



Compiler für Eingebettete Systeme

[CS7506]

Sommersemester 2014

Heiko Falk

Institut für Eingebettete Systeme/Echtzeitsysteme
Ingenieurwissenschaften und Informatik
Universität Ulm



Kapitel 3

Interner Aufbau von Compilern

Inhalte der Vorlesung

1. Einordnung & Motivation der Vorlesung
2. Compiler für Eingebettete Systeme – Anforderungen & Abhängigkeiten
- 3. Interner Aufbau von Compilern**
4. Prepass-Optimierungen
5. HIR Optimierungen und Transformationen
6. Instruktionsauswahl
7. LIR Optimierungen und Transformationen
8. Register-Allokation
9. Compiler zur WCET_{EST}-Minimierung
10. Ausblick

Inhalte des Kapitels

3. Interner Aufbau von Compilern

- Compilerphasen
 - *Frontend*: Lexikalische Analyse, syntaktische Analyse, semantische Analyse
 - *Backend*: Instruktions-Auswahl, Register-Allokation, Instruktions-Anordnung
- Interne Zwischendarstellungen
 - *High-Level, Medium-Level & Low-Level IRs*
 - Beispiele: ICD-C, MIR, LLIR
 - Struktur eines hochoptimierenden Compilers
- Optimierungen & Zielfunktionen
 - Abstraktionsebenen von Optimierungen
 - Durchschnittliche & *Worst-Case* Laufzeit
 - Codegröße
 - Energieverbrauch

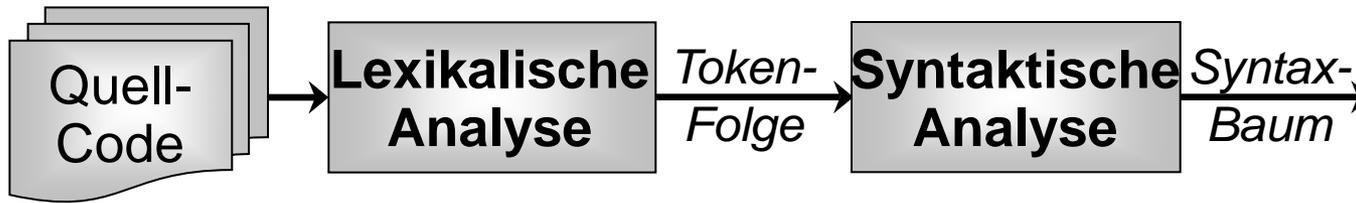
Das *Frontend* (Analysephase)



Lexikalische Analyse (*Scanner*)

- Zerlegung des Quellprogramms in lexikalische Einheiten (*Token*)
- Erkennung von *Token* (reguläre Ausdrücke, endliche Automaten)
- *Token*: Repräsentieren Zeichenfolgen von Bedeutung in der Quellsprache (z.B. Bezeichner, Konstanten, Schlüsselworte)

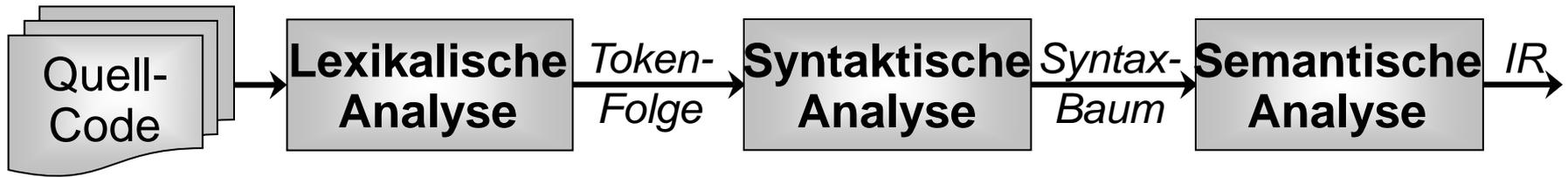
Das *Frontend* (Analysephase)



Syntaktische Analyse (*Parser*)

- Sei G Grammatik der Quellsprache
- Entscheidung, ob *Token-Folge* aus G ableitbar ist.
- Syntaxbaum: Baumförmige Darstellung des Codes anhand während Ableitung benutzter Regeln aus G
- Fehlerbehandlung

Das *Frontend* (Analysephase)



Semantische Analyse (*IR Generator*)

- Namensanalyse (z.B. Gültigkeitsbereiche von Symbolen)
- Prüfung, dass jeder Ausdruck korrekten Typs ist (*Typanalyse*)
- Aufbau von Symboltabellen (Abbildung von Bezeichnern zu deren Typen und Positionen)
- Erzeugung einer Internen Zwischendarstellung (*Intermediate Representation, IR*) zur weiteren Verarbeitung

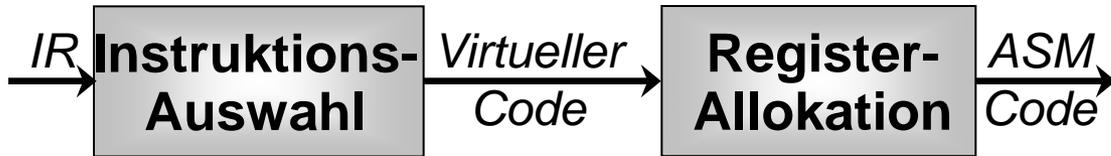
Das *Backend* (Synthesephase)



Instruktionsauswahl (*Code Selector*)

- Auswahl von Maschinenbefehlen zur Implementierung einer IR
- *Oft – Generierung von Virtuellem Code*: Nicht lauffähiger Assemblercode; Annahme unendlich vieler Virtueller Register, anstatt begrenzt vieler Physikalischer Register
- *Alternativ – Generierung von Code mit Stack-Zugriffen*: Lauffähiger Assemblercode; sehr eingeschränkte Nutzung von Registern; Variablen werden im Speicher gehalten (Bsp.: GCC)

Das *Backend* (Synthesephase)



Register-Allokation

- *Entweder*: Abbildung Virtueller auf Physikalische Register
- *Oder*: Ersetzen von *Stack*-Zugriffen durch Speicherung von Daten in Registern
- Einfügen von Speicher-Transfers (*Aus-/Einlagern, Spilling*), falls zu wenig physikalische Register vorhanden

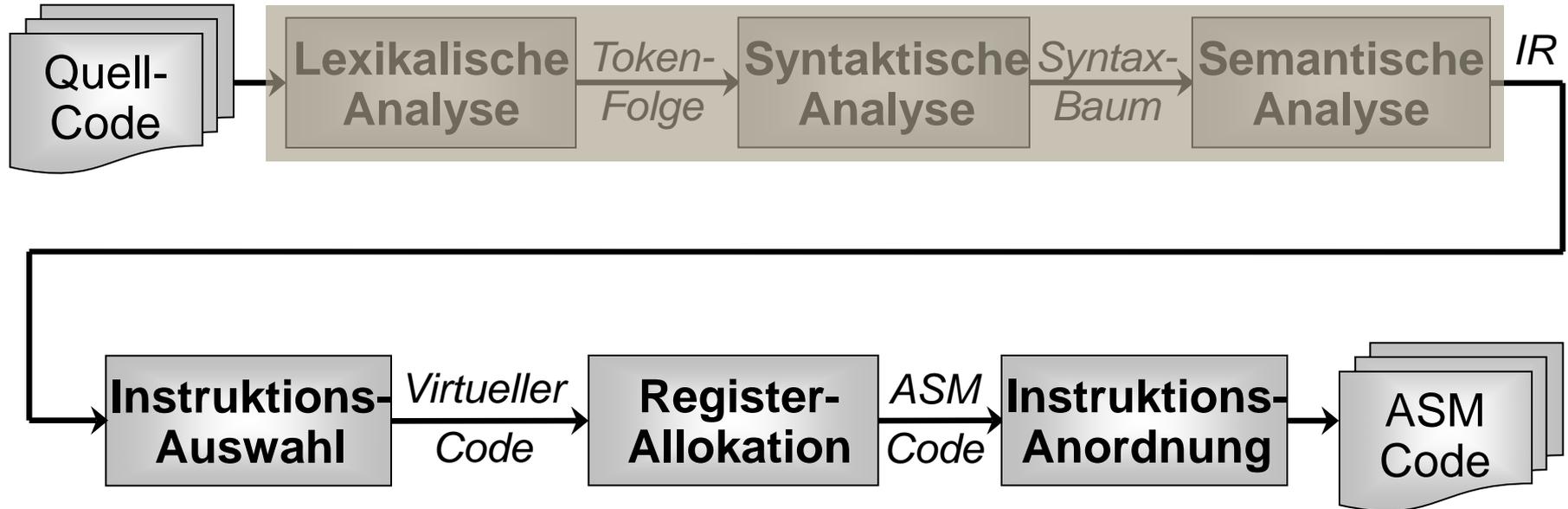
Das *Backend* (Synthesephase)



Instruktionsanordnung (*Scheduler*)

- Umordnen von Maschinenbefehlen zur Erhöhung der Parallelität
- Abhängigkeitsanalyse zwischen Maschinenbefehlen (Daten- & Kontroll-Abhängigkeiten)

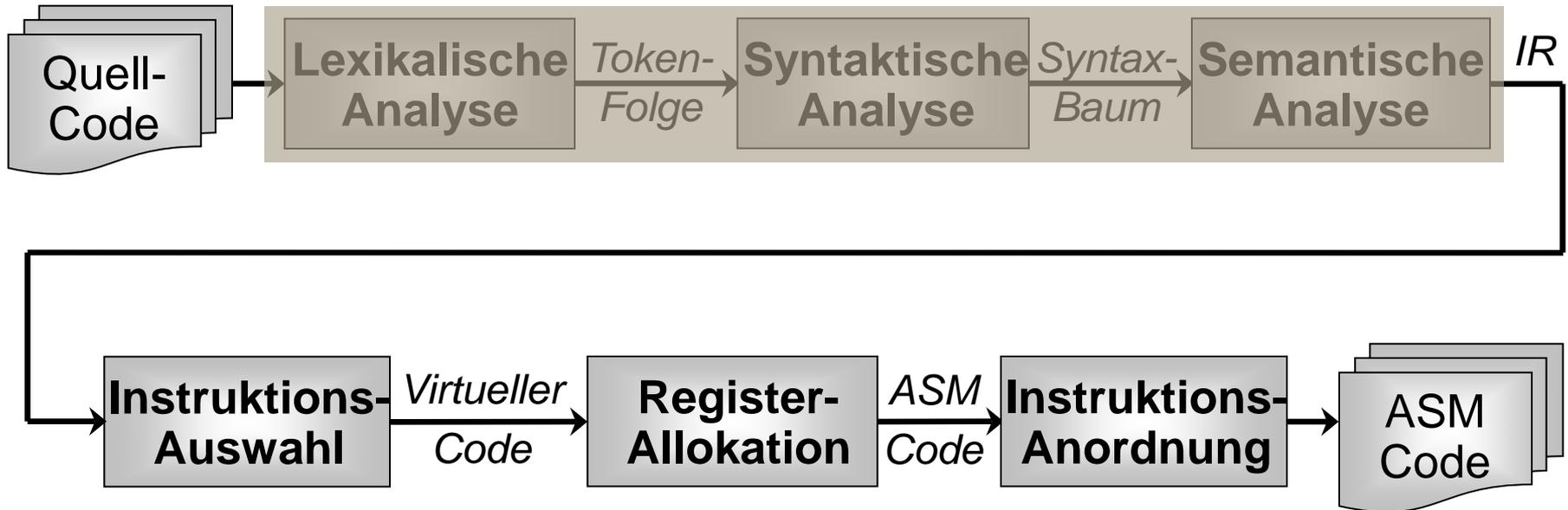
Frontend & Backend



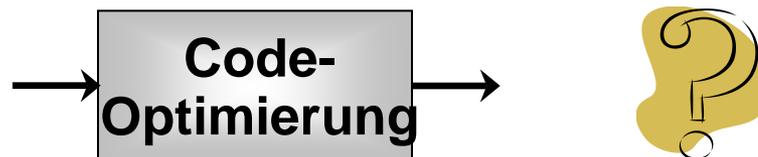
Vorlesung „Compiler für Eingebettete Systeme“

- *Frontend* nicht weiter betrachtet
(☞ Vorlesung „Grundlagen des Übersetzerbaus“)
- *Schwerpunkt: Backend & Compiler-Optimierungen*

Offene Frage



Wo sollten Compiler-Optimierungen angesiedelt sein?



Begriff „Code-Optimierung“

Definition

- Compilerphase, die Code einliest, ändert und ausgibt.
- Code-Änderung erfolgt mit Ziel der *Verbesserung* des Codes.

Bemerkungen

- Optimierungen erzeugen i.d.R. keinen *optimalen Code* (oft unentscheidbar), sondern (hoffentlich) *besseren Code*.
- Code-Verbesserung erfolgt bzgl. einer *Zielfunktion*.

Vorhandensein formaler Code-Analysen

- Code-Änderungen müssen wieder zu korrektem Code führen.
- Optimierung muss entscheiden, wann Änderungen am Code vorgenommen werden dürfen, und wann nicht.
- *Formale Code-Analysen* helfen bei dieser Entscheidung.
- Beispiele: Kontroll- & Datenflussanalyse, Abhängigkeitsanalyse, ...

Voraussetzung zur Code-Optimierung

Benötigte Infrastruktur zur Optimierung

- Effektive interne Darstellung von Code, die
 - Code-Manipulation leicht ermöglicht und
 - notwendige Analysen für Optimierungen bereitstellt.

☞ Interne Zwischendarstellung (*IR*)

Wo sollten Compiler-Optimierungen angesiedelt sein?

- *Optimierungen finden (i.d.R.) auf IR-Ebene im Compiler statt.*

Intermediate Representations (IRs)

- Interne Datenstrukturen des Compilers, die zu übersetzenden bzw. zu optimierenden Code repräsentieren.
- Gute IRs stellen zusätzlich zur Optimierung benötigte Code-Analysen bereit.

Inhalte des Kapitels

3. Interner Aufbau von Compilern

- Compilerphasen
 - *Frontend*: Lexikalische Analyse, syntaktische Analyse, semantische Analyse
 - *Backend*: Instruktions-Auswahl, Register-Allokation, Instruktions-Anordnung
- Interne Zwischendarstellungen
 - *High-Level, Medium-Level & Low-Level IRs*
 - Beispiele: ICD-C, MIR, LLIR
 - Struktur eines hochoptimierenden Compilers
- Optimierungen & Zielfunktionen
 - Abstraktionsebenen von Optimierungen
 - Durchschnittliche & *Worst-Case* Laufzeit
 - Codegröße
 - Energieverbrauch

Abstraktionsniveaus von IRs

```
float a[20][10];
... a[i][j+2] ...;
```

– High-Level

```
t1 ← a[i,j+2]
```

– Medium-Level

```
t1 ← j+2
```

```
t2 ← i*10
```

```
t3 ← t1+t2
```

```
t4 ← 4*t3
```

```
t5 ← addr a
```

```
t6 ← t5+t4
```

```
t7 ← *t6
```

– Low-Level

```
r1 ← [fp-4]
```

```
r2 ← r1+2
```

```
r3 ← [fp-8]
```

```
r4 ← r3*10
```

```
r5 ← r4+r2
```

```
r6 ← 4*r5
```

```
r7 ← fp-216
```

```
f1 ← [r7+r6]
```

Abstraktionsniveaus von IRs

High-Level IRs

- Repräsentation sehr nah am Quellcode
- Oft: Abstrakte Syntaxbäume
- Variablen & Typen zur Speicherung von Werten
- Erhaltung komplexer Kontroll- & Datenflussoperationen (insbes. Schleifen, *if-then / if-else* Ausdrücke, *Array-Zugriffe* [])
- Rücktransformation der *High-Level* IR in Quellcode leicht

[S. S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997]

Abstraktionsniveaus von IRs

Medium-Level IRs

- Drei-Adress-Code: $a_1 \leftarrow a_2 \text{ op } a_3$;
- IR-Code unabhängig von Quell-Sprache & Ziel-Prozessor
- Temporäre Variablen zur Speicherung von Werten
- Komplexe Kontroll- & Datenflussoperationen vereinfacht (*Labels* & Sprünge, Zeiger-Arithmetik)
- Kontrollfluss in Form von *Basisblöcken*

Definition: Ein *Basisblock* $B=(I_1, \dots, I_n)$ ist eine Befehlssequenz maximaler Länge, so dass

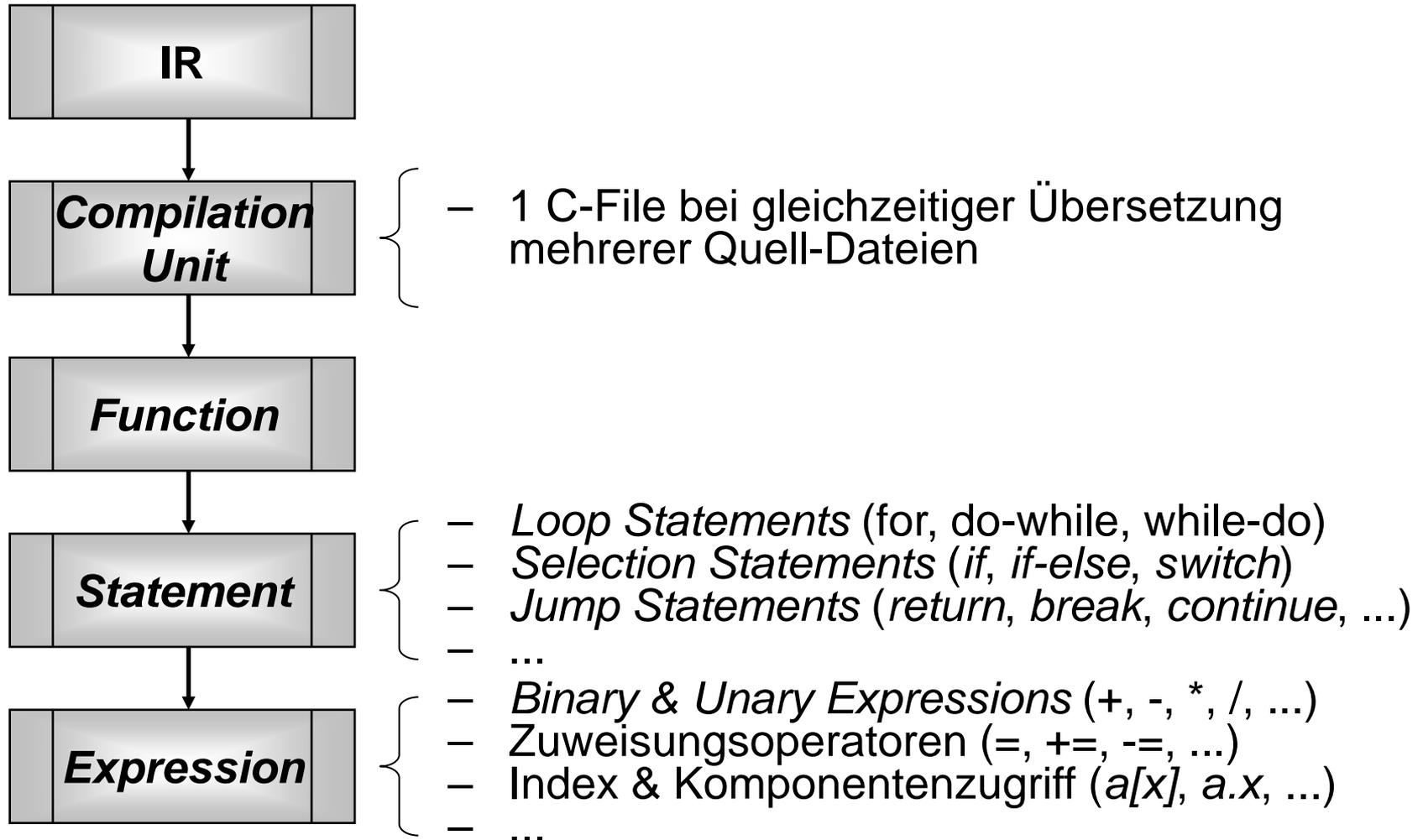
- B nur durch die erste Instruktion I_1 betreten wird, und
- B nur durch die letzte Instruktion I_n verlassen wird.

Abstraktionsniveaus von IRs

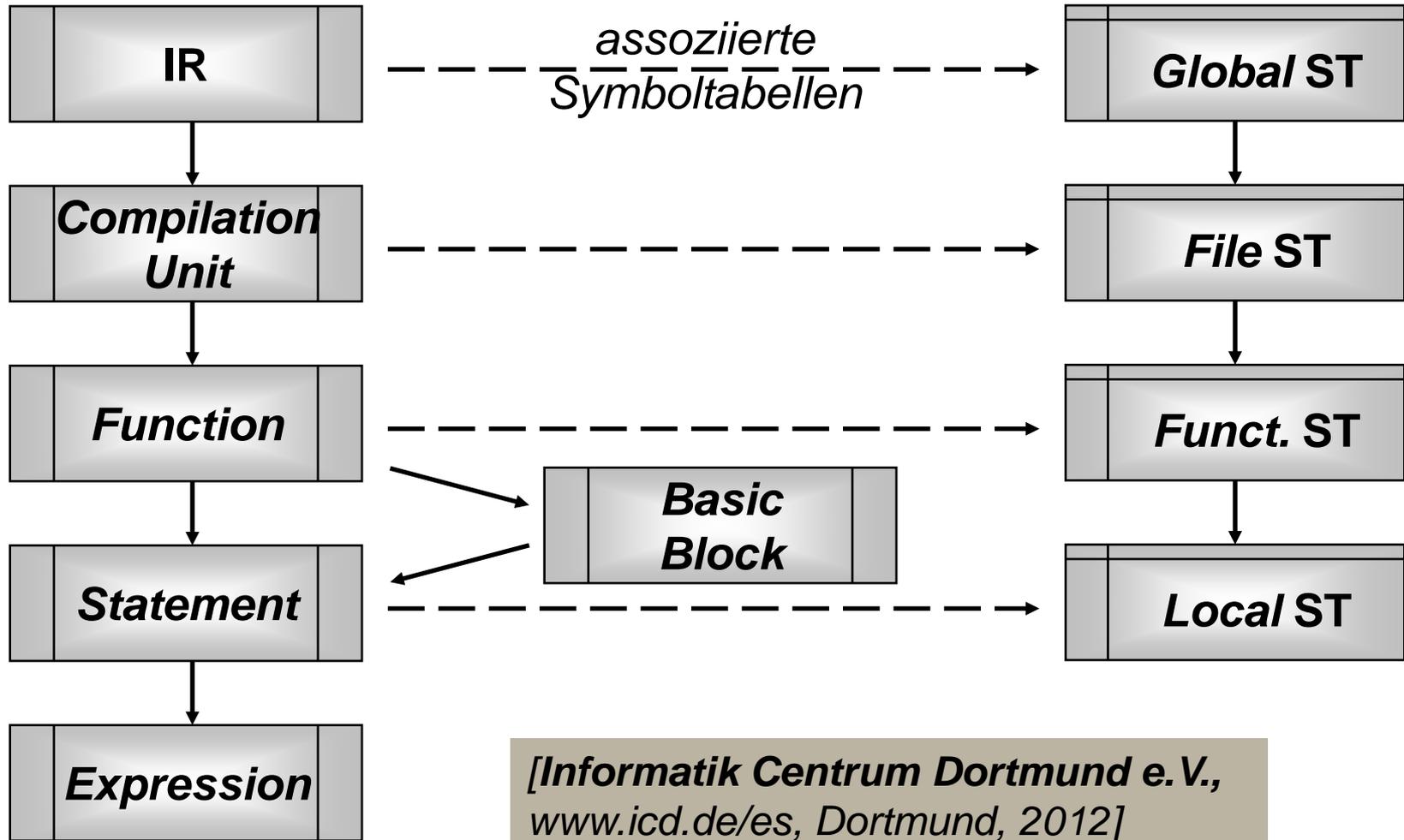
Low-Level IRs

- Repräsentation von Maschinen-Code
- Operationen entsprechen Maschinenbefehlen
- Register zur Speicherung von Werten
- Transformation der *Low-Level IR* in Assemblercode leicht

High-Level IR: ICD-C



High-Level IR: ICD-C

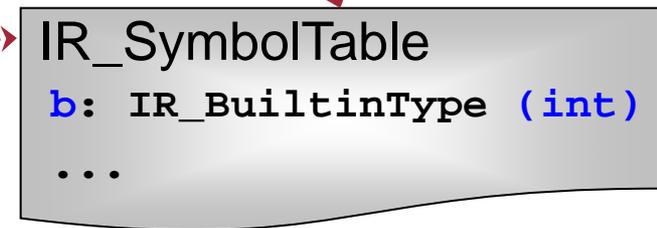
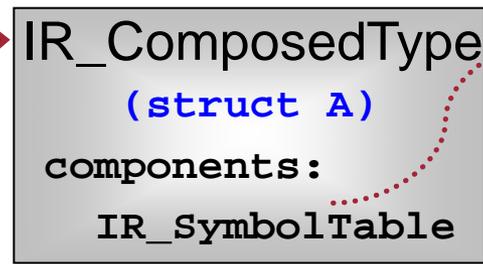
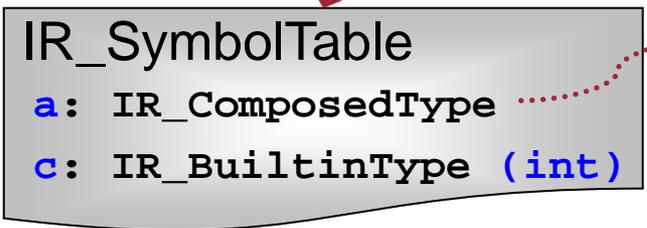
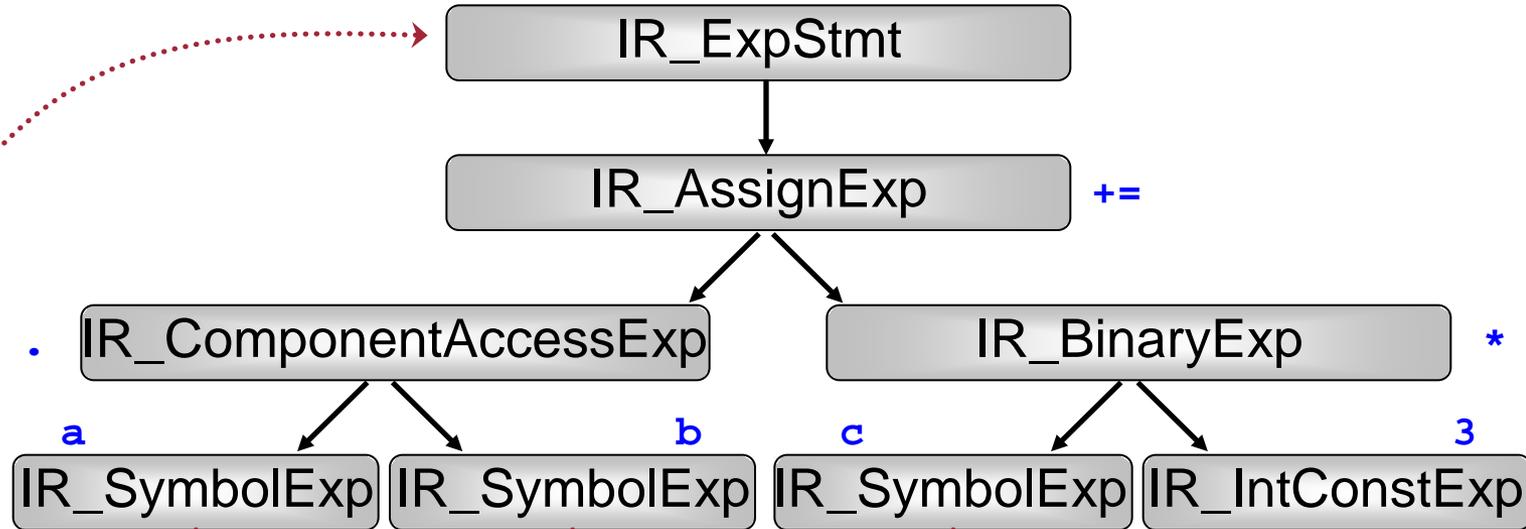


ICD-C: Code-Beispiel

```

struct A {
  int b;
  ...
} a;
int c;

...
a.b += c*3;
    
```



ICD-C: *Features*

- **ANSI-C Compiler *Frontend*:** C89 + C99 Standards
GNU *Inline-Assembler*
- **Enthaltene Analysen:** Datenflussanalysen
Kontrollflussanalysen
Schleifenanalysen
Zeigeranalyse
- **Schnittstellen:**
 - ANSI-C *Dump* der IR als Schnittstelle zu externen Tools
 - Schnittstelle zur Code-Selektion in Compiler-*Backends*
- **Interne Strukturen:**
 - Objektorientiertes Design (C++)

Medium-Level IR: MIR

- **MIR Program:** $1 - N$ *Program Units* (d.h. Funktionen)
- **Program Unit:** `begin MIRInst* end`
- **MIR-Instruktionen:**
 - **Quadrupel:** 1 Operator, 3 Operanden (*d.h. 3-Adress-Code*)
 - **Instruktionstypen:**
 - Zuweisungen, Sprünge (`goto`), Bedingungen (`if`), Funktionsaufruf & -rücksprung (`call`, `return`), Parameterübergabe (`receive`)
 - Können MIR-Ausdrücke (*Expressions*) enthalten

Medium-Level IR: MIR

– **MIR-Ausdrücke:**

- Binäre Operatoren: `+`, `-`, `*`, `/`, `mod`, `min`, `max`
- Relationale Operatoren: `=`, `!=`, `<`, `<=`, `>`, `>=`
- Schiebe- & Logische Operatoren: `shl`, `shr`, `shra`, `and`, `or`, `xor`
- Unäre Operatoren: `-`, `!`, `addr`, `cast`, `*`

– **Symboltabelle:**

- Enthält Variablen und symbolische Register
- Einträge haben Typen: `integer`, `float`, `boolean`

[S. S. Muchnick, Advanced Compiler Design & Implementation, Morgan Kaufmann, 1997]

MIR: Eigenschaften

- **MIR ist keine *High-Level IR***
 - Nähe zur Quellsprache fehlt
 - *High-Level* Konstrukte fehlen: Schleifen, *Array*-Zugriffe, ...
 - Nur wenige, meist simple Operatoren präsent

- **MIR ist keine *Low-Level IR***
 - Nähe zur Zielarchitektur fehlt: Verhalten von Operatoren ist maschinenunabhängig definiert
 - Konzept von Symboltabellen, Variablen & Typen nicht *low-level*
 - Abstrakte Mechanismen zum Funktionsaufruf, Rücksprung und Parameterübergabe

☞ ***MIR ist eine Medium-Level IR.***

Low-Level IR: LLIR

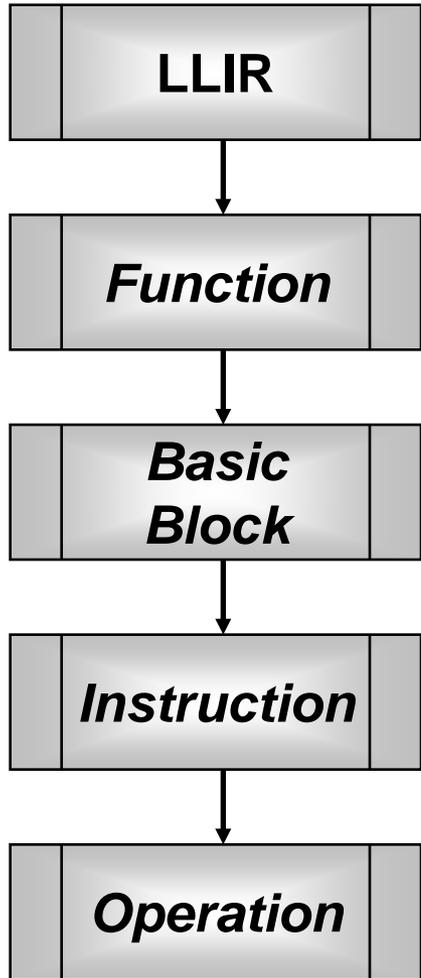


- Maschinen-Instruktion
- Enthält 1-N Maschinen-Operationen
- Operationen werden parallel ausgeführt (☞ VLIW)



- Maschinen-Operation
- Enthält Assembler-Opcode (z.B. **ADD**, **MUL**, ...)
- Enthält 0-M Parameter

Low-Level IR: LLIR

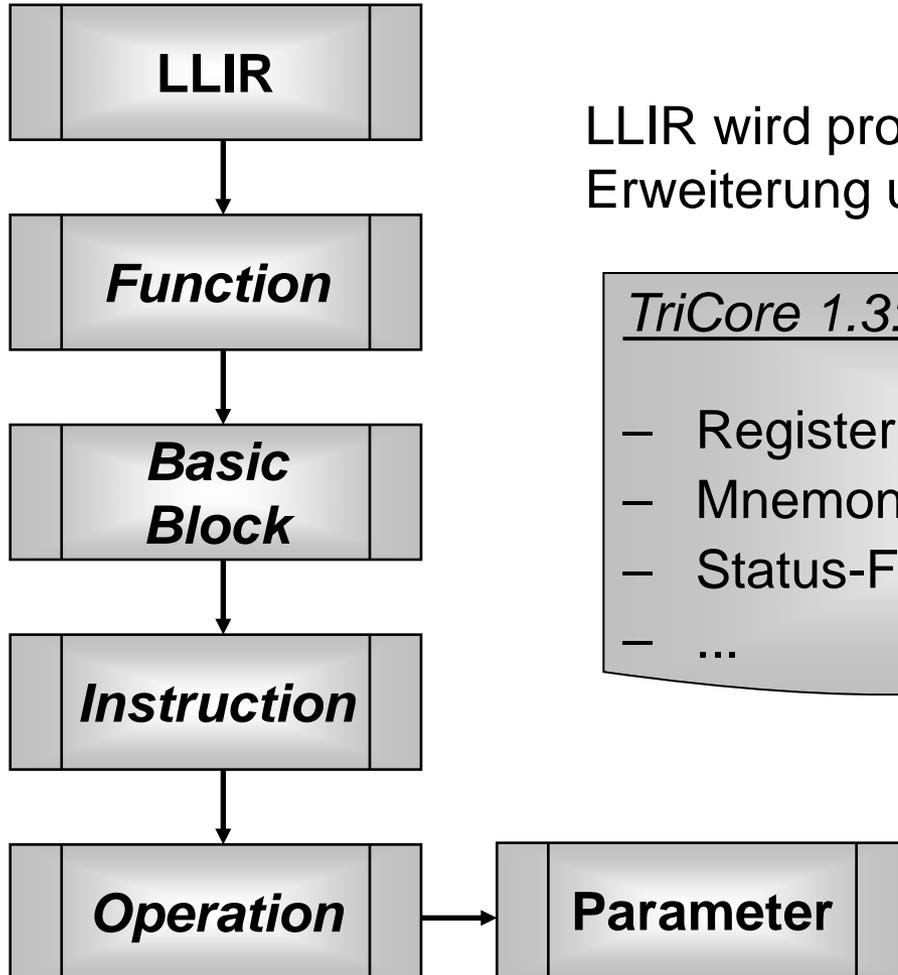


Diese LLIR-Struktur ist vollkommen *prozessor-unabhängig*:

- Eine LLIR besteht aus *irgendwelchen* generischen Funktionen
- Eine LLIR-Funktion besteht aus...
- Eine LLIR-Operation besteht aus *irgendwelchen* generischen Parametern

- Register
- Integer-Konstanten & Labels
- Adressierungsmodi
- ...

Low-Level IR: LLIR



LLIR wird prozessor-spezifisch durch Erweiterung um eine *Prozessor-Beschreibung*:

```

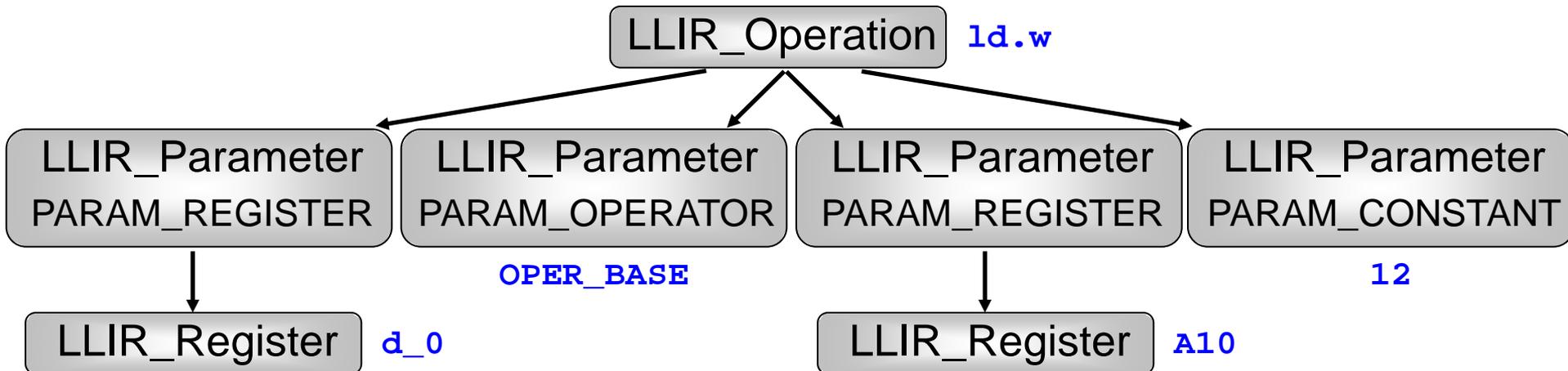
TriCore 1.3:
- Register = {D0, ..., D15, A0, ..., A15}
- Mnemonics = {ABS, ABS.B, ..., XOR.T}
- Status-Flags = {C, V, ..., SAV}
- ...
  
```

[Informatik Centrum Dortmund e.V., www.icd.de/es, Dortmund, 2012]

LLIR: Code-Beispiel (*Infineon TriCore 1.3*)

```
ld.w %d_0, [%A10] 12;
```

- Lade Speicherinhalt von Adresse [%A10] 12 in Register d_0
- Erinnerung: Register A10 = *Stack-Pointer* ➔ Physikalisches Register
- Adresse [%A10] 12 = *Stack-Pointer* + 12 Bytes
(sog. *Base + Offset-Adressierung*)
- TriCore hat kein Register d_0 ➔ Virtuelles Datenregister



LLIR: *Features*

– **Retargierbarkeit:**

- Anpassbarkeit auf verschiedenste Prozessoren (z.B. DSPs, VLIWs, NPUs, ...)
- ☞ Modellierung verschiedenster Befehlssätze
- ☞ Modellierung verschiedenster Registersätze

– **Enthaltene Analysen:**

- Datenflussanalysen
- Kontrollflussanalysen

– **Schnittstellen:**

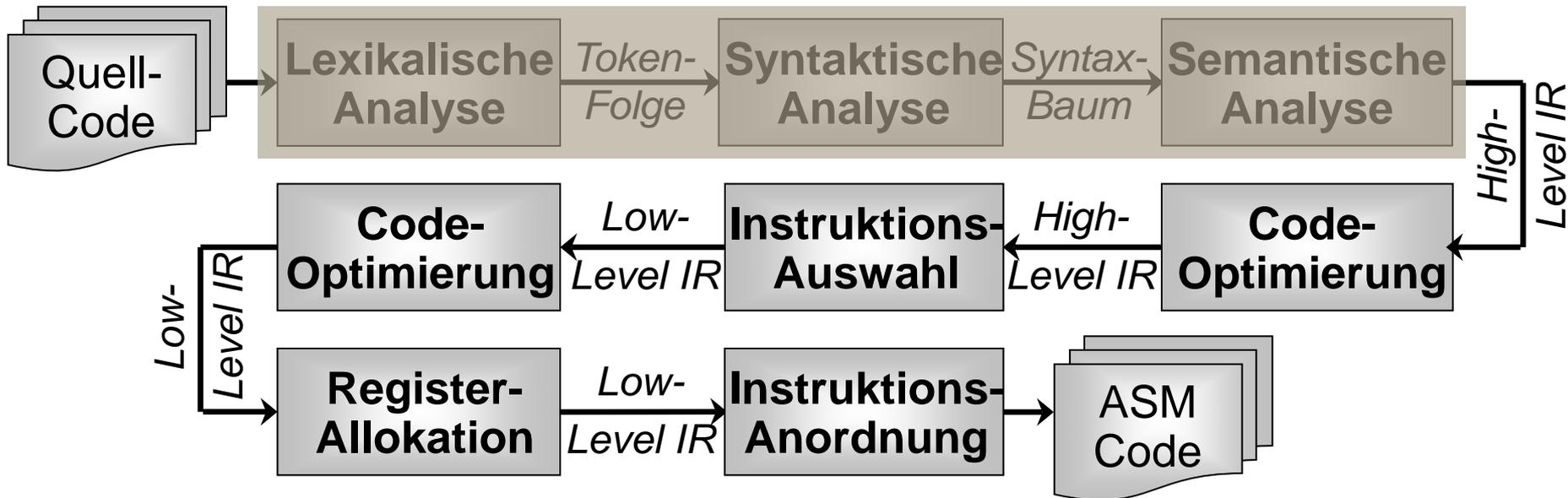
- Einlesen und Ausgabe von Assembler-Dateien
- Schnittstelle zur Code-Selektion

Zurück zur eigentlichen offenen Frage...

Wo sollten Compiler-Optimierungen angesiedelt sein?

- Optimierungen finden (i.d.R.) auf IR-Ebene im Compiler statt.

☞ **Struktur eines Optimierenden Compilers mit 2 IRs:**

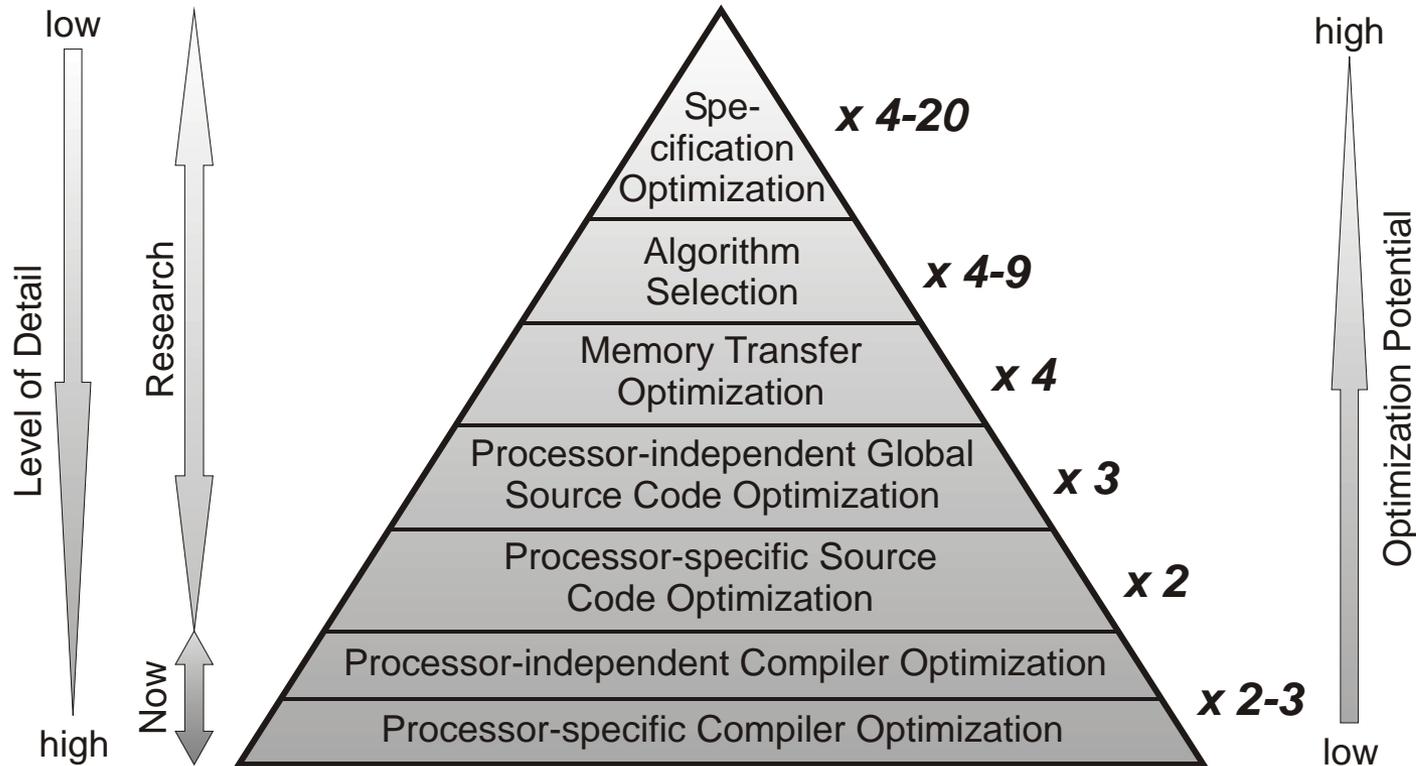


Inhalte des Kapitels

3. Interner Aufbau von Compilern

- Compilerphasen
 - *Frontend*: Lexikalische Analyse, syntaktische Analyse, semantische Analyse
 - *Backend*: Instruktions-Auswahl, Register-Allokation, Instruktions-Anordnung
- Interne Zwischendarstellungen
 - *High-Level, Medium-Level & Low-Level IRs*
 - Beispiele: ICD-C, MIR, LLIR
 - Struktur eines hochoptimierenden Compilers
- Optimierungen & Zielfunktionen
 - Abstraktionsebenen von Optimierungen
 - Durchschnittliche & *Worst-Case* Laufzeit
 - Codegröße
 - Energieverbrauch

Abstraktionsebenen von Optimierungen



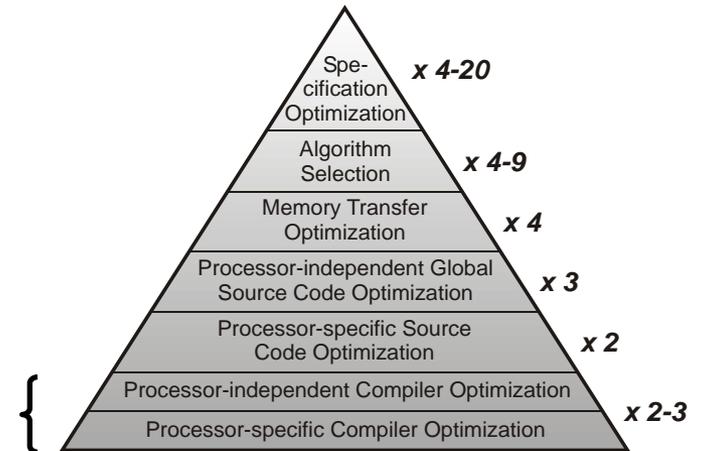
[H. Falk, Source Code Optimization Techniques for Data Flow Dominated Embedded Software, Kluwer, 2004]

Abstraktionsebenen von Optimierungen

Compiler-Optimierungen

- Alles, was in heutigen Compilern enthalten ist
- Prozessor-spezifisch: *low-level*
- Prozessor-unabhängig: *high-level*
- Typische *Speed-Ups*: insgesamt um Faktor 2 bis 3

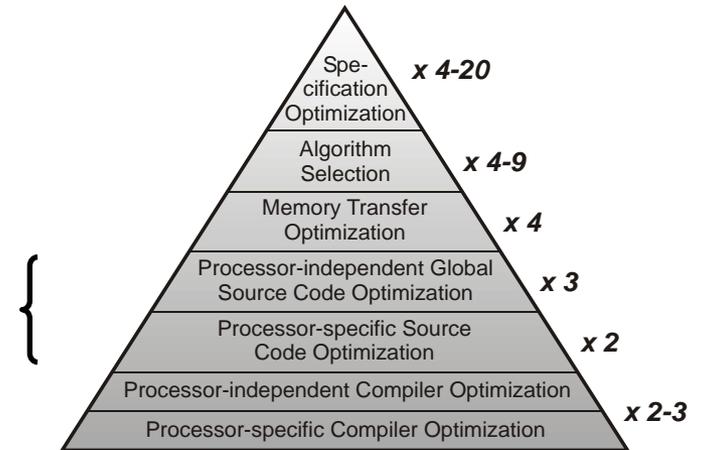
☞ *Mehr dazu in Kapiteln 5 - 9*



Abstraktionsebenen von Optimierungen

Quellcode-Optimierung

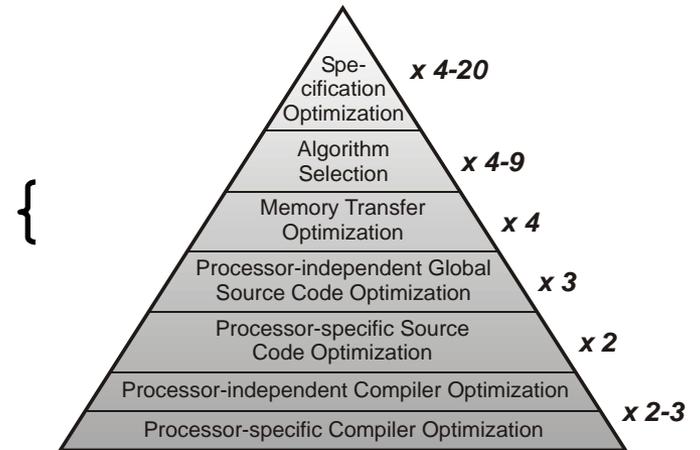
- Quellcode-Umschreiben, so dass Compiler effizienteren Code erzeugt
 - Prozessor-spezifisch: Unterstützung des Compilers bei Abbildung von Quell- auf Maschinensprache
 - Prozessor-unabhängig: Maschinen-unabhängiges Verbessern der Quellcode-Struktur
 - Teils automatisch, teils manuell
 - Typische *Speed-Ups*: je x2 oder x3
- ☞ *Mehr dazu in Kapitel 4*



Abstraktionsebenen von Optimierungen

Speichertransfer-Minimierung

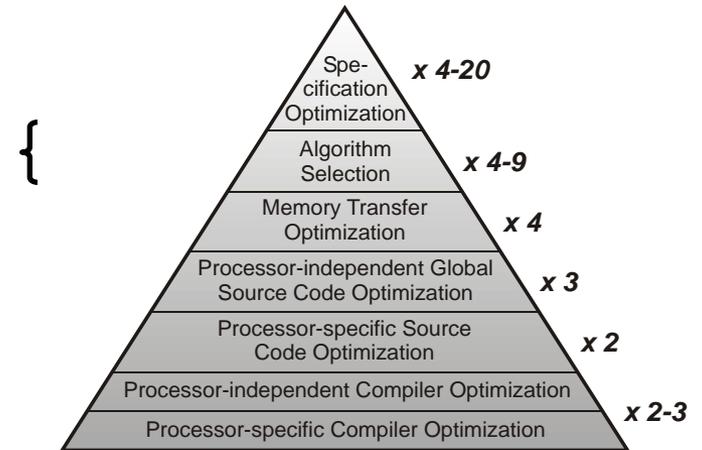
- Reduktion von Daten- & Code-Transfers vom Speicher zum Prozessor auf sehr abstraktem Niveau
- z.B. Umordnen der Datenstrukturen einer Applikation, Umordnung von (mehrdimensionalen) Feldinhalten im Speicher, Zusammenfassen & Teilen von Feldern
- Ausschließlich manuell
- Typische *Speed-Ups*: ca. Faktor 4



Abstraktionsebenen von Optimierungen

Algorithmen-Auswahl

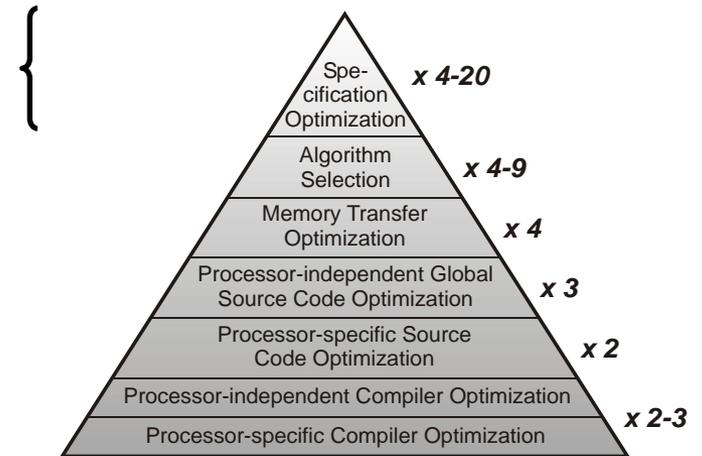
- Ersetzung ganzer Algorithmen einer Applikation durch andere, effizientere Implementierungen
- z.B. Bubblesort → Quicksort
- Ersetzung muss funktionales Verhalten der Applikation beibehalten
- Ausschließlich manuell
- Typische *Speed-Ups*: Faktor 4 – 9



Abstraktionsebenen von Optimierungen

Optimierung der Spezifikation

- Ersetzung von Algorithmen wie bei „*Algorithm Selection*“
- *Aber:* Ersetzung darf funktionales Verhalten der Applikation ändern
- z.B. Ersetzung von `double` Gleitkommazahlen durch einfache Genauigkeit oder `integer`; Ersetzung komplexer Formeln durch einfachere Approximationen (`sin`, `cos`)
- Ausschließlich manuell
- Typische *Speed-Ups*: Faktor 4 – 20



Zielfunktion: (Typische) Laufzeit

- **Durchschnittliche Laufzeit, *Average-Case Execution Time (ACET)***
Ein ACET-optimiertes Programm soll bei einer „typischen“ Ausführung (d.h. mit „typischen“ Eingabedaten) schneller ablaufen.
- **Die Zielfunktion optimierender Compiler schlechthin. Strategie:** „*Greedy*“, d.h. wo die Ausführung von Code zur Laufzeit eingespart werden kann, wird dies i.d.R. auch getan.
- **ACET-optimierende Compiler haben meist kein präzises Modell der ACET.**
Exakte Auswirkung von Optimierungen auf die effektive Laufzeit ist dem Compiler unbekannt.
- ☞ **ACET-Optimierungen sind meist vorteilhaft, manchmal aber auch bloß neutral oder sogar nachteilig**

Beispiel: *Function Inlining*

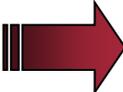
```

main() {
    ...
    a = min( b, c );
    ...
    ...min( f, g )...
}

int min( int i,
        int j ) {
    return(
        i < j ? i : j );
}

main() {
    ...
    a = b < c ? b : c;
    ...
    ...f < g ? f : g;
}

```



Potenzielle Laufzeit-Reduktion wegen:

- Wegfallenden Codes für Parameter- / Rückgabewert-Übergabe
- Wegfallenden Codes zum Sprung in die aufgerufene Funktion
- Evtl. wegfallender Speicherplatz-Allokation zu Beginn der aufgerufenen Funktion
- Evtl. Ermöglichung anderer Optimierungen, die sonst an Funktionsgrenzen scheitern

Zielfunktion: Codegröße

- **Erzeugung von minimal wenig Code, in Bytes gemessen**
- **Einfache Modellbildung:**
Compiler weiß, welche Instruktionen er generiert, und wie viele Bytes jede einzelne Instruktion benötigt.

Oft Zielkonflikt mit Laufzeit-Minimierung: *Bsp. Inlining*

- Kopieren des Funktionsrumpfes an Stelle des Funktionsaufrufs
- Bei großen Funktionen und/oder vielen Vorkommen von Aufrufen im Code: starke Erhöhung der Codegröße!
- Codegrößen-minimierende optimierende Compiler:
 - ☞ *Komplett deaktiviertes Function Inlining*

Zielfunktion: Energieverbrauch (1)

- Generierung von Code mit minimalem Energieverbrauch
- Modellbildung umfasst i.d.R. Prozessor und Speicher

Einfaches Energiemodell für Prozessoren:

- *Basiskosten* eines Befehls: Energieverbrauch des Prozessors bei Ausführung nur dieses einen Befehls
- Ermittlung der Basiskosten (z.B. für **ADD**-Befehl):

.L0:

...

ADD d0, d1, d2;

ADD d0, d1, d2;

ADD d0, d1, d2;

...

LOOP a5, .L0;

- Schleife, die zu untersuchenden Befehl *sehr oft* enthält.
- Ausführung auf realer Hardware
- Energiemessung: Amperemeter
- Ergebnis herunterrechnen auf einmal **ADD**
- Wiederholen für kompletten Befehlssatz

Zielfunktion: Energieverbrauch (2)

Einfaches Energiemodell für Prozessoren:

- *Inter-Instruktionskosten* zwischen zwei nachfolgenden Befehlen:
Modellieren Aktivierung und Deaktivierung Funktionaler Einheiten (FUs)
- Beispiel: **ADD** wird in ALU ausgeführt, **MUL** in Multiplizierer

`.L0:`

`ADD d0, d1, d2;`

`MUL d3, d4, d5;`

`ADD d0, d1, d2;`

`MUL d3, d4, d5;`

`...`

`LOOP a5, .L0;`

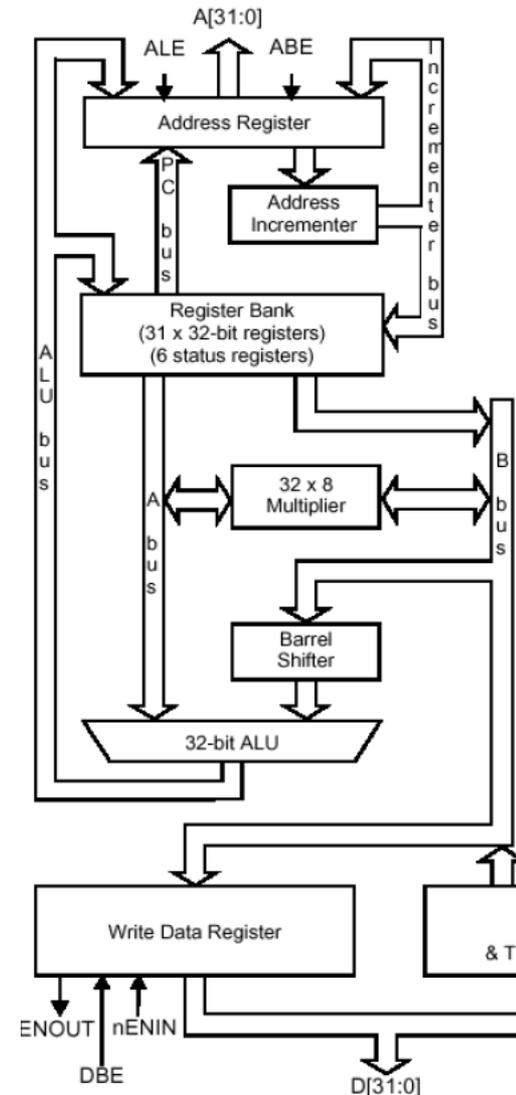
- Schleife, die zu untersuchendes Befehlspar
sehr oft enthält.
- Ausführung & Messung wie bei Basiskosten
- Ergebnis herunterrechnen auf ein Befehls-
paar **ADD** und **MUL**
- Wiederholen für alle Kombinationen von FUs

Zielfunktion: Energieverbrauch (3)

Funktionale Einheiten von ARM7-CPU

- Funktionale Einheiten: *Address Incrementer*, *32x8 Multiplier*, *Barrel Shifter* und ALU
 - *Beispiel von voriger Folie: ALU für ADD* mit Energie versorgen, Addition durchführen.
 - *Danach: Multiplier für MUL* anschalten, die Busse von/zum *Multiplier* aufladen.
 - *Schließlich: Nach MUL Multiplier* abschalten, die Busse entladen.
- ☞ An-/Abschalten & Auf-/Entladen kostet viel Energie!

[ARM Limited, ARM7TDMI Technical Reference Manual, 2004]



Zielfunktion: Energieverbrauch (4)

Berechnung der Prozessor-Energie durch Compiler

- Summiere Basiskosten aller generierten Instruktionen
- Summiere Inter-Instruktionskosten benachbarter Befehlspaare

[V. Tiwari et al., Power Analysis of Embedded Software: A First Step Towards Software Power Minimization, IEEE Transactions on VLSI, Dezember 1994]

Berechnung der Speicher-Energie durch Compiler

- Entweder anhand von Datenblättern oder durch Messungen
- Grundlage: Energieverbrauch pro Lade- und Speichervorgang
- Einfach für Statische RAMs (SRAM), schwer für Caches und Dynamische RAMs (DRAM)

[S. Steinke et al., An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations, PATMOS Workshop, September 2001]

Zielfunktion: *Worst-Case* Laufzeit (1)

Worst-Case Execution Time (WCET):

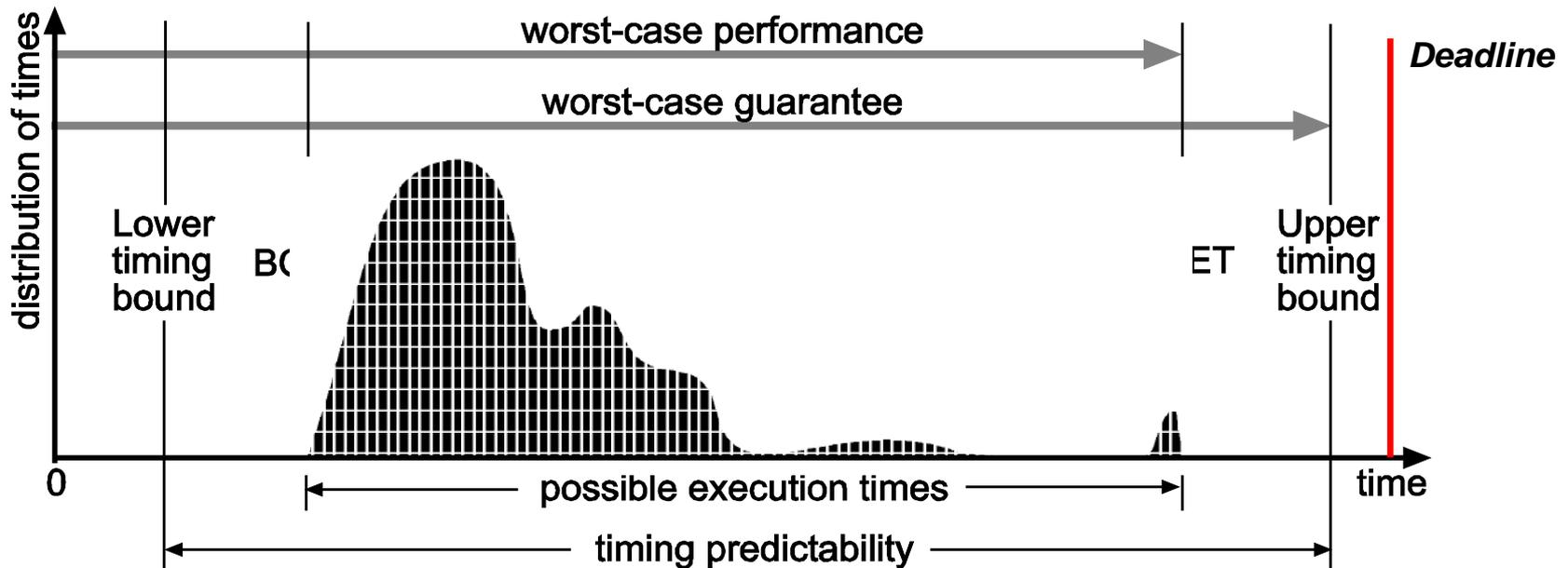
Die maximale Laufzeit eines Programms über alle denkbaren Eingabedaten hinweg.

Problem:

Ermittlung der WCET eines Programms nicht berechenbar! (Würde das Lösen des Halte-Problems beinhalten)

Zielfunktion: *Worst-Case Laufzeit* (2)

Lösung: Schätzung oberer Grenzen für die echte (unbekannte) WCET



Anforderungen an WCET-Abschätzungen:

- Sicherheit (*safeness*): $WCET \leq WCET_{EST}$!
- Präzision (*tightness*): $WCET_{EST} - WCET \rightarrow \text{minimal}$

Literatur

Compilerphasen und IRs

- Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
ISBN 1-55860-320-4
- Andrew W. Appel. *Modern compiler implementation in C*. Cambridge University Press, 1998.
ISBN 0-521-58390-X

Abstraktionsebenen von Optimierungen

- H. Falk, *Source Code Optimization Techniques for Data Flow Dominated Embedded Software*, Kluwer Academic Publishers, 2004.
ISBN 1-4020-2822-9

Zusammenfassung (1)

Compilerphasen

- Bedeutung einzelner Phasen eines Compilers
- Fokus auf Compiler-*Backend*
- Anordnung von Optimierungen innerhalb des Compilers

Interne Zwischendarstellungen

- Begriff der IR: effektive Compiler-interne Darstellung von Code; erleichtert Manipulation und Analyse von Code
- Verschiedene Abstraktionsniveaus: quellcodenah; unabhängig von Programmiersprache und Prozessorarchitektur; maschinennah

Zusammenfassung (2)

Optimierungen & Zielfunktionen

- Viele Arten von Code-Optimierungen mit sehr hohem Potential nicht automatisierbar
- Fokus auf Compiler- und Quellcode-Optimierungen
- *Average-Case Execution Time*: Zielfunktion nahezu jedes Compilers; Compiler enthalten jedoch kein ACET-Modell
- Codegröße: oft im Widerspruch zu ACET
- Energieverbrauch: Energiemodelle für Prozessoren (Basis- & Inter-Instruktionskosten) und Speicher
- *Worst-Case Execution Time*: nicht berechenbar; WCET-Abschätzung