TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Technische Informatik, Professur für VLSI-Entwurfssysteme, Diagnostik und Architektur

Erschließung von Just-in-Time-Compilierungstechniken in der Realisierung eines retargierbaren Architektursimulators

Diplomverteidigung

Marco Kaufmann
s9186072@mail.inf.tu-dresden.de

Dresden, 20.01.2010

Gliederung

1. Einführung
2. Simulationstechniken
3. Simulatorentwurf
4. Architekturbeschreibung
5. Implementierung
6. Ergebnisse
7. Ausblick

1. Einführung

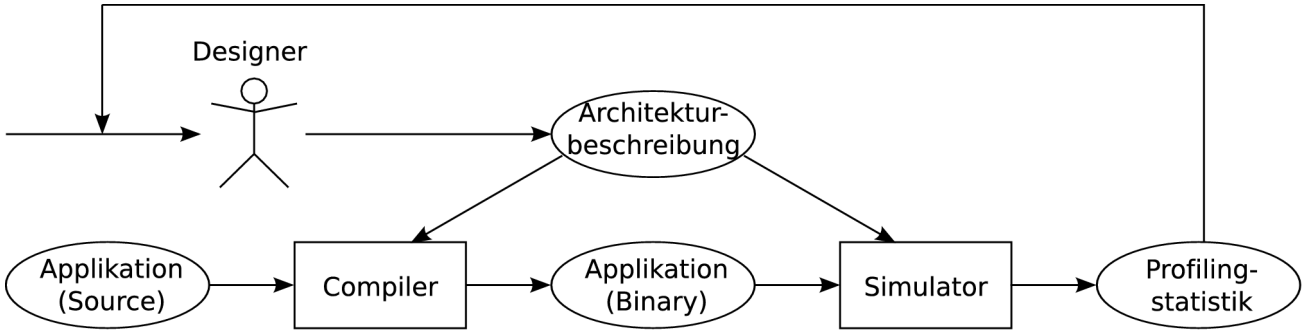
Aufgabenstellung

- Literaturstudium zu retargierbaren Architektursimulatoren und ihrer Beschreibungssprachen unter Berücksichtigung von JIT-Techniken
- Abgrenzung des Simulators und der von ihm unterstützten Techniken
- Entwurf des Simulators und Spezifikation der notwendigen Änderungen an TADL
- Implementierung des entworfenen Simulators
- Nachweis der Funktion und Bewertung der Implementierung
- Dokumentation des Entwurfs und der erzielten Ergebnisse

1. Einführung

Zweck der Simulation

- Validierung von Architekturdesigns
- Vorab Testen von Compiler / Software
- Evaluierung von Entwurfsentscheidungen



Entwurfszyklus nach [5]

Mächtiges Werkzeug für Architekturentwurf, DSE und Hardware / Software Codesign

1. Einführung

Anforderungen an Simulatoren

- Hohe Simulationsgeschwindigkeit
- Flexibilität
- Retargierbarkeit
- Portierbarkeit

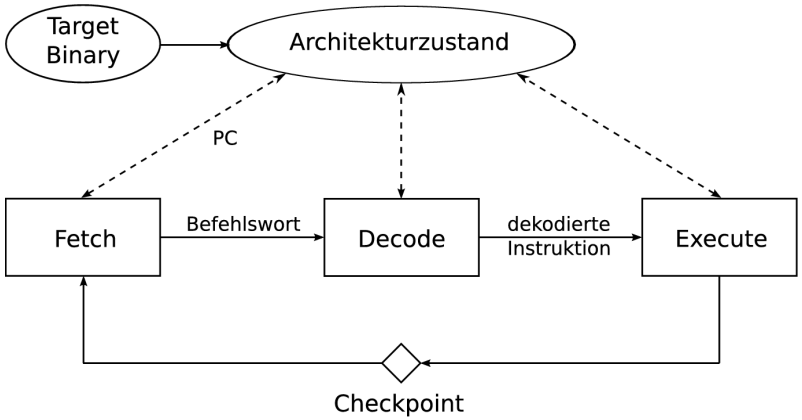
Im Unterschied zu Emulatoren:

- Beobachtbarkeit
- Grenze zwischen Simulation und Emulation verläuft jedoch unscharf

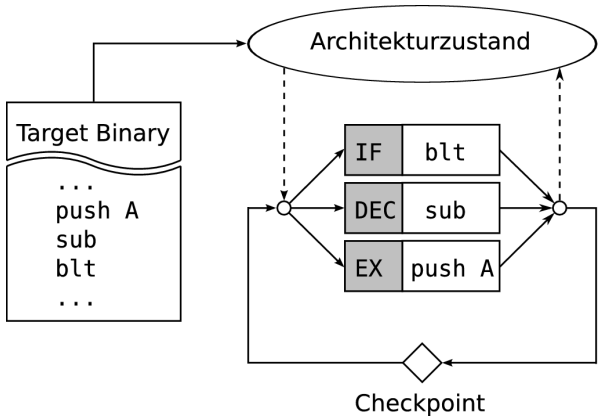
2. Simulationstechniken

Interpretierende Simulation

- Flexibelster, aber langsamster Ansatz
- Beispiel: DITO / TADL



Interpretierende Simulation, befehlsgenau



Interpretierende Simulation, zyklengenau

Die meiste Zeit wird mit dem Holen und Dekodieren von Befehlen verbracht, anstatt der eigentlichen Befehlsausführung!

2. Simulationstechniken

Statisch Compilierende Simulation

Idee:

- Simulationscompiler
- Möglichst viel Overhead von Laufzeit auf Compilierzeit verlagern
- Befehle vorab holen und dekodieren
- Zur Laufzeit lediglich Befehlsausführung

Sehr schnell, aber unflexibel!

2. Simulationstechniken

Dynamisch Compilierende Simulation

Intention:

- Flexibilität interpretierender mit Performance compilierender Simulation vereinen

Idee:

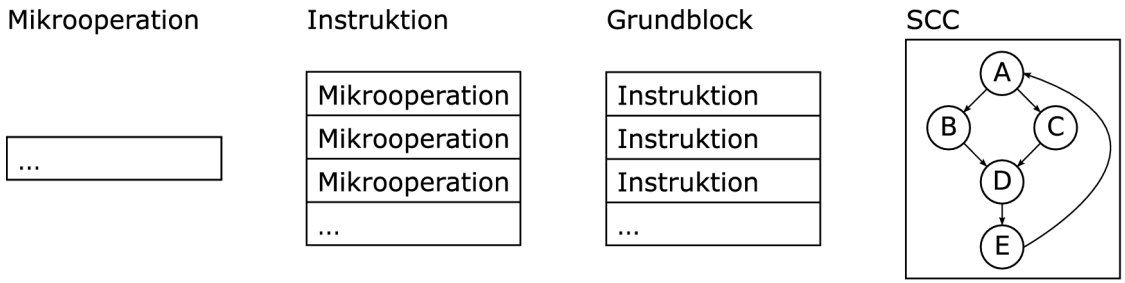
- Integration von Simulationscompiler und Simulator
- JIT-Compiler: Holen, Dekodieren und Übersetzen von Instruktionen Just-in-Time vor der ersten Ausführung

Schnell und flexibel, aber schlecht portierbar!

2. Simulationstechniken

Große Übersetzungseinheiten

- Bisher: Übersetzungseinheit = einzelner Befehl oder Mikrooperation
- Vorteil größerer Übersetzungseinheiten:
  - ➔ Weniger Zeit in Simulationsschleife
  - ➔ Größerer Bereich für Optimierung durch Compiler
- Beispiele: EHS, QEMU



Verringert jedoch die Beobachtbarkeit und geht somit bereits in Richtung Emulation!

3. Simulatorentwurf

Entwurfsziele

- Hochperformanter Simulator mit JIT-Compilierungstechniken
- Portierbarkeit
- Keine Einschränkung der Retargierbarkeit gegenüber DITO
- Flexibilität und Beobachtbarkeit dagegen nebenrangiges Ziel
  - ➔ Keine pipelinegenaue Beschreibung und Simulation
  - ➔ Vorerst kein selbstmodifizierender Code
  - ➔ Dynamisch nachgeladener Code nur mit Einschränkungen möglich

3. Simulatorentwurf

Java als Quasi-Hostplattform

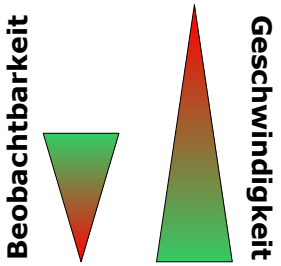
Schlechte Portierbarkeit von JIT-Simulatoren. Lösung:

- Verwendung der JRE als Quasi-Host!
  - ➔ JIT-Simulationscompiler generiert Java-Bytecode
  - ➔ Transformation in nativen Code für Host durch JVM
  - ➔ Backend somit für Vielzahl von Plattformen schon verfügbar
- Weiterer Vorteil: Optimierer bereits „gratis“ in JVM enthalten
  - ➔ Simulationscompiler kann auf eigenen Optimierer verzichten!
- Nachteil: eine Java-Klasse für jede generierte Translated Function
  - ➔ Laden von Klassen ist relativ teuer

3. Simulatorentwurf

Simulationsmodi

- Interpretermodus
- Cached-Interpretermodus
- Single-Instruction-JIT – Modus (SI-JIT)
- Basic-Block-JIT – Modus (BB-JIT)



➔ Sinvoll zur Anwendung bei der Simulation: JIT-Modi

➔ Interpretermodi zu Test- und Vergleichszwecken

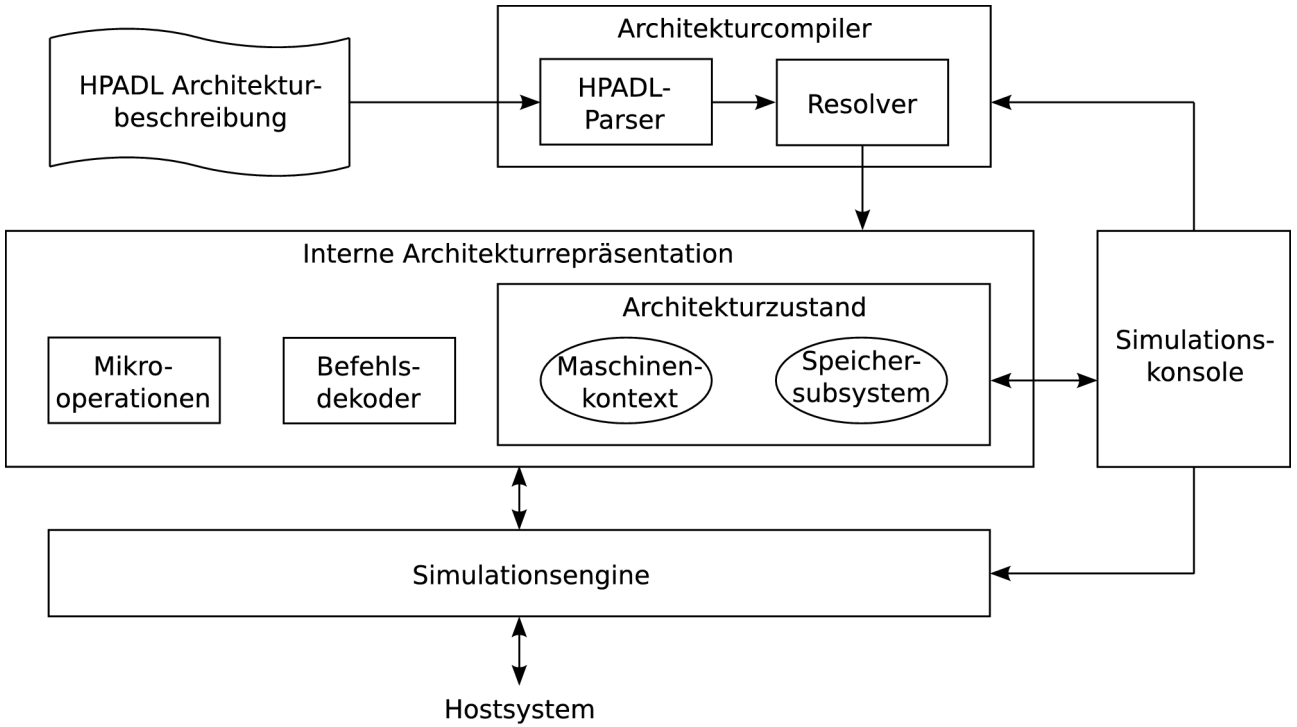
4. Architekturbeschreibungssprache

Eigenschaften von HPADL und Vergleich mit TADL

- Verzicht auf Pipelinebeschreibung
  - ➔ Kein Zeitverhalten an Ressourcen (Stalls, Ports, Verzögerungen, Requests)
  - ➔ Register = herkömmliche Variablen
- Neues Dekoder- und Befehlsbeschreibungsmodell
  - ➔ Strikte Trennung von Dekoder – und Befehlsbeschreibung
  - ➔ Dekodieralgorithmus als Bestandteil der Architekturbeschreibung
  - ➔ Dekoder generiert Sequenz von Mikrooperationen
    - Modularer Aufbau des Befehlssatzes
    - Keine Befehlshierarchie
- Typsystem (`bool`, `int`, `long`, `big`)
- Import externer Klassen möglich

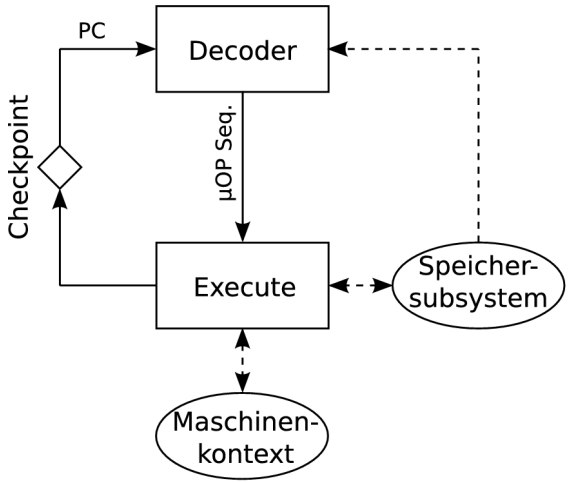
5. Implementierung

Module

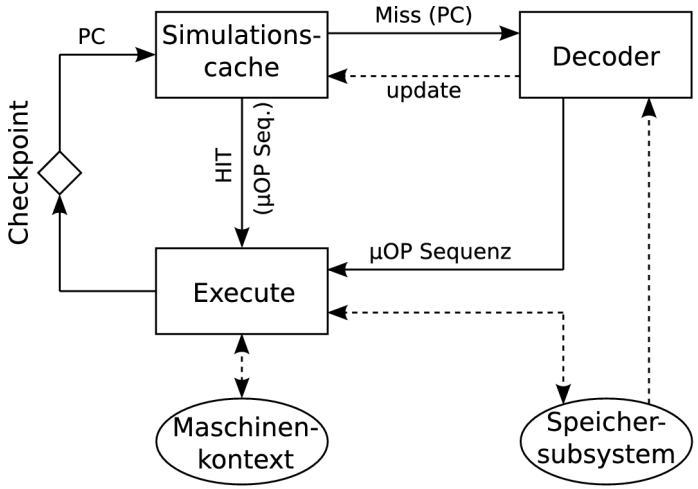


5. Implementierung

Simulationsengine – Interpretermodi



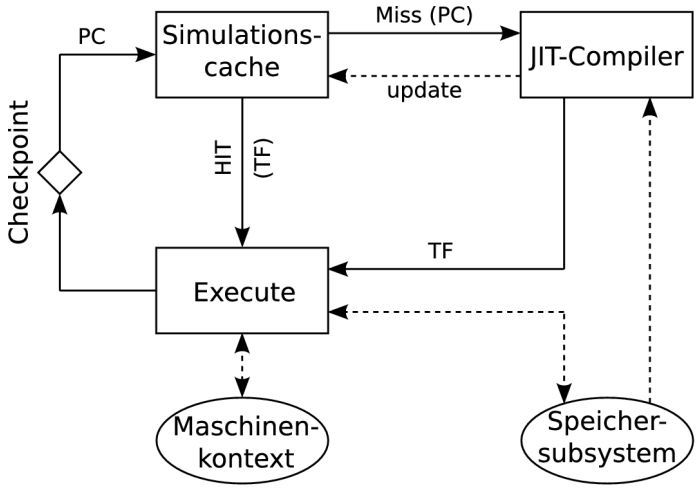
Interpretermodus



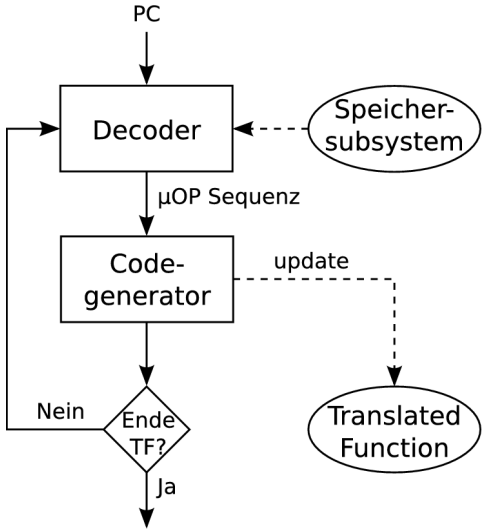
Cached-Interpretermodus

5. Implementierung

Simulationsengine – JIT-Modi



Simulationsschleife

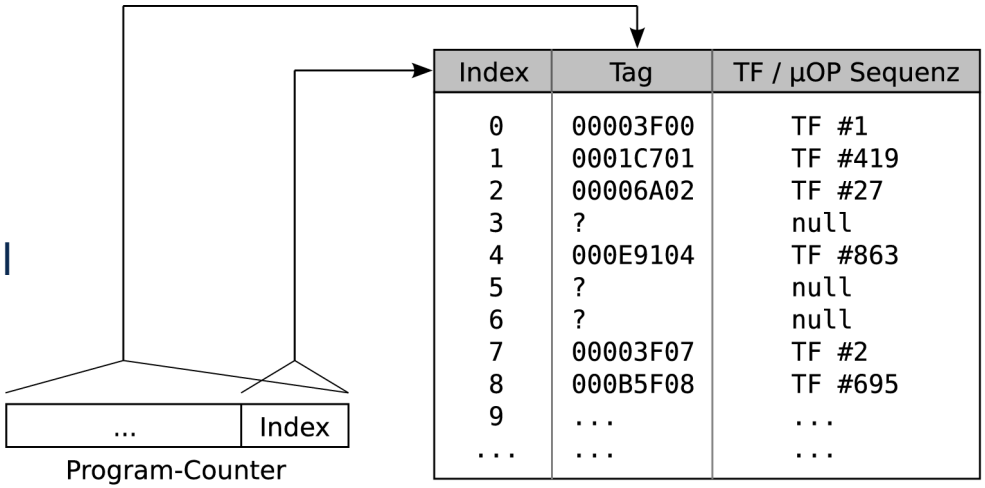


JIT-Compiler

5. Implementierung

Simulationsengine – Simulationscache

- Einfach assoziativ
- 64 Bit Tag
- Indexbreite und Cachegröße variabel



6. Ergebnisse

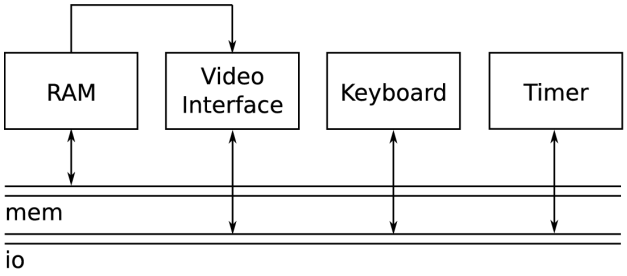
Funktionsnachweis – Zielarchitekturen und Testapplikationen

- DLX, Tweaked-DLX
  - ➔ mit Traps und Floating-Point-Support
  - ➔ Tweaked DLX: Verzicht auf Arrays
  - ➔ Testanwendungen: Add, PrimeNumbers
- 8086, 8086_io
  - ➔ mit Lazy Flag Evaluation
  - ➔ 8086_io: Speichersubsystem mit zusätzlichen IO-Geräten
  - ➔ Testanwendungen: PrimeNumbers, Tetris
- ARMv4, ARMv4_io
  - ➔ Ohne Koprozessorinstruktionen und Softwareinterrupts, nur User-Mode
  - ➔ ARMv4_io: Speichersubsystem mit zusätzlichen IO-Geräten
  - ➔ Testanwendungen: Sieve-Test, Tetris

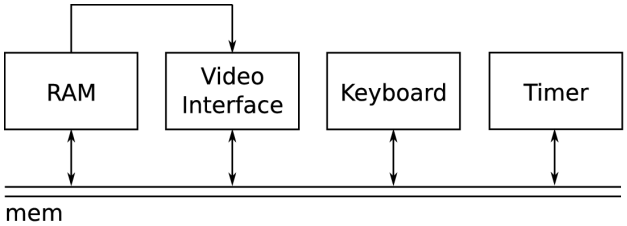
Getestet in allen Simulationsmodi!

6. Ergebnisse

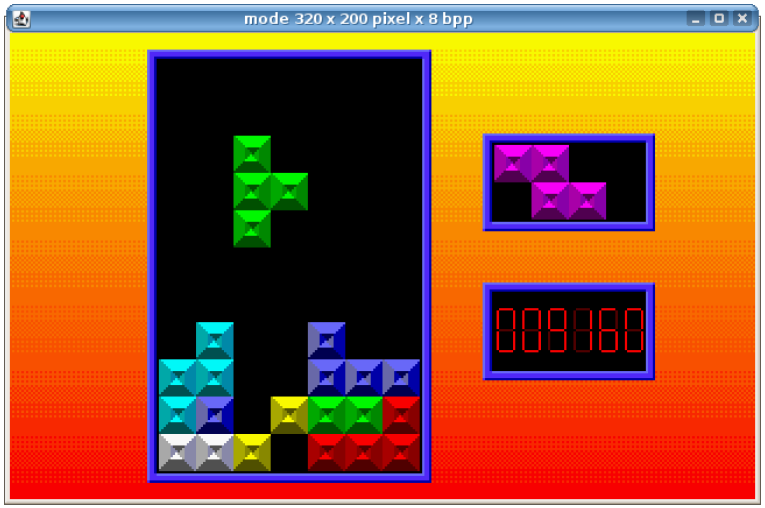
Funktionsnachweis – Speichersubsystem und IO-Geräte



8086_io - Speichersubsystem



ARMv4_io - Speichersubsystem



Tetris – ARMv4_io - Version

6. Ergebnisse

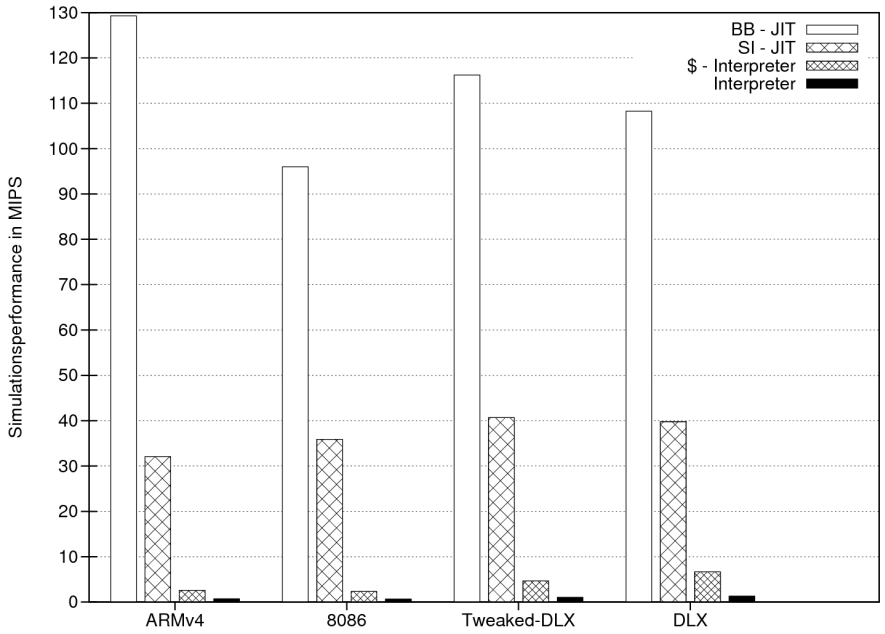
Benchmarktest

Modus	ARMv4	8086	T.-DLX	DLX
BB-JIT	129,3	96,0	116,2	108,2
SI-JIT	32,1	35,9	40,7	39,8
$-Interpreter	2,61	2,35	4,68	6,66
Interpreter	0,72	0,65	1,03	1,30

Simulationsperformance in MIPS

Modus	ARMv4	8086	T.-DLX	DLX
BB-JIT	179,6	147,7	112,8	83,2
SI-JIT	44,6	55,2	39,5	30,6
$-Interpreter	3,63	3,62	4,5	5,1

Speedup gegenüber Interpretermodus (Faktor)



Simulationsperformance

6. Ergebnisse

Benchmarktest – Vergleich mit DITO

Architektur	Takte/s	Instruktionen/s
ARM7v4	41 175	19 955

DITO - Simulationsperformance

Modus	MIPS	Faktor
BB-JIT*	161,4	8087
BB-JIT	129,3	6478
SI-JIT	32,1	1608
$-Interpreter	2,61	131
Interpreter	0,72	36

rISA Sim - Simulationsperformance

- Overhead durch takt- und pipelinegenaue Simulation sowie Listener
- Kein Simulationscache
- Keine Compilierung
- Keine großen Übersetzungseinheiten

6. Ergebnisse

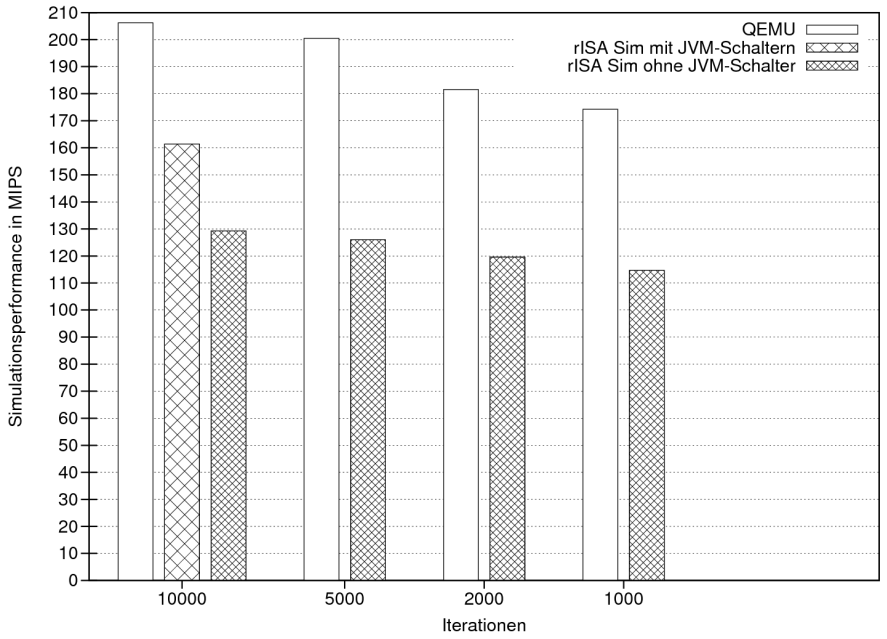
Benchmarktest – Vergleich mit QEMU

Iterationen	QEMU	RISA Sim	
	MIPS	MIPS	Prozent
10 000	206,2	129,3	62,7
5 000	200,5	126,0	62,9
2 000	181,6	119,6	62,9
1 000	174,2	114,7	62,8

QEMU und rISA Sim im Vergleich (ohne JVM-Schalter)

Iterationen	QEMU	RISA Sim	
	MIPS	MIPS	Prozent
10 000	206,2	161,4	78,3

QEMU und rISA Sim im Vergleich (mit JVM-Schaltern)



Simulationsperformance

7. Ausblick

Architektur der Simulationsengine:

- HotSpot-Engine für JIT-Simulationscompiler
- JIT-Compilierung des Speichersubsystems
- Bytecodeoptimierer

Funktionsumfang:

- Interrupts, Breakpoints
- Laufzeitdynamischer Code
- Integration von SI-JIT und BB-JIT Modus
- Automatisierte Abbildung von Arrays auf Einzelvariablen

Literaturangaben

[1] BELLARD, F.: *QEMU, a fast and portable dynamic translator*. In: *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, S. 41–41

[2] JONES, D.; TOPHAM, N.: *High Speed CPU Simulation Using LTU Dynamic Binary Translation*. In: *HiPEAC '09: Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg: Springer-Verlag, 2009. – ISBN 978–3–540–92989–5, S. 50–64

[3] MILLS, C.; AHALT, S. C.; FOWLER, J.: *Compiled Instruction Set Simulation*. In: *Software, Practice and Experience, Vol. 21 #8 (1991), S. 877–889*

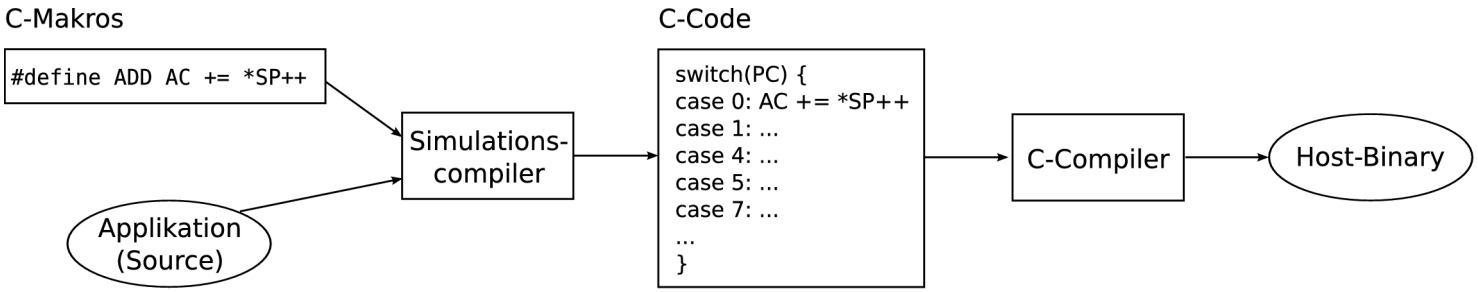
[4] NOHL, A.; BRAUN, G.; SCHLIEBUSCH, O.; LEUPERS, R.; MEYR, H.; HOFFMANN,A.: *A universal technique for fast and flexible instruction-set architecture simulation*. In: *DAC '02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, 2002. – ISBN 1–58113–461–4, S. 22–27

[5] PREUSSER, T. B.: *Entwicklung eines Prozessorsimulators unter besonderer Berücksichtigung des Organic Computing*, 2003

Vielen Dank
für Ihre Aufmerksamkeit!

Zu 2. Simulationstechniken

Statisch Compilierende Simulation mittels C-Makros

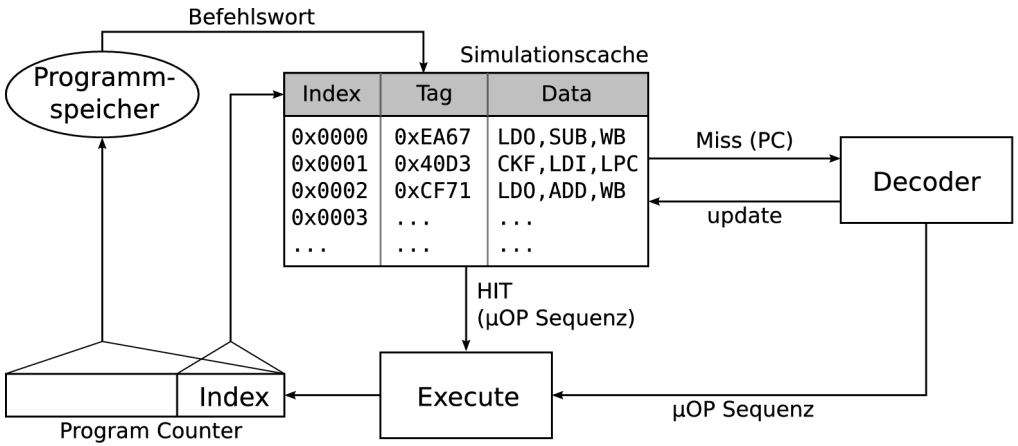


- Einzelne Instruktionen durch C-Makros repräsentiert
- Simulationscompiler generiert C-Code
- Optimierter Simulator durch C-Compiler
- Nicht takt- und pipelinegenau

Zu 2. Simulationstechniken

Dynamisch Compilierende Simulation - JIT-CCS

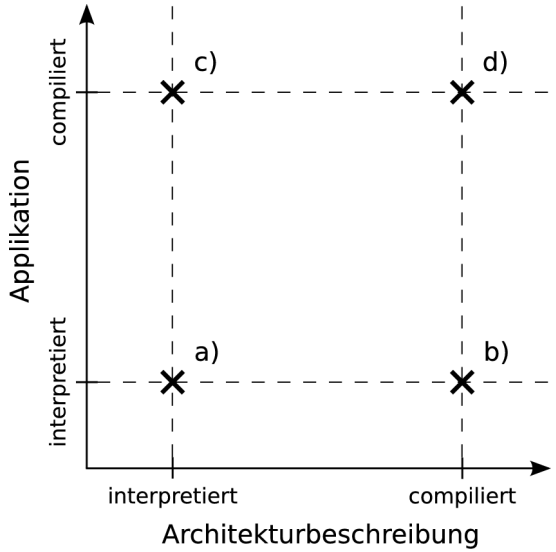
Simulationscache für bereits dekodierte Instruktionen



Simuliert takt- und pipelinegenau.

Zu 2. Simulationstechniken

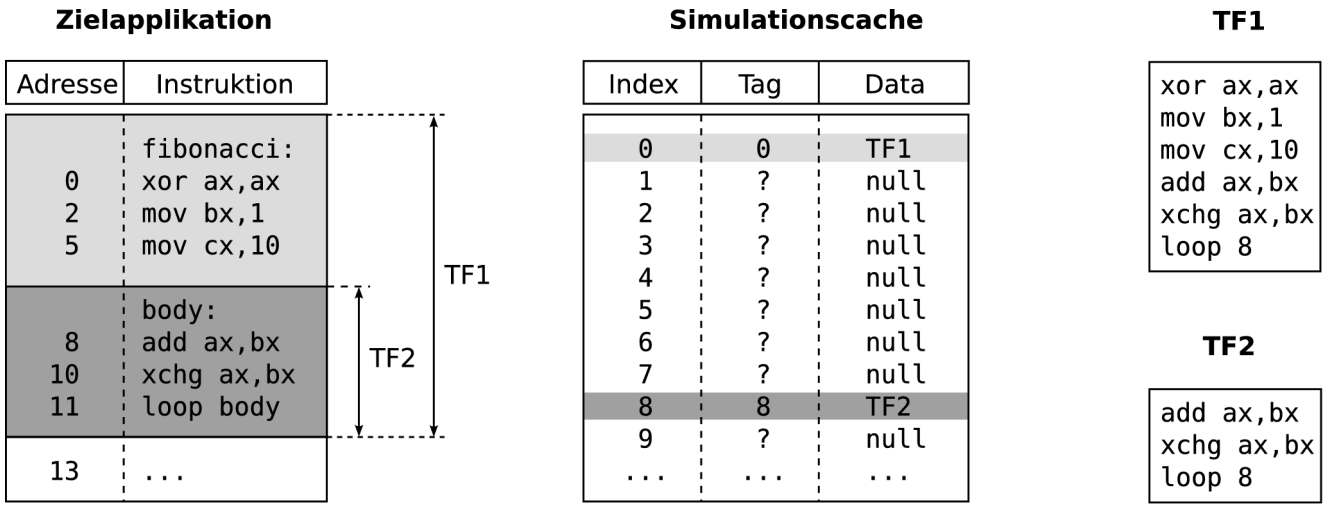
Interpretierende und compilierende Simulation



a) DITO, rISA Sim im Interpretermodus
b) µOPs compiliert, Applikation weiterhin interpretiert
   ➔ Kein Overhead für JIT-Compilierung
c) rISA Sim im Cached Interpretermodus
d) rISA Sim im SI-JIT und BB-JIT-Modus, JIT-CCS, QEMU, EHS

Zu 3. Simulatorentwurf

Überlappende Ausführungseinheiten im BB-JIT-Modus



Zu 4. Architekturbeschreibungssprache

Modellierungsaufwand – Vergleich mit DITO

- Zusätzlich: Implementierung des Dekodieralgorithmus
- Dafür keine Modellierung der Befehlspipeline

rISA Sim:

- DLX (mit FPU): 367 Zeilen
- 8086: 2398 Zeilen
- ARMv4 (Usermode): 1011 Zeilen

DITO:

- DLX (ohne FPU): 301 Zeilen
- ARM7v4 (Usermode): 1523 Zeilen

Zu 4. Architekturbeschreibungssprache

Modellierungsaufwand – Vergleich mit QEMU

- Spezielle Sprachkonstrukte zur effizienteren Architekturbeschreibung:
  - ➔ Speichersubsystem
  - ➔ Dekoderbeschreibung
  - ➔ Mikrooperationen

- QEMU: Standard-C
  - ➔ ARM926EJ-S (ARMv5TEJ), vollständig: ca. 17300 Zeilen
  - ➔ i386: ca. 21800 Zeilen
  - ➔ PowerPC: ca. 35800 Zeilen

Zu 6. Ergebnisse

Benchmarktest – Testplattformen

CPU	2 x AMD Opteron™ Dual-Core @2,4GHz
Cache	L1: 2 x 128 kB, L2: 2 x 1 MB
Hauptspeicher	4 GB
Betriebssystem	Ubuntu 9.04, Kernel 2.6.28-16-generic
JRE	Java™ SE Runtime Environment (build 1.6.0_16-b01), Java HotSpot™ 64-Bit Server VM (build 14.2-b01, mixed mode)

Testplattform I

CPU	Intel® Core™ i7 920 @ 2,67GHz
Cache	L1: 4 x 64kB, L2: 4 x 256 kB, L3: 8 MB shared
Hauptspeicher	6GB GB DDR3-1600 CL-7-7-7-20
Betriebssystem	Ubuntu 9.04, Kernel 2.6.28-16-generic
JRE	Java™ SE Runtime Environment (build 1.6.0_16-b01), Java HotSpot™ 64-Bit Server VM (build 14.2-b01, mixed mode)

Testplattform II

Zu 6. Ergebnisse

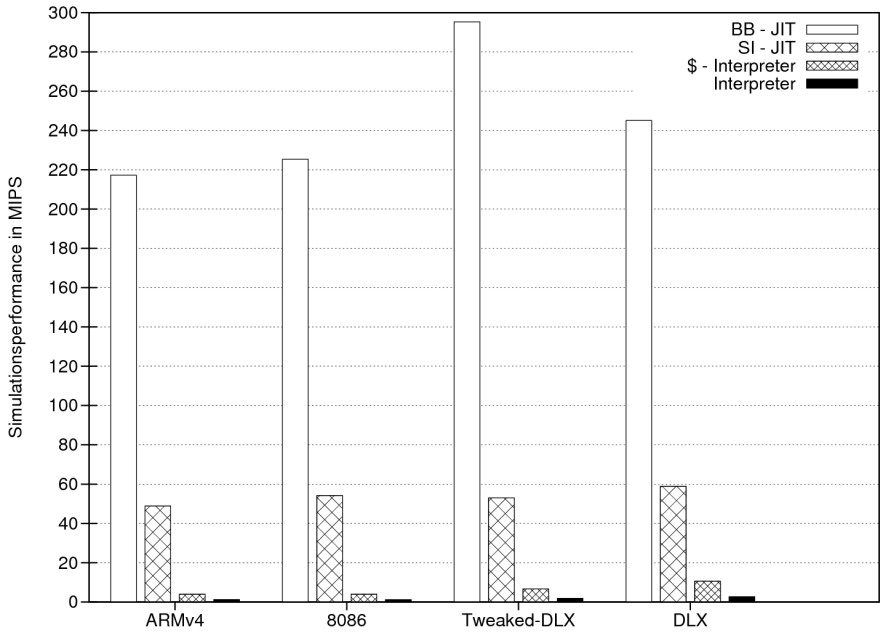
Benchmarktest – Testplattform II

Modus	ARMv4	8086	T.-DLX	DLX
BB-JIT	217,2	225,3	295,2	245,2
SI-JIT	48,9	54,2	53,0	58,8
$-Interpreter	3,92	3,93	6,69	10,6
Interpreter	1,25	1,18	1,92	2,65

Simulationsperformance in MIPS

Modus	ARMv4	8086	T.-DLX	DLX
BB-JIT	173,8	190,9	153,7	92,5
SI-JIT	39,1	45,9	27,6	22,2
$-Interpreter	3,1	3,3	3,5	4

Speedup gegenüber Interpretermodus (Faktor)



Simulationsperformance

Zu 6. Ergebnisse

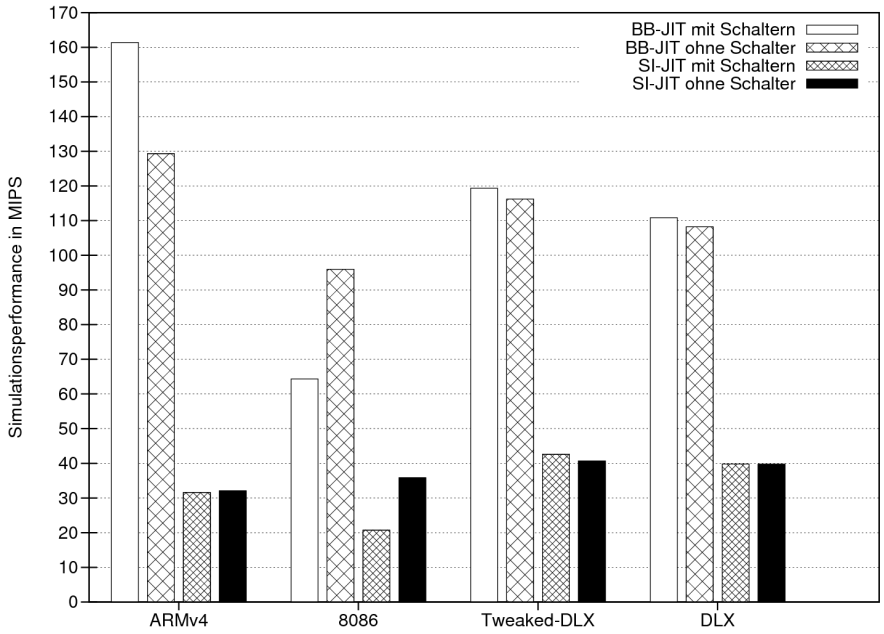
Benchmarktest – Einfluss der JVM-Konfiguration

Modus	ARMv4	8086	T.-DLX	DLX
BB-JIT	161,4	64,4	119,3	110,9
SI-JIT	31,6	20,7	42,6	39,8

Simulationsperformance mit JVM-Schaltern in MIPS

Modus	ARMv4	8086	T.-DLX	DLX
BB-JIT	+24,6	-2,55	+6,89	+1,48
SI-JIT	+2,44	+1,77	+15,2	-3,65

Speedup in Prozent



Simulationsperformance

Zu 6. Ergebnisse

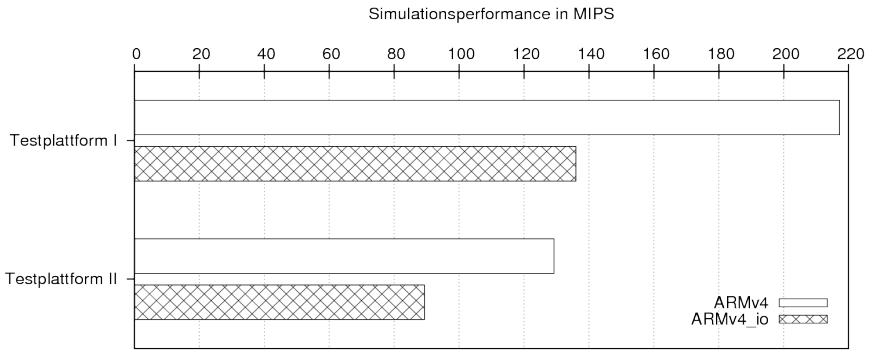
Benchmarktest – ARMv4 und ARMv4_io

Performance	In MIPs	In Prozent
ARMv4	129,3	100
ARMv4_io	89,3	69,1

Testplattform I (BB-JIT)

Performance	In MIPs	In Prozent
ARMv4	217,2	100
ARMv4_io	136,0	62,6

Testplattform II (BB-JIT)



Simulationsperformance – ARMv4 und ARMv4_io

Zu 6. Ergebnisse

Benchmarktest – Vergleich mit DITO

Architektur	Takte/s	Instruktionen/s
ARM7v4	41 175	19 955

DITO - Testplattform I

Architektur	Takte/s	Instruktionen/s
ARM7v4	55 496	26 895

DITO - Testplattform II

Modus	MIPS	Faktor
BB-JIT*	161,4	8087
BB-JIT	129,3	6478
SI-JIT	32,1	1608
$-Interpreter	2,61	131
Interpreter	0,72	36

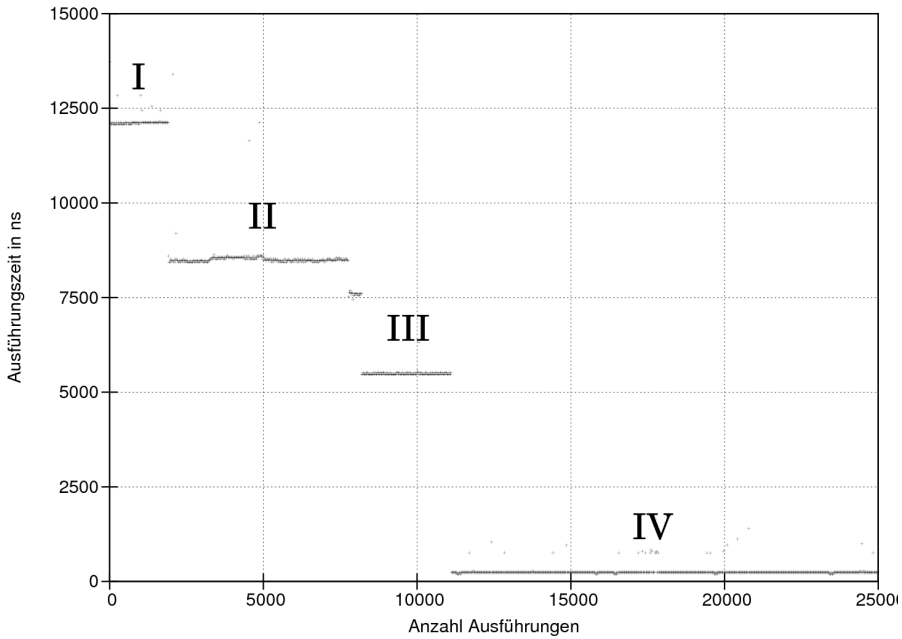
rISA Sim - Testplattform I

Modus	MIPS	Faktor
BB-JIT*	270,7	10065
BB-JIT	217,2	8077
SI-JIT	48,9	1817
$-Interpreter	3,92	145,8
Interpreter	1,25	46

rISA Sim - Testplattform II

Zu 6. Ergebnisse

HotSpot – JIT – Effekte (Beispiel: TF #26)



I : < 2000 Ausführungen, t = 12µs
II : < 8000 Ausführungen, t = 8,5µs
III : < 11000 Ausführungen, t = 5,5µs
IV : ≥ 11000 Ausführungen, t = 0,2µs

Zu 6. Ergebnisse

Kosten der JIT-Compilierung (Beispiel: TF#26)

Teilschritte:

I. Dekodieren der Zielinstruktionen
II. Transformation in Java Bytecode
III. Laden der Klasse und Instantiieren der TF

I	II	III	Gesamt
15,1	106,8	150,2	272,1

Laufzeit in µs

➔ TF#26 enthält 15 ARMv4-Instruktionen
➔ Entspricht 55 127 ARMv4 Instruktionen / Sekunde, bzw. 215 kB ARMv4 Programmcode / Sekunde

Zu 6. Ergebnisse

Vorteile gegenüber QEMU

- Portierbarkeit trotz JIT-Compilierung
  - ➔ JRE als Quasi-Hostplattform
- Spezielle Sprachkonstrukte zur effizienten Architekturbeschreibung
  - Generisches Speichersubsystem,
    beliebige Adressierungseinheit und Zugriffsbreiten
  - Dekoderbeschreibung
  - Datentyp „`big`“ nativ unterstützt
  - ➔ Bessere Retargierbarkeit, weniger Modellierungsaufwand
- Simulator, kein Emulator
  - ➔ Befehlsgenaue Simulation möglich
  - ➔ Simulationskonsole, direkte Inspektion und Manipulation des Architekturzustandes