

# Just-In-Time-Compiler (1)

Dr.-Ing. Volkmar Sieh

Department Informatik 4  
Verteilte Systeme und Betriebssysteme  
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2021/2022



Assemblerbefehle tun sehr viel mehr, als man im ersten Moment denkt. Z.B. `addl %eax, 4(%esp)`:

- Test, ob überhaupt Strom vorhanden ist
- Test, ob der letzte Befehl eine Exception ausgelöst hat
- Test, ob ein System-Management-Interrupt vorliegt
- Test, ob ein Interrupt vorliegt
- Test, ob die CPU im HALT-Zustand ist



- Holen der Instruktion
  - Berechnung der linearen Adresse oder „Segmentation Violation“
  - Berechnung der physikalischen Adresse oder „Page Fault“
  - Holen der Instruktion oder „Bus Error“
- Dekodieren der Instruktion; eventuell weitere Teile der Instruktion holen (s.o.)



- Holen der Parameter
  - Berechnung der virtuellen Adresse
  - Berechnung der linearen Adresse oder „Segmentation Violation“
  - Berechnung der physikalischen Adresse oder „Page Fault“
  - Holen der Parameter oder „Bus Error“
- Addieren
  - eigentliches Additionsergebnis
  - Flags (Carry, Overflow, Parity, Zero, ...)
- Schreiben des Ergebnisses oder „Page Fault“
- Inkrementieren des IP



Beispiel %eflags-Berechnung add-Befehl:

```
int cf, pf, af, zf, sf, of;
long src1, src2;

src1 = CC_SRC;
src2 = CC_DST - CC_SRC;
cf = (DATA_TYPE) CC_DST < (DATA_TYPE) src1;
pf = parity_table[(uint8_t) CC_DST];
af = (CC_DST ^ src1 ^ src2) & 0x10;
zf = ((DATA_TYPE) CC_DST == 0) << 6;
sf = lshift(CC_DST, 8 - DATA_BITS) & 0x80;
of = lshift((src1 ^ src2 ^ -1) & (src1 ^ CC_DST),
            12 - DATA_BITS) & CC_0;
return cf | pf | af | zf | sf | of;
```



Emulation wie bisher vorgestellt ist (sehr) langsam.

Hauptproblem: CPU (siehe letzte Folien)

**Idee:**

Statt z.B. in Schleifen wieder und wieder den auszuführenden Code zu holen und zu interpretieren, lieber beim ersten Holen compilieren und compilierten Code wieder und wieder ausführen.

*Just-In-Time Compiler* (JIT-Compiler)



Probleme:

- Holen und Dekodieren der Instruktionen macht nur Bruchteil der Laufzeit einer Instruktion aus
- Suchen der schon compilierten Instruktionen kostet auch Zeit

**Idee:**

Ganzen Block von Instruktionen am Stück compilieren.



Vorteil: Die Tests,

- ob Strom vorhanden ist,
- ob ein System-Management-Interrupt vorliegt,
- ob ein Interrupt vorliegt,
- ob die CPU im HALT-Zustand ist,

müssen nur einmal am Blockanfang durchgeführt werden.



- Holen und Dekodieren der Instruktionen

wird nur während des Compilierens einmal durchgeführt.

- Flags

müssen nur berechnet werden, wenn nachfolgende Instruktionen sie nicht vor einer Verwendung wieder überschreiben.

- Inkrementieren des IP

muss nur einmal am Block-Ende durchgeführt werden.



## Standard-Emulation:

```
void
cpu_step() {
    if (! power)
        ; /* Do nothing... */
    else if (int_pending)
        cpu_exec_interrupt();
    else
        cpu_exec_instruction();
}
```

## JIT-Variante:

```
void
cpu_step() {
    if (! power)
        ; /* Do nothing... */
    else if (int_pending)
        cpu_exec_interrupt();
    else {
        void (*block)(void);
        block = lookup(IP);
        if (! block)
            block = compile(IP);
        (*block)();
    }
}
```



Probleme:

- Code muss u.U. auch wieder invalidiert werden. Z.B.:
  - bei Schreiboperationen in den Code-Bereich
  - beim dynamischen (Ent-) Laden von Programmen/Treibern
  - beim Umprogrammieren der MMU
- Code-Cache ist nur begrenzt groß; Code im Cache muss verworfen werden können.
- Code im Cache muss *sehr* schnell über den IP wiedergefunden werden können.



Probleme der Methode:

- Exceptions lassen (eigentlich) keine linearen Blöcke zu.
- Instruktion-Hol- und Daten-Verarbeitungs-Exceptions kommen u.U. in anderer Reihenfolge
- Interrupts müssen (eigentlich) vor *allen* Instruktionen abgefragt werden.
- berechnete Sprünge und Return-Anweisungen begrenzen Block-Länge
- andere Teile des Rechners müssen auch weiter emuliert werden
- Blöcke i.A. sehr unterschiedlich lang => komplizierte Code-Cache-Verwaltung



„Exceptions lassen keine linearen Blöcke zu.“

Rückkehr in die Hauptschleife bei Exceptions kann das Problem lösen.

Aber:

Wird ein Block vorzeitig verlassen, greifen einige Optimierungen nicht mehr:

- Setzen des IPs am Ende es Blockes nicht mehr ausreichend.
- Berechnung der CC-Flags ausschliesslich vor Instruktionen, die die Werte benötigen, nicht mehr ausreichend.

Wegoptimierte Instruktionsteile müssen vor dem Zurückkehren aus einem compilierten Block ggf. nachgeholt werden.



## Statt

```
A:  
  InstA;  
  FLAGS = ...;  
  IP = ...;  
  return;  
  
B:  
  InstB;  
  FLAGS = ...;  
  IP = ...;  
  return;
```

## Besser

```
AB:  
  InstA;  
  if (exception_pending) {  
    FLAGS = ...;  
    IP = ...;  
    return;  
  }  
  InstB;  
  FLAGS = ...;  
  IP = ...;  
  return;
```

Der Test `if (exception_pending)` kann entfallen, wenn `InstA` keine Exceptions generieren kann.



„Instruktion-Hol- und Daten-Verarbeitungs-Exceptions kommen u.U. in anderer Reihenfolge“

Beispiel:

```
movl $0, %ebx
idivl %ebx, %eax
ret
```

Werden Instruktionen der Reihe nach geholt und ausgeführt, käme im Beispiel erst ein Division-durch-Null-Fehler beim Ausführen der zweiten Instruktion.

Werden erst alle Instruktionen geholt und dann alle ausgeführt, könnte im Beispiel zunächst ein Page-Fault beim Holen der Instruktion `ret` kommen, wenn z.B. `ret` auf einer neuen Page liegt.



„Instruktion-Hol- und Daten-Verarbeitungs-Exceptions kommen u.U. in anderer Reihenfolge“

Aber:

Moderne/große CPUs holen Instruktionen frühzeitig um sie mit Out-of-Order-Execution auszuführen (Superskalar-CPU's). Dabei können ähnliche Effekte auftreten.

Alte/kleine CPUs kennen keine Page-Faults. Dann kann der Effekt auch nicht auftreten.

Problematisch: „mittlere“ CPUs.

Ähnliches Problem: Setzen der Access-/Dirty-Bits in der MMU.



„Interrupts müssen vor *allen* Instruktionen abgefragt werden.“

Möglich wäre:

```
AB:
  InstA;
  if (interrupt_pending) {
    FLAGS = ...;
    IP = ...;
    return;
  }
  InstB;
  FLAGS = ...;
```

(Zu) Großer Aufwand!



„Interrupts müssen vor *allen* Instruktionen abgefragt werden.“

Aber:

Niemand wird es merken, wenn *externe* Interrupts um wenige Takte/Instruktionen verzögert werden.

Es reicht i.A. vor der Ausführung eines Code-Blockes Interrupt-Requests zu behandeln.

Zur Zeit ungelöst: von der CPU selbst ausgelöste Interrupts (z.B. System-Management-Interrupts (SMI) oder Inter-Processor-Interrupts (IPI)).

**Vorsicht:**

=> Blöcke dürfen keine (Endlos-) Schleifen enthalten!



„Berechnete Sprünge und Return-Anweisungen begrenzen Block-Länge.“

Beispiele:

```
movl ..., %eax  
jmp *%eax
```

```
pushl ...  
ret
```

Der berechnete Sprung bzw. der `ret`-Befehl begrenzen die mögliche Block-Länge. Die Befehle vor dem `jmp` bzw. vor dem `ret` müssen ausgeführt werden, damit der nachfolgende Befehl den neuen IP setzen kann. *Erst dann* steht die nächste auszuführende Instruktion fest.

=> Funktionen sind i.A. die maximalen Blöcke!



Wie lang sollte ein übersetzter Block sein?

Möglichkeiten:

- immer  $N$  Instruktionen
- jeweils eine ganze Unterfunktion
- alle Instruktionen bis zum nächsten Sprung
- ...
- oder Kombinationen davon

Um Interrupt-Request zeitlich einigermaßen richtig zu emulieren, dürfen Blöcke nicht zu lang sein und keine Schleifen enthalten.



Blocklänge QEMU / FAUmaschine:

- alle Instruktionen bis zum nächsten Sprung  
aber
- maximal 4096 Bytes



## Funktion:

```
func:
    movl $1, %eax
    movl $1, %ecx
    jmp .L2
.L1:
    imull %ecx, %eax
    incl %ecx
.L2:
    cmpl 4(%esp), %ecx
    jne .L1
.L3:
    ret
```

## 1. Block:

```
func:
    movl $1, %eax
    movl $1, %ecx
    jmp .L2
```

## 2. Block:

```
.L2:
    cmpl 4(%esp), %ecx
    jne .L1
.L3:
```

## 3. Block:

```
.L1:
    imull %ecx, %eax
    incl %ecx
    cmpl 4(%esp), %ecx
    jne .L1
.L3:
```

## 4. Block

```
.L3:
    ret
```



## 1. Block

```
func :  
    movl $1, %eax  
    movl $1, %ecx  
    jmp  .L2
```

## JIT-Übersetzung 1. Block

```
func :  
    movl $1, reg_eax  
    movl $1, reg_ecx  
    movl $.L2, reg_eip  
    ret
```



## 2. Block

```
.L2:  
  cmpl 4(%esp), %ecx  
  jne .L1  
.L3:
```

## JIT-Übersetzung 2. Block

```
.L2:  
  movl reg_esp, %eax  
  addl $4, %eax  
  ...  
  movb ..., reg_zero_flag  
  cmpb $1, reg_zero_flag  
  jne label  
  movl $.L3, reg_eip  
  ret  
label:  
  movl $.L1, reg_eip  
  ret
```



Beobachtung:

- Blöcke sind häufig recht kurz

Der 3. Block könnte wie folgt umgebaut werden (Loop-Unrolling):

```
.L1:  
    imull %ecx, %eax  
    incl %ecx  
    cmpl 4(%esp), %ecx  
    jne .L1  
.L3:
```

```
.L1:  
    imull %ecx, %eax  
    incl %ecx  
    cmpl 4(%esp), %ecx  
    je .L3  
    imull %ecx, %eax  
    incl %ecx  
    cmpl 4(%esp), %ecx  
    je .L3  
    ...  
    imull %ecx, %eax  
    incl %ecx  
    cmpl 4(%esp), %ecx  
    jne .L1  
.L3:
```



Um wiederholtes Suchen und Starten von Blöcken zu vermeiden, kann man versuchen, die einzeln compilierten Blöcke zu verbinden.

normales Block-Ende	unverbindbares Block-Ende	verbindbares Block-Ende
<pre>movl \$const, IP ret</pre>	<pre>movl var, IP movl \$0, %eax ret</pre>	<pre>.L1:   jmp .L2 .L2:   movl \$const, IP   movl \$.L1, %eax   ret</pre>

`jmp .L2` kann durch einen Sprung direkt zum nächsten Block-Anfang ersetzt werden.

Diese Methode verdoppelt die Emulationsgeschwindigkeit!



Das Verbinden von Blöcken führt i.A. zu Schleifen. D.h.:

- Interrupt-Requests werden nicht mehr abgearbeitet.
- Andere Rechner-Komponenten werden nicht mehr weiteremuliert.

Lösungsmöglichkeiten:

- **QEMU:** Kette wird im Signal-Handler (Interrupt) unterbrochen (gute Performance).
  - Andere Rechner-Komponenten reagieren nur während in- und out-Befehlen der CPU.

- **FAUmachine:** Blöcke zählen Instruktion-Count herunter. Wenn 0 erreicht wird, wird zur Hauptschleife zurückgekehrt (deterministisch).
  - Andere Rechner-Komponenten können dann weitersimuliert werden.



Problem:

Moderne CPUs kennen  $>1000$  Instruktionen mit ggf. jeweils mehreren Operanden, die wiederum  $>10$  verschiedene Adressierungsarten verwenden können.

=>

Es existieren *sehr* viele verschiedene Instruktionen.

=>

Vorhaltung von allen vorcompilierten Instruktionen unmöglich.

=>

Unterteilung der Instruktionen in Mikro-Instruktionen.



Mikro-Instruktionen Beispiel:

```
add %eax, 4(%esp)
```

ersetzen durch

```
mov %esp, %tmp2  
add $4, %tmp2  
load (%tmp2), %tmp0  
mov %eax, %tmp1  
add %tmp1, %tmp0  
store %tmp0, (%tmp2)
```

Entspricht im Wesentlichen RISC-Befehlen.



## Mikro-Instruktionen

- sind einfach
  - mov-Befehle, die zwischen %tmp- und Architektur-Registern Daten kopieren
  - load- und store-Befehle
  - Arithmetik-Befehle auf %tmp-Registern
- müssen aber parametrisierbar sein (z.B. `add $imm, %reg`)

Beispiel FAUmachine: x86\_64: 1352 Mikro-Instruktionen



- parametrisierbare Mikro-Instruktionen vorcompiliert vorhalten
- Compile-Funktion „klebt“ dann nur noch vorcompilierte Code-Schnipsel zusammen, setzt Parameter ein und fügt ein `ret` an.

Beispiel FAUmachine: Compile-Funktion sehr komplex  
(ca. 7000 Code-Zeilen)



Problem: Wie generiert man für ca. 1300 Mikro-Instruktionen vorcompilierte Code-Schnippel?

Idee QEMU: Mikro-Instruktionen als C-Funktionen programmieren, vom C-Compiler übersetzen und durch ein kleines Tool anpassen lassen.

Anpassung: ret am Ende der Funktion weglassen.



# JIT – Code-Generierung – Beispiel 1

---

Beispiel:

Mikro-Instruktion:

```
add %eax, %tmp0
```

C-Code:

```
void add_eax_tmp0(struct state *state) {  
    state->tmp0 += state->eax;  
}
```



C-Compiler generierter Code:

```
add_eax_tmp0:  
    movl 4(%esp), %eax  
    movl off_eax(%eax), %edx  
    addl %edx, off_tmp0(%eax)  
    ret
```

Vom Tool angepasster Code:

```
add_eax_tmp0_begin:  
    movl 4(%esp), %eax  
    movl off_eax(%eax), %edx  
    addl %edx, off_tmp0(%eax)  
add_eax_tmp0_end:
```



Untersucht man die Code-Schnipsel genauer, merkt man, dass

- die Zeile `movl 4(%esp), %eax` ähnlich in jedem Code-Schnipsel vorkommt
- drei temporäre Register sehr(!) häufig gebraucht werden.

Idee (QEMU): state-Variable und die temporären Register als globale Register-Variablen deklarieren.



# JIT – Code-Generierung – Beispiel 1

Beispiel mit globalen Register-Variablen (GNU-C-Extension):

Mikro-Instruktion:

```
add %eax, %tmp0
```

C-Code:

```
struct state *state asm("ebp");
int tmp0 asm("edi");
int tmp1 asm("esi");

add_eax_tmp0() {
    tmp0 += state->eax;
}
```



Vom C-Compiler generierter Code:

```
add_eax_tmp0:  
    addl off_eax(%ebp), %edi  
    ret
```

Vom Tool angepasster Code:

```
add_eax_tmp0_begin:  
    addl off_eax(%ebp), %edi  
add_eax_tmp0_end:
```

Eine Mikro-Instruktion entspricht einer realen Assembler-Instruktion!



Problem: Compiler optimiert. Beispiel:

Mikro-Instruktion

```
struct state *state asm("ebp");
int tmp0 asm("edi");
int tmp1 asm("esi");

div_tmp1_tmp0() {
    if (tmp1 == 0)
        state->exception_pending = 1;
    else
        tmp0 /= tmp1;
}
```



Vom C-Compiler generierter Code:

```
div_tmp1_tmp0:  
  cmpl $0, %esi  
  jne .L1  
  movl $1, off_exception_pending(%ebp)  
  ret                                <<<----- Problem!  
.L1:  
  divl %esi, %edi  
  ret
```



Anderer vom C-Compiler generierter Code:

```
div_tmp1_tmp0:
  cml $0, %esi
  jne .L2
  movl $1, off_exception_pending(%ebp)
.L1:
  ret <<<----- Problem!
.L2:
  divl %esi, %edi
  jmp .L1 <<<----- Problem!
```



Anderer vom C-Compiler generierter Code:

```
foo:
  jmp bar                <<<----- Problem!
```

statt

```
foo:
  call bar
  ret
```

Weitere ähnliche Probleme!

FAUmachine/QEMU: „gelöst“ durch das Ausschalten entsprechender Compiler-Optimierungen.



Vor dem Zusammenkleben der Schnipsel vom JIT-Compiler nicht benötigte Schnipsel entfernen. Beispiel:

```
addl %eax, %edx    movl %edx, %tmp0
                   movl %eax, %tmp1
                   addl %tmp1, %tmp0
                   call eflags_calculation_add <<<--- Omit!
cmp1 $0, %eax      movl %tmp0, %edx
                   movl %eax, %tmp0
                   movl $0, %tmp1
                   subl %tmp1, %tmp0
                   call eflags_calculation_sub
jne .L1            ...
```



Manche Instruktionen können zusammen optimiert werden. Beispiel:

```
    cmp1 $0, %eax
    jne .L1
    add1 $1, %eax
    jmp .L2
.L1:
    sub1 $1, %eax
.L2:
    ...
```

- jne-Instruktion braucht nur Zero-Flag
- add- und sub-Instruktionen berechnen Flags neu

=> cmp1-Instruktion braucht nicht alle Flags berechnen.



Für die zusammengeklebten Code-Schnipsel wird Speicherplatz benötigt.

Probleme:

- Es ist unklar, wieviel Platz für einen Code-Block benötigt wird.
- Neue Code-Blöcke kommen hinzu.
- Alte Code-Blöcke müssen invalidiert werden.
- Block-Allozieren/-Freigeben muss *sehr* schnell sein.

Maximale Code-Block-Größe kann offline berechnet/abgeschätzt werden.



- Es existiert *ein* großer Puffer.
- Neue Code-Blöcke werden einfach an bestehende angehängt.
- Invalidierte Blöcke verbleiben einfach im Puffer.
- Wenn kein maximal-großer Code-Block mehr in Puffer passt, werden *alle* Code-Blöcke invalidiert.
  
- Vorteil: Alloc/Free sehr schnell.
- Nachteil: Manche Blöcke werden unnötig invalidiert.



- Es existieren  $N$  Puffer, die jeweils einen maximal großen Code-Block aufnehmen können.
- Puffer werden in einer Round-Robin-Weise vergeben/invalidiert.
- Vorteil: Alloc/Free sehr schnell.
- Nachteil: Puffer sind i.A. unnötig sehr viel zu groß.



Wenn in einen Speicherbereich geschrieben wird, muss der dazugehörige pre-compilierte Code invalidiert werden.

Aufgabe der `virt_store`-Funktion:

- Test auf non-aligned Zugriff.
- Überprüfung ob pre-compilierter Code invalidiert werden muss.
- Lookup der physikalischen Adresse im TLB.
- Schreiben
  - direkt in Speicher
  - oder über Funktionsaufruf.

=> **sehr aufwändig!**

`virt_load`-Funktion ähnlich.



Idee QEMU:

```
virt_store32(uint32_t vaddr, uint32_t val) {
    unsigned int i = (vaddr >> 12) % TLB_SIZE;
    if (tlb[i].vaddr == (vaddr & 0xffff003)) {
        /* Fast Case */
        uint32_t *haddr = tlb[i].haddr + (vaddr & 0xfff);
        *haddr = val;
    } else {
        /* Slow Case */
        ...
    }
}
```

**Drei** Tests mit **einem** einfachen if!



- Fast Case:
- Virtuelle Adresse richtig (Bits 31-12)
  - Bits 11-2 nicht gesetzt
  - Alignment richtig (Bits 1-0)

Slow Case: sonst...

- Fast-Case bleibt „fast“.
- Slow-Case durch Setzen eines der Bits 11-2 erzwingbar. Direkter Schreibzugriff auf Pages mit Code damit verhinderbar. (ROMs damit z.B. auf read-only einstellbar!)



Direkte Schreibzugriffe auf eine Page nach dem Muster der letzten Folie verhindertbar.

Lesezugriffe damit aber auch verlangsamt.

=> Dreifache Trennung:

- Data-Read-TLB
- Data-Write-TLB
- Code-Execute-TLB



Problem Windows-XP: Interrupt-Handler:

```
.text
...
incl interrupt_count
...
iret
interrupt_count:
    /* <<<--- ".data" missing!!! */
    .long 0
```

Code und Daten in *einer* Page.



**Erkennung:** Wird Code sehr häufig invalidiert und wieder compiliert, liegt vermutlich obiger Fall vor (Zähler für jede Page).

**Lösung (FAUmachine/QEMU):** Für diese Page Bitmap mit 4096 Bits anlegen. Einzelne Bits zeigen an, ob das zugehörige Byte der Page Code enthält (1) oder Daten (0).  
Invalidierung dann byteweise möglich.



Moderne, von Compilern generierte Programme

- haben getrennte Code- und Daten-Pages
- haben read-only Code-Pages
- haben Daten mit richtigem Alignment

=> „Fast-Case“ ist in der Praxis der Normal-Fall.

- Bei Linux und \*BSD keine Ausnahme bekannt.
- Windows hat eine (wichtige!) Ausnahme (s.o.).
- DOS und viele DOS-Programme mit vielen Ausnahmen.



Invalidierung daher im Allgemeinen nur

- beim Terminieren/Nachladen von Programmen (selten),
- beim Kontext-Wechsel (häufig!).



Lösung für Kontextwechsel:

- Code *nicht* invalidieren!
- Code geschützt lassen.
- Code wiederverwenden, wenn wieder auf den Kontext zurückgeschaltet wird.

=> Extra Cache-Liste



Code wird z.B. bei FAUmachine/QEMU i.A. Page-weise invalidiert.

Einzelne Instruktionen können aber auf zwei Pages verteilt sein. (Nur bei CISC-Rechnern!)

Code-Blöcke könnten mehrere Pages umfassen.

=> Code-Länge für Blöcke auf 4KB begrenzen

=> Jeder Block erstreckt sich über max. 2 Pages.

=> Invalidierungsaufwand begrenzt.



Erfahrungen mit dem FAUmachine-Projekt:

- Hauptschleife braucht ca. 30% der Rechenzeit
- Compilieren braucht ca. 30% der Rechenzeit
- Lauf der compilierten Blöcke braucht ca. 30% der Rechenzeit
- Grafik-Aufbau braucht ca. 10% der Rechenzeit

