

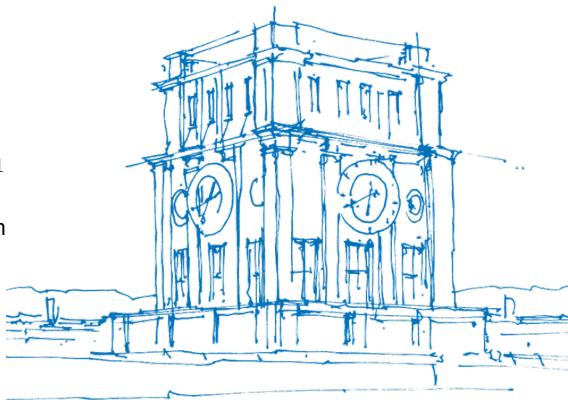
# Dynamische Binärübersetzung: RISC-V $\rightarrow$ x86-64

## 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)

26. Oktober 2020



TUM Uhrenturm

# 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

# 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 Eingangscodem

# 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)

Flowchart goes here. . .

```
./translator
```