

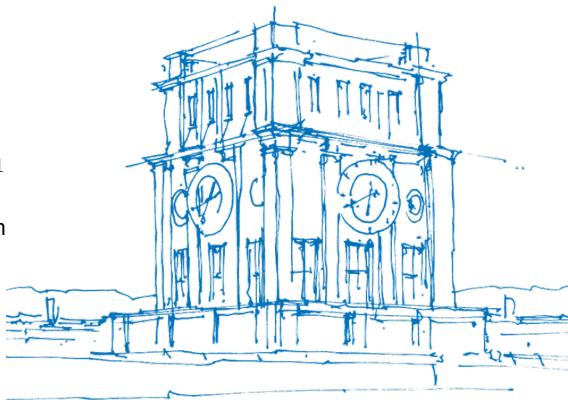
# Dynamische Binärübersetzung: RISC-V $\rightarrow$ x86-64

## Endpräsentation

Noah Dormann<sup>1</sup>, Simon Kammermeier<sup>1</sup>,  
Johannes Pfannschmidt<sup>1</sup>, Florian Schmidt<sup>1</sup>

<sup>1</sup>Fakultät für Informatik, Technische Universität München  
(TUM)

27. Oktober 2020



# Gliederung

- 1 Einführung
  - Problembeschreibung
  - RISC-V vs. x86-64
  - Dynamische Binärübersetzung
- 2 Ansatz
  - Programmablauf
  - Partitionierung des Codes
  - Codegenerierung und Cache
  - Registernutzung
  - Optimierungen
- 3 Ergebnisse und Performanz
  - SPEC CPU 2017
  - Optimierungen
- 4 Demo

## Problembeschreibung

**RISC-V:** Offene ISA, die dem Reduced Instruction Set Computer (RISC) Schema folgt.

**Problem:**

- Verfügbarkeit von RISC-V Prozessoren ist begrenzt.
- Entwickler, die Code für RISC-V als Zielplattform kompilieren, können diesen nicht ausführen.

**Lösung:** Emulieren des RISC-V Befehlsatzes auf einem x86-64 Prozessor

### Warum x86-64?

x86-64 ist der derzeitige Standard für Prozessoren in Laptops und Desktop-PCs.

# RISC-V vs. x86-64

## Gegenüberstellung

### RISC-V Übersicht:

- RISC Schema
- Load-Store-Architektur
- 31 General Purpose Register
- 32 Floating Point Register
- 3-Operanden Adressform
- Spezielles Zero-Register

### x86-64 Übersicht:

- CISC Schema
- Register-Memory-Architektur
- 16 General Purpose Register
- 16 Floating Point (XMM) Register
- 2-Operanden Adressform

# Dynamische Binärübersetzung

## Interpretation

### Vorgehen

- Laden des nächsten Befehls
- Übersetzung
- Ausführen der Seiteneffekte des Befehls (e.g. Änderung eines Registers)

### Vorteile

- Einfach zu implementieren
- Keine Erzeugung von JIT Assembler nötig

### Nachteile

- Mehrfache Übersetzung der selben Instruktion
- Wenig Optimierungspotential

# Dynamische Binärübersetzung

## Statische Binärübersetzung

### Vorgehen

- Einmaliges Übersetzen der gesamten Auszuführenden Datei
- Ausführen der Übersetzung

### Vorteile

- Einmaliger Übersetzungsaufwand
- Theoretisch fast native Geschwindigkeit erreichbar

### Nachteile

- Schwierig zu implementieren (Halteproblem)
- Problematisch bei Änderungen der Assembly zur Runtime

# Dynamische Binärübersetzung

## Dynamische Binärübersetzung

### Vorgehen

- Einlesen eines Blocks von Befehlen
- (Optional) Optimierungen
- Übersetzen des einzelnen Blocks
- Ausführen des Übersetzten Blocks

### Vorteile

- Nur tatsächlich notwendige Teile des Programms werden übersetzt
- Instruktionen werden nur einmal Übersetzt
- Blöcke von Instruktionen eignen sich zur Optimierung

### Nachteile

- Problematisch bei Änderungen der Assembly zur Runtime

# Partitionierung des Codes

## Grundlagen

**Ziel:** Finden von sinnvollen Übersetzungseinheiten

### Überlegung:

- einzelne Instruktionen übersetzen zu aufwändig
- keine Übersetzung des ganzen Programmes

⇒ Übersetzung von *Basic Blocks*

### Definition: Basic Block

- einziger Ein- und Ausgangspunkt
- enthaltene Instruktionen der Reihe nach ausgeführt



# Partitionierung des Codes

## Finden von Blockgrenzen

**Blockende** durch folgende Instruktionen erreicht:

- Unbedingte Sprünge & Funktionsaufrufe (`j`, `call`, `ret`)
- Bedingte Sprünge (`beq`, `bne`, `blt`, `bge`, `bltu`, `bgeu`)
- System Calls (`ecall`)

## Optimierungspotenzial:

- Sprünge folgen
- rekursive Übersetzung von Sprungzielen
- Schwierigkeiten bei bedingten Sprüngen

# Partitionierung des Codes

## Beispiel

**Sprungverfolgung** zu `label`,  
**Blockende** durch `ecall`.

```
add x6, x6, x7
slli x6, x6, 3
xori x7, x7, -1
j label
```

```
label:
addi a0, x0, 0
addi a7, x0, __NR_exit
ecall
```

# Codegenerierung

## Grundlagen

**Ziel:** Generieren von äquivalentem Code

**Prinzipieller Ansatz:** Instruktions-Mapping x86-64  $\implies$  RISC-V

- Übersetzungen jeder Instruktion der Quellarchitektur
- Probleme durch architektonische Unterschiede
  - ☐ *load-store- vs. register-memory-Architektur*
  - ☐ *Zwei- bzw. Dreiadressform*
- Mustererkennung im Eingangscod

# Codegenerierung

## Beispiel: Architektonische Unterschiede

**Problem:** ein Operand ist implizites Zielregister (x86)

`sub rd, rs1, rs2`

$\Rightarrow$

`mov rd, rs1`

`sub rd, rs2`

# Codegenerierung

## Beispiel: Optimierte Übersetzung

**Optimierungsmöglichkeit:** äquivalente native Instruktion existiert

`xori rd, rd, -1`



`not rd`

# Codegenerierung

## Beispiel: Macro Operation Fusion

**Optimierungsmöglichkeit:** mehrere Instruktionen bündeln

```
lui rd, imm1  
addi rd, rd, imm2
```

$\Rightarrow$

```
mov rd, (imm1 + imm2)
```

# Code Cache

## Konzept

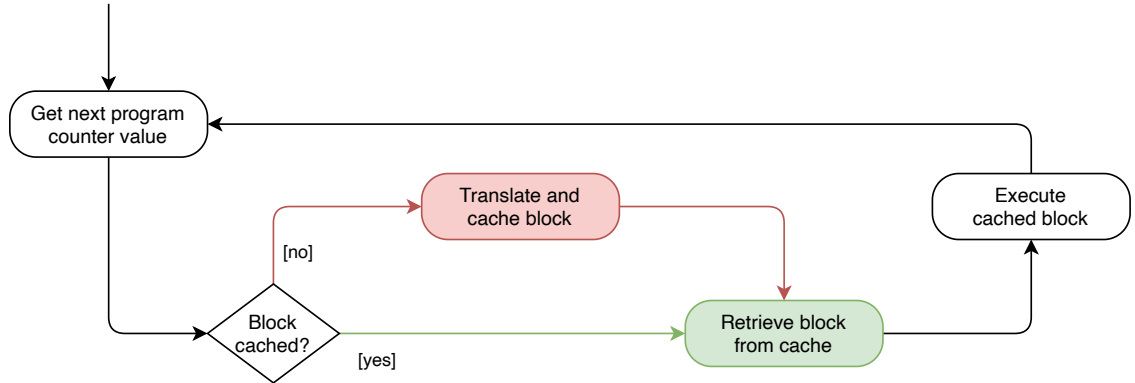
**Hintergrund:** Angetroffene Basic Blocks sollen nur ein Mal übersetzt werden.

### Code Cache

- Speicherregion, in die generierter Code geschrieben wird
- Index für die Speicherregion für schnellen Lookup (→ Hash-Tabelle, TLB)

## Nutzung:

- Block wird nach erstem Übersetzen in den Cache geschrieben
- Lookup vollzieht Adressübersetzung RISC-V → x86
- kein Löschen von übersetzten Blöcken (→ Optimierungen)





# Registernutzung

## Grundlagen

**Ziel:** möglichst effizientes Emulieren der RISC-V-Register

### Definition: Registerdatei im Speicher

- Speicherbereich, der die Registerwerte des Gastprogramms hält (264 Byte)
- Permanenter Speicherbereich, der über Kontextwechsel erhalten bleibt

**Problem:** viele Speicherzugriffe  $\implies$  **ineffizient**

# Registernutzung

## Ansatz

**Idee:** Werte in Registern halten

### Register bei RISC-V und x86-64

#### ■ RISC-V

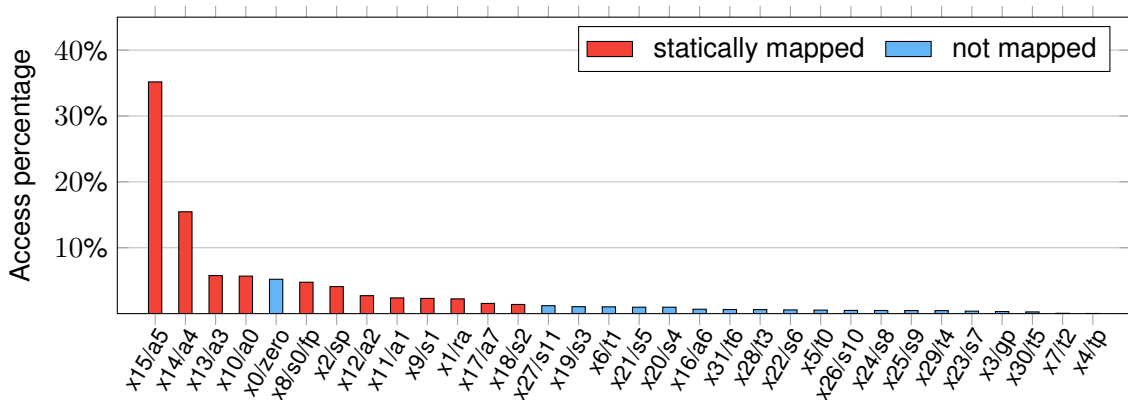
- ☐ 32 general-purpose Register
- ☐ x0, x1–x31
- ☐ festes Nullregister

#### ■ x86-64

- ☐ 16 general-purpose Register
- ☐ rax–rdx, rsp, rbp, rsi, rdi, und r8–r15

- zu wenige Register  $\implies$  statische Abbildung nur teilweise möglich
- rax, rcx, rdx speziell benötigt; rsp unpraktisch

**Überlegung:** Welche 12 Register werden häufig verwendet?



# Registernutzung

## Vorgehen

**Idee:** Speicherzugriffe minimieren

### Register-Handling-Strategie

- statische Abbildung der 12 zugriffshäufigsten Register (außer x0)
  - ☐ a0–a5, a7, s1–s2, ra, fp, sp
  - ☐ bleiben über Blockgrenzen hinweg erhalten (→ Kontextwechsel)
- dynamische Allokation in die restlichen 3 x86-Register
  - ☐ dynamisch in rax, rcx, rdx (*least recently used, lazy write-back*)
  - ☐ an Blockgrenzen zurückgeschrieben

**Ziel:** Überprüfung der Korrektheit

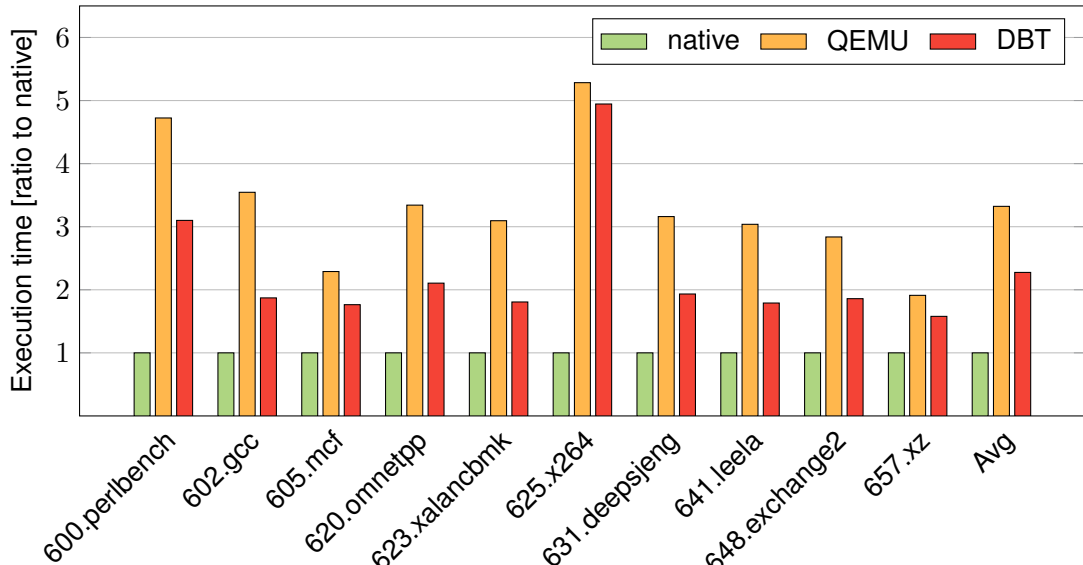
**Ansätze:**

- Extensive Unittests (durch Parametrisierung über 30.000 Test cases)
- Ausführen einfacher bzw. komplizierterer Programme und Überprüfen des Outputs

- Kommerzielle Benchmark Suite mit einer großen Vielfalt an verschiedenen Workloads.
- Realitätsnahe Gestaltung

SPECspeed Benchmark	Workload
600.perlbench	Perl interpreter
602.gcc	GNU C compiler
605.mcf	Route planning
620.omnetpp	Discrete Event simulation – computer network
623.xalancbmk	XML to HTML conversion via XSLT
625.x264	Video compression
631.deepsjeng	Artificial Intelligence: alpha-beta tree search (Chess)
641.leela	Artificial Intelligence: Monte Carlo tree search (Go)
648.exchange2	Artificial Intelligence: recursive solution generator (Sudoku)
657.xz	General data compression

# SPEC CPU 2017 intspeed Results



`./translator`