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

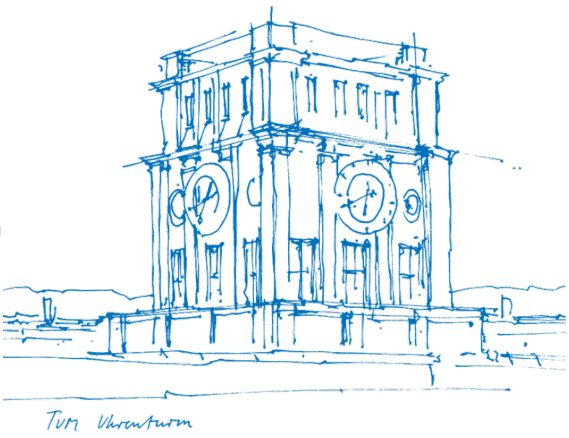
Dynamische Binärübersetzung: RISC–V → x86–64

Endpräsentation

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

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

28. Oktober 2020



Gliederung

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 Zielplatform 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 vs. dynamische Übersetzung

Vorteile

- Einfach zu implementieren
- Keine Erzeugung von JIT Assembler nötig

Nachteile

- Mehrfache Übersetzung der selben Instruktion
- Wenig Optimierungspotential

Vorteile

- Einfach zu implementieren
- Keine Erzeugung von JIT Assembler nötig

Dynamische Binärübersetzung

Statische Binärübersetzung

Vorgehen

- Eimaliges Übersetzen der gesamten Auszuführenden Datei
- Ausführen der Übersetzung

Vorteile

- Einmaliger Übersetzungsaufwand
- Theoretisch fast native Geschwindigkeit erreichbar

Nachteile

- Schwierig zu implementieren (Halteproblem)
- Problematisch bei Änderungen der Assembly zur Runtime

Dynamische Binärübersetzung

Dynamische Binärübersetzung

Vorgehen

- Einlesen eines Blocks von Befehlen
- (Optional) Optimierungen
- Übersetzen des einzelnen Blocks
- Ausführen des Übersetzten Blocks

Vorteile

- Nur tatsächlich notwendige Teile des Programms werden übersetzt
- Instruktionen werden nur einmal Übersetzt
- Blöcke von Instruktionen eigenen sich zur Optimierung

Nachteile

- Problematisch bei Änderungen der Assembly zur Runtime

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

$\Longrightarrow$ Übersetzung von *Basic Blocks*

Partitionierung des Codes

Grundlagen

Ziel: Finden von sinnvollen Übersetzungseinheiten

Überlegung:

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

$\Longrightarrow$ 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 $\Longrightarrow$ 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        ⟹        mov rd, rs1
                                  sub rd, rs2
```

Codegenerierung

Beispiel: Macro Operation Fusion

Optimierungsmöglichkeit: mehrere Instruktionen bündeln

```
  lui rd, imm1                 ⟹        mov rd, (imm1 + imm2)
addi rd, rd, imm2
```

Code Cache

Konzept

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 ($\rightarrow$ 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 ($\rightarrow$ Hash-Tabelle, TLB)

Nutzung:

- Block wird nach erstem Übersetzen in den Cache geschrieben
- Lookup vollzieht Adressübersetzung RISC–V $\rightarrow$ x86
- kein Löschen von übersetzten Blöcken ($\rightarrow$ 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 $\Longrightarrow$ **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 $\Longrightarrow$ statische Abbildung nur teilweise möglich
- `rax`, `rcx`, `rdx` speziell benötigt; `rsp` unpraktisch

Registernutzung

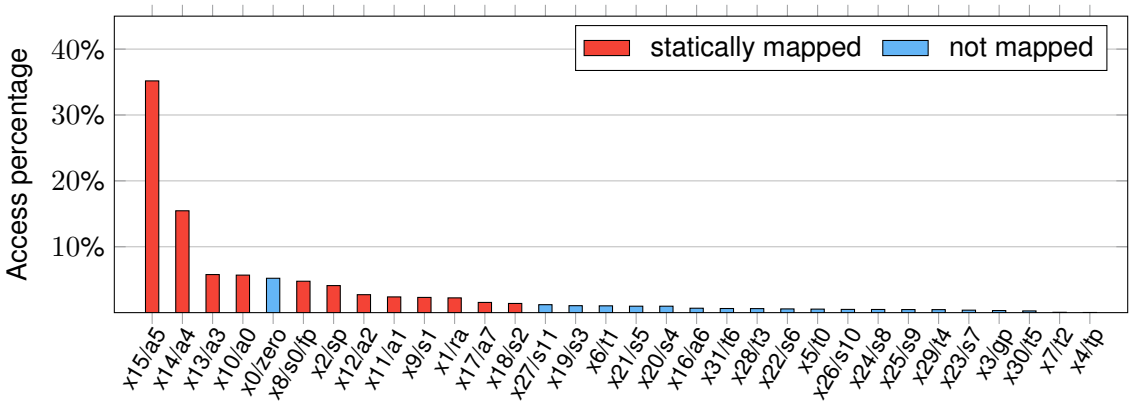
Ansatz

Überlegung: Welche 12 Register werden häufig verwendet?

Registernutzung

Ansatz

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 ($\rightarrow$ 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 ($\rightarrow$ Kontextwechsel)
- dynamische Allokation in die restlichen 3 x86-Register
  - dynamisch in `rax`, `rcx`, `rdx` (*least recently used*, *lazy write-back*)
  - an Blockgrenzen zurückgeschrieben

Testing

Ziel: Überprüfung der Korrektheit

Ansätze:

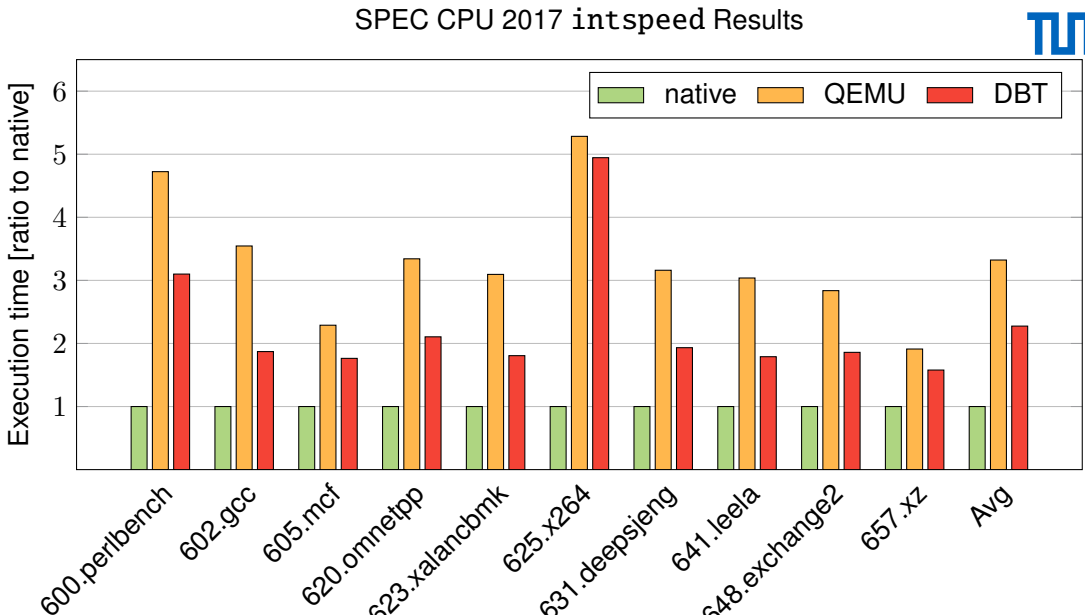
- Extensive Unittests (durch Parametrisierung über 30.000 Test cases)
- Ausführen einfacher bzw. komplizierterer Programme und Überprüfen des Outputs

Benchmarks

SPEC CPU 2017

- Kommerzielle Benchmark Suite mit einer großen Vielfalt an verschieden 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 intspeed Results
native
QEMU
DBT
Execution time [ratio to native]
1
2
3
4
5
6
600.perlbench
602.gcc
605.mcf
620.omnetpp
623.xalancbmk
625.x264
631.deepsjeng
641.leela
648.exchange2
657.xz
Avg

Demo

But can it run Crysis?

```
./translator
```