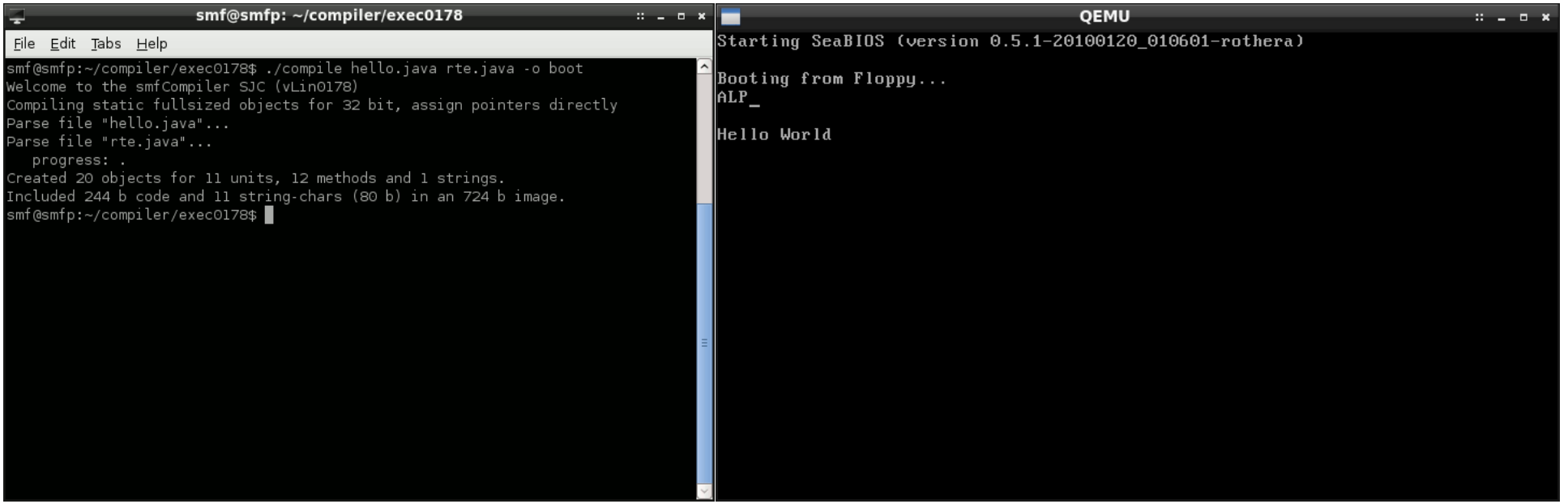
4. Beispielanwendungen

1. Motivation
2. Zielsetzung
3. Implementierung
4. Beispielanwendungen
5. Zusammenfassung und Fazit

4.1 Minimal-Hello-World

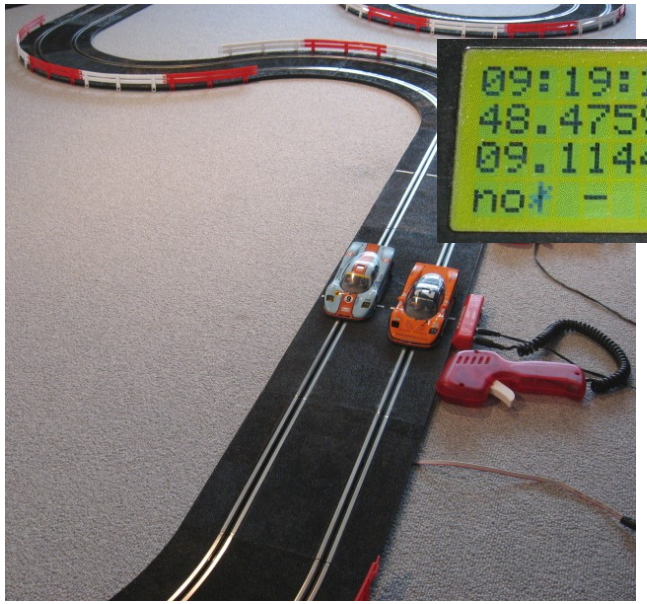
- „Hello World“ x86-bootfähig in Java
  - 16 Zeilen für vollständiges „Hello World“
  - 59 Zeilen für Minimal-Laufzeitumgebung
  - Speicherabbild insgesamt 724 Bytes groß*

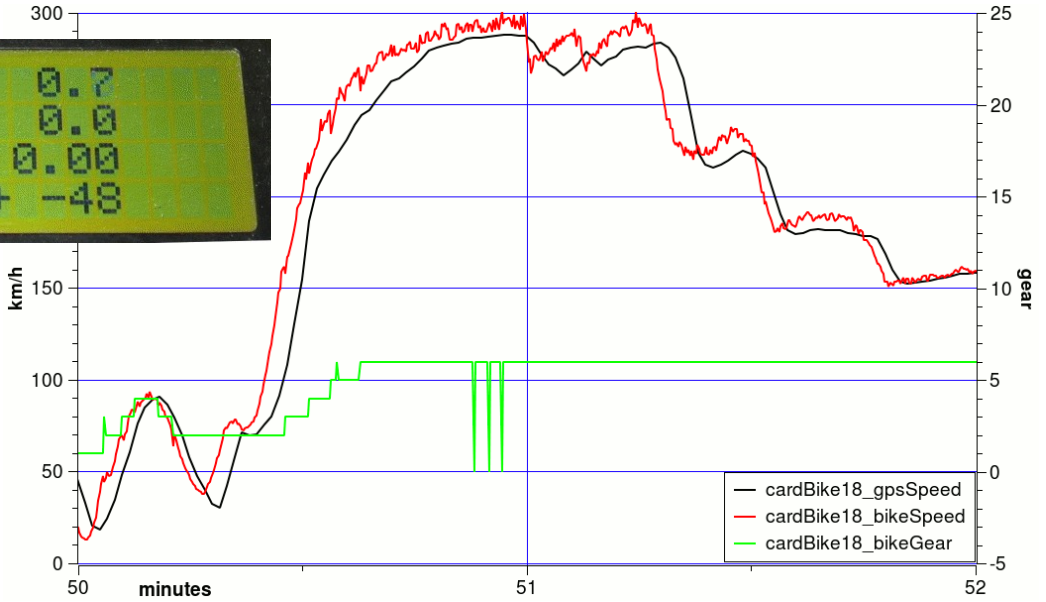
* Bootsektor der Diskette: +512 Bytes



4.2 Eingebettete Systeme

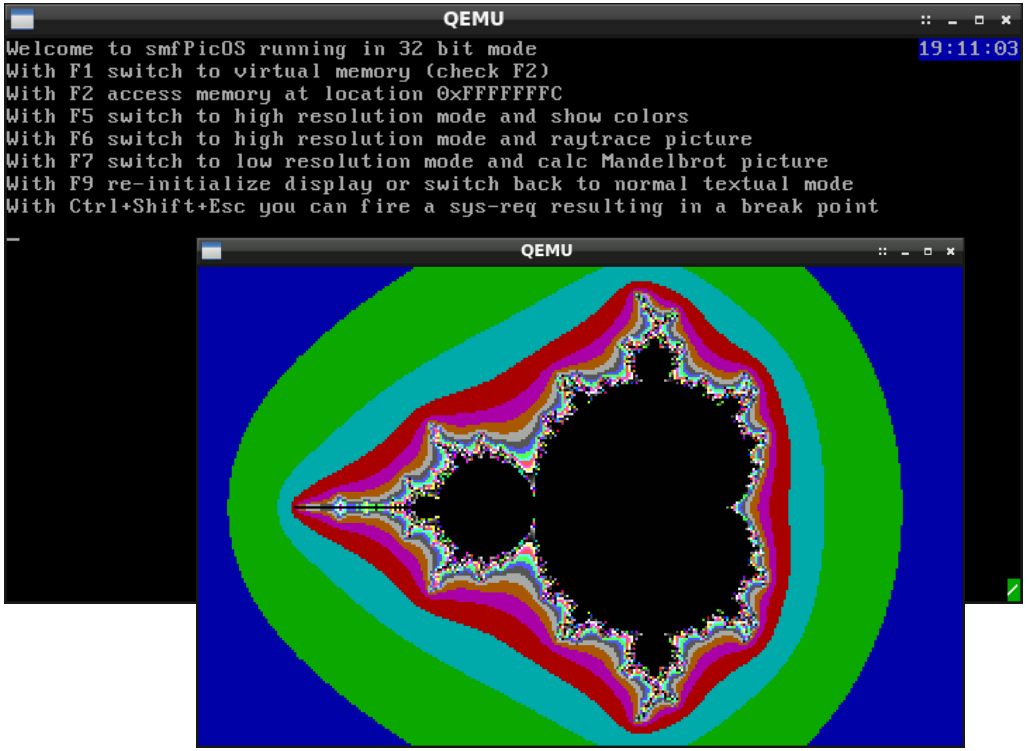
- Zeitmessung für Rennbahn
  - 8 Bit ATmega8 mit 8 kb Flash, 1 kb RAM
- Eingebettete mobile Datenaufzeichnung auf SD
  - 8 Bit ATmega32 mit 32 kb Flash, 2 kb RAM

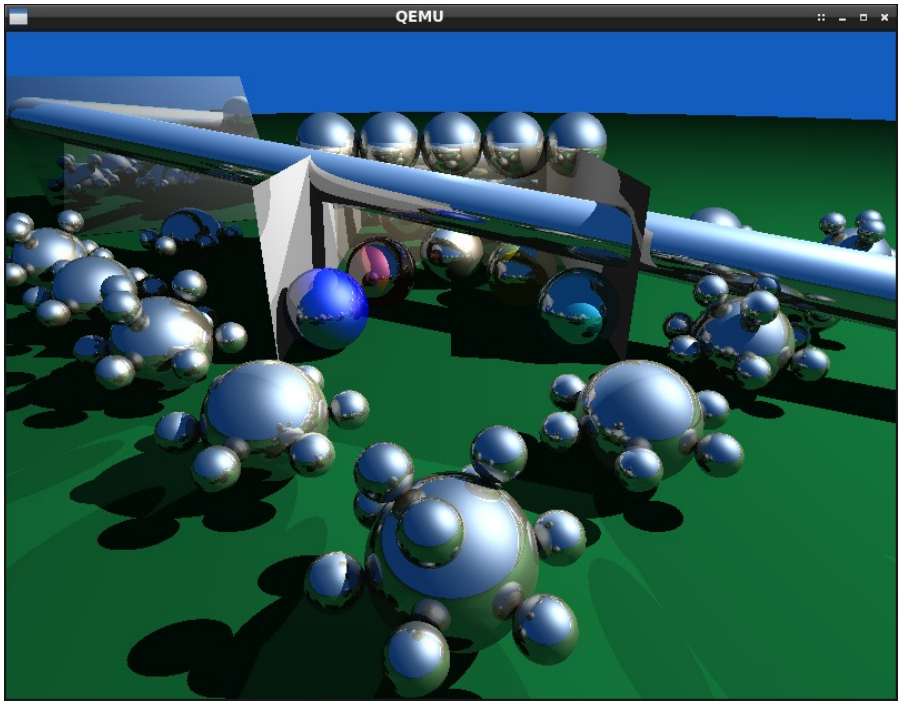




4.3 Test-Betriebssystem

- Testsystem „PicOS“ nur 34 bis 40 kb groß
  - 32 oder 64 Bit mit gemeinsamer Codebasis
  - inkl. Raytracer, Fraktalgenerator, MMU...





4.4 Forschungsprojekte

- Forschungsprojekt „Rainbow“ der Uni Ulm
  - Verteiltes transaktionales 64 Bit Cluster-System
    - vollständiges Betriebssystem inkl. Treiber (nic, gpu...)
  - Derzeit diverse Dissertationen auf Basis von SJC
    - demnächst: Dissertation zu Code- und Instanzevolution
- „Wissenheim“-Demonstrator als 3D-Umgebung
  - Teilpaket des EU-Forschungsprojekts „XtreemOS“
    - läuft mit SJC übersetzt nativ unter Linux im Grid
    - gleicher Quellcode als JDK-Applet im Browser
  - Kooperation mit Uni Ulm / Uni Düsseldorf

4.5 Lehrverwendungen

- Lehr-Einsatz an der Hochschule Kempten
  - Vorlesung „Betriebssystem im Eigenbau“
  - Vorlesung „Verteiltes BS im Eigenbau“
  - Vielleicht bald mehr? Sehr gern!
- Lehr-Einsatz an der Uni Ulm
  - VL „Systemprogrammierung“
  - VL „Technische Informatik“
  - PK „BS im Eigenbau“
  - VL „GPS-Logger“

5. Zusammenfassung und Fazit

1. Motivation
2. Zielsetzung
3. Implementierung
4. Beispielanwendungen
5. Zusammenfassung und Fazit

5.1 SJC in Stichworten

- SJC übersetzt Java-Quellen in Maschinencode
  - Unterstützt werden diverse Systeme 8 bis 64 Bit
- SJC ist einsetzbar für Forschung und Lehre
  - Unterstützt werden diverse Speicherlayouts
- SJC ist schlank und modular aufgebaut
  - Leicht erweiterbar und gut verständlich

- Fazit: viel Arbeit, aber viel Spaß :-)

Vielen Dank für Ihre Aufmerksamkeit

SJC steht unter GPLv3 und ist erhältlich unter www.fam-frenz.de/stefan/compiler.html

SJC ist entstanden als Ein-Personen-Freizeitprojekt. Für die produktiven Diskussionen vor allem mit P. Schulthess, M. Schöttner und P. Schmidt möchte ich mich herzlich bedanken.