

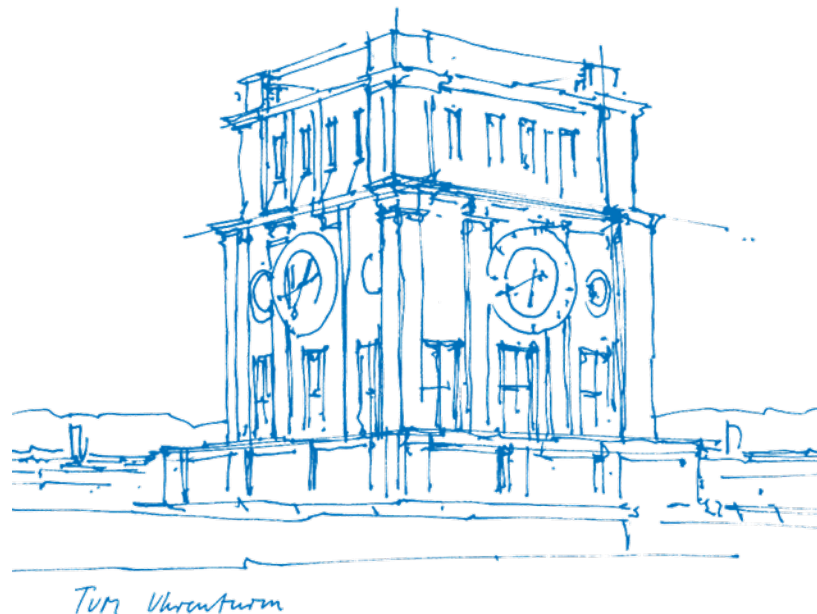
Dynamische Binärübersetzung: RISC-V \rightarrow x86-64

Zwischenpräsentation

Noah Dormann¹, Simon Kammermeier¹,
Johannes Pfannschmidt¹, Florian Schmidt¹

¹Fakultät für Informatik, Technische Universität München (TUM)

21. Juli 2020



Gliederung

1. Einführung

- 1.1 Dynamische Binärübersetzung
- 1.2 Grobüberblick über die RISC-V ISA
- 1.3 Angebot

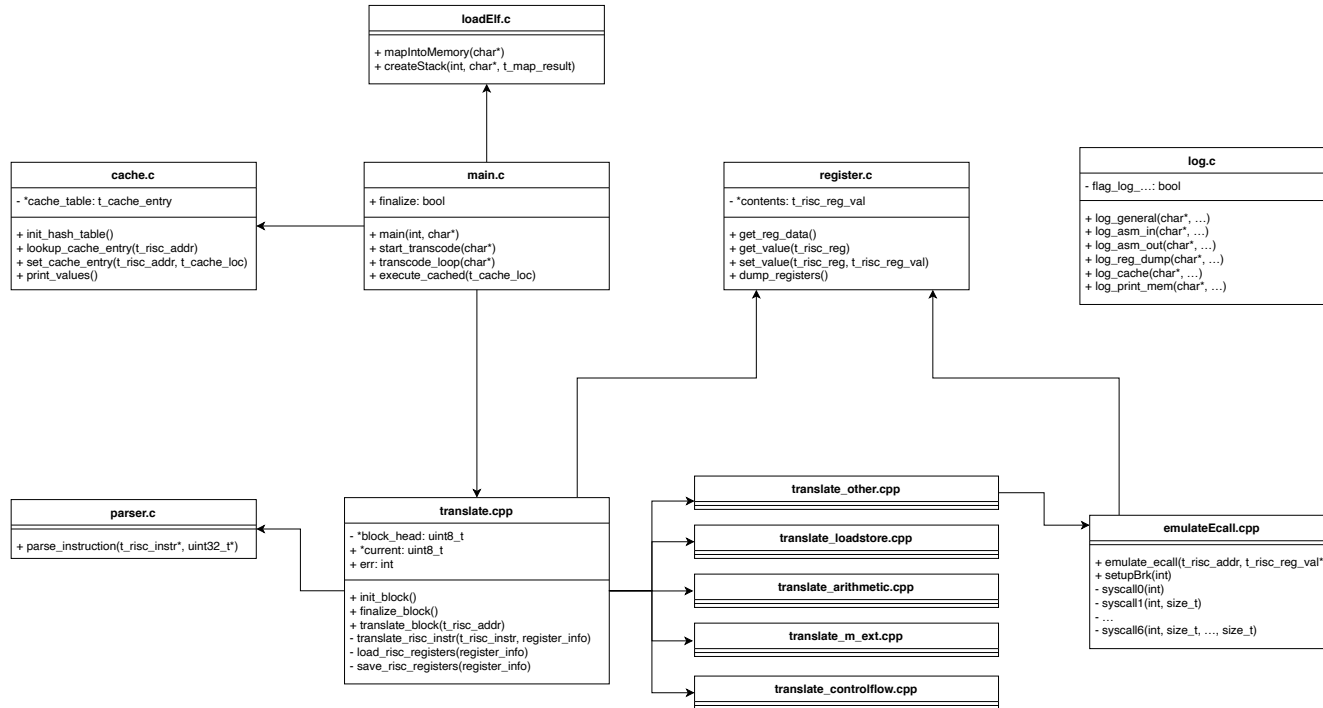
2. Systemarchitektur

- 2.1 ELF-Loader
- 2.2 Parser
- 2.3 Register File
- 2.4 Block Loader
- 2.5 Code Generator
- 2.6 Code Cache

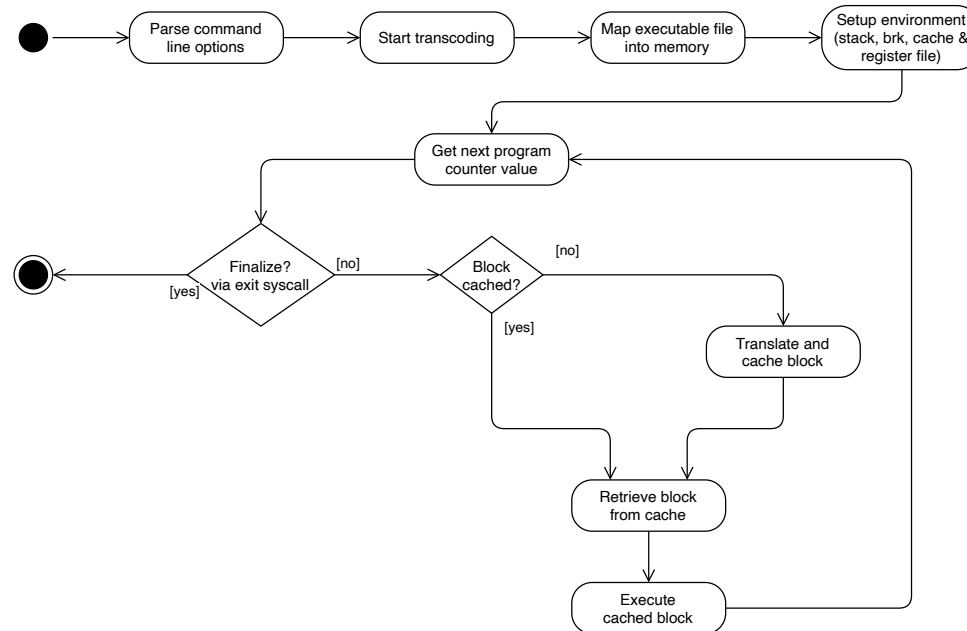
3. Anhang

Einführung

Systemarchitektur



Vorgehen



Parser

Überblick

Ziel: Decodieren der 4 byte großen codierten RISC-V-Befehle aus der geladenen elf-Datei.

→ (Zählen der Anzahl von Zugriffen auf einzelne Register für spätere Optimierungen.)

Input: Zeiger auf RISC-V-Befehl im Speicher

Output: Ausgefüllte `t_risc_instr` Struktur mit allen relevanten Informationen

- Nutzen der Instruction-Structs des Parsers
- Instruktionsmapping RISC-V → x86
- einzelne Übersetzungsfunktionen für jede Instruktion

- allokierte Speicherseite für die x86-Assembly
- Encoding der Instruktionen in den Speicherbereich

```
typedef struct {  
    t_risc_addr addr;  
    t_risc_mnem mnem;  
    t_risc_optype optype;  
    t_risc_reg reg_src_1;  
    t_risc_reg reg_src_2;  
    t_risc_reg reg_dest;  
    t_risc_imm imm;  
} t_risc_instr;
```

Parser

Ansatz

Ziel: Decodieren der 4 byte großen codierten RISC-V-Befehle aus der geladenen elf-Datei.

```
void parse_instruction(t_risc_instr *p_instr_struct, uint32_t *reg_count);
```

Parser

Ansatz

Ziel: Decodieren der 4 byte großen codierten RISC-V-Befehle aus der geladenen elf-Datei.

```
void parse_instruction(t_risc_instr *p_instr_struct, uint32_t *reg_count);
```

31	27	26	25	24	20	19	15	14	12	11	7	6	0	
funct7				rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]						rs1		funct3		rd		opcode		I-type
imm[11:5]				rs2		rs1		funct3		imm[4:0]		opcode		S-type
imm[12 10:5]				rs2		rs1		funct3		imm[4:1 11]		opcode		B-type
imm[31:12]										rd		opcode		U-type
imm[20 10:1 11 19:12]										rd		opcode		J-type

```
typedef struct {
    t_risc_addr addr;
    t_risc_mnem mnem;
```


Register File

Ziel: Speicherung der Registerinhalte des RISC-V-Programmes

Register File

Ziel: Speicherung der Registerinhalte des RISC-V-Programmes

Emulieren der Register x0 bis x31 sowie pc in

```
t_risc_reg_val contents[33];
```

Register File

Ziel: Speicherung der Registerinhalte des RISC-V-Programmes

Emulieren der Register x0 bis x31 sowie pc in

```
t_risc_reg_val contents[33];
```

und Zugriff via Startpointer und den convenience methods:

```
t_risc_reg_val *get_reg_data(void);  
t_risc_reg_val get_value(t_risc_reg reg);  
void set_value(t_risc_reg reg, t_risc_reg_val val);
```

z.T.: Caching der Inhalte in Hardware-x86-Registern je nach Registermapping für die Basic Blocks.

Dynamische Codegenerierung

Überblick

Input: gepackte RISC-V-Instruktionen eines Basic Blocks

Output: übersetzte x86-Instruktionen für diesen Block

Dynamische Codegenerierung

Überblick

Input: gepackte RISC-V-Instruktionen eines Basic Blocks

Output: übersetzte x86-Instruktionen für diesen Block

- Nutzen der Instruction-Structs des Parsers
- Instruktionsmapping RISC-V \rightarrow x86
- einzelne Übersetzungsfunktionen für jede Instruktion

- allokierte Speicherseite für die x86-Assembly
- Encoding der Instruktionen in den Speicherbereich

```
typedef struct {  
    t_risc_addr addr;  
    t_risc_mnem mnem;  
    t_risc_optype optype;  
    t_risc_reg reg_src_1;  
    t_risc_reg reg_src_2;  
    t_risc_reg reg_dest;  
    t_risc_imm imm;  
} t_risc_instr;
```

Dynamische Codegenerierung

Ansatz

Übersetzung aller Instruktionen im Basic Block in einen x86-Buffer,

```
//aus translate_block(t_risc_addr), translate.cpp
init_block();

for (int i = 0; i < instructions_in_block; i++) {
    translate_risc_instr(block_cache[i], r_info);
}

return finalize_block();
```

anschließend

- Finalisieren des Blocks (ret anhängen, etc.)
- Rückgabe des Blocks an den Cache (später).

Dynamische Codegenerierung

Metadaten

Register-Mapping als Parameter für die Übersetzerfunktionen, basierend auf Zuteilung des Block Loaders:

```
/**  
 * Register information for the translator functions.  
 */  
struct register_info {  
    asmjit::x86::Gp *map;  
    bool *mapped;  
    uint64_t base;  
};
```

Dynamische Codegenerierung

Metadaten

Register-Mapping als Parameter für die Übersetzerfunktionen, basierend auf Zuteilung des Block Loaders:

```
/**
 * Register information for the translator functions.
 */
struct register_info {
    asmjit::x86::Gp *map;
    bool *mapped;
    uint64_t base;
};
```

- Synchronisierung der zugewiesenen Register mit register file
- Lesen/Schreiben an Basic-Block-Grenzen
- Unterschiedliche Instruktionsübersetzungen je nach Mapping

Dynamische Codegenerierung

Dispatch

Verteilung der Übersetzung auf einzelne Funktionen für jede Instruktion:

```
//aus translate.cpp
void translate_risc_instr(const t_risc_instr &instr, const register_info &r_info) {
    switch (instr.mnem) {
        //...
        case OR:
            translate_OR(instr, r_info);
            break;
        case AND:
            translate_AND(instr, r_info);
            break;
        case SLLIW:
            translate_SLLIW(instr, r_info);
            break;
        //...
    }
```

Dynamische Codegenerierung

Übersetzerfunktionen (1)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Dynamische Codegenerierung

Übersetzerfunktionen (1)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Einfache Instruktionen, z.B. ADD:

```
//aus translate_arithmetic.cpp
void translate_ADD(const t_risc_instr &instr, const register_info &r_info) {
    if (r_info.mapped[instr.reg_dest] && r_info.mapped[instr.reg_src_1] &&
        r_info.mapped[instr.reg_src_2]) {
        //...
    } else {
        a->mov(x86::rax, x86::ptr(r_info.base + 8 * instr.reg_src_1));
        a->add(x86::rax, x86::ptr(r_info.base + 8 * instr.reg_src_2));
        a->mov(x86::ptr(r_info.base + 8 * instr.reg_dest), x86::rax);
    }
}
```

Dynamische Codegenerierung

Übersetzerfunktionen (1)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Einfache Instruktionen, z.B. ADD:

```
//aus translate_arithmetic.cpp
void translate_ADD(const t_risc_instr &instr, const register_info &r_info) {
    if (r_info.mapped[instr.reg_dest] && r_info.mapped[instr.reg_src_1] &&
        r_info.mapped[instr.reg_src_2]) {
        //...
    } else {
        a->mov(x86::rax, x86::ptr(r_info.base + 8 * instr.reg_src_1));
        a->add(x86::rax, x86::ptr(r_info.base + 8 * instr.reg_src_2));
        a->mov(x86::ptr(r_info.base + 8 * instr.reg_dest), x86::rax);
    }
}
```

→ Load-Store-Architektur vs. Register-Memory-Architecture

Dynamische Codegenerierung

Übersetzerfunktionen (2)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Notwendigkeit von Fallunterscheidungen, z.B. `REM`: (Semantik nach¹, S. 44f.)

¹**riscv-spec.**

Dynamische Codegenerierung

Übersetzerfunktionen (2)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Notwendigkeit von Fallunterscheidungen, z.B. REM: (Semantik nach¹, S. 44f.)

```
mov rax, [r_info.base + 8 * instr.reg_src_1]
cmp qword ptr [r_info.base + 8 * instr.reg_src_2], 0
jnz not_div_zero
mov [r_info.base + 8 * instr.reg_dest], rax
jz div_zero
```

```
not_div_zero:
xor rdx, rdx
idiv qword ptr [r_info.base + 8 * instr.reg_src_2]
mov [r_info.base + 8 * instr.reg_dest], rdx
```

```
div_zero:
```

¹riscv-spec.

Dynamische Codegenerierung

Übersetzerfunktionen (3)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Emulierung der system calls für ECALL:

Dynamische Codegenerierung

Übersetzerfunktionen (3)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Emulierung der system calls für ECALL:

```
void translate_ECALL(const t_risc_instr &instr, const register_info &r_info) {  
    save_risc_registers(r_info);  
    a->mov(x86::rdi, instr.addr);  
    a->mov(x86::rsi, r_info.base);  
    typedef void emulate(t_risc_addr addr, t_risc_reg_val *registerValues);  
    emulate *em = &emulate_ecall;  
    a->call(reinterpret_cast<uintptr_t>(em));  
}
```


Dynamische Codegenerierung

Übersetzerfunktionen (3)

Realisierung der RISC-V-Instruktionen mit x86-64-Assembly.

Emulierung der system calls für ECALL:

```
void translate_ECALL(const t_risc_instr &instr, const register_info &r_info) {  
    save_risc_registers(r_info);  
    a->mov(x86::rdi, instr.addr);  
    a->mov(x86::rsi, r_info.base);  
    typedef void emulate(t_risc_addr addr, t_risc_reg_val *registerValues);  
    emulate *em = &emulate_ecall;  
    a->call(reinterpret_cast<uintptr_t>(em));  
}
```

- Behandlung von system calls zur Laufzeit
- Übersetzung, Adaptieren bzw. Emulieren der benötigten Funktionalität

Code Cache

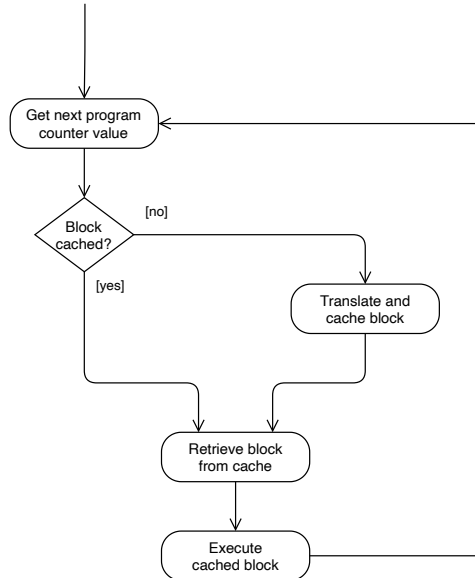
Überblick

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)

Code Cache

Überblick

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)



Code Cache

Ansatz

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)

Idee: Hashtable für schnellen Lookup der Blöcke, Startadresse des RISC-V-Blocks als Key

Code Cache

Ansatz

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)

Idee: Hashtable für schnellen Lookup der Blöcke, Startadresse des RISC-V-Blocks als Key

Einträge speichern RISC-V-Blockstartadresse sowie die Adresse des übersetzten Blocks:

```
typedef struct {  
    t_risc_addr risc_addr;  
    t_cache_loc cache_loc;  
} t_cache_entry;
```

Code Cache

Ansatz

Ziel: Caching bereits übersetzter Basic Blocks für nochmalige Ausführung (teure Übersetzung nur einfach)

Idee: Hashtable für schnellen Lookup der Blöcke, Startadresse des RISC-V-Blocks als Key

Einträge speichern RISC-V-Blockstartadresse sowie die Adresse des übersetzten Blocks:

```
typedef struct {  
    t_risc_addr risc_addr;  
    t_cache_loc cache_loc;  
} t_cache_entry;
```

Lookup als Open Hashing mit linearem Sondieren, via

```
inline size_t hash(t_risc_addr risc_addr) {  
    return (risc_addr & 0x0000FFF0u) >> 4u;  
}
```

Code Cache

Einsatz im System

Zugriff auf den Cache von außen via

```
t_cache_loc lookup_cache_entry(t_risc_addr risc_addr);  
void set_cache_entry(t_risc_addr risc_addr, t_cache_loc cache_loc);
```

wobei UNSEEN_CODE von `lookup_cache_entry(...)` einen nicht im Cache enthaltenen Block anzeigt.

Code Cache

Einsatz im System

Zugriff auf den Cache von außen via

```
t_cache_loc lookup_cache_entry(t_risc_addr risc_addr);  
void set_cache_entry(t_risc_addr risc_addr, t_cache_loc cache_loc);
```

wobei UNSEEN_CODE von `lookup_cache_entry(...)` einen nicht im Cache enthaltenen Block anzeigt.

→ Dynamische Reallokation der Größe bei Kapazitätsgrenzen

Code Cache

Einsatz im System

Zugriff auf den Cache von außen via

```
t_cache_loc lookup_cache_entry(t_risc_addr risc_addr);  
void set_cache_entry(t_risc_addr risc_addr, t_cache_loc cache_loc);
```

wobei UNSEEN_CODE von lookup_cache_entry(...) einen nicht im Cache enthaltenen Block anzeigt.

→ Dynamische Reallokation der Größe bei Kapazitätsgrenzen

Ausführung bereits übersetzter Blöcke via

```
typedef void (*blk)(void);  
((blk) loc)();
```

Literaturverzeichnis