

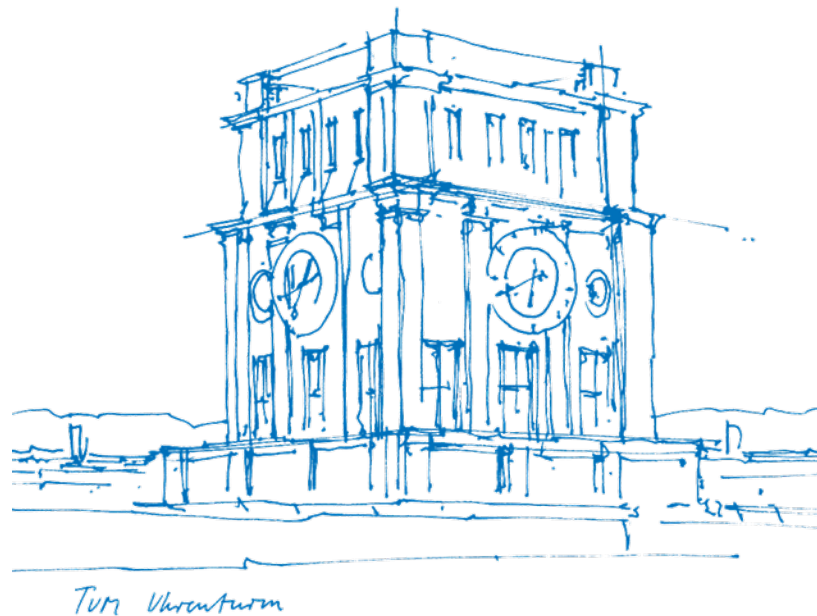
Dynamische Binärübersetzung: RISC-V \rightarrow x86-64

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

Dynamische Binärübersetzung

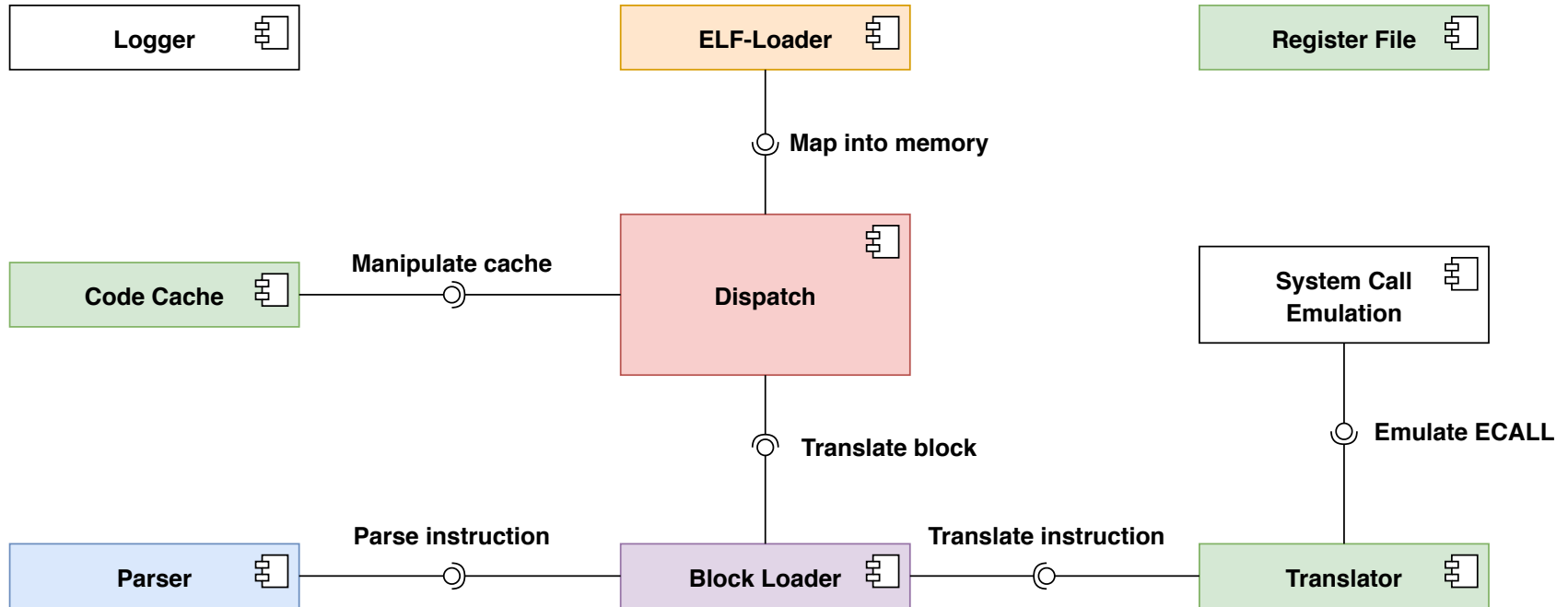
Ideen: - warum dynamic? - vorteil - nachteil

Einführung

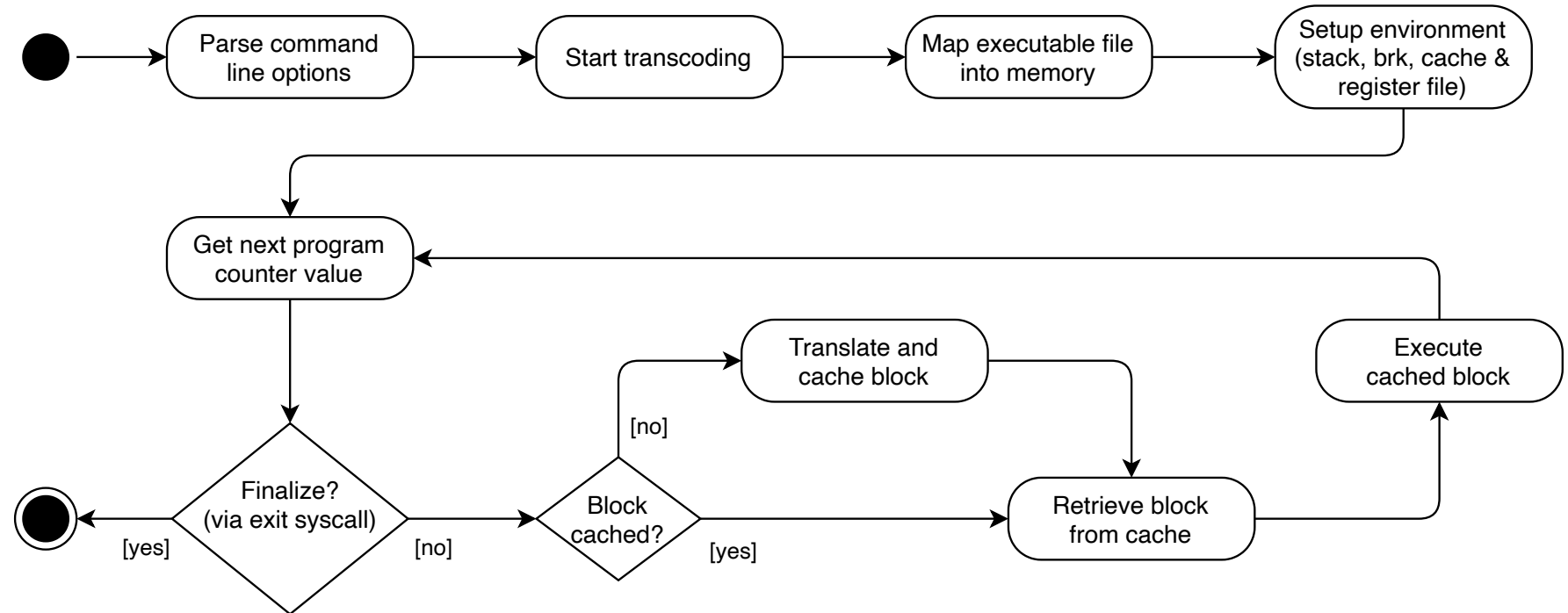
Grobüberblick über die RISC-V ISA

- x86-64: CISC → complex instruction set
 - Register-Memory-Architektur
 - 16 GPRs
 - 2-Adressform
- RISC-V: RISC → reduced instruction set
 - Load-Store-Architektur
 - 31 GPRs
 - 3-Adressform
 - spezielles Zero-Register

Systemarchitektur



Programmablauf



ELF-Loader

Überblick

Aufgabe: Speicher für das Gastprogramm anlegen und mit Startwerten, Instruktionen, etc. befüllen

Aufgeteilt in:

- Memory mapping
- Stack Allocation

ELF-Loader

Memory mapping

Ziel: Die Binärdatei einlesen und alle Segmente, die mit „load“ gekennzeichnet sind, an die korrekten Orte im Speicher laden.

Input: String des Pfads der Binärdatei.

Output: Einsprungsadresse des geladenen RISC-V-Programms (plus einige Metadaten) in einem `t_risc_elf_map_result` struct.

```
typedef struct {  
    bool valid;  
    t_risc_addr entry;  
    t_risc_addr phdr;  
    Elf64_Half ph_count;  
    Elf64_Half ph_entsize;  
    t_risc_addr dataEnd;  
} t_risc_elf_map_result;
```


ELF-Loader

Memory mapping

Ziel: Die Binärdatei einlesen und alle Segmente, die mit „load“ gekennzeichnet sind, an die korrekten Orte im Speicher laden.

Input: String des Pfads der Binärdatei.

Output: Einsprungsadresse des geladenen RISC-V-Programms (plus einige Metadaten) in einem `t_risc_elf_map_result` struct.

- Laden des ELF-Headers vom Anfang der Datei.
- Checken der Flags auf nicht unterstützte RISC-V ABIs
- Iterieren über alle Segment-Header um den Addressbereich des Binärs zu erhalten
- Allokieren des gesamten benötigten Addressbereichs an der nativen Adresse
- Laden aller „load“ Segmente an die richtigen Speicheradressen

```
typedef struct {  
    bool valid;  
    t_risc_addr entry;  
    t_risc_addr phdr;  
    Elf64_Half ph_count;  
    Elf64_Half ph_entsize;  
    t_risc_addr dataEnd;  
} t_risc_elf_map_result;
```

ELF-Loader

Stack Allocation

Ziel: Stack für das Gastprogramm allozieren und mit den üblichen Daten initialisieren.

Input: Argumentanzahl des Gastprogramms, Argumentarray des Gastprogramms, der Output des memory mappings

Output: Die Adresse des Stackpointers nach dem Initialisieren

ELF-Loader

Stack Allocation

Ziel: Stack für das Gastprogramm allozieren und mit den üblichen Daten initialisieren.

Input: Argumentanzahl des Gastprogramms, Argumentarray des Gastprogramms, der Output des memory mappings

Output: Die Adresse des Stackpointers nach dem Initialisieren

- Allozieren von Speicher für das Stack entsprechend des stack limits des Kernels (oder 8 MiB als Standardwert)

ELF-Loader

Stack Allocation

Ziel: Stack für das Gastprogramm allozieren und mit den üblichen Daten initialisieren.

Input: Argumentanzahl des Gastprogramms, Argumentarray des Gastprogramms, der Output des memory mappings

Output: Die Adresse des Stackpointers nach dem Initialisieren

- Guard page am unteren Ende des Stacks um Overflow abzufangen.
- Alignment, damit der resultierende Stack Pointer die ABI erfüllt (16 Byte aligned)
- Kopieren der Werte des auxiliary vectors (aus Resultat des memory mappings bzw. Daten des Hostprogramms)
- Kopieren des environment vectors des Hostprogramms
- Kopieren des Argumentarrays und der Argumentanzahl

⋮ (Nicht Teil des Stacks)
Stack Alignment
Null terminierter auxiliary vector
Null terminierter environment vector
Null terminierter Argumentarray
Argumentanzahl (Stack Pointer am Ende auf hier)
⋮
Guard page

Parser

Überblick

Ziel: Dekodieren der 4 Byte großen kodierten 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

Parser

Überblick

Ziel: Dekodieren der 4 Byte großen kodierten 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

```
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;
```

- `addr` → Adresse des originalen riscv-Befehls im Speicher
- `mnem` → Mnemonic der Instruktion (Zusammengesetzt aus Opcode und weiteren *funct* Blöcken)
- `optype` → Optype (Kategorie der Instruktion)
- `reg_(dest/src1/src2)` → Registernummern der Quell- und Ziel Register
- `imm` → Immediate der Instruktion

Parser

Extrahieren der Blöcke

Ziel: Dekodieren der 4 Byte großen kodierten RISC-V-Befehle aus der geladenen ELF-Datei.

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

Parser

Extrahieren der Blöcke

Ziel: Dekodieren der 4 Byte großen kodierten RISC-V-Befehle aus der geladenen ELF-Datei.

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

Um

die einzelnen Teile der Instruktion zu extrahieren verwenden wir kompakte inline Funktionen.

```
// extract U-Type immediate bit[31:12] -> mask lower 12 bit [11:0] with zeros
static inline int extract_imm_U(int32_t instr) { return instr & ~(0xfff); }
```


Parser

Umsetzung

Ziel: Dekodieren der 4 Byte großen kodierten RISC-V-Befehle aus der geladenen ELF-Datei.

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

Zunächst wird der Opcode extrahiert, dieser legt das Format (R/I/S/B/U/J-type) fest.

Teilweise lässt sich die Mnemonic schon exakt aus dem Opcode auslesen (U/J-type).

```
int32_t raw_instr = *(int32_t *) p_instr_struct->addr; //cast and dereference
p_instr_struct->reg_dest = extract_rd(raw_instr); //fill basic struct
t_opcodes opcode = raw_instr >> 2 & 0x1f; //extract opcode bits[6:2]
switch (opcode) {
    case OP_LUI:
        p_instr_struct->optype = UPPER_IMMEDIATE;
        p_instr_struct->mnem = LUI;
        p_instr_struct->imm = extract_imm_U(raw_instr);
        break;
    //...
}
```

Parser

Umsetzung

Ziel: Dekodieren der 4 Byte großen kodierten RISC-V-Befehle aus der geladenen ELF-Datei.

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

Bei einigen Instruktionen muss zusätzlich zwischen den *funct* Codes unterschieden werden.

```
case OP_OP_IMM_32:
    p_instr_struct->optype = IMMEDIATE;
    switch (extract_funct3(raw_instr)) {
        case 0:
            p_instr_struct->mnem = ADDIW;
            p_instr_struct->imm = extract_imm_I(raw_instr);
            break;
        //...
    }
    break;
```

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 Zugriffsmethoden:

```
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.

Block Translator

Überblick

Aufgabe: Basic Blocks parsen, übersetzen, und im Cache ablegen

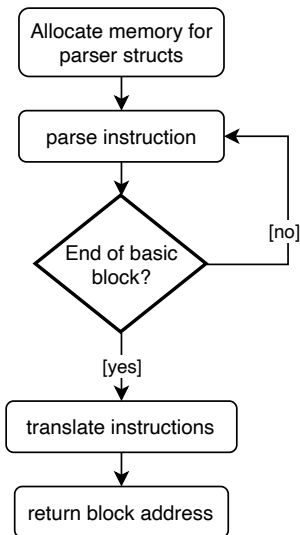
```
t_cache_loc translate_block(t_risc_addr risc_addr)
```

Block Translator

Überblick

Aufgabe: Basic Blocks parsen, übersetzen, und im Cache ablegen

```
t_cache_loc translate_block(t_risc_addr risc_addr)
```



Block Translator

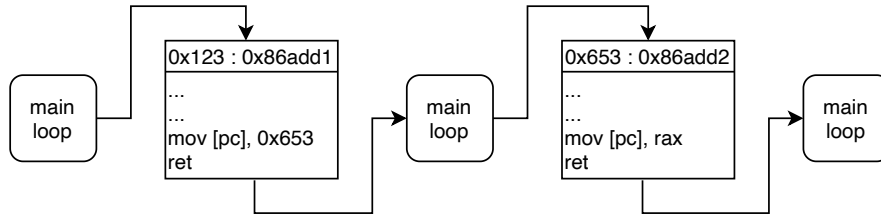
Chaining

Ziel: weniger Overhead durch main loop und Cache Lookup

Block Translator

Chaining

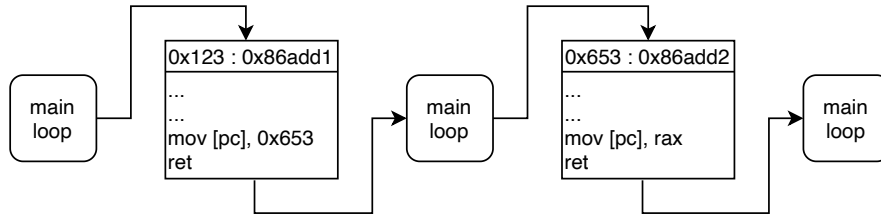
Ziel: weniger Overhead durch main loop und Cache Lookup Ohne Chaining:



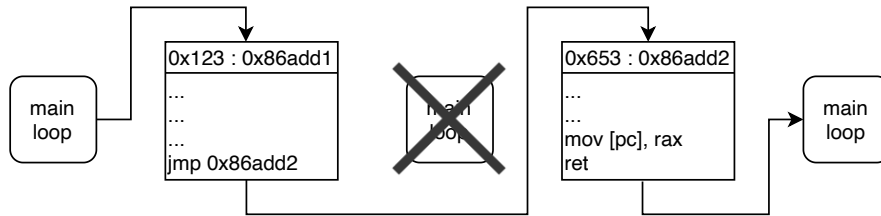
Block Translator

Chaining

Ziel: weniger Overhead durch main loop und Cache Lookup Ohne Chaining:



Mit Chaining:



Block Translator

Chaining durch recursive block translation

Block Translator

Chaining durch recursive block translation

- Bereits beim Parsen die Sprungziele rekursiv übersetzen
- und in den Cache legen.
- Bis jetzt nur für JAL

- direkter Sprung im Branch wenn Ziel bereits im Cache

<2>

```
if (block_cache[parse_pos].reg_dest !=  
    t_risc_reg::x0) {  
    instructions_in_block++;  
    t_risc_addr target = risc_addr +  
        block_cache[parse_pos].imm;  
    t_cache_loc cache_loc =  
        lookup_cache_entry(target);  
    if (cache_loc == UNSEEN_CODE) {  
        //flag?  
        cache_loc = translate_block(target);  
        set_cache_entry(target, cache_loc);  
    }  
    goto PARSE_DONE;  
}
```

<3>

Dynamische Codegenerierung

Überblick

Input: gepackte RISC-V-Instruktionen eines Basic Blocks

Output: übersetzte x86-Instruktionen für diesen Block

Dynamische Codegenerierung

Überblick

Input: geparte 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 {
    FeReg *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 {
        err |= fe_enc64(&current, FE_MOV64rm, FE_AX, FE_MEM_ADDR(r_info.base + 8 * instr.reg_src_1));
        err |= fe_enc64(&current, FE_ADD64rm, FE_AX, FE_MEM_ADDR(r_info.base + 8 * instr.reg_src_2));
        err |= fe_enc64(&current, FE_MOV64mr, FE_MEM_ADDR(r_info.base + 8 * instr.reg_dest), FE_AX);
    }
}
```

→ 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.)

¹A. Waterman u. a. (2017). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Version 2.2.*

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:
```

¹A. Waterman u. a. (2017). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Version 2.2.*

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);  
    err |= fe_enc64(&current, FE_MOV64ri, FE_DI, instr.addr);  
    err |= fe_enc64(&current, FE_MOV64ri, FE_SI, r_info.base);  
    typedef void emulate(t_risc_addr addr, t_risc_reg_val *registerValues);  
    emulate *em = &emulate_ecall;  
    err |= fe_enc64(&current, FE_CALL, reinterpret_cast<uintptr_t>(em));  
}
```

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

Code Cache

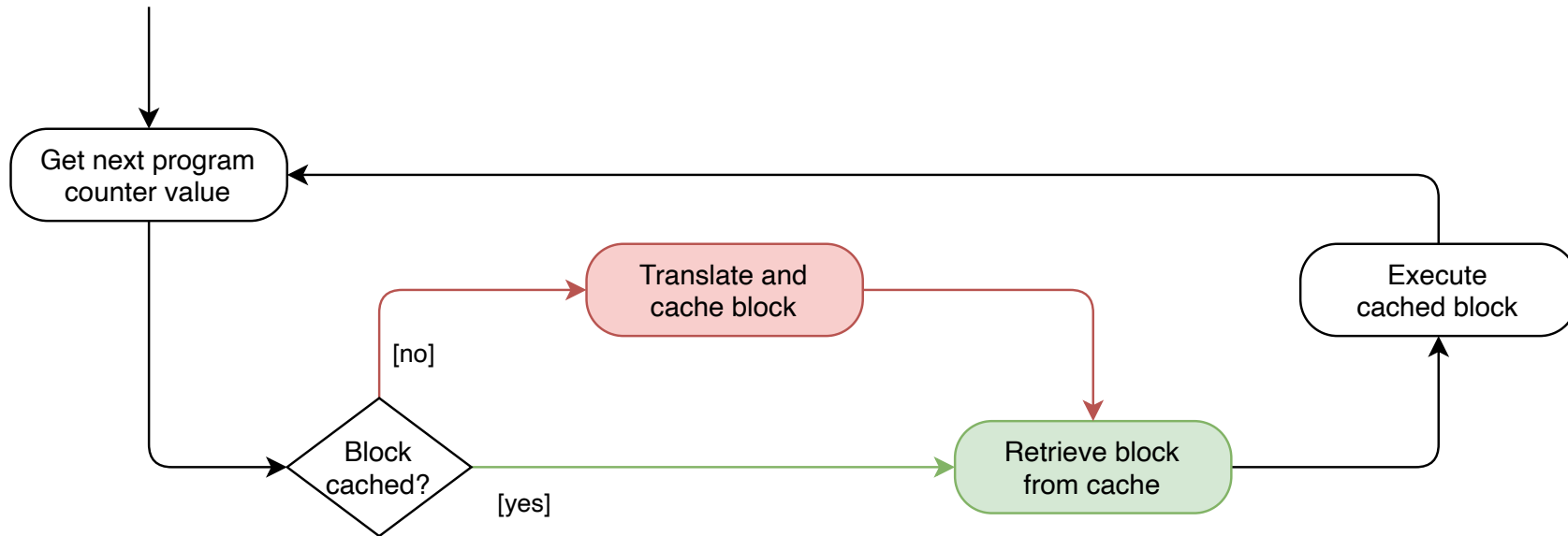
Überblick

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 & 0x00003FFCu) >> 2u;  
}
```

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

 [Waterman, A. u. a. \(2017\)](#). *The RISC-V Instruction Set Manual. Volume I: User-Level ISA. Version 2.2.*