

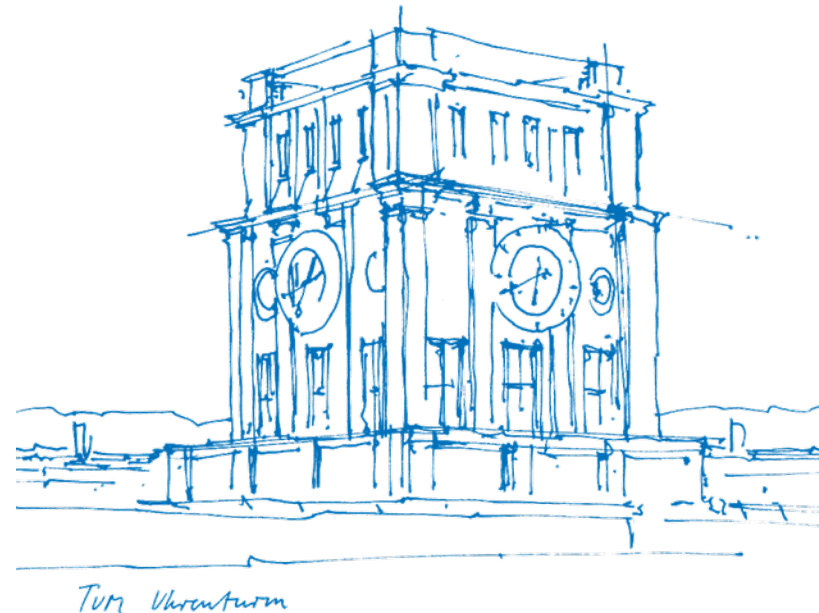
# x86-64 zu RISC-V Binärübersetzer

## Abschlusspräsentation

Knud Haase, Björn Boss Henrichsen, Michael Plainer, Justus Polzin

Lehrstuhl für Rechnerarchitektur & Parallele Systeme,  
Technische Universität München (TUM)

17 Oktober 2019



# Was ist RISC-V?

- Reduced Instruction Set Computer
- Kostenlose Open-Source ISA
- Erste Veröffentlichung 2010
- Will zukunftsicher sein (z.B. durch 128 bit Unterstützung)
- (Fast) alle Instruktionen 32 bit lang

# Was ist RISC-V?

- Reduced Instruction Set Computer
- Kostenlose Open-Source ISA
- Erste Veröffentlichung 2010
- Will zukunftsicher sein (z.B. durch 128 bit Unterstützung)
- (Fast) alle Instruktionen 32 bit lang
- Wenig Hilfe neben der offiziellen Dokumentation
- (Noch) sehr kleine Community

# Problemstellung

- Ziel: Ausführen einer x86-64-Elf-Datei auf RISC-V
- Aufruf auf einem RISC-V-Computer

```
— oxtra gzip_x86_64 -a "<arguments>"
```

# Problemstellung

- Ziel: Ausführen einer x86-64-Elf-Datei auf RISC-V
- Aufruf auf einem RISC-V-Computer
  - `oxtra gzip_x86_64 -a "<arguments>"`
- Nicht portierte closed-source Anwendungen ausführen
  - Legacy Anwendungen
  - Neue ISA; (noch) wenig offizieller Support

# Problemstellung

- Ziel: Ausführen einer x86-64-Elf-Datei auf RISC-V
- Aufruf auf einem RISC-V-Computer
  - `oxtra gzip_x86_64 -a "<arguments>"`
- Nicht portierte closed-source Anwendungen ausführen
  - Legacy Anwendungen
  - Neue ISA; (noch) wenig offizieller Support
- Emulation, Statische Binärübersetzung, **Dynamische Binärübersetzung**

# Problemstellung

- Ziel: Ausführen einer x86-64-Elf-Datei auf RISC-V
- Aufruf auf einem RISC-V-Computer
  - `oxtra gzip_x86_64 -a "<arguments>"`
- Nicht portierte closed-source Anwendungen ausführen
  - Legacy Anwendungen
  - Neue ISA; (noch) wenig offizieller Support
- Emulation, Statische Binärübersetzung, **Dynamische Binärübersetzung**
- Fokus auf essentielle Instruktionen und damit Performance

# x86-64 vs. RISC-V — Instruktionen

- x86-64: CISC → viele sehr versatile Instruktionen
  - `add reg, reg`
  - `add reg, mem`
  - `add reg, imm`



# x86-64 vs. RISC-V — Instruktionen

- x86-64: CISC → viele sehr versatile Instruktionen
  - `add reg, reg`
  - `add reg, mem`
  - `add reg, imm`
- RISC-V: RISC → wenige eingeschränkte Instruktionen
  - `add reg, reg, reg`
  - `addi reg, reg, imm`

# x86-64 vs. RISC-V — Instruktionen

- x86-64: CISC → viele sehr versatile Instruktionen
  - `add reg, reg`
  - `add reg, mem`
  - `add reg, imm`
- RISC-V: RISC → wenige eingeschränkte Instruktionen
  - `add reg, reg, reg`
  - `addi reg, reg, imm`
- Speicherzugriff
  - x86-64: Register-Memory-Architektur
  - RISC-V: Load-Store-Architektur

# x86-64 vs. RISC-V — Register

- x86-64: 16 GPRs
  - rax, rbx, rcx, rdx
  - rsp, rbp, rdi, rsi
  - r8 - r15
- RISC-V: 31 GPRs
  - ra, sp, gp, tp
  - a0 - a7
  - s0 - s11
  - t0 - t6

# Überblick — Übersetzung

x86-64:

```
1 mov rdx, 0xf8abc1f128fccbb1
```

# Überblick — Übersetzung

x86-64:

```
1 mov rdx, 0xf8abc1f128fccbb1
```

RISC-V:

```
1 lui a0,0xff8ac
2 addiw a0,a0,-993
3 slli a0,a0,0xc
4 addi a0,a0,297
5 slli a0,a0,0xc
6 addi a0,a0,-51
7 slli a0,a0,0xc
8 addi a0,a0,-1103
```

# Überblick — Übersetzung

x86-64:

```
1 push  qword ptr [rax+rsi*2+0x1234]
```

# Überblick — Übersetzung

x86-64:

```
1 push qword ptr [rax+rsi*2+0x1234]
```

RISC-V:

```
1 addi sp, sp, -0x8(-8)
2 slli t0, a1, 0x1(1)
3 add t1, t0, a7
4 lui t0, 0x1(1)
5 xori t0, t0, 0x234(564)
6 add t1, t1, t0
7 ld t0, 0(t1)
8 sd t0, 0(sp)
```

# Überblick — Übersetzung

x86-64:

```
1 imul rdx
```



# Überblick — Übersetzung

x86-64:

```
1 imul rdx
```

RISC-V:

```
1 mulh t4, a7, a2
```

```
2 mul a7, a7, a2
```

```
3 addi a2, t4, 0x0(0)
```

# Übersetzungseinheit

- Einzelne Instruktionen ausführen sehr zeitaufwendig
- Lösung: Mehrere Instruktionen bündeln

# Übersetzungseinheit

- Einzelne Instruktionen ausführen sehr zeitaufwendig
- Lösung: Mehrere Instruktionen bündeln

## Basic Block

- (Lange) lineare Folge von Instruktionen
- Kontrollsteuerende Instruktionen markiert Ende (`jmp`, `call ...`)

# Übersetzungseinheit

- Einzelne Instruktionen ausführen sehr zeitaufwendig
- Lösung: Mehrere Instruktionen bündeln

## Basic Block

- (Lange) lineare Folge von Instruktionen
- Kontrollsteuerende Instruktionen markiert Ende (`jmp`, `call ...`)
- Kann interpretiert werden wie **eine** komplexe Instruktion
- Klar definierter Ein- und Ausgang

# Basic Blocks

```
1  add  rax ,  rbx
2  mov  rbx ,  rax
3  cmp  rbx ,  rax
4  jne  fatal
5  mov  rax ,  60
6  xor  rdi ,  rdi
7  syscall
```

# Basic Blocks

```
1  add  rax ,  rbx
2  mov  rbx ,  rax
3  cmp  rbx ,  rax
4  jne  fatal
5  mov  rax ,  60
6  xor  rdi ,  rdi
7  syscall
```

# Basic Blocks

```
1  add  rax , rbx
2  mov  rbx , rax
3  cmp  rbx , rax
4  jne  fatal
5  mov  rax , 60
6  xor  rdi , rdi
7  syscall
```

# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel anfangs unbekannt



# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel anfangs unbekannt

## Statisches Ziel

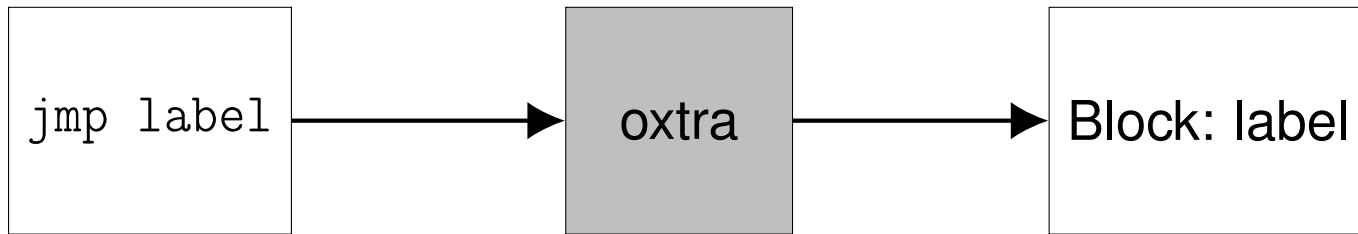
- Sprung zu einer statischen Adresse
- Ziel muss nur einmal evaluiert werden

# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel anfangs unbekannt

## Statisches Ziel

- Sprung zu einer statischen Adresse
- Ziel muss nur einmal evaluiert werden

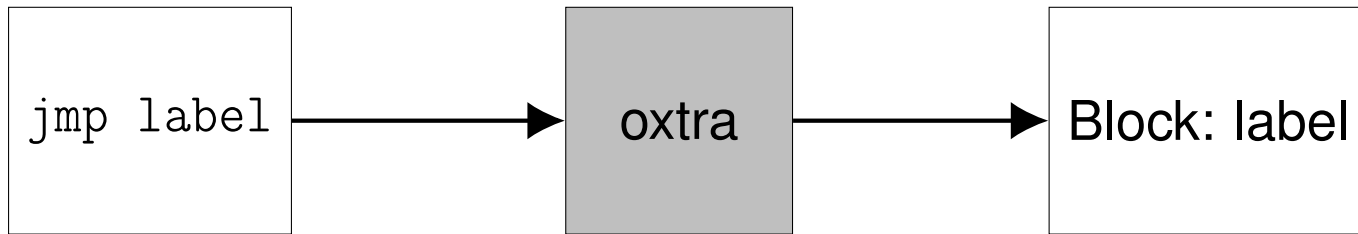


# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel anfangs unbekannt

## Statisches Ziel

- Sprung zu einer statischen Adresse
- Ziel muss nur einmal evaluiert werden



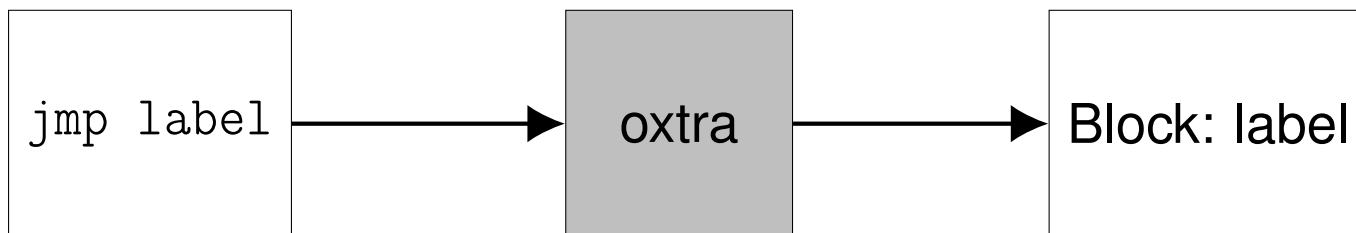
Nach erster Evaluierung Blöcke statisch verbinden (**Block Chaining**)

# Basic Blocks verbinden

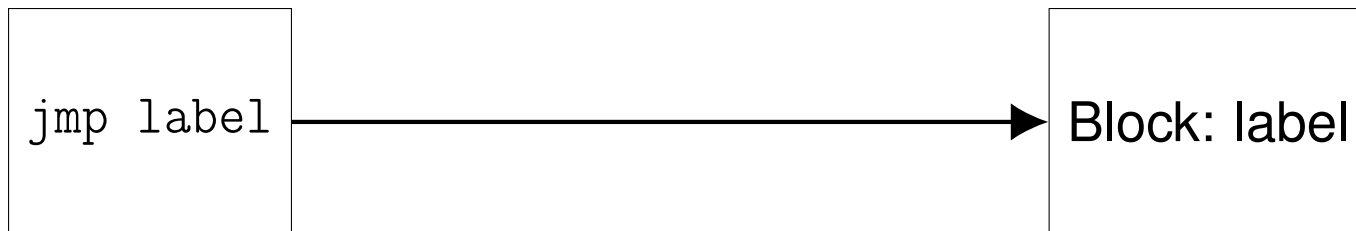
- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel anfangs unbekannt

## Statisches Ziel

- Sprung zu einer statischen Adresse
- Ziel muss nur einmal evaluiert werden



Nach erster Evaluierung Blöcke statisch verbinden (**Block Chaining**)



# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel (anfangs) unbekannt

## **Dynamisches Ziel**

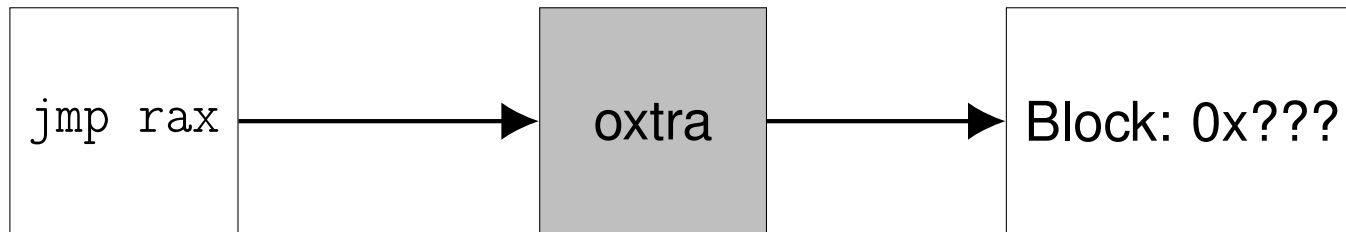
- z.B.: Sprung zu einem Register
- Ziel muss jedes mal neu evaluiert werden

# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel (anfangs) unbekannt

## Dynamisches Ziel

- z.B.: Sprung zu einem Register
- Ziel muss jedes mal neu evaluiert werden

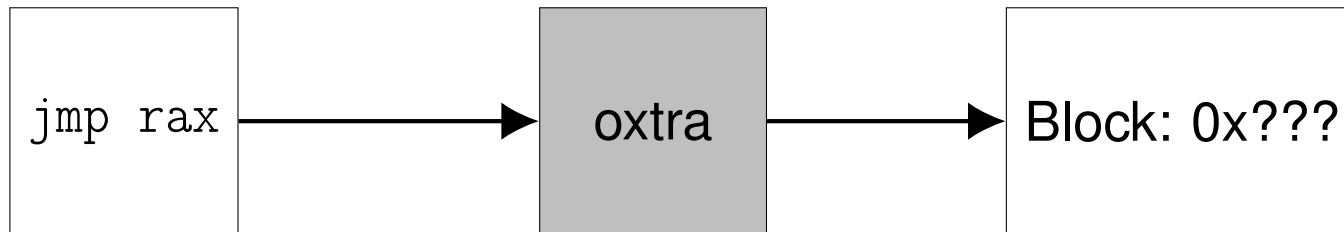


# Basic Blocks verbinden

- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel (anfangs) unbekannt

## Dynamisches Ziel

- z.B.: Sprung zu einem Register
- Ziel muss jedes mal neu evaluiert werden



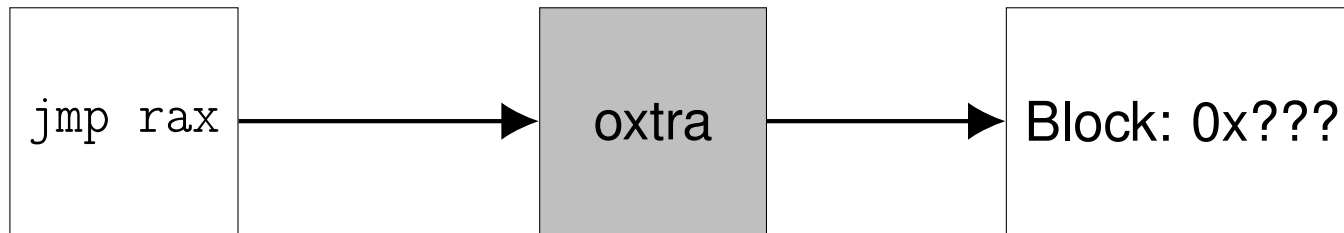
Blöcke bleiben indirekt (dynamisch) verbunden

# Basic Blocks verbinden

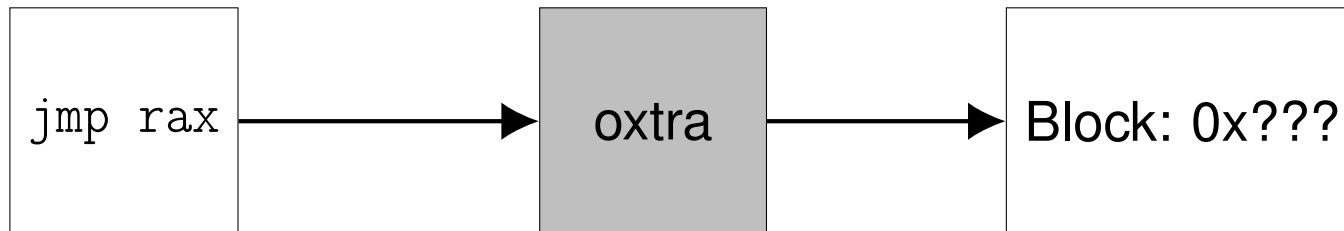
- Letzte Instruktion eines Basic Blocks "zeigt" auf andere(n)
- Konkretes Ziel (anfangs) unbekannt

## Dynamisches Ziel

- z.B.: Sprung zu einem Register
- Ziel muss jedes mal neu evaluiert werden



Blöcke bleiben indirekt (dynamisch) verbunden





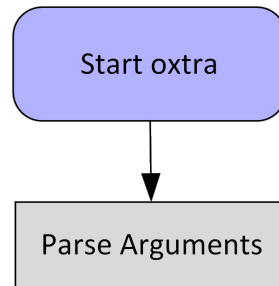
# Context

- Vollständiger Status der CPU (RISC-V)
- Alle Register und Stack

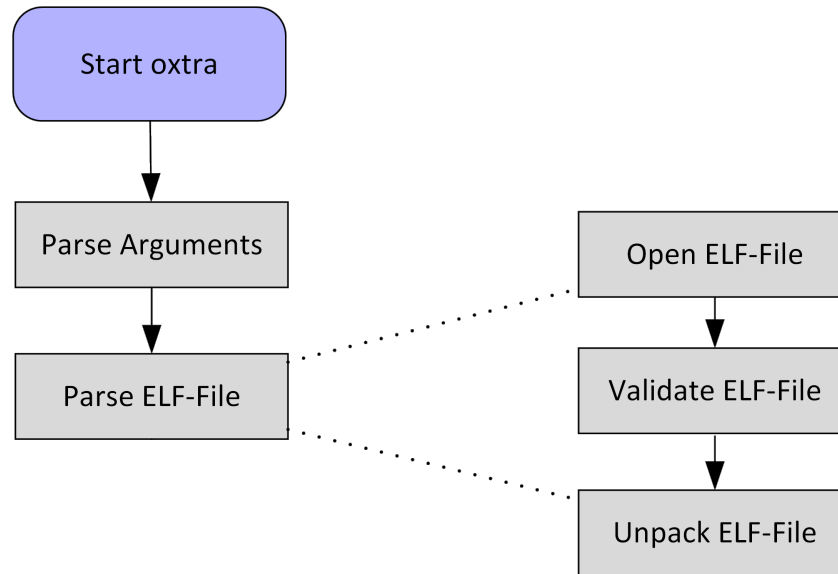
# Context

- Vollständiger Status der CPU (RISC-V)
- Alle Register und Stack
- Unterscheidung: Guest- und Hostcontext
- Wechsel der Contexte notwendig

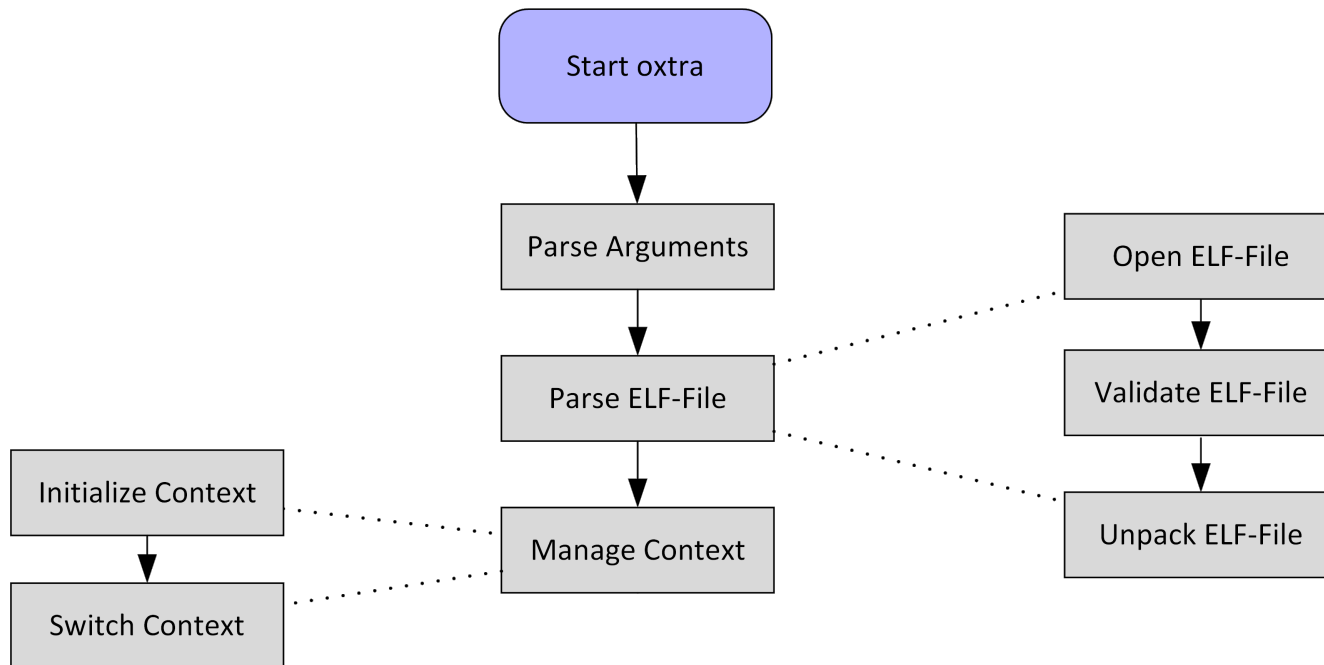
# Technische Umsetzung — Ablauf



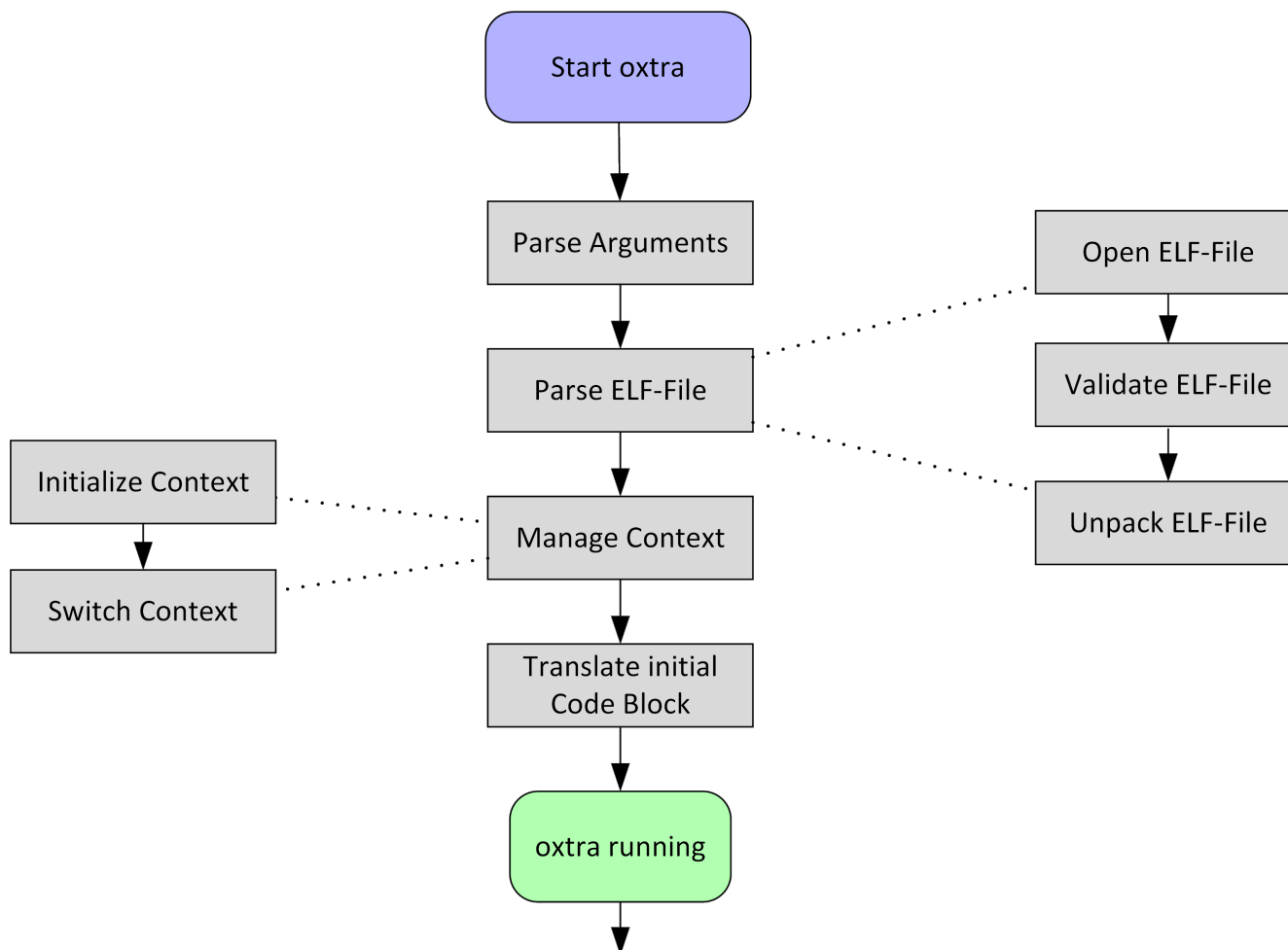
# Technische Umsetzung — Ablauf



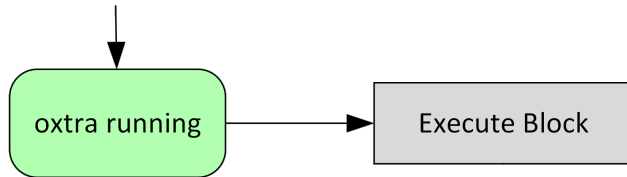
# Technische Umsetzung — Ablauf



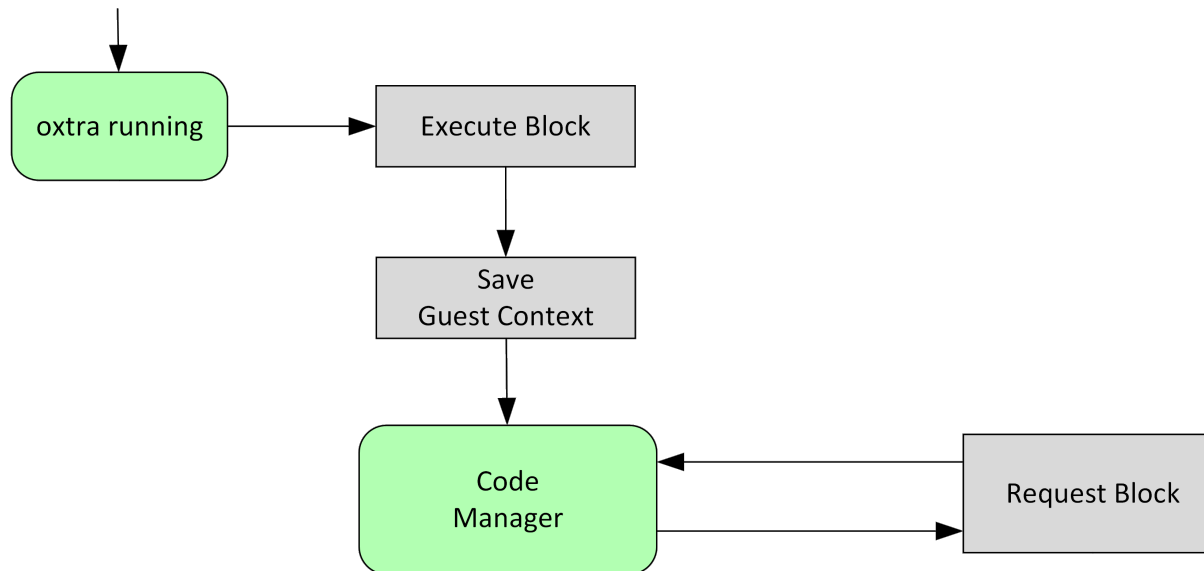
# Technische Umsetzung — Ablauf



# Technische Umsetzung — Ablauf

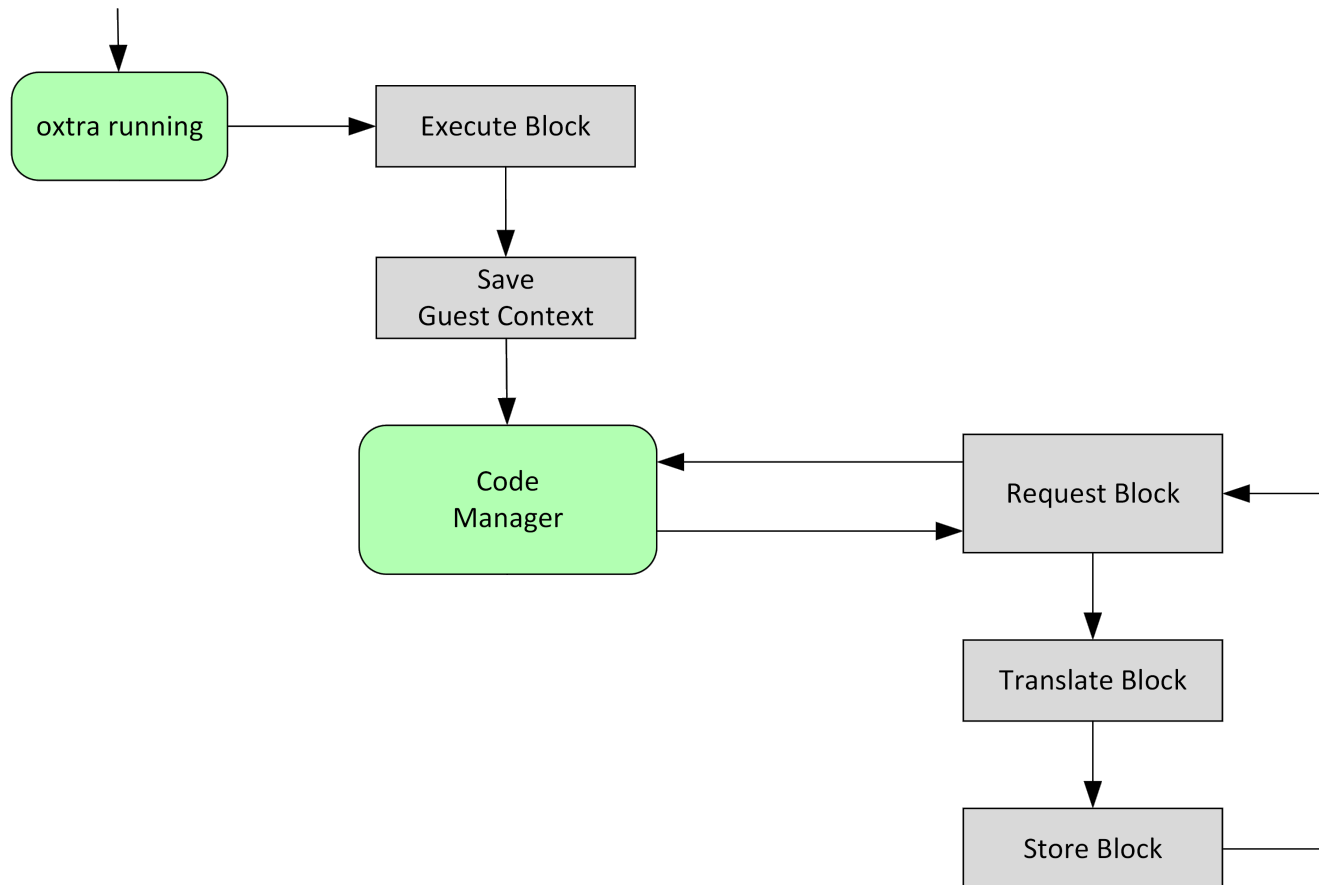


# Technische Umsetzung — Ablauf

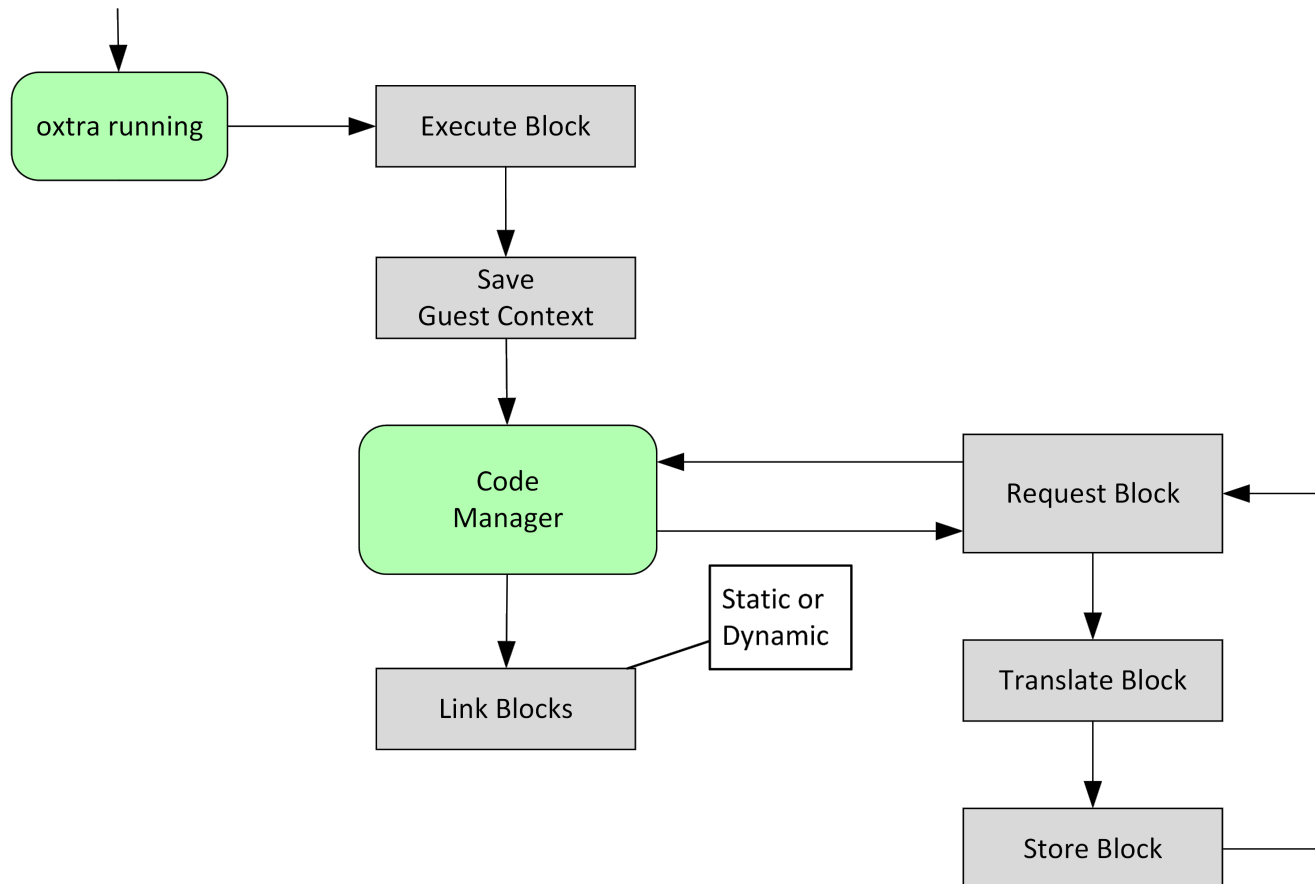




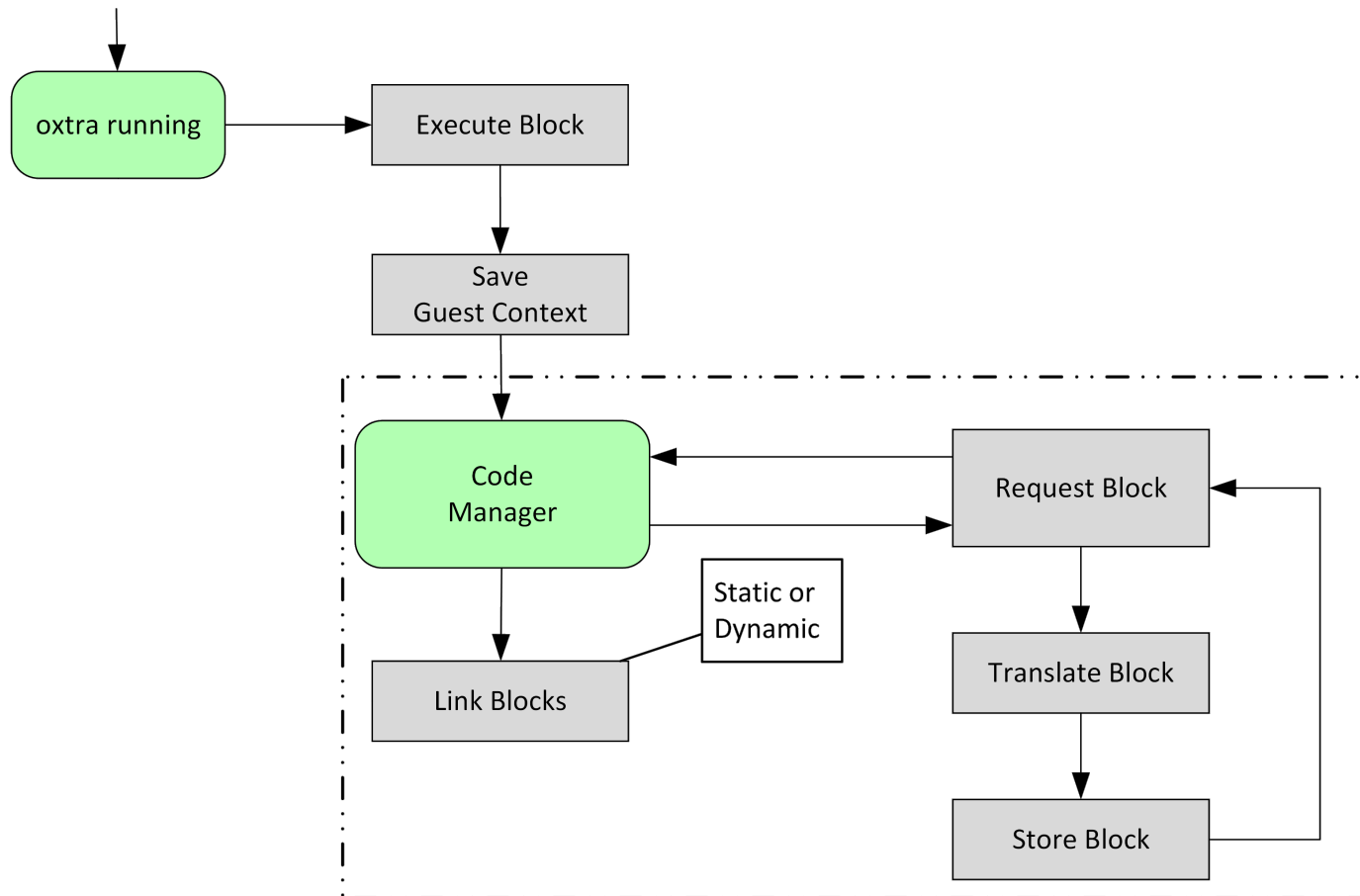
# Technische Umsetzung — Ablauf



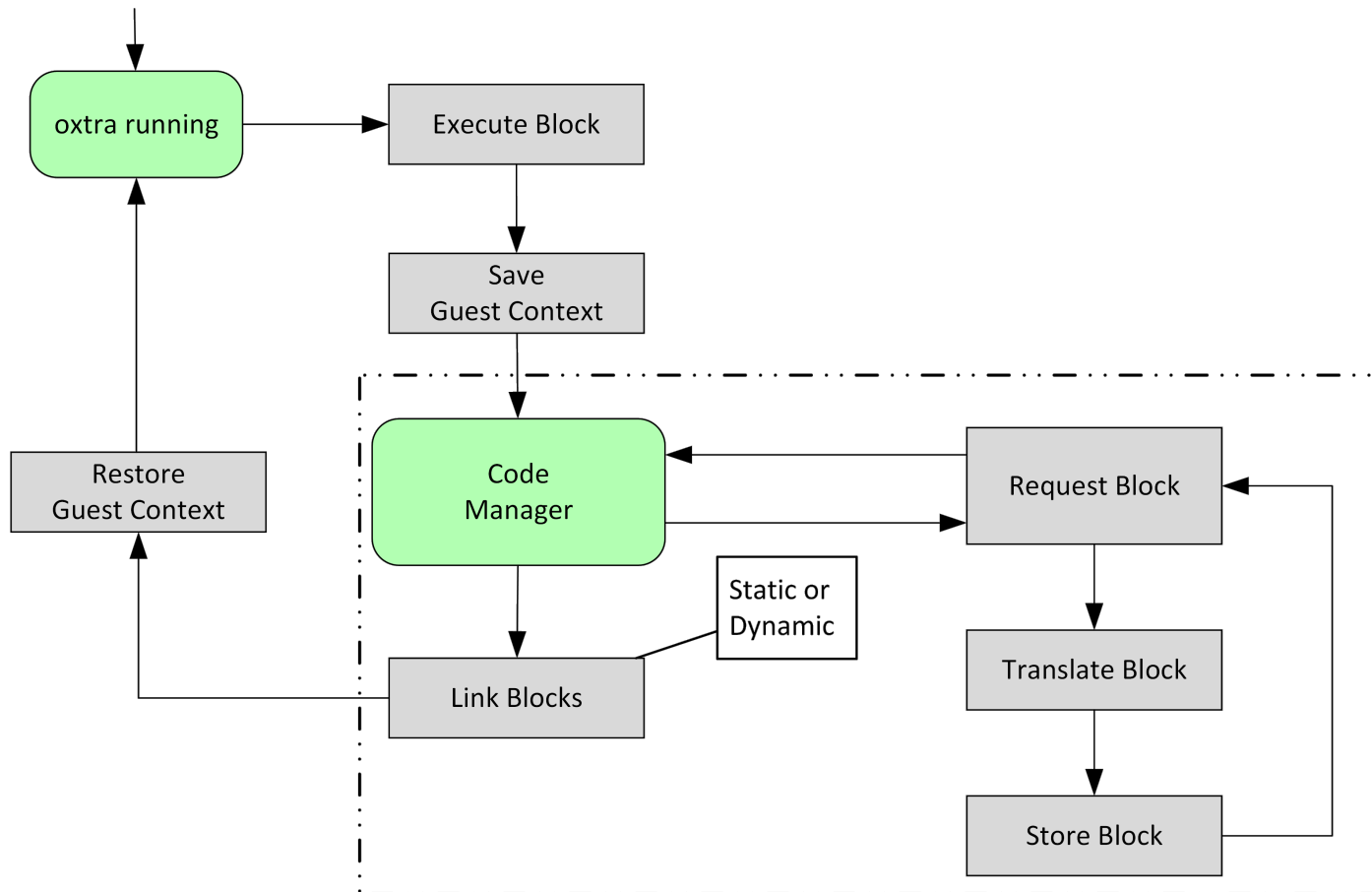
# Technische Umsetzung — Ablauf



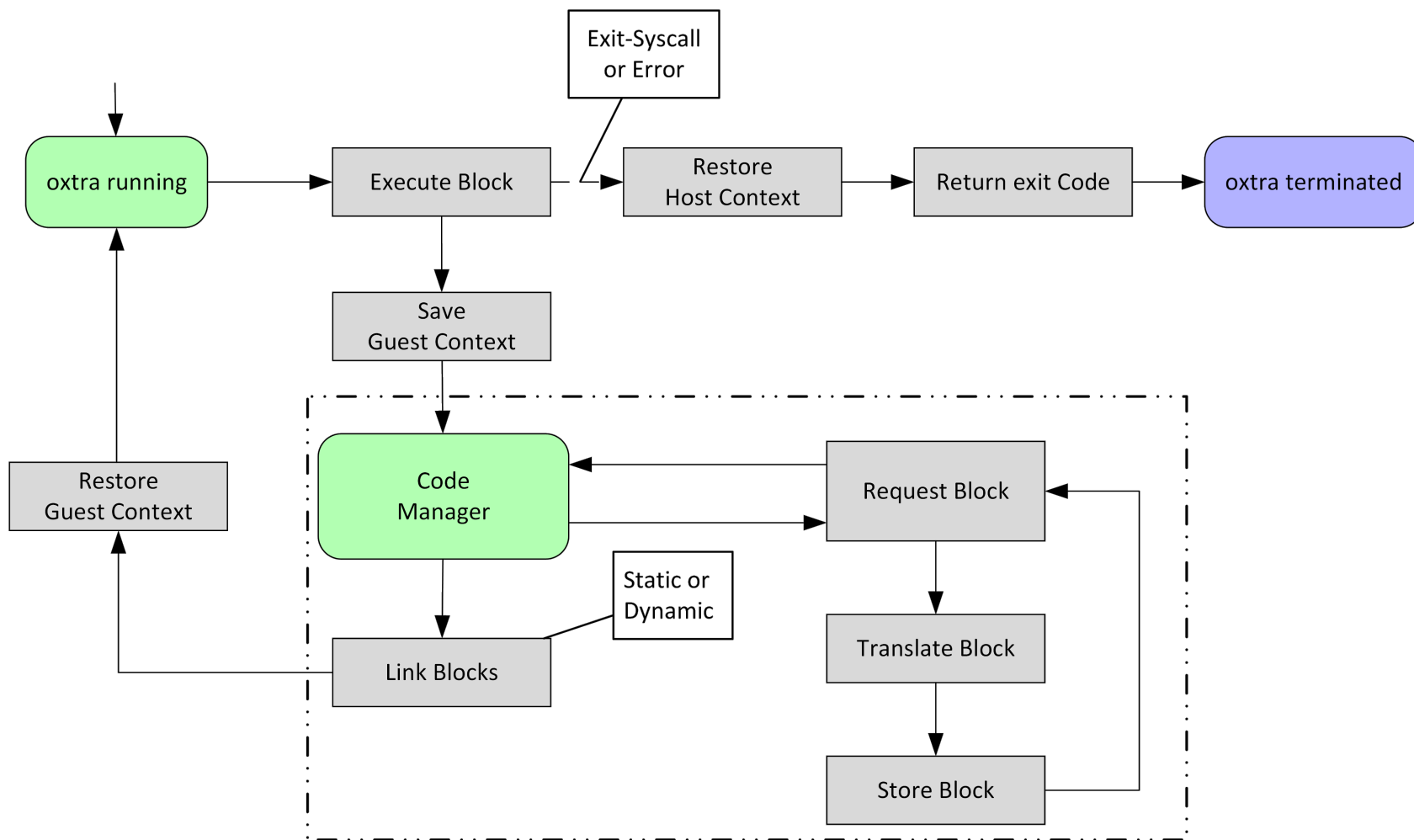
# Technische Umsetzung — Ablauf



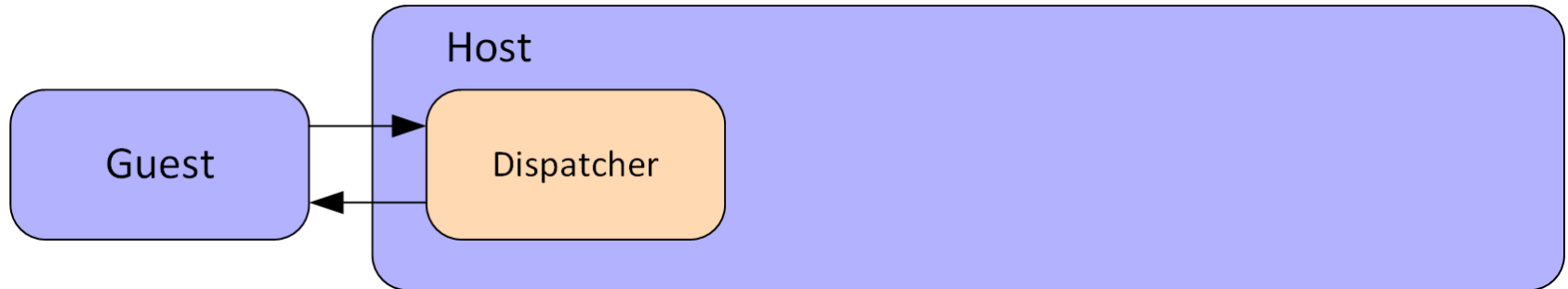
# Technische Umsetzung — Ablauf



# Technische Umsetzung — Ablauf



# Technische Umsetzung



# Technische Umsetzung — Dispatcher

- Implementiert das Interface zwischen Guest und Host

# Technische Umsetzung — Dispatcher

- Implementiert das Interface zwischen Guest und Host
- Steuert das Übersetzen und Linken der Blöcke



# Technische Umsetzung — Dispatcher

- Implementiert das Interface zwischen Guest und Host
- Steuert das Übersetzen und Linken der Blöcke
- Initialisiert Stack und Register ABI konform

# Technische Umsetzung — Dispatcher

- Implementiert das Interface zwischen Guest und Host
- Steuert das Übersetzen und Linken der Blöcke
- Initialisiert Stack und Register ABI konform
- Großteil der Implementierung in Assembler

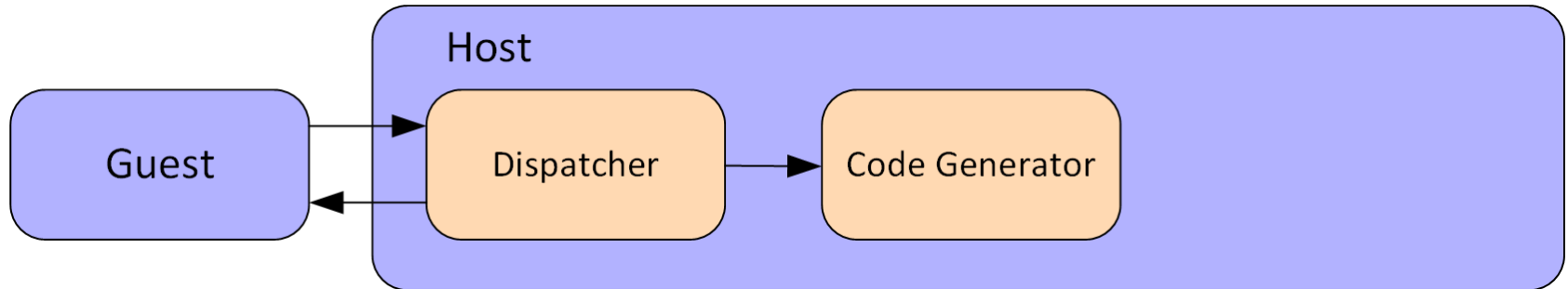
# Technische Umsetzung — Dispatcher

- Implementiert das Interface zwischen Guest und Host
- Steuert das Übersetzen und Linken der Blöcke
- Initialisiert Stack und Register ABI konform
- Großteil der Implementierung in Assembler
- Implementiert Verarbeitung von System Calls

# Technische Umsetzung — Dispatcher

- Implementiert das Interface zwischen Guest und Host
- Steuert das Übersetzen und Linken der Blöcke
- Initialisiert Stack und Register ABI konform
- Großteil der Implementierung in Assembler
- Implementiert Verarbeitung von System Calls
  - Syscall-Indices Zuordnung
  - Syscall an den Kernel weiterleiten

# Technische Umsetzung



# Technische Umsetzung — Code Generator

- Implementiert den Kern der Übersetzung
- Generiert ausführbaren RISC-V Code

# Technische Umsetzung — Code Generator

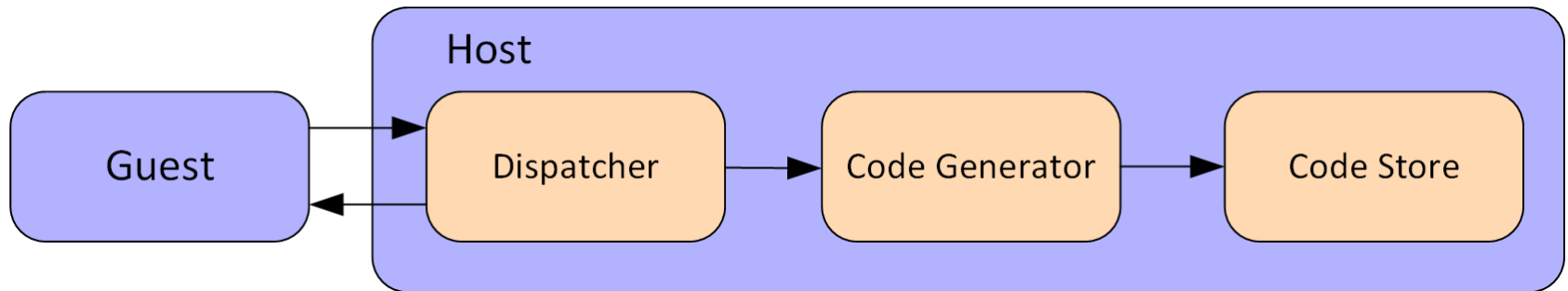
- Implementiert den Kern der Übersetzung
- Generiert ausführbaren RISC-V Code
- Definiert die Register-Zuordnung (→ später mehr)

# Technische Umsetzung — Code Generator

- Implementiert den Kern der Übersetzung
- Generiert ausführbaren RISC-V Code
- Definiert die Register-Zuordnung (→ später mehr)
- Bietet Hilfsfunktionen, die Standard-Operationen benötigen
  - Speicher-Operanden evaluieren
  - Immediates laden
  - ...



# Technische Umsetzung



# Technische Umsetzung — Code Store

- Speichert und verwaltet Blöcke

# Technische Umsetzung — Code Store

- Speichert und verwaltet Blöcke
- Gibt Möglichkeit nach Blöcken zu suchen
- Ermöglicht übersetzen von Teil-Blöcken
- Verwendet Paging-Verfahren zur Block-Verwaltung

# Technische Umsetzung — Code Store

- Speichert und verwaltet Blöcke
- Gibt Möglichkeit nach Blöcken zu suchen
- Ermöglicht übersetzen von Teil-Blöcken
- Verwendet Paging-Verfahren zur Block-Verwaltung
- Optimierungsbedingte Restriktionen:
  - Inhalte bleiben für immer erhalten
  - Speicher darf **nicht** reallokiert werden

# Technische Umsetzung — Register

- 16 x86-64 Register auf 31 RISC-V Register abbilden
- System Call Argumente möglichst in den richtigen Registern
- Verbleibende Register für zusätzliche Daten verwendet

# Technische Umsetzung — Register

- 16 x86-64 Register auf 31 RISC-V Register abbilden
- System Call Argumente möglichst in den richtigen Registern
- Verbleibende Register für zusätzliche Daten verwendet

RISC-V Register	Verwendung
a7	rax
a6	rcx
a2	rdx
s2	rbx
sp	rsp
s0/fp	rbp
a1	rsi
a0	rdi
a4	r8

RISC-V Register	Verwendung
a5	r9
a3	r10
s3-s7	r11-r15
s1	Return Stack
s8	Call Table
s9	TLB
s10	Jump Table
s11	Context
t0-t6	Temporary

# Optimierung — Call/Return

- Return Adresse ist auf dem Stack → dynamischer Link

# Optimierung — Call/Return

- Return Adresse ist auf dem Stack → dynamischer Link
- Nur in Ausnahmefällen modifiziert der Guest die Return Adresse
- Effiziente Lösung: Return Stack



# Optimierung — Call/Return

- Return Adresse ist auf dem Stack → dynamischer Link
- Nur in Ausnahmefällen modifiziert der Guest die Return Adresse
- Effiziente Lösung: Return Stack
- Call speichert Return Adresse in separatem Stack
- Return vergleicht tatsächliche Adresse mit Return Stack Adresse
- Falls gleich: Ausführung fortsetzen
- Falls nicht gleich: Dispatcher aufrufen, dynamische Evaluierung

# Kontrollfluss — Flags

- x86-64 hat ein Status (Flags) Register
- RISC-V hat vergleiche-und-verzweige Instruktionen  
→ Flags müssen explizit berechnet werden

# Kontrollfluss — Flags

- x86-64 hat ein Status (Flags) Register
- RISC-V hat vergleiche-und-verzweige Instruktionen  
→ Flags müssen explizit berechnet werden
- Jede arithmetische Instruktion in x86-64 verändert die Flags
- Nur die letzte arithmetische Operation vor einer Flags-verwendenden Instruktion ist relevant  
→ “Lazy” Evaluation

# Kontrollfluss — Flags

- x86-64 hat ein Status (Flags) Register
- RISC-V hat vergleiche-und-verzweige Instruktionen  
→ Flags müssen explizit berechnet werden
- Jede arithmetische Instruktion in x86-64 verändert die Flags
- Nur die letzte arithmetische Operation vor einer Flags-verwendenden Instruktion ist relevant  
→ “Lazy” Evaluation

```
1 add rax, rbx
2 update flags
3 add rax, rcx
4 update flags
5 add rax, rdx
6 update flags
7 jcc label
```

(a) Eager Flag Evaluation

```
1 add rax, rbx
2 add rax, rcx
3 add rax, rdx
4 update flags
5 jcc label
```

(b) Lazy Flag Evaluation

# Optimierung — Flags Vorhersagen

- Problem: Flags ausrechnen ist zeitintensiv

# Optimierung — Flags Vorhersagen

- Problem: Flags ausrechnen ist zeitintensiv
- Lösung: Flags nur zum Ende eines Blocks konsistent halten

# Optimierung — Flags Vorhersagen

- Problem: Flags ausrechnen ist zeitintensiv
- Lösung: Flags nur zum Ende eines Blocks konsistent halten
- Problem: Kleine Blöcke haben größeren Anteil an Flag-Code als der “echter” Code

# Optimierung — Flags Vorhersagen

- Problem: Flags ausrechnen ist zeitintensiv
- Lösung: Flags nur zum Ende eines Blocks konsistent halten
- Problem: Kleine Blöcke haben größeren Anteil an Flag-Code als der “echter” Code
- Lösung: Rekursive Flagnutzungsvorhersage über den Blockrand hinaus um nur relevante Flags zu aktualisieren

```
1 add rax, rbx
2 jmp label
3 ...
4 label:
5 cmp rcx, rdx
```



# Benchmarking — Methodik

- Keine Hardware → Benchmarking in einem Emulator
- Laufzeit (nicht präzise wegen Emulation)
- Ausgeführte Instruktionen (unabhängig von Hardware/Emulation)
- Speicherverbrauch
- Generierte Instruktionen
- Blocklänge

# Benchmarking — Methodik

- Keine Hardware → Benchmarking in einem Emulator
- Laufzeit (nicht präzise wegen Emulation)
- Ausgeführte Instruktionen (unabhängig von Hardware/Emulation)
- Speicherverbrauch
- Generierte Instruktionen
- Blocklänge
- Testprogramme
  - Primzahlen unter einer Schwelle  $n$  berechnen und ausgeben
  - gzip

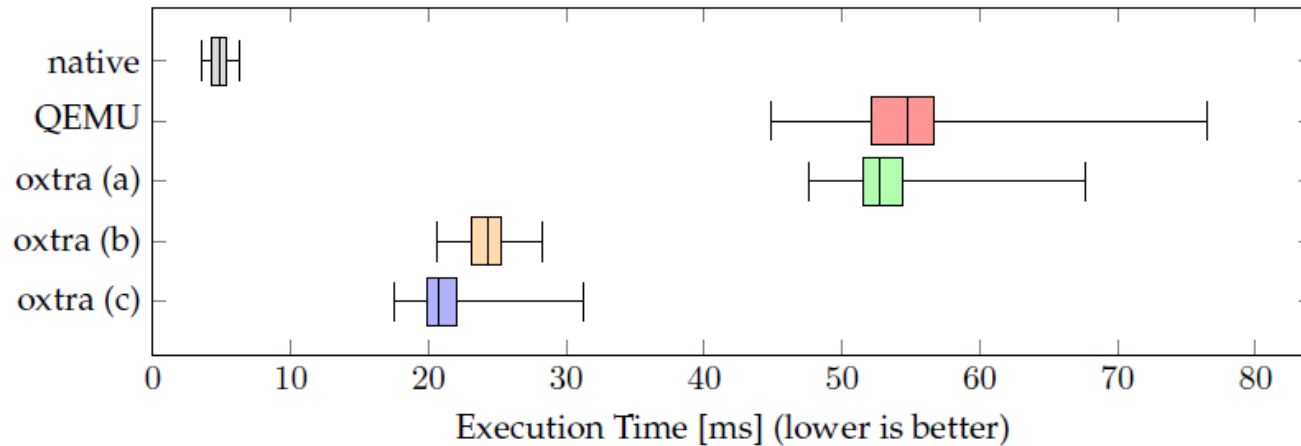
# Benchmarking — Versionen

- oxtra (a): Erste Version die gzip ausführen konnte
- oxtra (b): Return Stack
- oxtra (c): Aktuelle Version
  - System Call Optimierung
  - Flag Prediction
  - Blockerweiterung

# Benchmarking — Versionen

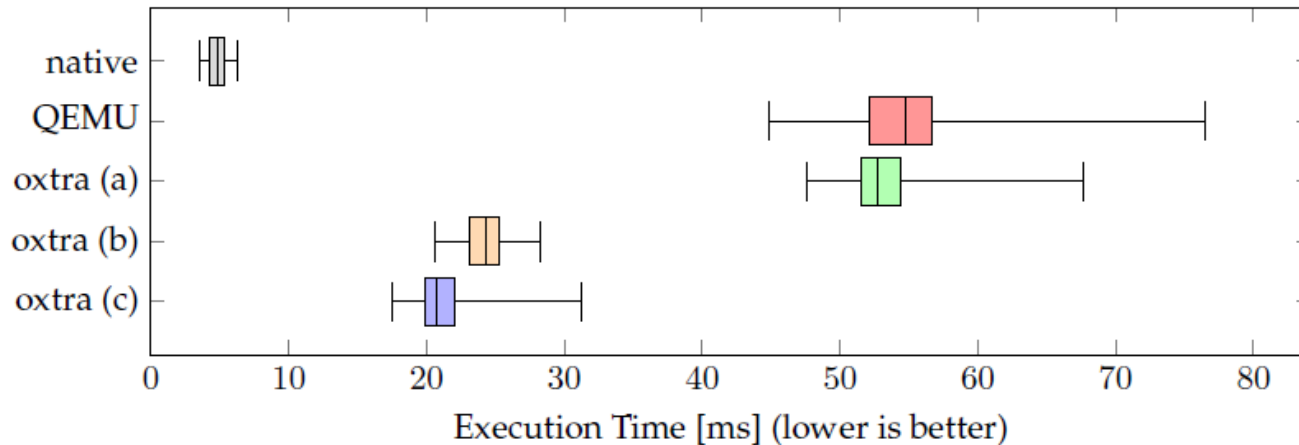
- oxtra (a): Erste Version die gzip ausführen konnte
- oxtra (b): Return Stack
- oxtra (c): Aktuelle Version
  - System Call Optimierung
  - Flag Prediction
  - Blockerweiterung
- native: direkt kompiliert für RISC-V
- QEMU 4.1.0

# Benchmarking — Laufzeit — Primzahlen

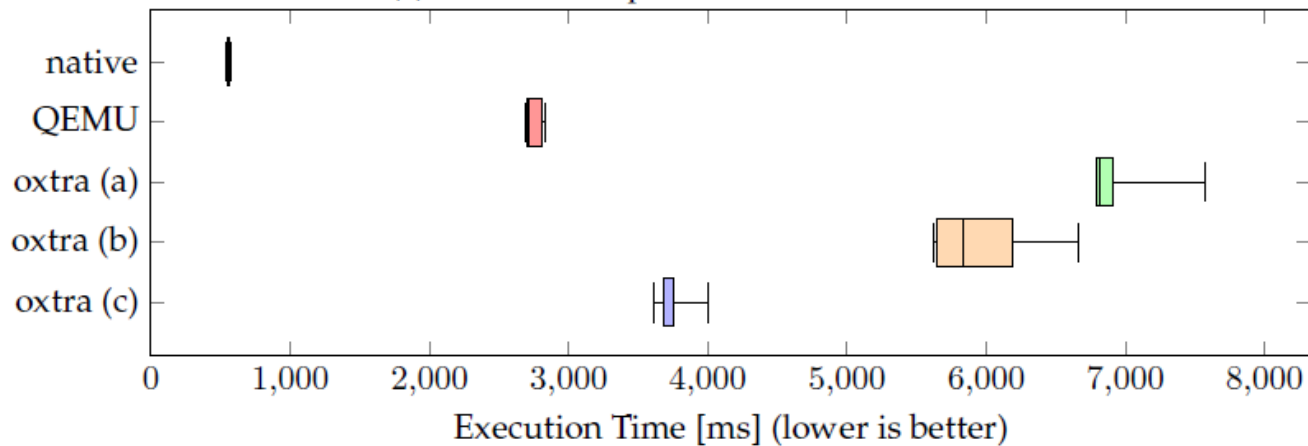


(a) 100 runs, first primes below 10,000

# Benchmarking — Laufzeit — Primzahlen

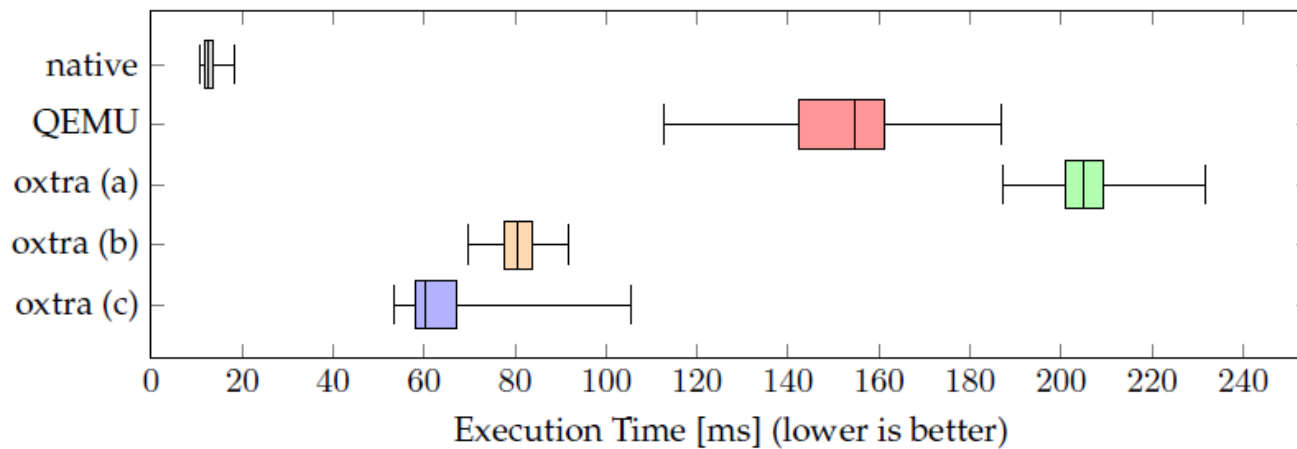


(a) 100 runs, first primes below 10,000



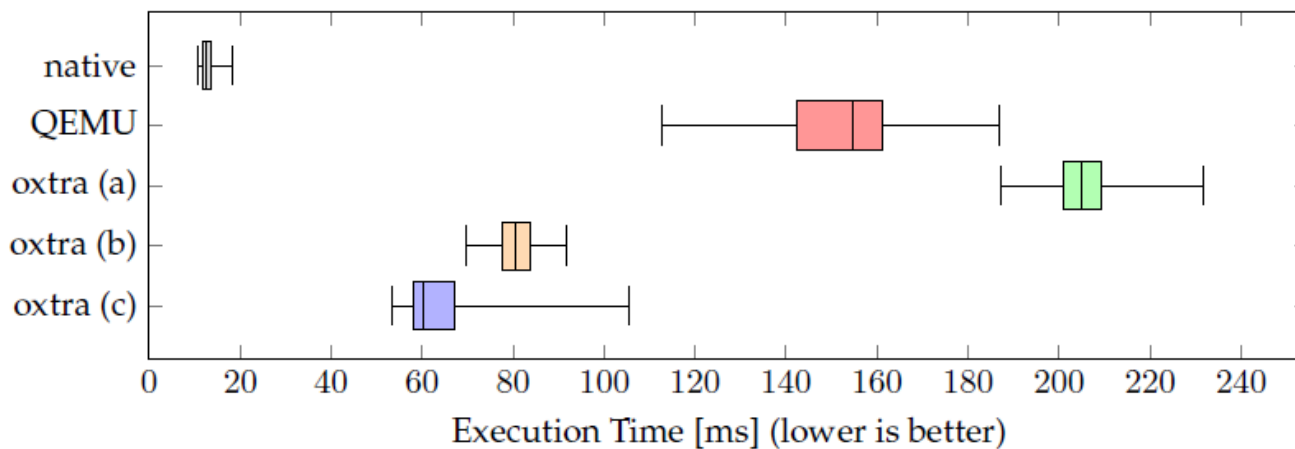
(b) 5 runs, first primes below 1,000,000

# Benchmarking — Laufzeit — gzip

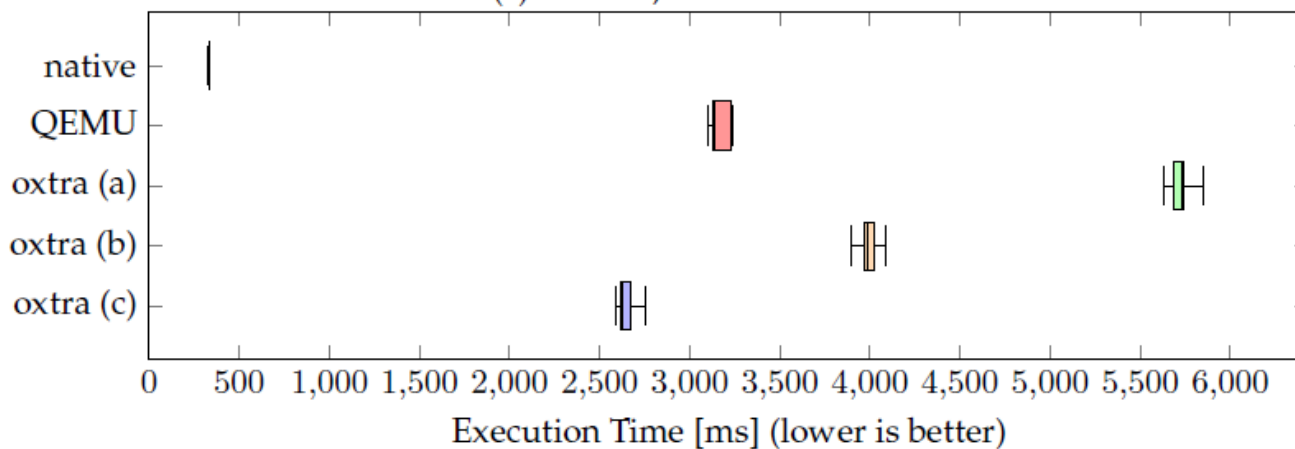


(a) 100 runs, 132kB file

# Benchmarking — Laufzeit — gzip



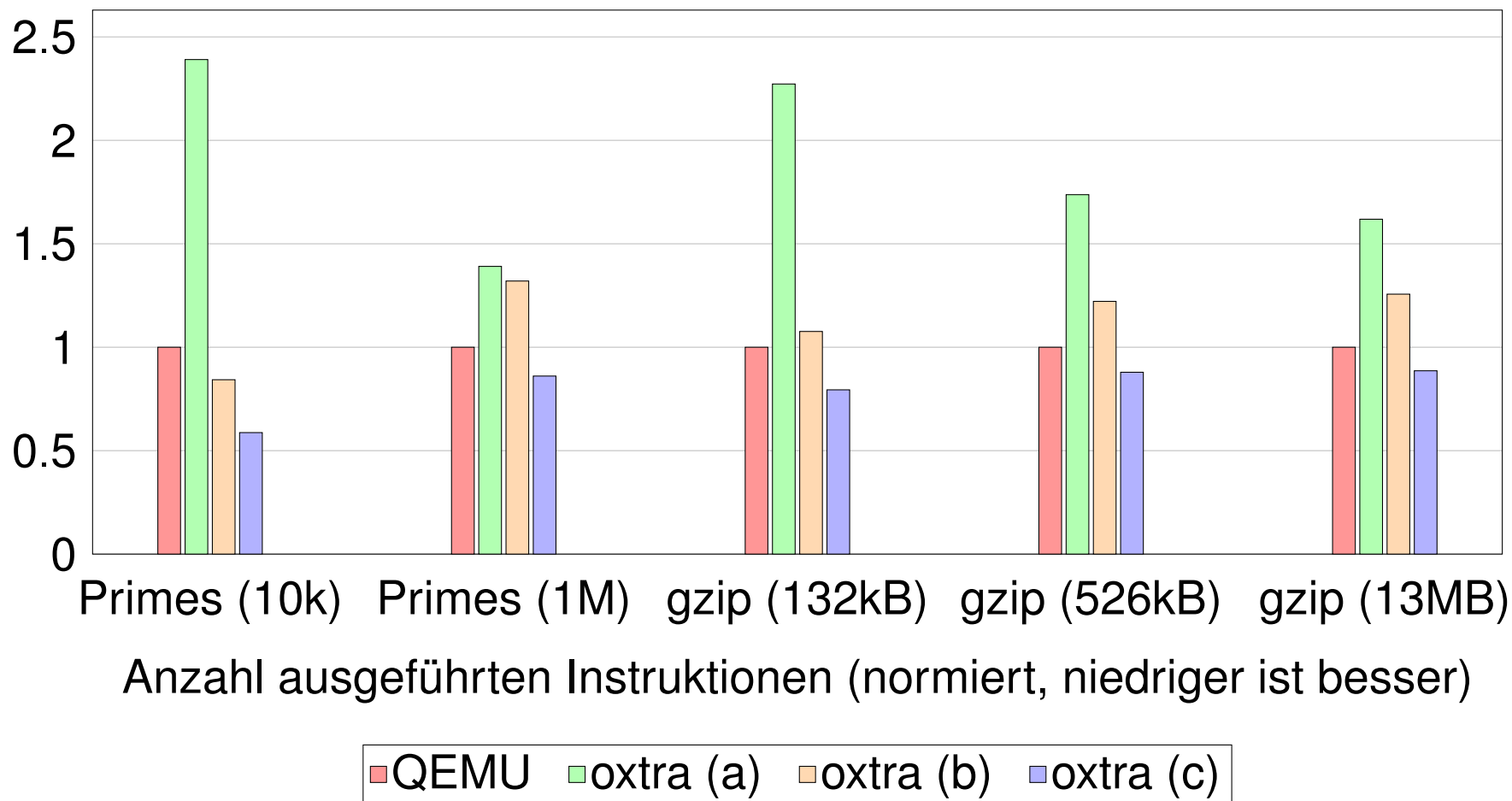
(a) 100 runs, 132kB file



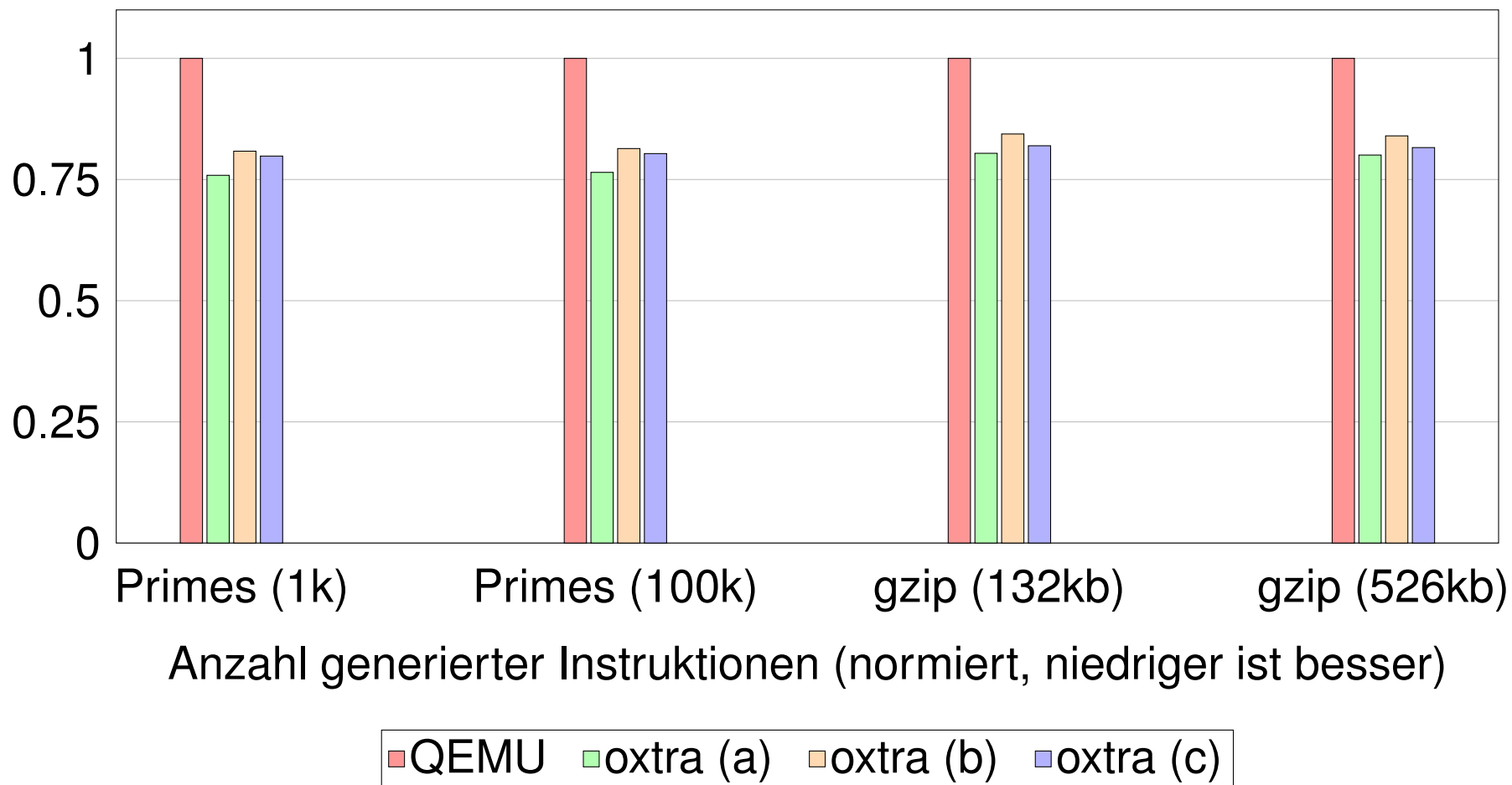
(c) 5 runs, 13MB file



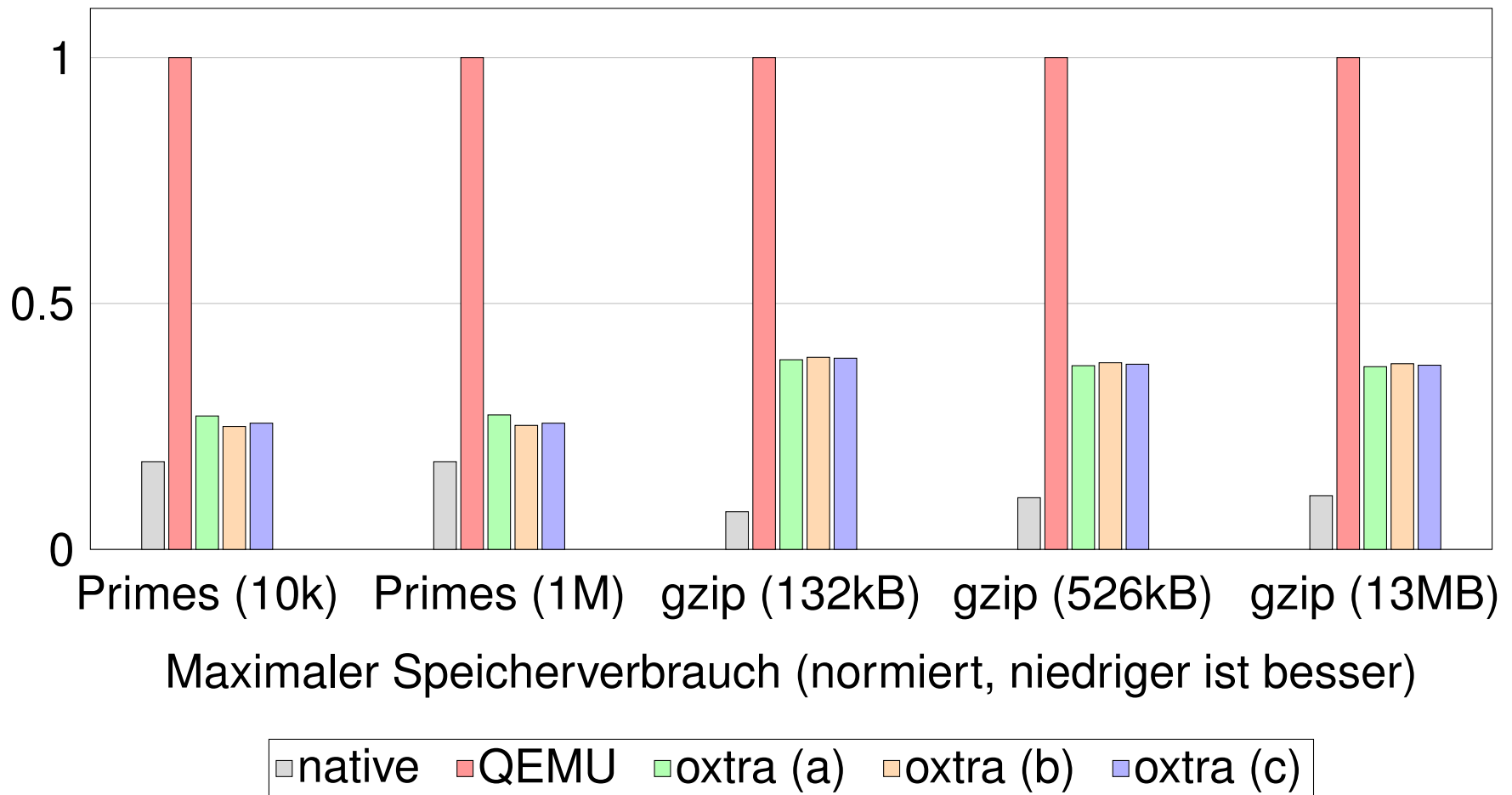
# Benchmarking — Ausgeführte Instruktionen



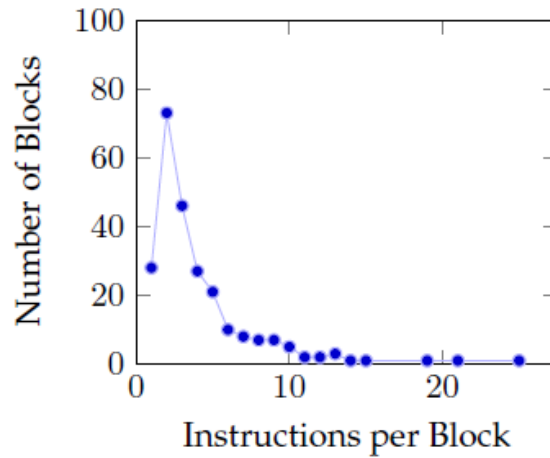
# Benchmarking — Erzeugte Instruktionen



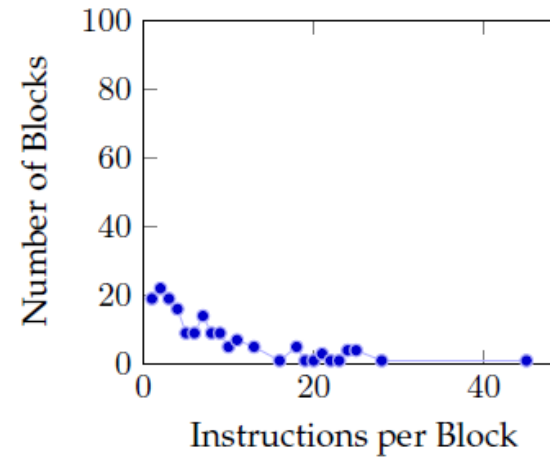
# Benchmarking — Speicherverbrauch



# Benchmarking — Instruktionen pro Block

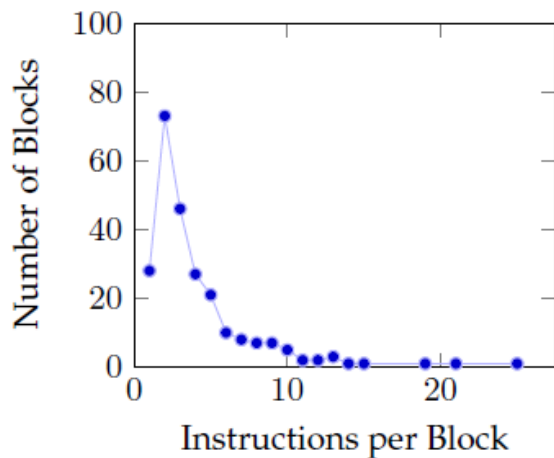


(a) Primes, oxtra (b)

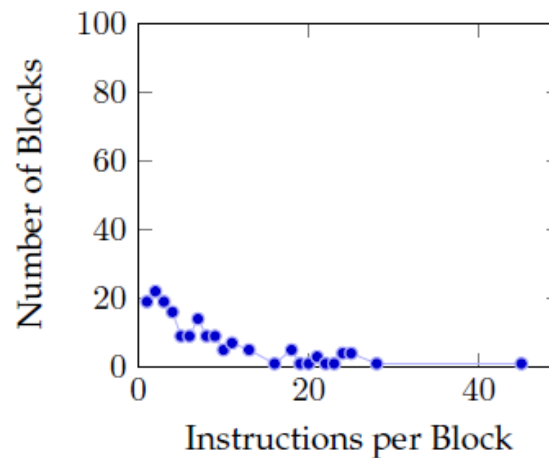


(b) Primes, oxtra (c)

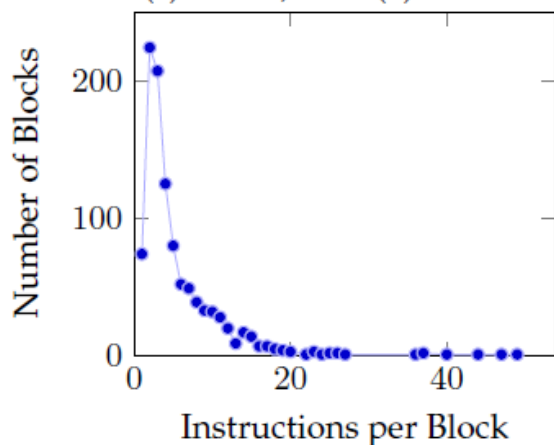
# Benchmarking — Instruktionen pro Block



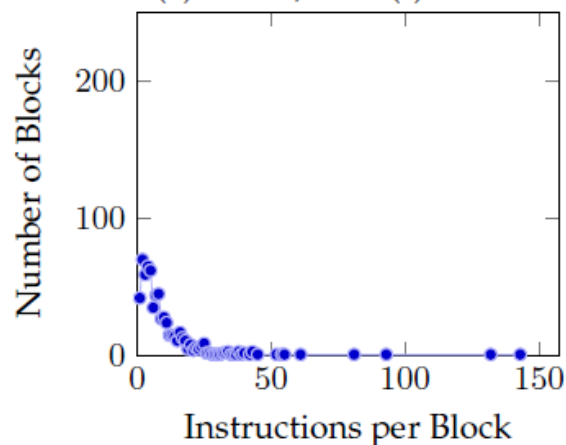
(a) Primes, oxtra (b)



(b) Primes, oxtra (c)

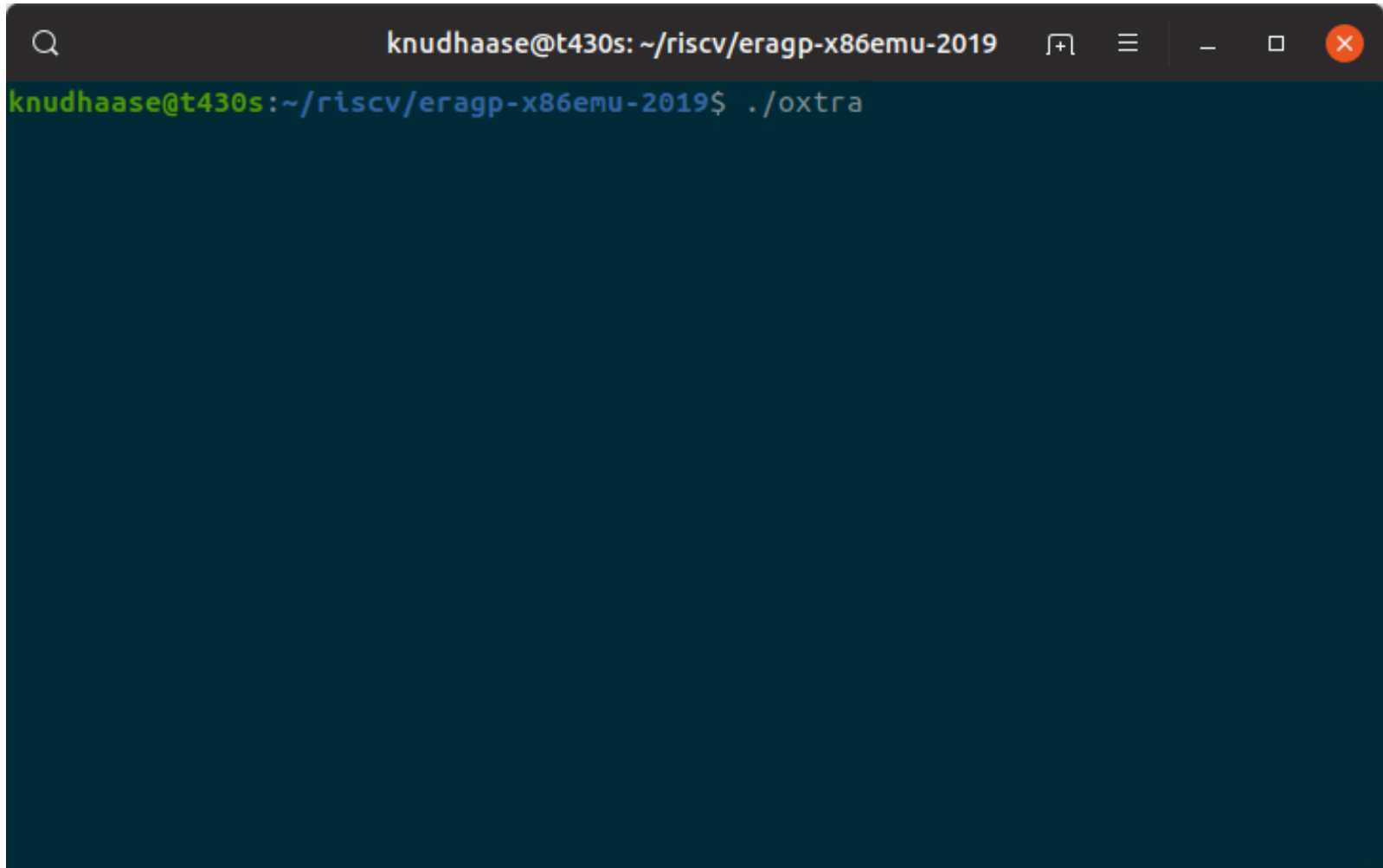


(c) gzip, oxtra (b)



(d) gzip, oxtra (c)

# Live Demo



A terminal window with a dark background. The title bar at the top shows the user 'knudhaase' on host 't430s' in the directory '~/riscv/eragp-x86emu-2019'. The terminal content shows the prompt 'knudhaase@t430s:~/riscv/eragp-x86emu-2019\$' followed by the command './oxtra'.

```
knudhaase@t430s: ~/riscv/eragp-x86emu-2019
knudhaase@t430s:~/riscv/eragp-x86emu-2019$ ./oxtra
```