

Portierung eines JIT-Compilers auf FPGAs

Georg Acher

acher@in.tum.de, http:/wwwbode.in.tum.de/∼acher

Lehrstuhl für Rechnertechnik und Rechnerorganisation / Parallelrechnerarchitektur(LRR-TUM)

Institut für Informatik der Technischen Universität München

20.9.99, JavaDays '99

Inhalt:

- Motivation
- Anforderungen an das JIT-System
- Jiffy-Systemübersicht
- Performanceergebnisse
- Zusammenfassung und Ausblick

Motivation

Ausführung von Java/JVM-Code:

1) JVM Interpreter (langsam, portabel)

2) JVM Just-In-Time Kompilation (schnell, ressourcenintensiv)

3) Java-CPU

4) Java/JVM Precompilierung

Idee:
JIT Kompilation auf spezieller HW
Ausführung auf normaler CPU

HW: Field Programmable Gate Array (beliebig oft rekonfigurierbar)

CPU: i86-kompatible, ARM, Alpha etc.
Anwendungsbereich: NCs, Embedded Systems

Vorteile:

- Benötigt weniger CPU-Ressourcen
- $\rightarrow$ gute Performance trotz 'schwacher' CPUs/wenig Speicher
- Bessere 'Responsiveness' bei interaktiven Anwendungen
- Zusatznutzen durch rekonfigurierbares FPGA (Classloader, GC)

Anforderungen an einen JIT-Algorithmus für FPGAs (I)

Ziel: (Rechen)-Performance >50% C/C++

Hardwarebedingte Anforderungen:

- Möglichst einfache Logik → wenig Gatter → preiswertes FPGA
- Weitestgehende CPU-Unabhängigkeit

→ Tabellengestützte Übersetzung + Peephole-Optimierung

- FPGA-Speicher vs. System-Speicher: Beachtung der Zugriffsmuster

→ Restriktionen für mögliche Algorithmen

- Flexible Anbindung an das Zielsystem (PCI, CPU-Bus, USB(?))

Anforderungen an einen JIT-Algorithmus für FPGAs (II)

Softwarebedingte Anforderungen:

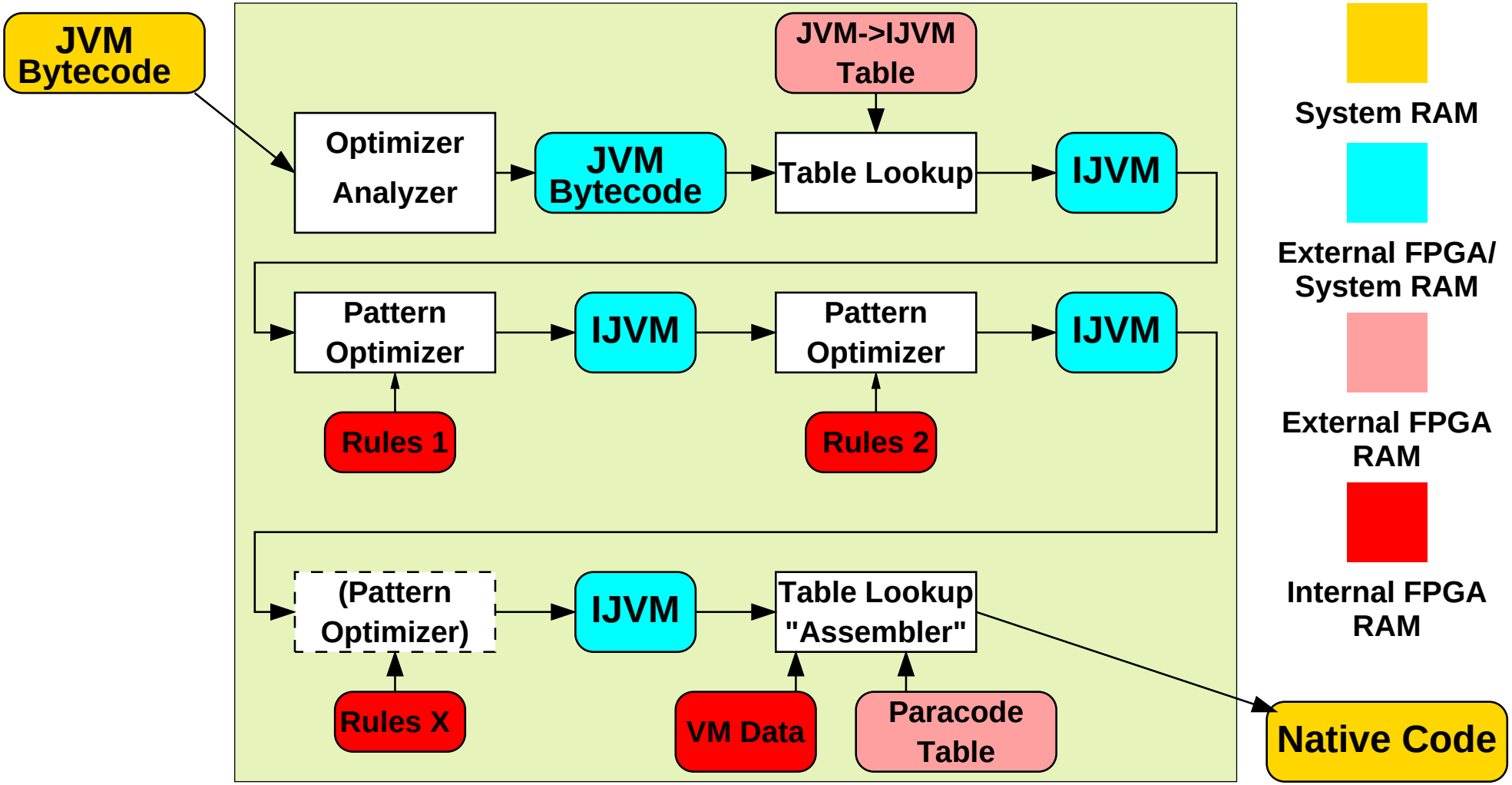
- HW Unabhängig von JVM-Umgebung

$\rightarrow$ kein direkter Zugriff auf JVM-Datenstrukturen in HW

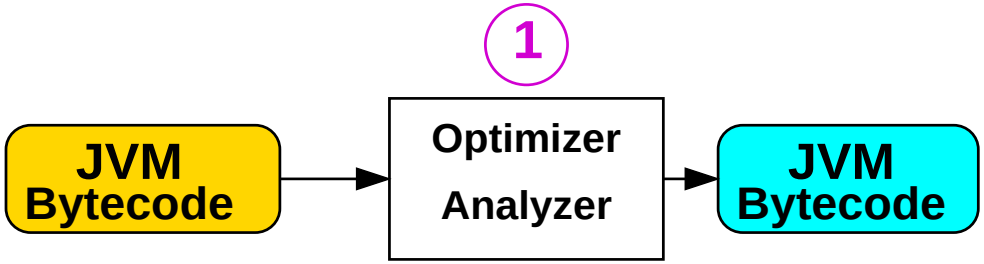
$\rightarrow$ Resolving ausschließlich zur Laufzeit

- Einfache Einbindung in JVM-Umgebung (z.B. Plugin)
- Einfache Anpassung an neue CPU-Architekturen

Jiffy Systemübersicht (I)

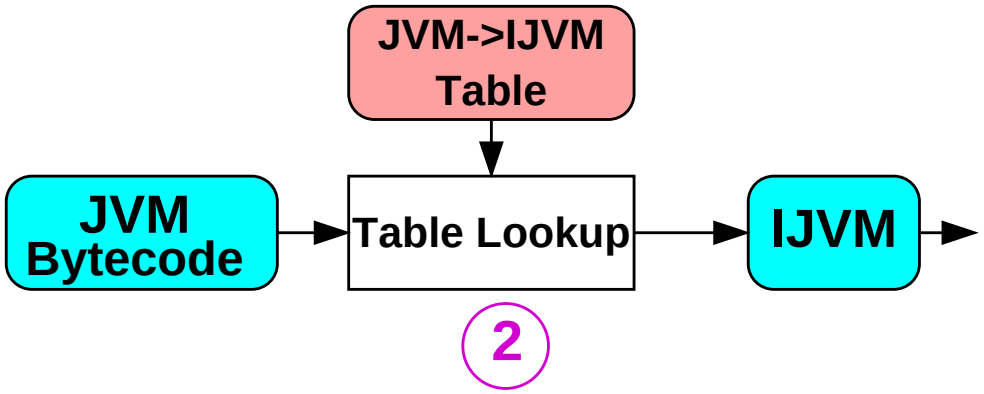


Jiffy Systemübersicht (II)



1) Analyse und Optimierung des JVM-Bytecodes
(Sprungziele, Nutzung lokaler Variablen, Test auf Leaf-Procedure etc.)
Zusammenfassung von häufigen Bytecode-Folgen zu 'Quick'-Opcodes
Beispiel: `dcmplt;ifeq` $\rightarrow$ `if_dcmplt`

Jiffy Systemübersicht (III)



2) Übersetzung von JVM-Bytecode in ein Zwischenformat (IJVM)

IJVM: Registerbasiert, expliziter Stack

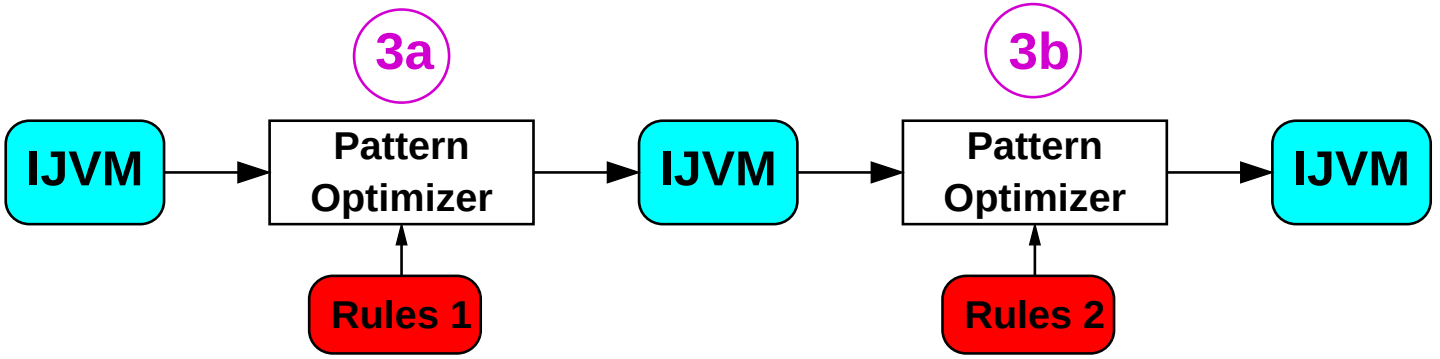
Fixes Befehlsformat (ähnlich RISC)

IJVM-Befehle haben JVM-Komplexität

Beispiel:

`iadd` → `pop_i %r1;pop_i %r2; add_i %r1,%r2;push_i %r2`

Jiffy Systemübersicht (IV)



3) Optimierung nach Peepholeprinzip auf IJVM-Pipeline

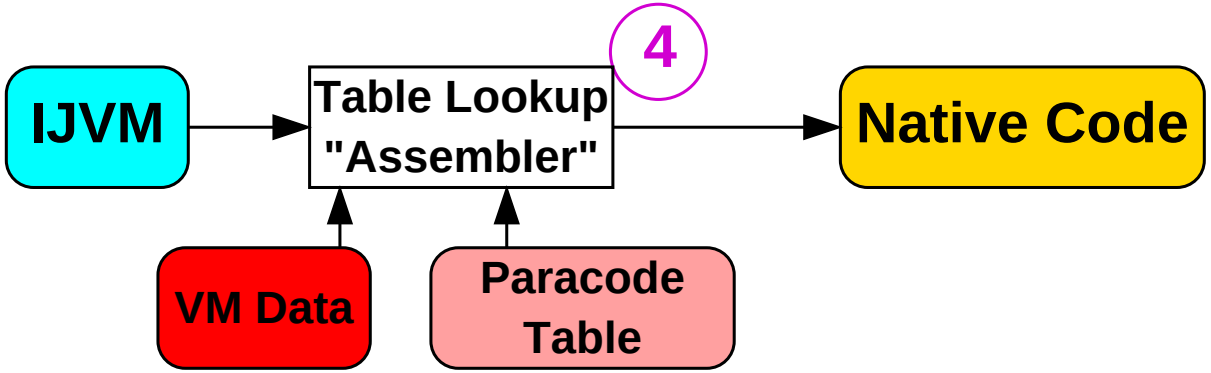
Kaskadierung mit unterschiedlichen Regeln

Anwendung in-place möglich

Beispiel:

`movec_i 2,%r1;push_i %r1;pop_i %r2` → `movec_i 2,%r2;nop;nop`

Jiffy Systemübersicht (V)



4) Übersetzung in nativen Code mit parametrisierbaren Assemblercode für jeden IJVM-Befehl

Assembler-HW unabhängig von CPU, einheitliches Tabellenformat
Resolving nur während der Ausführung durch CPU über Stub-Funktionen
Beispiel:
`moverlv_i %r2,5` → `movl %ebx,8+4*5(%ebp)`

Performanceergebnisse der Jiffy-Implementierung in SW

Testumgebung:

x86, PII-233Mhz	Alpha, 21164-500Mhz
gcc/egcs-1.0.2	gcc/egcs-1.0.2
JDK1.2(x86, Linux, Jiffy)	JDK1.2(alpha, Linux, Jiffy)
JDK1.2(x86, Linux, sunwjit)	CACAO
JDK1.2(x86, Linux, TYA 1.3)	
Kaffe V1.0b4 (x86)	
Symantec JIT V210.065	
MSIE 4.0	

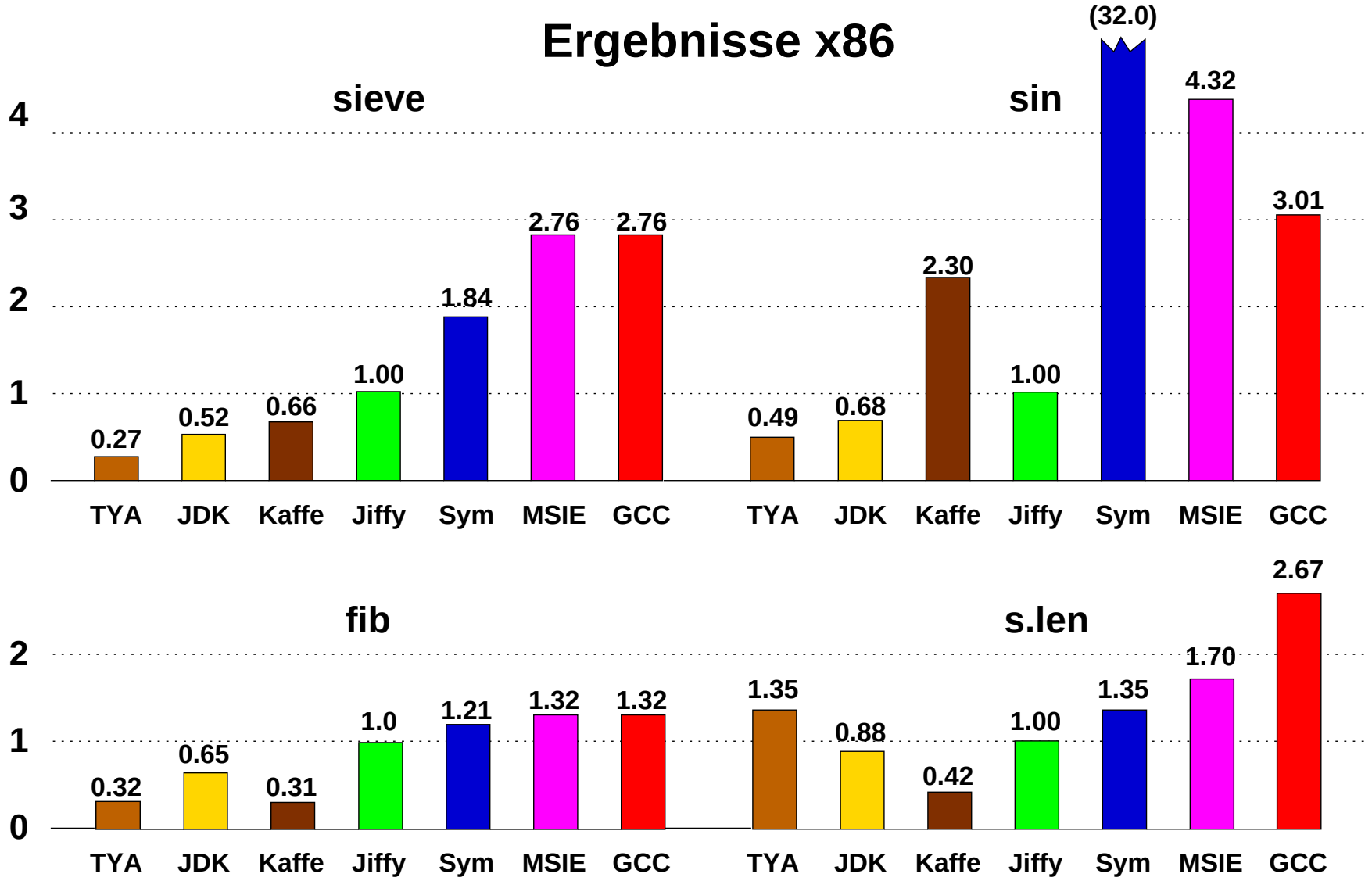
(Mikro)-Benchmarks:

sieve: Sieb des Eratosthenes, testet Arrayzugriffe

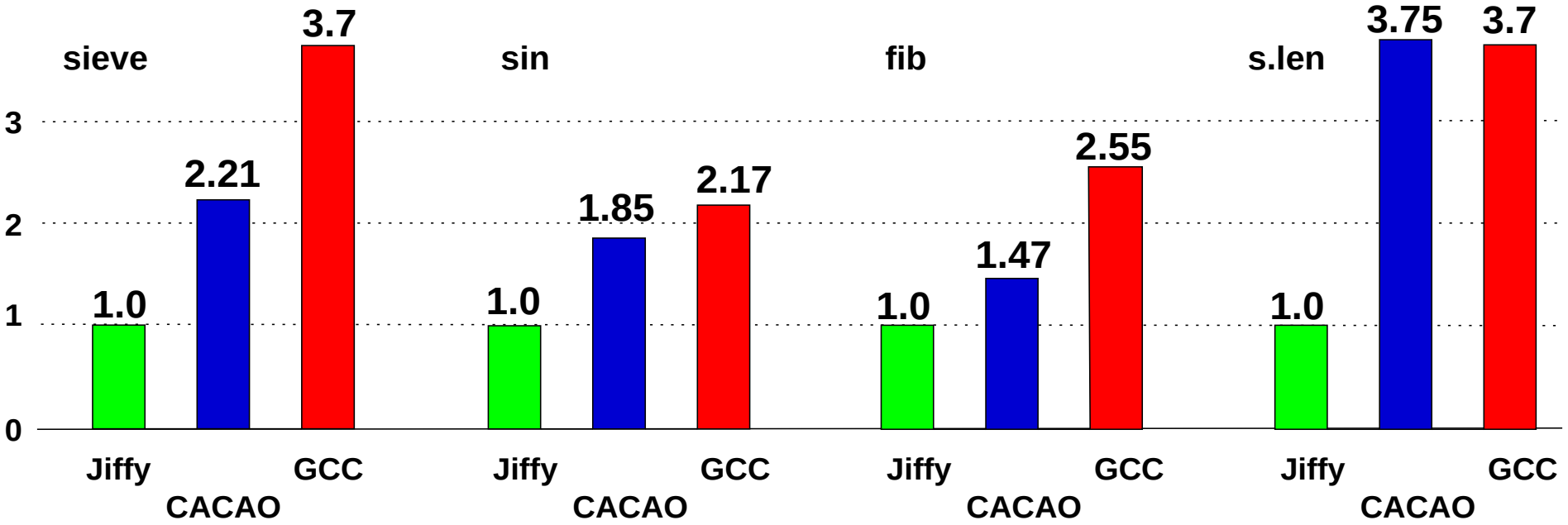
sin: Math.sin(), testet Native-Aufrufe

fib: Rekursive Fibonacci-Berechnung, testet invokestatic

s.len: Testet invokevirtual und getfield


Ergebnisse x86
sieve
sin
fib
s.len
TYA
JDK
Kaffe
Jiffy
Sym
MSIE
GCC
0.27
0.52
0.66
1.00
1.84
2.76
2.76
0.49
0.68
2.30
1.00
(32.0)
4.32
3.01
0.32
0.65
0.31
1.0
1.21
1.32
1.32
1.35
0.88
0.42
1.00
1.35
1.70
2.67

Ergebnisse Alpha



Zu erwartende Übersetzungsgeschwindigkeit der Hardware:
(aufgrund SW-Instrumentierung, nicht pipelined)

40-50 Takte pro JVM-Instruktion
$\rightarrow$ ca. 600000-700000 Instruktionen/s bei 30MHz

Wenig Portierungsaufwand der Architektur-spezifischen Teile

Zusammenfassung und Ausblick

- Jiffy bietet gute Geschwindigkeit des Kompilats bei sehr schneller Übersetzung
- Entlastung der CPU
- Einfache Portierung

Weitere Arbeiten:

- Bessere Anbindung an JDK
- Weitere Optimierungen
- Umsetzung der Übersetzungsstufen in FPGA-Logik
- Entwicklung eines Prototypen auf FPGA-Basis

Ende...


anke für die Aufmerksamkeit!