

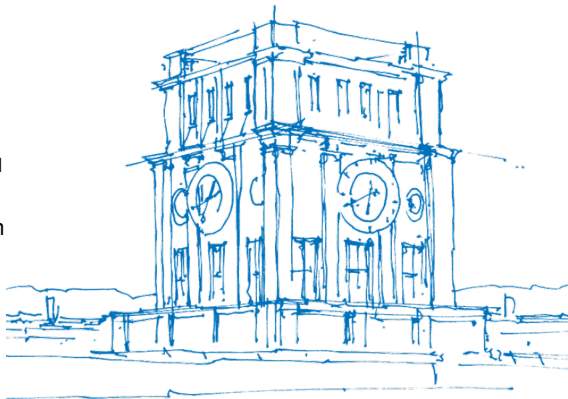
Dynamische Binärübersetzung: RISC-V \rightarrow x86-64

Endpräsentation

Noah Dormann¹, Simon Kammermeier¹,
Johannes Pfannschmidt¹, Florian Schmidt¹

¹Fakultät für Informatik, Technische Universität München
(TUM)

28. 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:

- RISC-V Prozessoren sind noch nicht weit verbreitet.
- Entwickler, die Code für RISC-V als Zielplattform kompilieren wollen, können diesen nicht nativ 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

Instruction Set Emulation

Interpretation

Das Assembly wird konsekutiv abgearbeitet und jeder Befehl wird einzeln übersetzt.

Vorteile

- Einfach zu implementieren, portable
- Keine Erzeugung von JIT Assembler nötig

Nachteile

- Geringe Performance
- Wenig Optimierungspotential (e.g. threaded interpretation)

Instruction Set Emulation

Dynamische Binärübersetzung

Das Assembly wird schrittweise in die Ziel Architektur übersetzt und dann ausgeführt.

Vorteile

- Einmaliger Übersetzungsaufwand
- Höhere Performanz, Optimierung des nativen Codes

Nachteile

- Unterstützt keinen selbstverändernden Code
- Eingeschränkt auf ein Quell- und Zielplattformpaar

Instruction Set Emulation

Statische Binärübersetzung

Übersetzen der gesamten Quellassembly in die Zielarchitektur. Ausführen des übersetzten Codes.

Vorteile

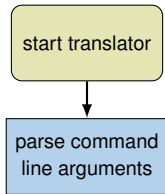
- Einmalige Übersetzung des gesamten Codes, keine Übersetzung zur Laufzeit
- Hohe Performanz

Nachteile

- Code Discovery Problem
- Code Location Problem

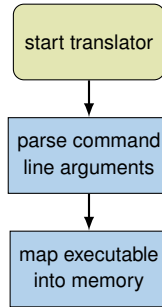
Programmablauf

Initialisierung



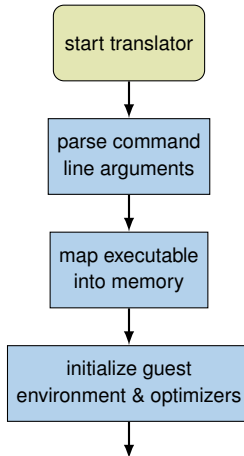
Programmablauf

Initialisierung



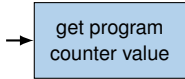
Programmablauf

Initialisierung



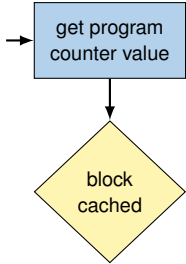
Programmablauf

Transcode loop



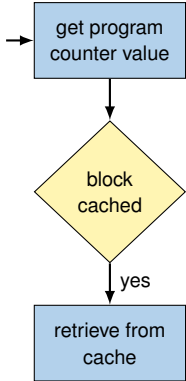
Programmablauf

Transcode loop



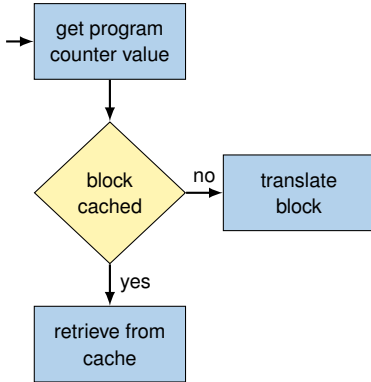
Programmablauf

Transcode loop



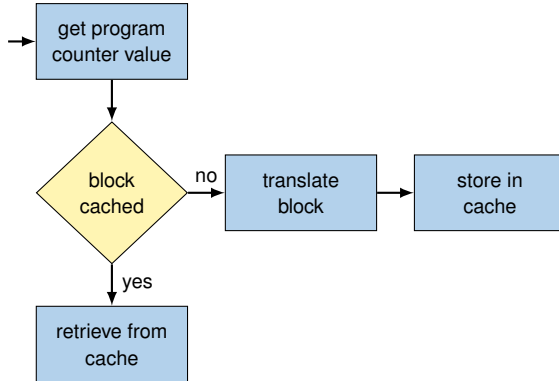
Programmablauf

Transcode loop



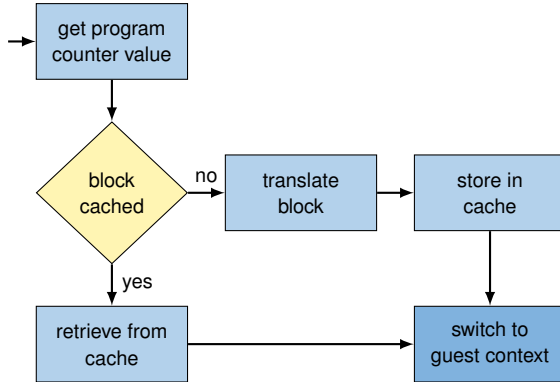
Programmablauf

Transcode loop



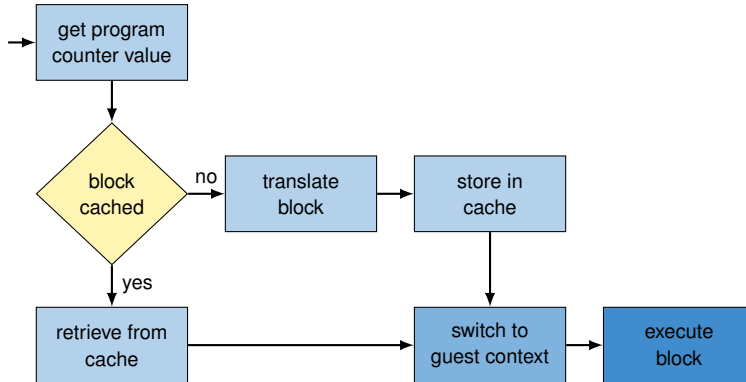
Programmablauf

Transcode loop



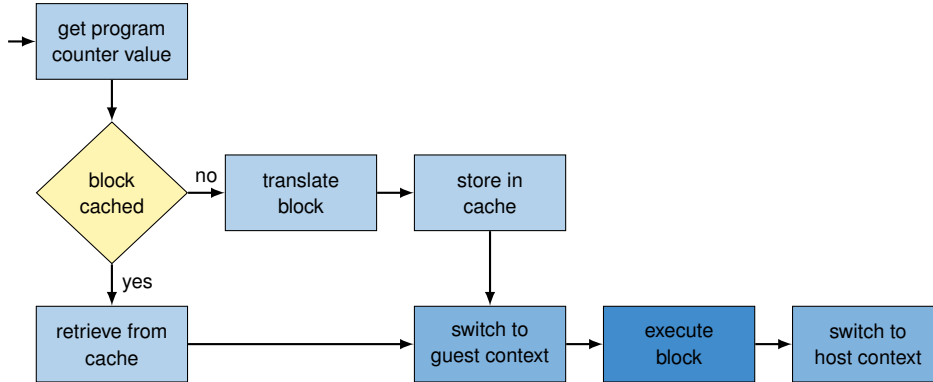
Programmablauf

Transcode loop



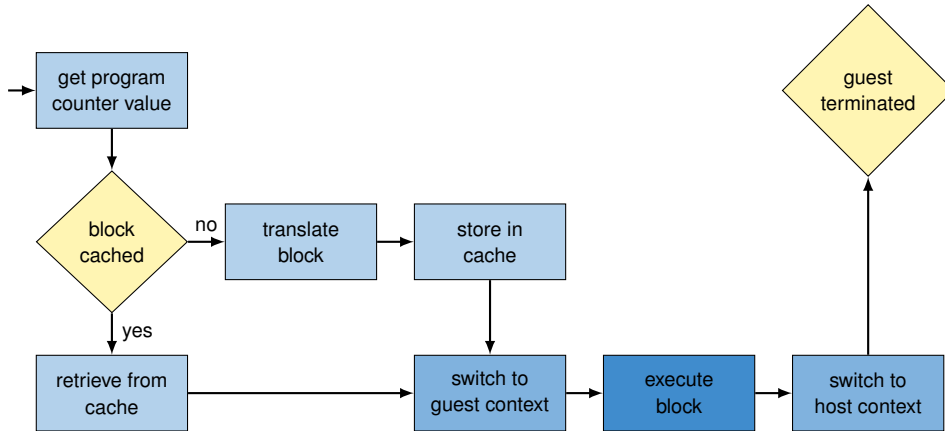
Programmablauf

Transcode loop



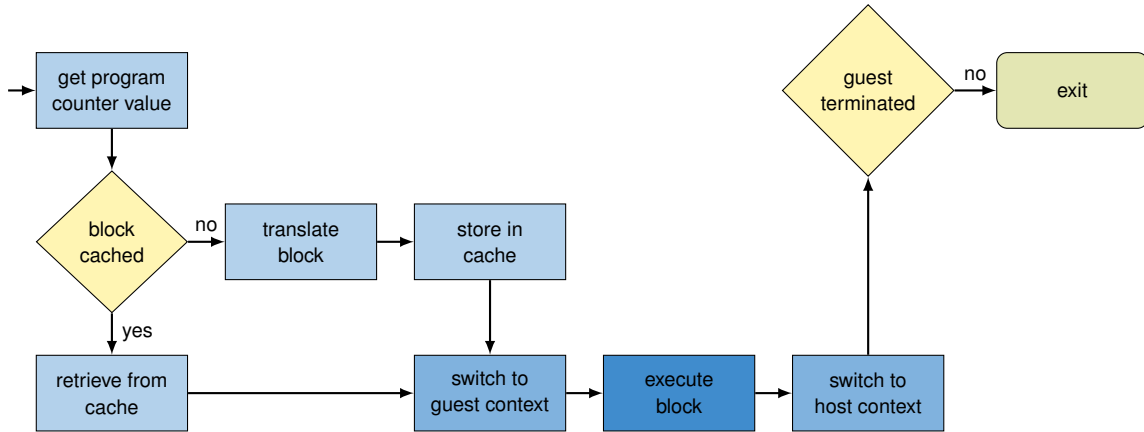
Programmablauf

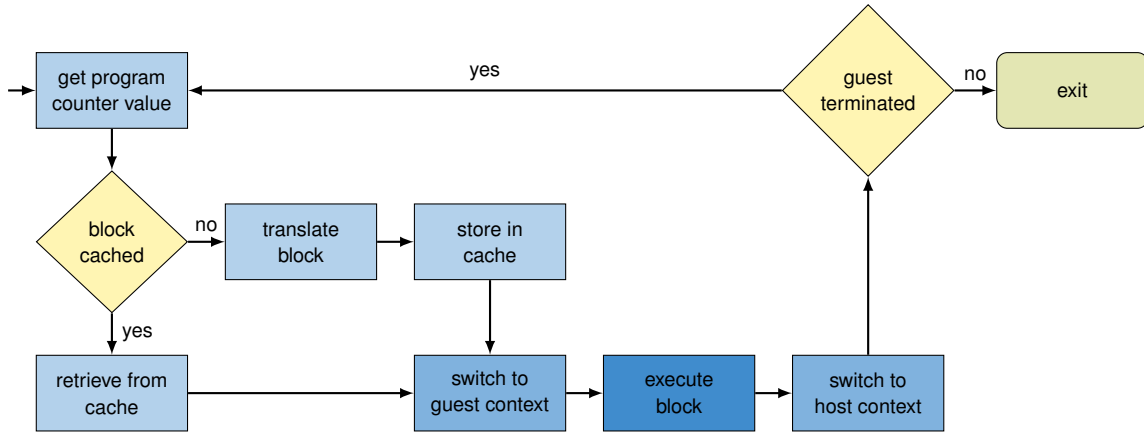
Transcode loop



Programmablauf

Transcode loop





Partitionierung des Codes

Grundlagen

Ziel: Finden von sinnvollen Übersetzungseinheiten

Partitionierung des Codes

Grundlagen

Ziel: Finden von sinnvollen Übersetzungseinheiten

Überlegung:

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

⇒ Übersetzung von *Basic Blocks*

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 erreicht durch...

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

Einheiten zusammen übersetzt, als **Blöcke** abgelegt.

Codegenerierung

Grundlagen



Ziel: Generieren von äquivalentem Code

Codegenerierung

Grundlagen

Ziel: Generieren von äquivalentem Code

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

Übersetzungsansatz

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

Codegenerierung

Beispiel: Architektonische Unterschiede

Problem: ein Operand ist implizites Zielregister (x86)

```
sub rd, rs1, rs2
```

\Rightarrow

```
mov rd, rs1  
sub rd, rs2
```

Codegenerierung

Beispiel: Macro Operation Fusion

Optimierungsmöglichkeit: mehrere Instruktionen bündeln

<code>lui rd, imm1</code>	\Rightarrow	<code>mov rd, (imm1 + imm2)</code>
<code>addi rd, rd, imm2</code>		

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

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)

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

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

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 \Rightarrow **ineffizient**

Registernutzung

Ansatz

Idee: Werte in Registern halten

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

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

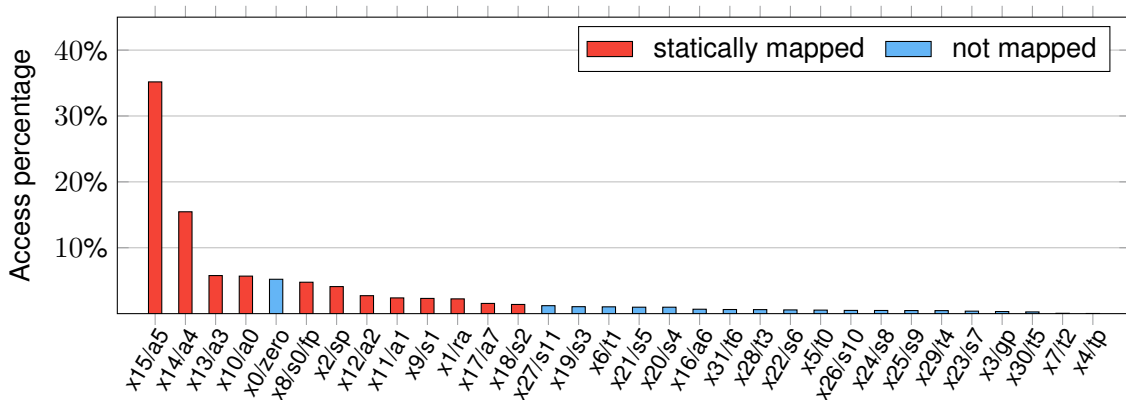
■ rax, rcx, rdx speziell benötigt; rsp unpraktisch

Registernutzung

Ansatz

Überlegung: Welche 12 Register werden häufig verwendet?

Überlegung: Welche 12 Register werden häufig verwendet?



Registernutzung

Vorgehen

Idee: Speicherzugriffe minimieren

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)

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

Optimierungen

Performance-Einbußen bei Dynamischer Binärübersetzung

- Blockweise Übersetzung und Ausführung -> ständiger Wechsel zwischen guest und host
 - viele Kontextwechsel
 - viele Cache-Lookups
- ISA-Unterschiede
 - Mapping einzelner instruktionen ineffizient

Optimierungsstrategie

- Wechsel zwischen guest und host vermeiden
 - mehrere Blöcke direkt hintereinander ausführen
- Besseres Instruction-Mapping
 - mehrere Instruktionen betrachten

Ziel: Überprüfung der Korrektheit

Ansätze:

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

Benchmarks

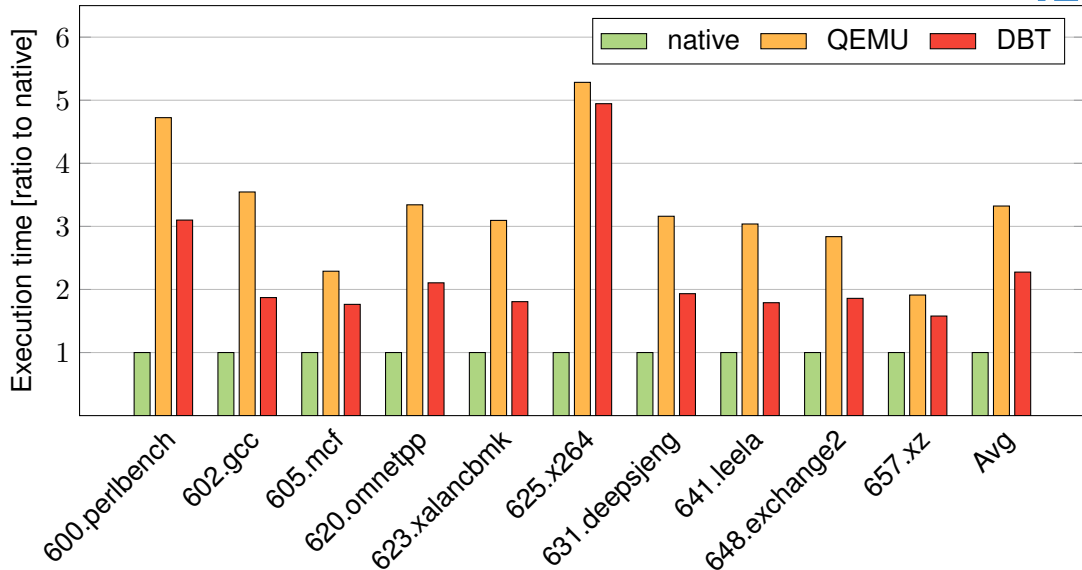
SPEC CPU 2017

- Industriell genutzte Benchmark Suite mit einer großen Vielfalt an verschiedenen Workloads
- Realitätsnahe Gestaltung

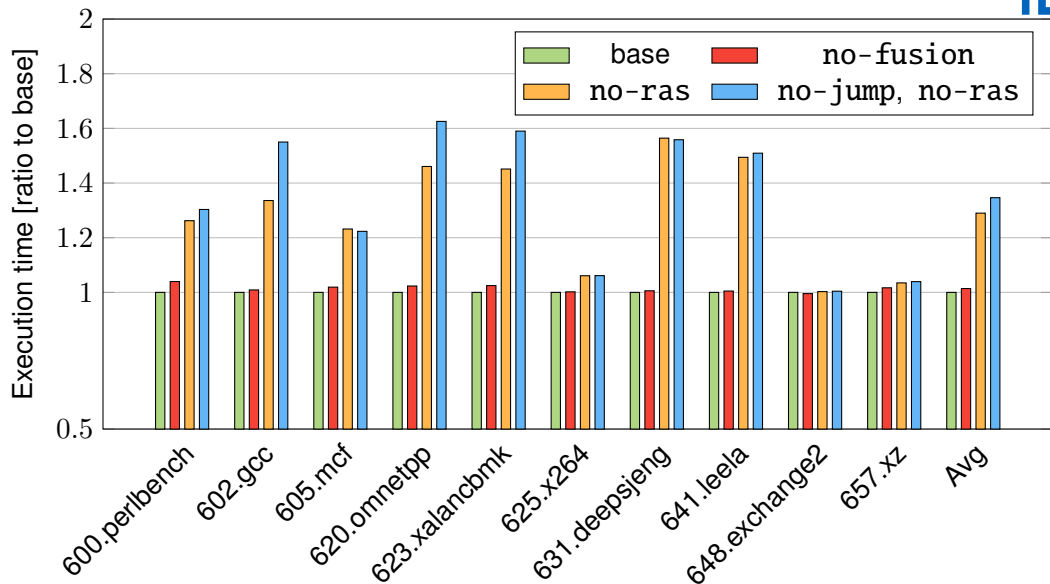
- Industriell genutzte Benchmark Suite mit einer großen Vielfalt an verschiedenen Workloads
- Realitätsnahe Gestaltung

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 intspeer Results



Optimisation option evaluation



Benchmark	no-fusion	no-ras	no-jump, no-ras	none
600.perlbench	1.04	1.26	1.30	7.32
602.gcc	1.01	1.34	1.55	7.89
605.mcf	1.02	1.23	1.22	3.67
620.omnetpp	1.02	1.46	1.63	5.22
623.xalancbmk	1.02	1.45	1.59	7.66
625.x264	1.00	1.06	1.06	2.46
631.deepsjeng	1.01	1.56	1.56	7.40
641.leela	1.00	1.49	1.51	5.54
648.exchange2	1.00	1.00	1.00	8.71
657.xz	1.02	1.03	1.04	4.09
Avg	1.01	1.29	1.35	6.00

Vergleich

Vorteile im Vergleich zu **QEMU**:

- Vermeidung einer Zwischendarstellung
- Zusätzliche Optimierungen
- Statisches Registermapping

Vergleich

Vorteile im Vergleich zu **QEMU**:

- Vermeidung einer Zwischendarstellung
- Zusätzliche Optimierungen
- Statisches Registermapping

Nachteile gegenüber **native**:

- Architektonische Unterschiede
- Overhead durch Parsen, Instruktionsgenerierung, etc.

`./translator`