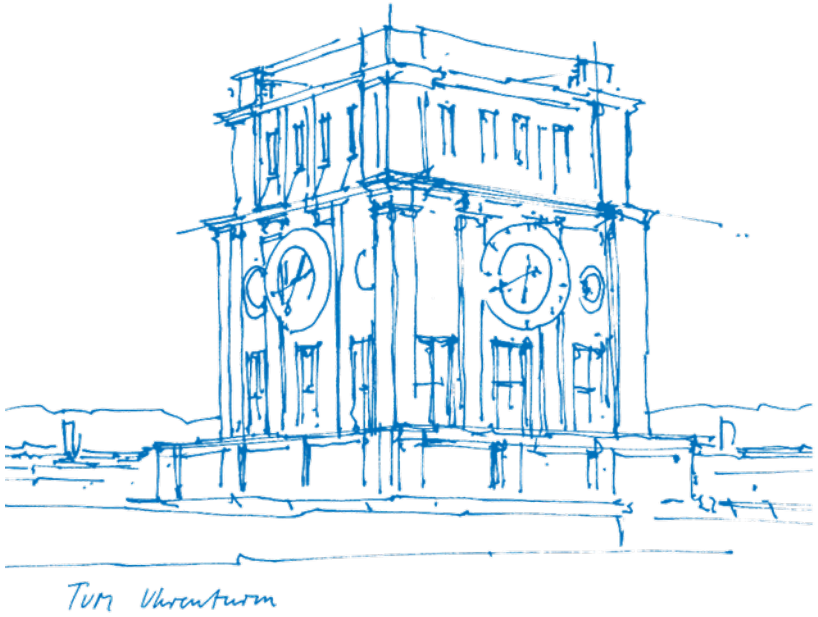
Statische Binärübersetzung von RISC-V in x86-64

Lukas Döllerer, Jonathan Hettwer, Johannes Maier, Tobias Schwarz, Felix Solcher

Rechnerarchitektur-Großpraktikum 2021
Garching, 16. Juli 2021



Inhalt

Dynamische und Statische Binärübersetzung

Gemeinsames Ziel: Ausführung von Binärdateien einer Architektur auf einer anderen Architektur.

Dynamische Übersetzung

- Instruktionen werden zur Laufzeit übersetzt
- Übersetzte Instruktionen können zwischengespeichert werden

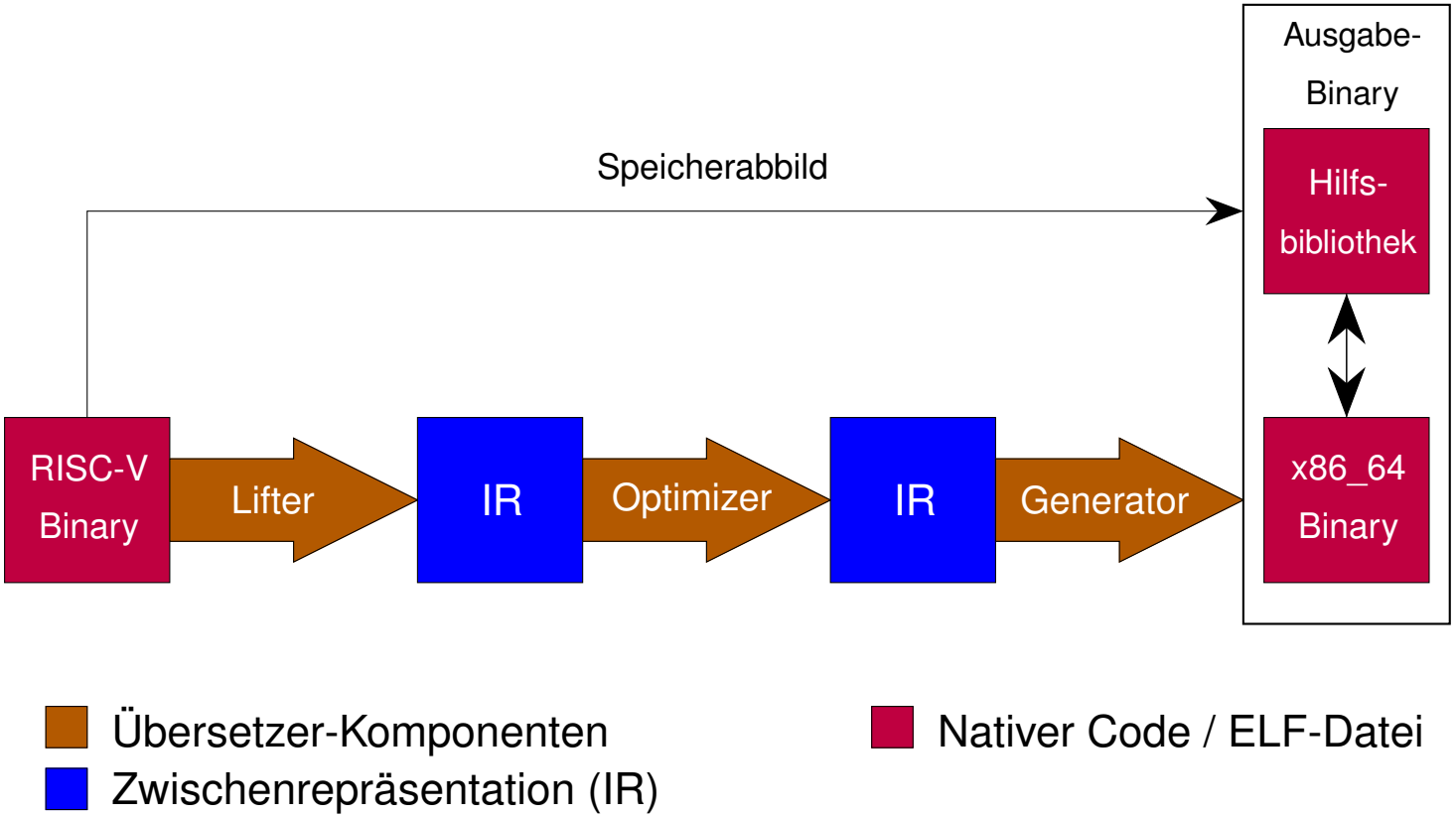
Statische Übersetzung

- Binärdatei wird erst zur Zielarchitektur übersetzt
- Resultat kann unabhängig von Übersetzer ausgeführt werden

RISC-V Übersicht

- Relativ neuer Befehlssatz
- Aufgeteilt in Base und Extensions
- Feste Befehlslänge von 32-Bit (optional auch 16-Bit)
- Reduced Instruction Set Computer $\rightarrow$ Wenige, einfache Befehle
- Load-Store-Architektur

Programmübersicht



Intermediate Representation

Aufbau

- Programm aus **Basic Blocks**
- Basic Blocks enthalten Variablen
- Eingaben in Form von Statics
- Verbindung durch Kontrollflussoperationen
- **Static Single Assignment (SSA)** Form

Intermediate Representation

Operationen

Instruktionen:

- Speicher: `store`, `load`
- Arithmetisch: `add`, `sub`, `mul`, ...
- Logisch: `and`, `or`, `shl`, ...
- Sonstige: Typkonvertierung, ...
- Gleitkommaarithmetik

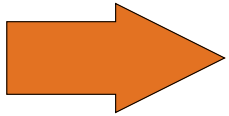
Kontrollflussoperationen:

- Sprünge: `jump`, `ijump`, `cjump`
- Aufrufe: `call`, `icall`, `return`
- `unreachable`
- `syscall`

Intermediate Representation

Beispiel

```
[...]
addi a0, x0, 100
j somewhere_else
[...]
```



```
block b1(/* inputs */) {
    imm v1 <- immediate 0
    imm v2 <- immediate 100
    i64 v3 <- add imm v1, imm v2
} => [(jump, [b2, i64 v3])]
```

Constant Folding

Beispiel

```
block b1(/* inputs */) {
    imm v1 <- immediate 0
    imm v2 <- immediate 100
    i64 v3 <- add imm v1, imm v2
} => [(jump, [b2, i64 v3])]
```

```
block b1(/* inputs */) {
    i64 v3 <- immediate 100
} => [(jump, [b2, i64 v3])]
```

Constant Folding

- Berechnung von Operationen, deren Eingaben bekannt sind (also Zahlen)
- Propagieren von bekannten und berechneten Werten
- Vereinfachung einiger Operationen

Dead Code Elimination

Beispiel

```
block b1(/* inputs */) {
    i64 v3 <- immediate 100
} => [(jump, [b2, i64 v3])]

block b2(i64 v1) <= [b1] {
} => [(jump, [b3])]
```

Dead Code Elimination

Löschen von nicht benötigten Variablen und Entfernung von unbenutzten Parametern

- Start bei Operationen mit Nebeneffekten, wie stores oder indirekte Jumps
- “Graph-Suche” nach allen erreichbaren Variablen
- Nicht erreichte Variablen sind überflüssig.

Common Subexpression Elimination

Beispiel

```
block b1(i64 v1) {
    imm v2 <- immediate 100
    i64 v3 <- add i64 v1, imm v2
    imm v4 <- immediate 100
    i64 v5 <- add i64 v1, imm v4
} => [(jump, [b2, i64 v5])]
```

```
block b1(i64 v1) {
    imm v2 <- immediate 100
    i64 v3 <- add i64 v1, imm v2
} => [(jump, [b2, i64 v3])]
```

Common Subexpression Elimination

Löschen von redundanten Operationen und doppelten Immediates

- Variablen von oben nach unten überprüfen
- Falls andere Variable mit gleichem Typ und gleichen Inputs existiert, ersetzen

Lifter

ELF Binärdatei laden und Instruktionsbytes decodieren

1. ELF File prüfen.
2. Program Header, Sections und Symbole auslesen.
   - 2.1 Instruktionen aus *ausführbaren* Sektionen mit **frvdec** decodieren.
   - 2.2 Falls keine Sektionen verfügbar ist, werden Program Header verwendet
   - 2.3 Daten aus *lesbaren* Sektionen die **nicht** *ausführbar* sind werden byteweise als reine Daten geladen.

Lifter

RISC-V Instruktionen sequentiell in IR Code umwandeln

Wiederhole solange ungeliftete Instruktionen existieren:

1. Starte neuen Basic Block bei der nächsten, ungelifteten Instruktion
2. geladene Instruktionen in IR parsen
3. Wiederhole solange bis:
   - 3.1 eine **Kontrollfluss ändernde Instruktion** auftritt
   - 3.2 der Start eines neues Basic Blocks an der nächsten Instruktion registriert ist
4. Beende Basic Block $\rightarrow$ markiere Start neuer Basic Blöcke an evtl. Sprungzielen

Lifter

Aufteilen eines Basic Blocks

- Situation: Sprung **in** einen bereits eingelesenen Basic Block
- Aufteilung an Sprungadresse in zwei Basic Blöcke
- Verbindung mit einem direkten Sprung
- Komplexe Operation, da alle Referenzen auf den geteilten Block betrachtet werden müssen

Lifter Subroutinen Erkennung

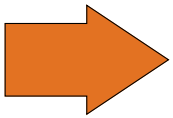
Erkennung der Assembler Pseudoinstruktionen `call` und `ret`.

- RISC-V Subroutinen Aufrufe funktionieren anders als bei x86_64
- Standard Calling Convention definiert `x1` und `x5` als *Linkregister*
- `JAL` := *Jump and Link*
- `JALR` := *Jump and Link Register*
- `JAL (x1 | x5), ..., ...` ≡ `call`
- `JALR x0, x1, 0` ≡ `ret`
- Laufzeitüberprüfung der Rücksprungziele durch Returnadressen-Stack

Lifter Sprungtabellenerkennung

Switch-Case Kontrollstrukturen werden in Sprungtabellen umgewandelt.

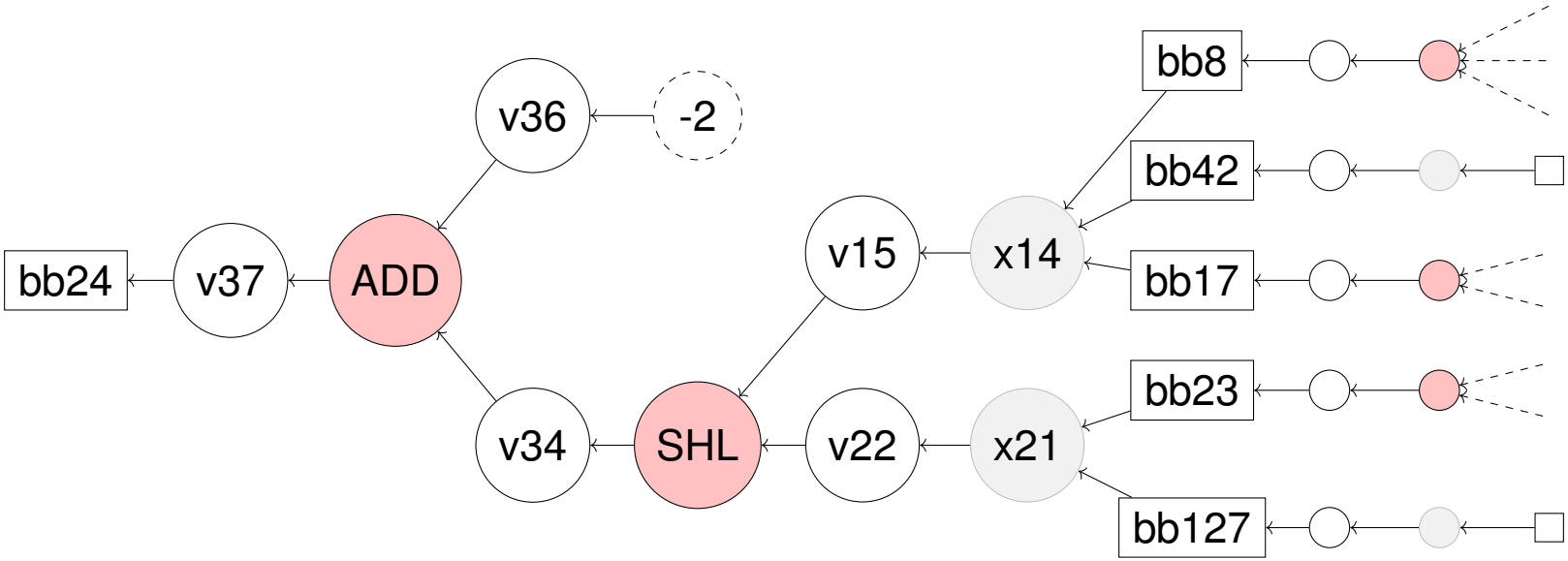
```
switch(instr.mnem) {
    case 0:
        fun0();
        break;
    case 1:
        fun3();
        break;
    default:
        fun4();
        break;
}
```



```
/* [...] */
    bgtu a4, 1, .L4
    slli a4, a5, 2
    lui a5, %hi(.L1)
    addi a5, a5, %lo(.L1)
    add a5, a4, a5
    lw a5, 0(a5)
    jr a5
/* [...] */
.L1:
    .word .L2
    .word .L3
```

Lifter Backtracking

Rücklverfolgung einer indirekten Sprungadresse
= rekursive Rückverfolgung der absoluten Werte aller Operationsparameter



Code Generator

Übersetzung der IR zu x86_64

- Fertige Binary enthält Assembly für Basic Blocks, originale Binary, Stack und Runtime-Helper
- Zwei Implementierungen: Simple und Advanced
- Simple Codegen optimiert wenig, dient Debugging/Development-Zwecken
- Advanced Codegen allocated Register für die Operationsergebnisse, optional Merging von Operationen

Beispiel

IR-Operation, für die Assembly generiert werden soll:

```
i64 v6 <- add i64 v5, imm v1
```

Wenn rbx freies Register ist:

```
lea rbx, [rax + 10]
```

Wenn rbx belegt und vor Variable in rax wieder benutzt:

```
mov [rsp], rax
add rax, 10
```

IJump Lookup

Datenstruktur für effizienten Zugriff auf Basic Block Startadressen

Lookup Table

- Speichere für jede mögliche Jump-Addresse die korrespondierende Basic Block Startadresse
- Wenn diese nicht existiert, füge stattdessen 0 ein
- Liste dieser Wert baut Adressraum der originalen Binary nach

Hashing

- Ordne Startadressen in Hashtabelle ein
- Zweistufiges Hashing durch Einsortieren in Buckets
- Langsamer als Lookup Table, aber besserer Platzverbrauch

Helper-Library

- Architektur-spezifische Funktionen, die für die übersetzte Binary erforderlich sind
- Wird zusammen mit dem Assembly des Generators gelinked
- Stack-Intialisierung
- Syscall-Translation
- Interpreter

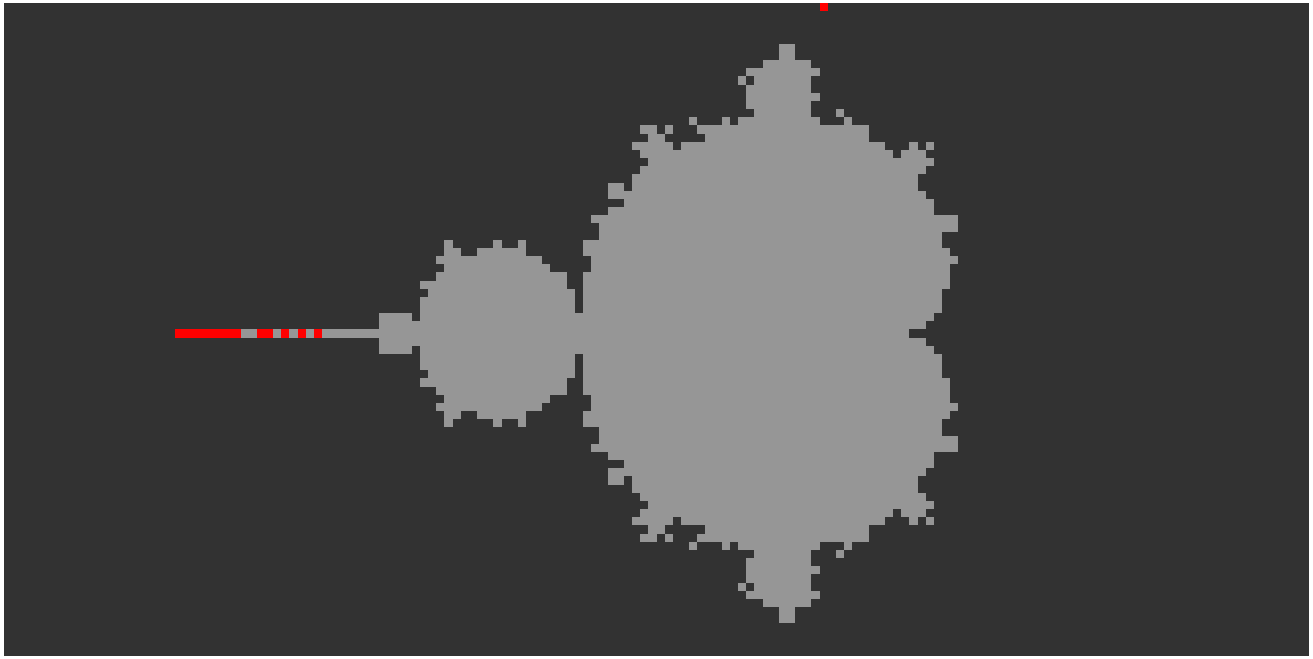
Interpreter

- Dient zur Auflösung indirekter Sprünge ohne erkanntem Sprungziel
- Interpreteriert die RISC-V Instruktionen an gegebener Addresse
- Springt zurück zu übersetzem Basic Block, wenn möglich -> Ijump Lookup Table
- Kann theoretisch mit JIT Compileren und Selbstmodifizierender Code umgehen

Gleitkommaarithmetik

- Unterstützung für F und D Standard Extensions
- Übersetzung zu SSE und SSE2 Instruktionen
- Ziel: Unterstützung der Gleitkommaarithmetik, aber nicht optimiert
- Aber: Unterschiede in Architekturen führen zu leicht anderen Ergebnissen
- Beachtung des Rundungsmodus bei jeder Instruktion: zu langsam -> nur bei Konvertierung:
  - Veränderung des **MXCSR** Registers, oder
  - Nuzung von **roundss**/**rounsd**
- Nutzung von FMA3 Instruktionen erhöht die Genauigkeit

Rundungsprobleme



Übersetzungprobleme bei der Gleitkommaarithmetik

- Einige Instruktionen können nicht direkt auf x86_64 abgebildet werden
- Vorzeichenlose Ganzzahlkonvertierung: nur von AVX512 nativ unterstützt
- Für **Sign Injection** und **Classify** gibt es gar keine Äquivalente
- Diese Instruktion müssen mittels Bitarithmetik berechnet werden

Benchmarks

- SPECSpeed®2017 Integer Benchmark Suite
- Vergleich zu QEMU, RIA-JIT
- Korrektheit mittels Tests

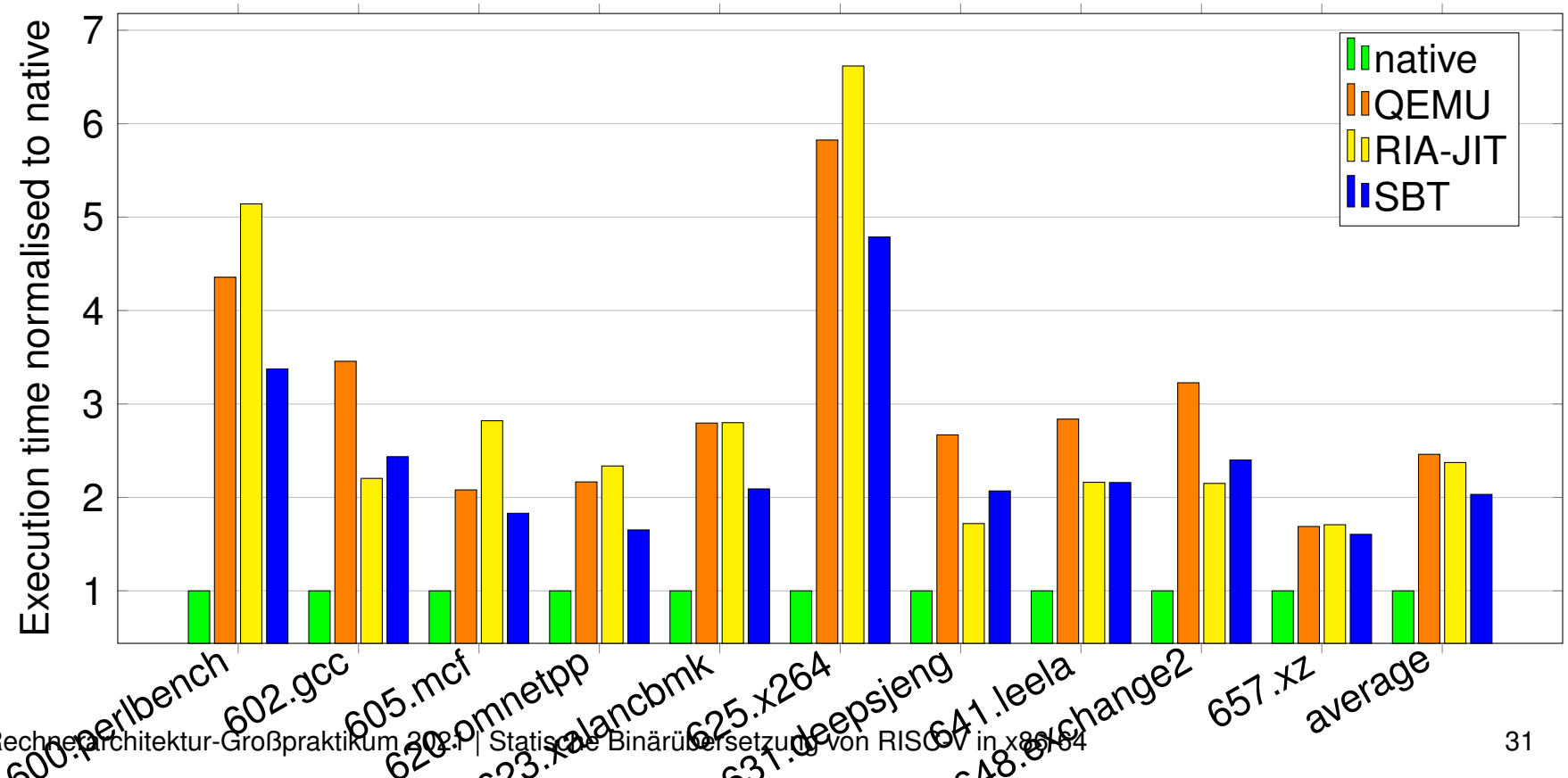
Benchmarks

Probleme

- Hashing zu langsam
- Translation Blöcke in manchen Fällen wichtig

Benchmarks

Ergebnisse



Erreichte Ziele

Erreichte Hauptziele und Ausblick

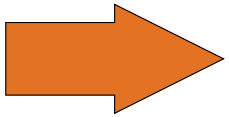
- Korrektes Übersetzen einfacher freistehender Programme
- Übersetzen von unmodifizierter musl libc
- Unterstützung von RV64I, RV64C, . . .
- Auflösung indirekter Sprünge zur Laufzeit

Noch Fragen?

Generator

```
        block b1(/* inputs */) <=
[/* predecessors */] {

        } => []
```



```
b1:
push rbp
mov rbp, rsp
sub rsp, 32

[...]
```

Basic Block Splitting

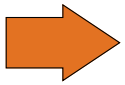
```
    [...]
    addi a1, x0, 100
loop:
    addi a1, a1, -1
    bne a1, x0, loop
    [...]
```

```
block b1(inputs) <= [predecessors] {
    i64 v0 <- @1
    [...] // more statics
    imm v33 <- immediate 0
    imm v34 <- immediate 100
    i64 <- add i64 v33, i64 v33
    [...]
} => [(jump, [b2, ...])]

block b2(inputs) <=
[predecessors] { // loop
    [...] // statics
    imm v33 <- immediate -1
    i64 v34 <- add i64 v11, i64 v33
    [...]
} => [(cjump, [b2, ...]), (jump, [...])]
```

Lifter Basic Blöcke

```
/* [...] */
0x6: add a5, a5, a4
0x8: bltu a5, a4, 10
0xc: lw a5, 0(a4)
0xe: sll a4, a5, a4
/* [...] */
```



```
/* [...] */
i64 v33 <- add v13, v14
imm v34 <- immediate 0x10
imm v35 <- immediate 0xc
} => [/* cjump */]

block b2(/**/) <= [/**/] {
imm v33 <- immediate 0
i32 v34 <- load v13, v33
i64 v35 <- sll v13, v14
/* [...] */
```

Technische Demonstration

Live Demonstration des aktuellen Standes

```
./big_tests/sysroot/bin/riscv64-linux-gnu-gcc \
    -static \
    examples/helloworld3.c \
    -o examples/helloworld3

./build/src/translate \
    --debug=true \
    --output-binary=examples/helloworld3.bin \
    --output=examples/helloworld3_translated.s \
    examples/helloworld3
```

```
gcc \
    -c examples/helloworld3_translated.s
    -o examples/helloworld3_translated.o

ld \
    -T src/generator/x86_64/helper/link.
    examples/helloworld3_translated.o \
    build/src/generator/x86_64/helper/li
    -o examples/helloworld3_translated

./examples/helloworld3_translated
```