Just-In-Time-Compiler (2)

Department Informatik 4
Systemsoftware
Friedrich-Alexander-Universität Erlangen-Nürnberg

WS 2026/2027



Just-In-Time-Compiler (2)

Hauptproblem JIT:

Viele reale Assembler-Instruktionen lassen sich nur aufwändig emulieren.

Z.B. berechnen fast alle Arithmetik-Befehle die Condition-Codes/Flags neu. Gebraucht werden die Condition-Codes/Flags aber i.A. nur von bedingten Sprüngen.

=> **Idee:** neu entworfene, speziell auf JIT angepasste „Hardware"



Just-In-Time-Compiler (2)

Schwierig/aufwändig zu emulieren:

- Exception
- Interrupts
- Condition-Code-Flags
- MMU/Segmentierung
- berechnete Sprünge
- selbst-modifizierender Code



Just-In-Time-Compiler (2) – Exceptions

- nahezu jede Instruktion kann Fehler produzieren
- Fehler unterbrechen linearen Code-Block
- für einen Retry muss kompletter CPU-Zustand zum Fehlerzeitpunkt bekannt sein (CC-Flags, IP, Register, ...)

=> Instruktionen müssen praktisch komplett und der Reihe nach emuliert werden.

=> Exceptions sind schwierig zu emulieren.

=>

- Fehler vermeiden (z.B. keine MMU, Typ-sichere Pointer, ...)
- Retry einzelner Instruktionen nicht zulassen (statt dessen z.B. `try`/`catch`-Blöcke)



Just-In-Time-Compiler (2) – Interrupts

- Interrupts müssen vor Ausführung jeder Instruktion abgefragt werden. *Sehr* aufwändig.
- Interrupts müssen vor Ausführung jedes Blocks abgefragt werden. In kurzen Schleifen *sehr* aufwändig.

Idee:

- Interrupts weglassen
- nebenläufige Prozesse zulassen



Just-In-Time-Compiler (2) – Condition-Code-Flags

Auf echter Hardware werden die Condition-Code-Flags parallel zum Rechenergebnis mitberechnet (kein Zeit-Overhead).

Auf emulierter Hardware nicht parallel möglich => großer Aufwand.

In den meisten Fällen werden die Condition-Code-Flags jedoch von der nächsten Instruktion überschrieben. In der Praxis zu >95% nur zwischen `cmp`-Befehl und nachfolgendem bedingten Sprung gebraucht.

=> JIT-freundliche Hardware berechnet Condition-Codes nur mit `cmp`-Befehl. Alle anderen (Arithmetik-) Befehle brauchen Condition-Code-Flags nicht berechnen.



Just-In-Time-Compiler (2) – MMU

Auf realer Hardware dient MMU u.a. zum gegenseitigen Schutz verschiedener User-Prozesse.

- Schutz nicht notwendig, wenn nur eine Applikation läuft.
- Schutz auch möglich, wenn sichere Pointer verwendet werden:
  - Typ-Prüfung der Assembler-Instruktionen notwendig; Pointer dürfen nicht „berechnet" werden; damit Typ-Prüfung möglich ist, muss Code bestimmten Kriterien genügen
  - Garbage-Collection notwendig



Just-In-Time-Compiler (2) – berechnete Sprünge

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

```
movl ..., %eax
call *%eax
```

Problem: JIT kann hier nicht weiter-compilieren. %eax kann hier 2^{32} Werte annehmen. Weiter ist nichts bekannt.



Just-In-Time-Compiler (2) – berechnete Sprünge

Beim Übersetzen moderner Programmiersprachen wird `jmp *%eax` nur für `switch`-Konstrukte gebraucht.

Idee:

`%eax` kann nur wenige Werte annehmen (die Werte der `case`-Statements). Spezielle „switch"-Assembler-Instruktion einführen.

=> mögliche Sprungziele dem JIT-Compiler bekannt

=> JIT-Compiler kann über die „switch"-Anweisung hinaus weiter-compilieren.



Just-In-Time-Compiler (2) – berechnete Sprünge

`call *%eax` braucht einer Compiler beim Übersetzen moderner Programmiersprachen nur für Aufrufe virtueller Methoden.

Die Anzahl der virtuellen Methoden, die an einer Stelle im Code aufgerufen werden können, ist i.A. klein und zur Compile-Zeit bekannt.

Idee:

Virtuellen Methoden vor dem Erzeugen eines Objektes JIT-compilieren. Adresse der virtuellen Methoden beim Erzeugen eines Objektes im Objekt speichern.

=> JIT-Compiler kann davon ausgehen, dass bei einem Aufruf einer virtuellen Methode, diese schon compiliert wurde.

=> JIT-Compiler kann über die „call virtual"-Anweisung hinaus weiter-compilieren.



Just-In-Time-Compiler (2) – Selbst-modifizierender Code

Reale Hardware gestattet es i.A., dass Software sich selbst modifiziert.

Dies führt zu Problemen:

- Modifikation muss detektiert werden, um generierten Code verwerfen zu können.
- Code-Cache muss es erlauben, Code verwerfen zu können.

=> JIT-freundliche Hardware verbietet, einmal compilierten Code zu invalidieren.



Just-In-Time-Compiler (2)

Werden alle obigen Punkte berücksichtigt, kann JIT-Compiler ganze Funktionen oder ganze Applikationen „am Stück” übersetzen.

=> optimale Optimierungsmöglichkeiten für JIT-Compiler

=> JIT-compilierter Code läuft in der VM etwa so schnell wie der gleiche Code compiliert für den Host



Just-In-Time-Compiler (2)

Nachteile:

- JIT muss Code bei jedem Programmstart neu übersetzen.
- JIT muss Code bei jedem Programmstart neu optimieren.

Vorteile:

- JIT kann sich auf Code beschränken, der wirklich gebraucht wird.
- JIT muss nur die Code-Bereiche optimieren, die häufig durchlaufen werden.
- JIT kann Laufzeit-Wissen nutzen, das normaler Compiler nicht hat.
- JIT kann auf die aktuelle Hardware optimieren.

JIT-Code kann effizienter sein als direkt für den Host compilierter Code!

