Lehrstuhl für Rechnerarchitektur und parallele Systeme
Fakultät für Informatik
Technische Universität München

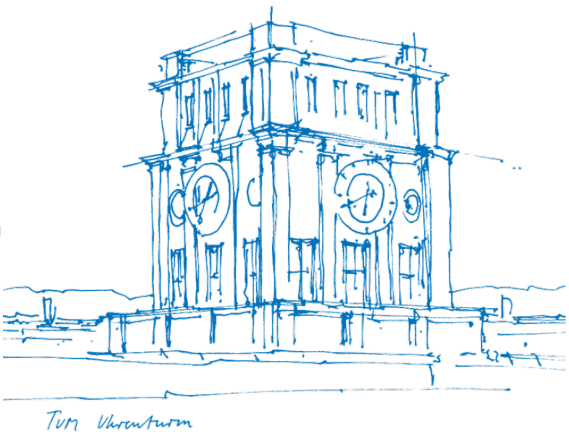
ria-jit: Dynamische Binärübersetzung RISC–V → x86–64

Projektvorstellung

Noah Dormann[1], Simon Kammermeier[1], Johannes Pfannschmidt[1], Florian Schmidt[1]

[1]Fakultät für Informatik, Technische Universität München (TUM)

2. Februar 2021



Gliederung

1. Motivation
2. Technischer Überblick
3. Demo
4. Kontakt

Warum Großpraktikum?

Motivation

Warum Großpraktikum?

Arbeitsaufwand bedeutend höher als beim ERA-P.

Motivation

Warum Großpraktikum?

Arbeitsaufwand bedeutend höher als beim ERA-P. **Aber:**

Lessons learned

- tiefgreifendes Einarbeiten in ein komplexes Thema
- langfristige Arbeit an einem großen Softwareprojekt
- viel Erfahrung mit Git ($\sim$ 1000 commits)
- Aufrechterhalten einer großen Codebase ($\sim$ 19.000 LOC)
- Benchmarking und Testing mit professionellen Tools/Hardware

Motivation

Warum Großpraktikum?

Zudem:

Soft Skills

- Teambuilding und kollaborative Aufgabenverteilung
- Projektmanagement und langfristige Zeitplanung
- Wissenschaftliches Schreiben

Motivation

Warum Großpraktikum?

Sehr angenehm:

Vorteile im Großpraktikum

- **bedeutend freiere** Aufgabenstellung
- persönliche **Betreuung** und Feedback am Lehrstuhl
- *Bragging rights*

Dynamische Binärübersetzung RISC–V $\rightarrow$ x86–64

Technischer Überblick

Dynamische Binärübersetzung RISC–V $\rightarrow$ x86–64

x86–64

- 64-Bit Version der x86-ISA

RISC–V

- Neu entwickelte offene RISC-ISA
- 32- und 64-Bit-Varianten

Technischer Überblick

Dynamische Binärübersetzung RISC–V $\rightarrow$ x86–64

Binärübersetzung

- Keine weit verbreiteten RISC–V-Prozessoren
- Für RISC–V kompilierte/geschriebene Programme können nicht getestet werden
- Weitere Gründe: Verwendung von legacy software, ...
- Binärübersetzung ermöglicht es RISC–V-Programme auf x86–64 auszuführen
- Direkte Übersetzung in x86–64-Instruktionen

Technischer Überblick

Dynamische Binärübersetzung RISC–V $\rightarrow$ x86–64

Dynamische Binärübersetzung

- Problem: Wie in der Datei Befehle von Daten unterscheiden?
- Problem: Wie Sprungziele indirekter Sprünge bestimmen?
- Lösung: Befehle erst übersetzen, wenn benötigt.
- Außerdem: Vermeidung der Übersetzung von nicht ausgeführtem Code

Blockweises Übersetzen

Mit Ausnahme von Sprüngen sequentieller Ablauf $\Longrightarrow$ Übersetzen bis nächsten Sprung:

1. Instruktionen des Blocks parsen
2. Jeweils äquivalente Instruktion(en) in x86–64 erzeugen.
   Architekturunterschiede beachten:
   - unterschiedliche Registeranzahl
   - separates Null-Register
   - 3-Operanden-Adressform
3. Block in code cache speichern, um erneutes Übersetzen zu vermeiden
4. Neuen code anspringen

Technischer Überblick

Dynamische Binärübersetzung RISC–V $\rightarrow$ x86–64

Zusätzliche Aufgaben

- Verwalten der Laufzeitumgebung des übersetzten Programms (Register, Stack, System calls)
- Verwalten der Laufzeitumgebung des Übersetzers (Code cache, cache index)

Optimierungen

- Sprungziele oft vorher bekannt $\Longrightarrow$ Blöcke direkt aneinanderhängen
- Bessere Instruktionsabbildung

Demo

But can it run Crysis?

```
./translator
```

Kontakt

Repository: https://github.com/ria-jit/ria-jit

- Noah Dormann
  n.dormann@tum.de
- Simon Kammermeier
  simon.kammermeier@tum.de
- Johannes Pfannschmidt
  johannes.pfannschmidt@tum.de
- Florian Schmidt
  fs.schmidt@tum.de