

Compilerbau

Wintersemester 2010 / 2011

Dr. Heiko Falk

Technische Universität Dortmund

Lehrstuhl Informatik 12

Entwurfsautomatisierung für Eingebettete Systeme

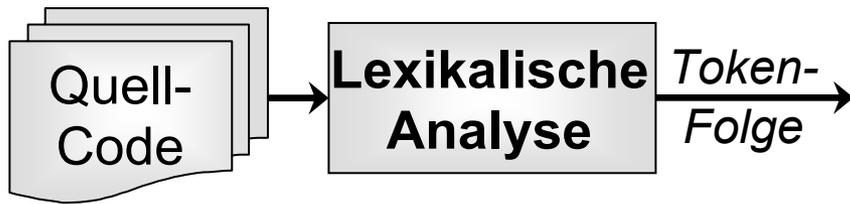
Kapitel 2

Interner Aufbau von Compilern

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
 - **Compilerphasen**
 - Strukturen Optimierender Compiler
 - Zielfunktionen von Code-Optimierungen
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

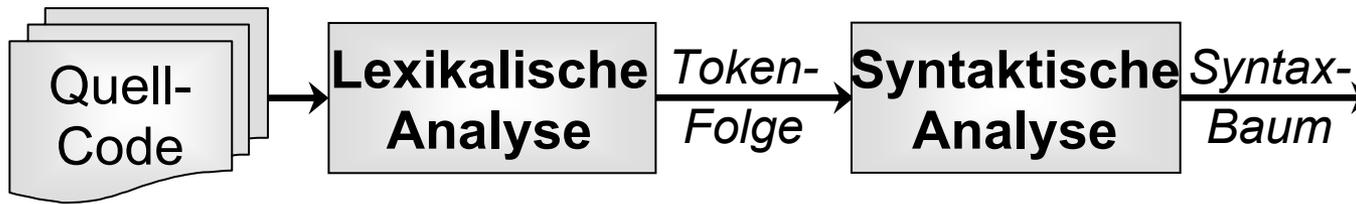
Das Frontend (Analysephase)



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)

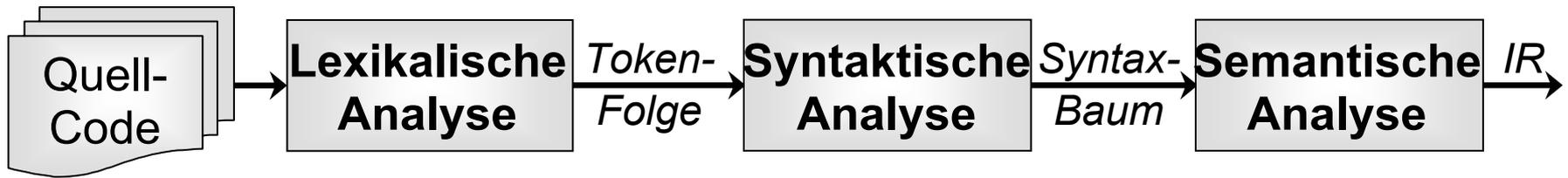
Das Frontend (Analysephase)



Syntaktische Analyse (*Parser*):

- Sei G Grammatik der Quellsprache
- Entscheidung, ob Tokenfolge 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

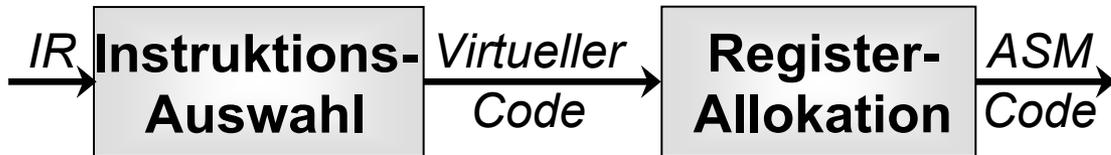
Das Backend (Synthesephase)



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)

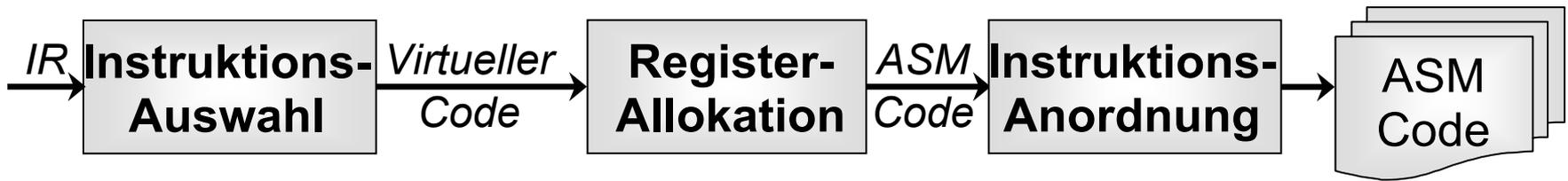
Das Backend (Synthesephase)



Registerallokation:

- *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

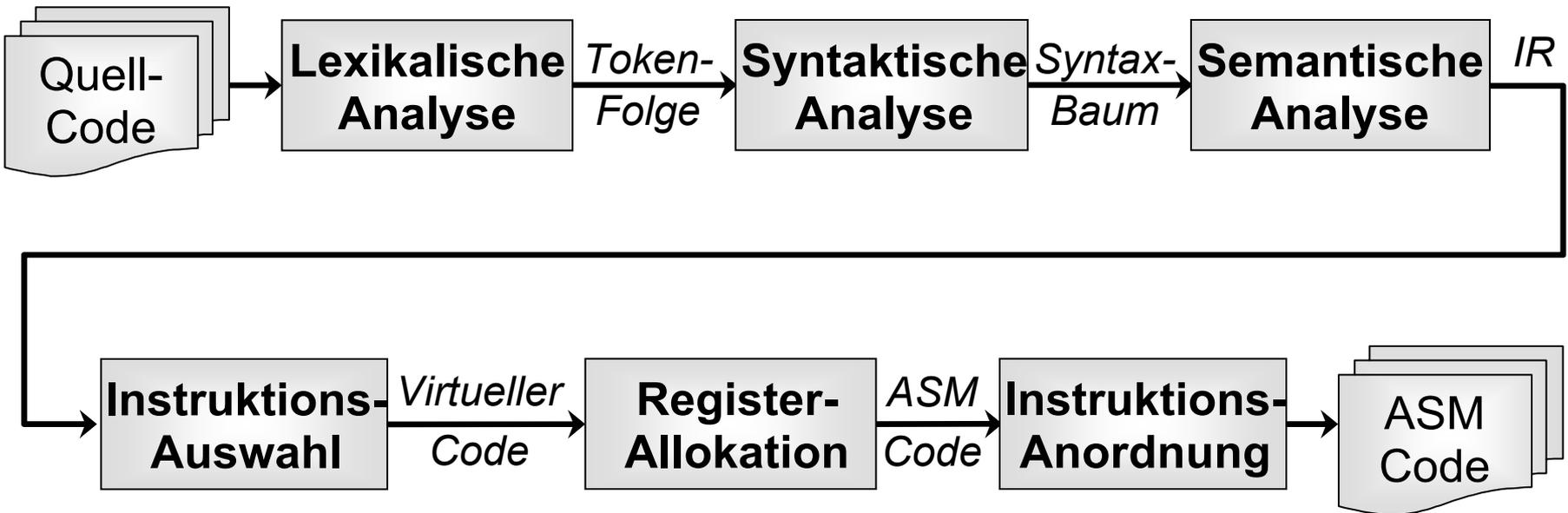
Das Backend (Synthesephase)



Instruktionsanordnung (*Scheduler*):

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

Aufbau eines einfachen Compilers



Wo finden in diesem Compiler nun Code-Optimierungen statt?

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
 - Compilerphasen
 - **Strukturen Optimierender Compiler**
 - Zielfunktionen von Code-Optimierungen
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

Begriff „Code-Optimierung“

Definition (*Optimierung*):

- 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*.

Infrastruktur zur Code-Optimierung

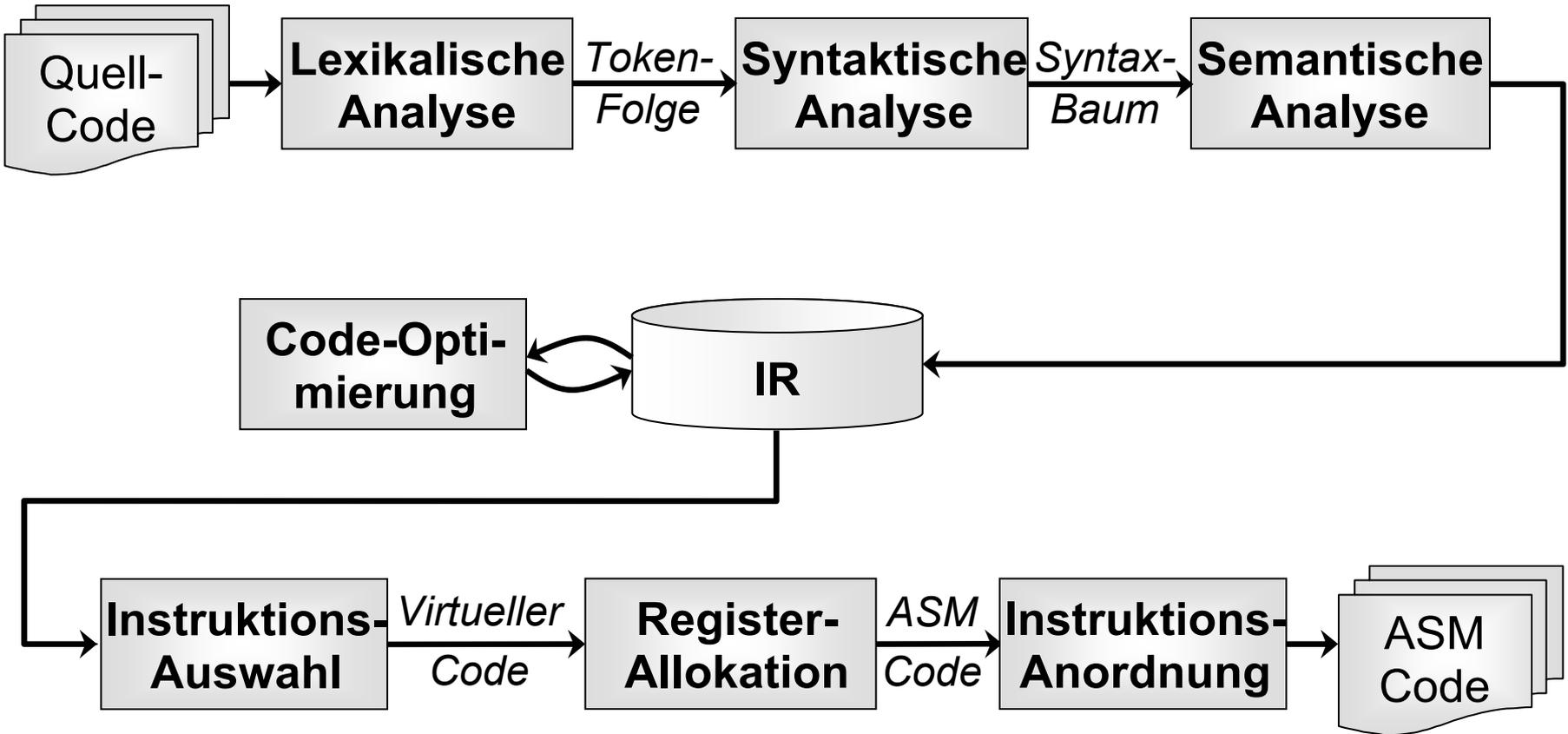
Vorhandensein formaler Code-Analysen:

- 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, ...

Intermediate Representations (IRs):

- Interne Datenstrukturen des Compilers, die zu übersetzenden bzw. zu optimierenden Code repräsentieren. (☞ [Kapitel 5](#))
- Gute IRs stellen zusätzlich zur Optimierung benötigte Code-Analysen bereit.

Aufbau eines optimierenden Compilers



Alternative Compiler-Strukturen (1)

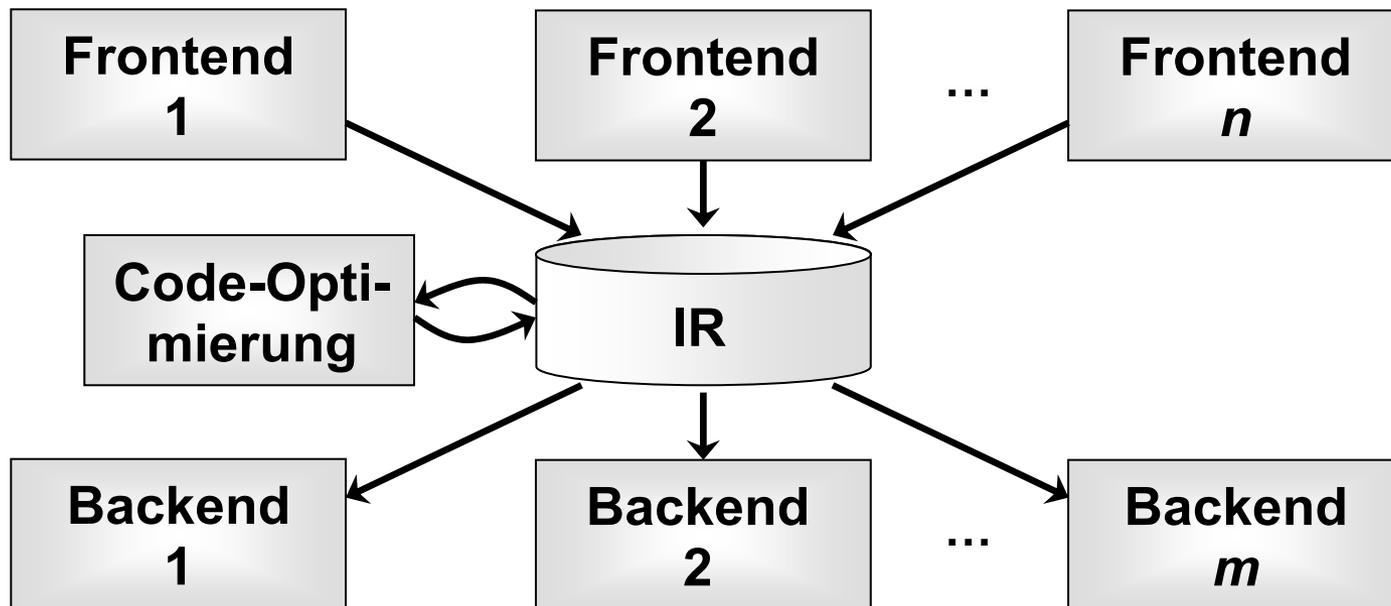
Standardcompiler für Desktop- bzw. Server-Systeme:

- Können oft mehrere Quellsprachen übersetzen (z.B. C, C++, Java, Pascal, ...).
- ☞ Benötigen daher mehrere Frontends, pro Quellsprache eines.
- Können oft Maschinencode für mehrere Zielprozessoren erzeugen (z.B. für i386, x86_64, sparc v8, sparc v9, ppc, ...).
- ☞ Benötigen daher mehrere Backends, pro Zielarchitektur eines.

- Informationsaustausch zwischen n Frontends und m Backends erfolgt über zentrale Schnittstelle, nämlich die IR des Compilers. Diese abstrahiert hinreichend von den diversen Quellsprachen und Prozessoren weg.

Alternative Compiler-Strukturen (2)

Standardcompiler für Desktop- bzw. Server-Systeme:



Beispiele:

- GNU Compiler Collection GCC
- LLVM
- Sun Studio Compiler

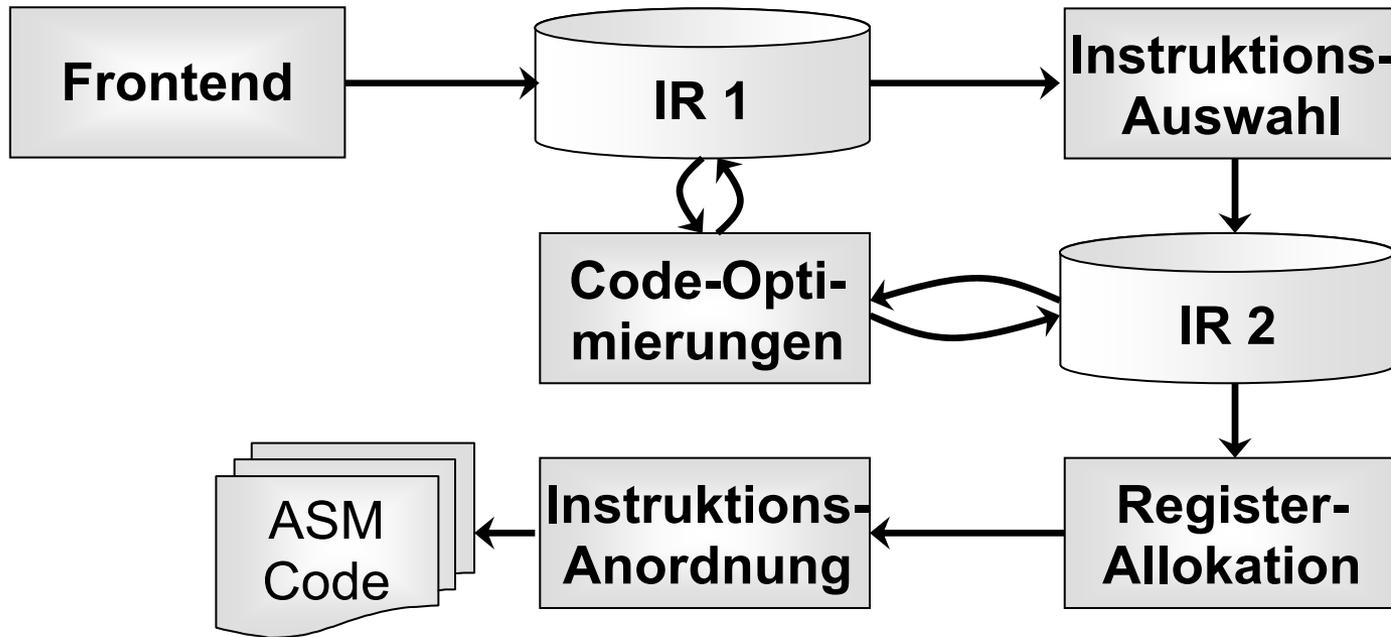
Alternative Compiler-Strukturen (3)

Compiler für Eingebettete Systeme:

- Spezialcompiler, die hohe Flexibilität der Standardcompiler bzgl. verschiedener Quellsprachen und Zielprozessoren nicht brauchen.
- ANSI-C als Quellsprache im Bereich Eingebetteter Systeme etablierter „de facto-Standard“.
- ☞ Ein ANSI-C Frontend genügt.
- ☞ Es genügt i.d.R. auch ein Backend, da Code nur für fest vorgegebene Zielarchitektur zu erzeugen ist.
- Compiler für Eingebettete Systeme sind nicht auf maximale Flexibilität ausgerichtet, sondern auf hoch-optimierten Code.
- ☞ Massive Code-Optimierung mit Hilfe mehrerer IRs.

Alternative Compiler-Strukturen (4)

Compiler für Eingebettete Systeme:



Beispiel:

- WCET-Aware C Compiler WCC

<http://ls12-www.cs.tu-dortmund.de/research/activities/wcc>

Gliederung der Vorlesung

- **Kapitel 1: Compiler – Abhängigkeiten und Anforderungen**
- **Kapitel 2: Interner Aufbau von Compilern**
 - Compilerphasen
 - Strukturen Optimierender Compiler
 - Zielfunktionen von Code-Optimierungen
- **Kapitel 3: Lexikalische Analyse (Scanner)**
- **Kapitel 4: Syntaktische Analyse (Parser)**
- **Kapitel 5: Semantische Analyse**
- **Kapitel 6: Instruktionsauswahl**
- **Kapitel 7: Register-Allokation**
- **Kapitel 8: Code-Optimierung**
- **Kapitel 9: Ausblick**

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;

...

JMP .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;

...

JMP .L0;

- Schleife, die zu untersuchendes Befehls-paar *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 (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)

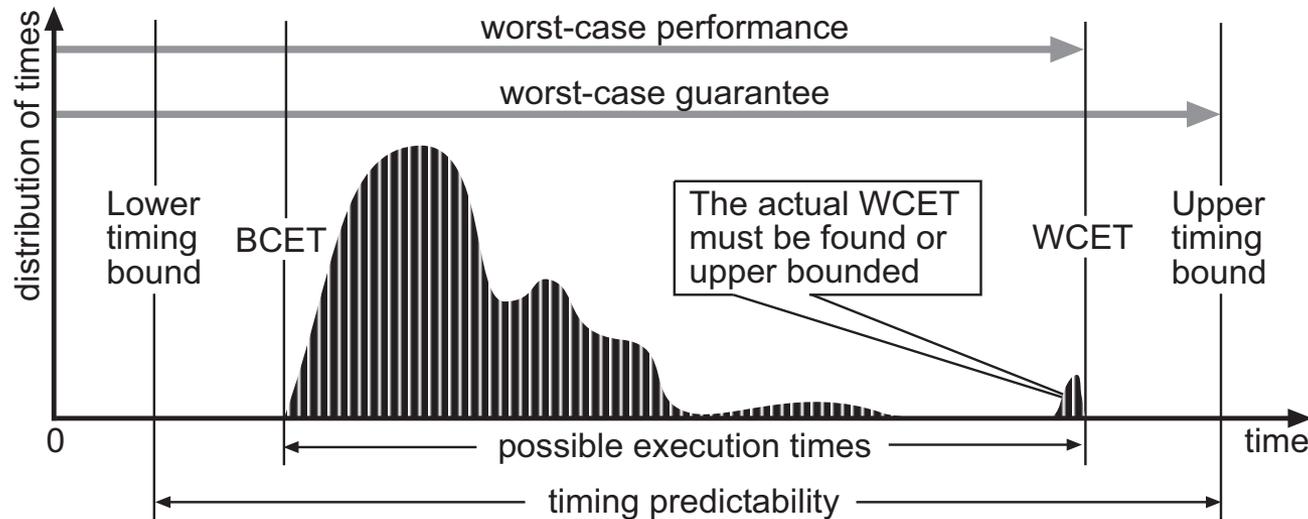
[S. Steinke et al., An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations, PATMOS Workshop, September 2001]

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 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

- 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

Zusammenfassung

- **Bedeutung einzelner Phasen eines Compilers**
- **Anordnung von Optimierungen innerhalb des Compilers**
- **Zielfunktionen:
ACET, Codegröße, Energieverbrauch, WCET**

