

Echtzeitsysteme

Zeitliche Analyse von Echtzeitanwendungen

Peter Ulbrich

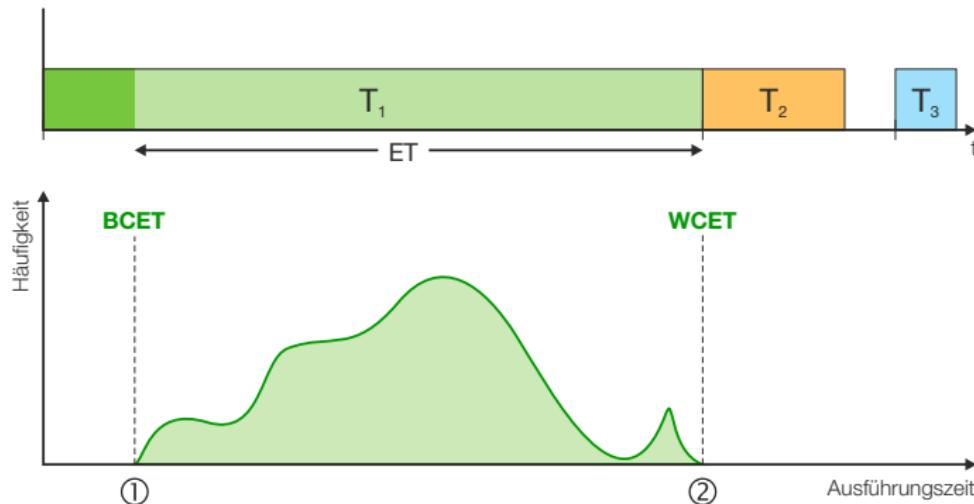
Lehrstuhl Informatik 4

28. Oktober 2014

Gliederung

- 1 Übersicht
- 2 Problemstellung
- 3 Dynamische WCET-Analyse (Messbasiert)
- 4 Statische WCET-Analyse
- 5 Hardware-Analyse – Die Maschinenprogrammebene
- 6 Zusammenfassung

Fragestellungen



- Alle sprechen von der **maximalen Ausführungszeit**
 - Worst Case Execution Time (WCET)** e_i (vgl. Folie III-2/28)
 - Unabdingbares Maß für **zulässigen Ablaufplan** (vgl. Folie III-2/33)
- Tatsächliche Ausführungszeit bewegt sich zwischen:
 - bestmöglicher Ausführungszeit (**Best Case Execution Time, BCET**)
 - schlechtest möglicher Ausführungszeit (besagter **WCET**)

Gliederung

- 1 Übersicht
- 2 Problemstellung
- 3 Dynamische WCET-Analyse (Messbasiert)
- 4 Statische WCET-Analyse
- 5 Hardware-Analyse – Die Maschinenprogrammebene
- 6 Zusammenfassung

Warum ist es so schwierig, **e** zu bestimmen?

Anders: Wovon hängt die maximale Ausführungszeit eigentlich ab?

Beispiel: Bubblesort

```
void bubbleSort(int a[], int size) {
    int i, j;

    for(i = size - 1; i > 0; --i) {
        for (j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j], &a[j+1]);
            }
        }
    }

    return;
}
```

Programmiersprachenebene:

- Anzahl der Schleifendurchläufe hängt von der Größe des Feldes `a[]` ab
 - Anzahl der Vertauschungen (`swap`) hängt vom Inhalt des Feldes ab
- ~ **exakte Vorhersage ist kaum möglich**
- sowohl die Größe als auch der Inhalt des Felds kann zur Laufzeit variieren

Die **Maschinenprogrammebene** liefert Dauer der Elementaroperationen:

- wie lange dauert ein ADD, ein LOAD, ...
- ist **prozessorabhängig** und für moderne Prozessoren sehr schwierig
 - **Cache** ~ Liegt die Instruktion/das Datum im schnellen Cache?
 - **Pipeline** ~ Wie ist der Zustand der Pipeline an einer Instruktion?
 - **Out-of-Order-Execution, Branch-Prediction, Hyper-Threading**, ...

Gliederung

- 1 Übersicht
- 2 Problemstellung
- 3 Dynamische WCET-Analyse (Messbasiert)
- 4 Statische WCET-Analyse
- 5 Hardware-Analyse – Die Maschinenprogrammebene
- 6 Zusammenfassung

Messbasierte WCET-Analyse [2]

Idee: der Prozessor selbst ist das präziseste Hardware-Modell
~~> Führe das Programm aus und beobachte die Ausführungszeit!

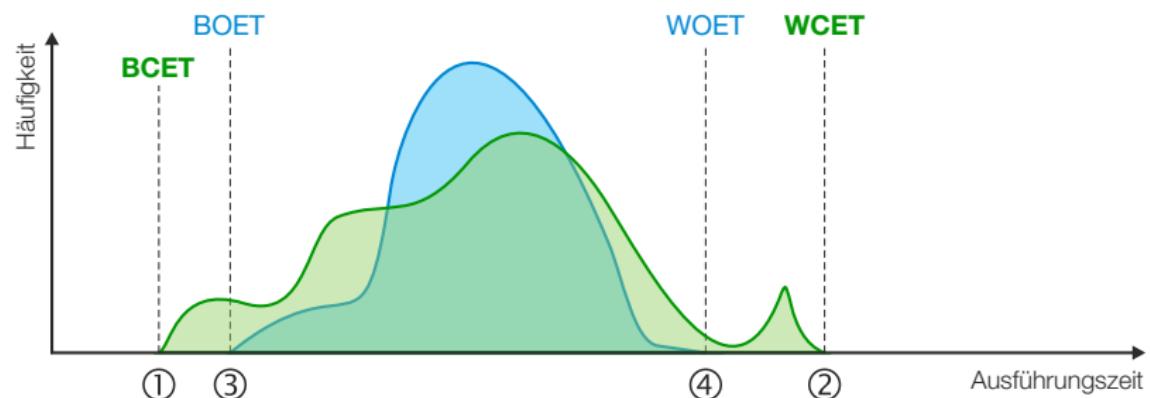
Es besteht **Bedarf** für messbasierte Methoden:

- **gängige Praxis** in der Industrie
- nicht alle Echtzeitsysteme benötigen eine sichere WCET
 - z. B. Echtzeitsystem mit **weichen Zeitschränken**
- lassen sich leicht an **neue Hardwareplattformen** anpassen
 - häufig ist kein geeignetes statisches Analysewerkzeug verfügbar
- **geringer Aufwand für Annotationen**
 - verschafft leicht Orientierung über die tatsächliche Laufzeit
- **sinnvolle Ergänzung** zur statischen WCET-Analyse
 - **Validierung** statisch bestimmter Werte
 - Ausgangspunkt für die Verbesserung der statischen Analyse

☞ **Allerdings** sollte man nicht „einfach drauf los messen“

- ~~> z. B. immer Pfade vermessen (d. h. Ablauf und Zeit)
- ~~> auf einen definierten Startzustand achten

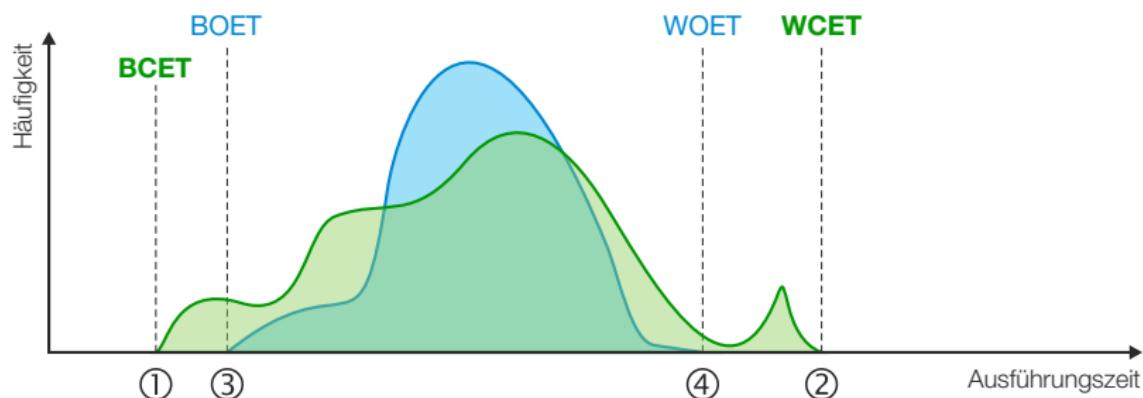
Messbasierte WCET-Analyse [2]



Dynamische WCET-Analyse liefert **Messwerte**:

- ③ Bestmögliche beobachtete Ausführungszeit
(Best Observed Execution Time, **BOET**)
- ④ Schlechteste mögliche beobachtete Ausführungszeit
(Worst Observed Execution Time, **WOET**)

Messbasierte WCET-Analyse [2]



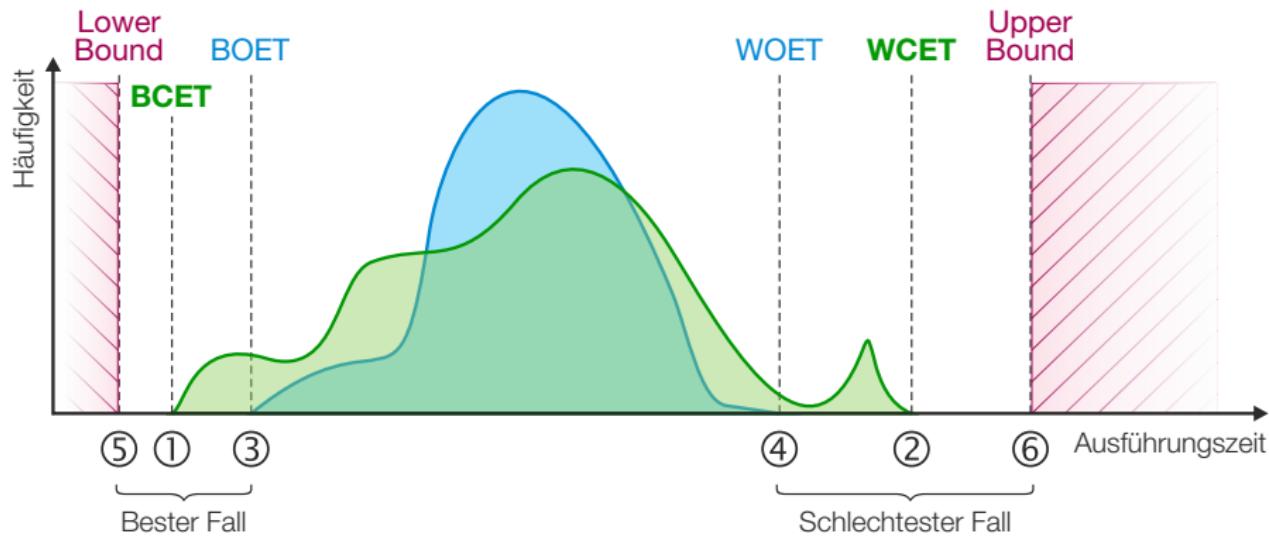
Probleme messbasierter Ansätze

- in der Praxis ist es unmöglich alle relevanten Pfade zu betrachten
 - gewählte Testdaten führen nicht unbedingt zum längsten Pfad
 - seltene Ausführungsszenarien werden nicht abgedeckt
 - abschnittsweise WCET-Messung $\not\rightarrow$ globalen WCET
 - Wiederherstellung des Hardwarezustandes schwierig/unmöglich
- ☞ messbasierte Ansätze unterschätzen die WCET meistens
- ☞ systematischere, messbasierte Analysetechniken sind vonnöten

Gliederung

- 1 Übersicht
- 2 Problemstellung
- 3 Dynamische WCET-Analyse (Messbasiert)
- 4 Statische WCET-Analyse
- 5 Hardware-Analyse – Die Maschinenprogrammebene
- 6 Zusammenfassung

Überblick: Statische WCET-Analyse



Statische WCET-Analyse liefert **Schranken**:

- ⑤ Geschätzte untere Schranke (Lower Bound)
- ⑥ Geschätzte obere Schranke (Upper Bound)
- Die Analyse ist **sicher** (sound) falls $\text{Upper Bound} \geq \text{WCET}$

Problem: Den längsten Weg durch ein Programm finden

Beispiel: Bubblesort

```
void bubbleSort(int a[], int size) {
    int i, j;

    for(i = size - 1; i > 0; --i) {
        for (j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j], &a[j+1]);
            }
        }
    }

    return;
}
```

betrachte Aufrufe: bubbleSort(a, size)

- Anzahl von Durchläufen, Vergleichen und Vertauschungen (engl. **Swap**)
- $a = \{1, 2\}$, $size = 2$
 $\rightsquigarrow D = 1, V = 1, S = 0$;
- $a = \{1, 3, 2\}$, $size = 3$
 $\rightsquigarrow D = 3, V = 3, S = 1$;
- $a = \{3, 2, 1\}$, $size = 3$
 $\rightsquigarrow D = 3, V = 3, S = 3$;

- ist für den **allgemeinen Fall nicht berechenbar** \rightsquigarrow **Halteproblem**
 - Wieviele Schleifendurchläufe werden benötigt?
- \rightsquigarrow in Echtzeitsystemen ist dieses Problem aber häufig lösbar
 - **kanonische Schleifenkonstrukte:** `for(int i = 0; i < X; ++i)`
 - X ist oft eine Konstante oder zumindest beschränkt
 - ggf. muss die obere Schranke manuell annotiert werden
 - die **maximale**, nicht die **exakte Pfadlänge** ist von Belang

Den längsten Weg durch ein Programm finden (2)

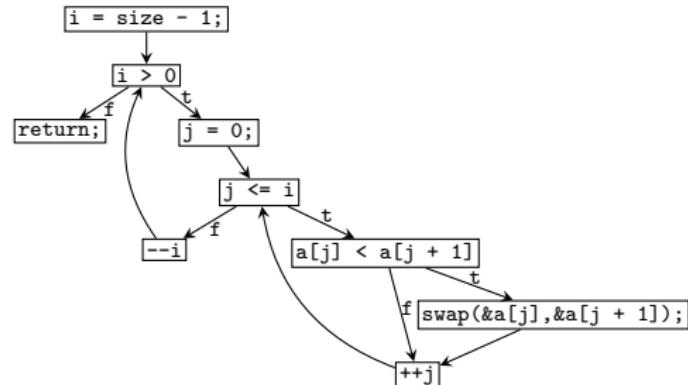
Die möglichen Wege lassen sich durch Kontrollflussgraphen beschreiben

Ein **Kontrollflussgraphen** (engl. *control flow graph*) ist ein gerichteter Graph und setzt sich aus **Grundblöcken** (engl. *basic blocks*) zusammen

- Grundblöcke sind sequentielle „Code-Schnipsel“
 - hier wird gearbeitet \leadsto Grundblöcke verbrauchen Rechenzeit
- Kanten im Kontrollflussgraphen \leadsto Sprünge zwischen Grundblöcken

```
void bubbleSort(int a[],int size) {
    int i,j;

    for(i = size - 1; i > 0; --i) {
        for (j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j],&a[j+1]);
            }
        }
    }
    return;
}
```



Lösungsweg₁: Timing Schema \rightsquigarrow Abstrakter Syntaxbaum

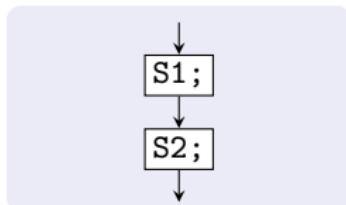
Ableitung des maximalen Pfads anhand der Programmstruktur

Sequenzen \rightsquigarrow Hintereinanderausführung

`s1();
s2();`

Summation der WCETs:

$$e_{seq} = e_{s1} + e_{s2}$$

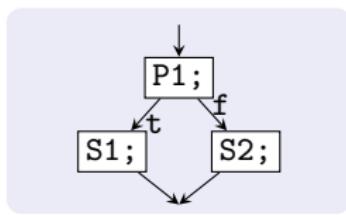


Verzweigung \rightsquigarrow bedingte Ausführung

`if(P1()) s1();
else s2();`

Abschätzung der Gesamtausführungszeit:

$$e_{cond} = e_{P1} + \max(e_{s1}, e_{s2})$$

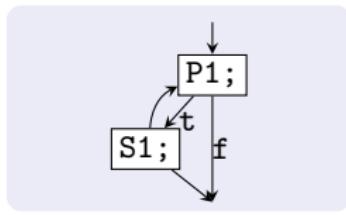


Schleifen \rightsquigarrow wiederholte Ausführung

`while(P1())
 s1();`

Schleifendurchläufe berücksichtigen:

$$e_{loop} = e_{P1} + n(e_{P1} + e_{s1})$$



Beispiel: Bubblesort

```
void bubbleSort(int a[], int size) {
    int i, j;

    for(i = size - 1; i > 0; --i) {
        for (j = 0; j < i; ++j) {
            if(a[j] > a[j+1]) {
                swap(&a[j], &a[j+1]);
            }
        }
    }

    return;
}
```

- Verzweigung $C_1: P_3 = a[j] > a[j + 1]$
 - $S_1 = \text{swap}(&a[j], &a[j + 1])$
 - $e_{C_1} = e_{P_3} + e_{S_1}$
- Funktionsaufruf $S_1 = \text{swap}(&a[j], &a[j + 1])$
 - analog zum hier vorgestellten Verfahren

- Schleife $L_1: P_1 = i > 0$
 - Rumpf: $L_2: --i;$
 - Durchläufe: $\text{size} - 1$
 - $e_{L_1} = e_{P_1} + (\text{size} - 1)(e_{P_1} + e_{L_2} + e_{--i})$
- Schleife $L_2: P_2 = j < i$
 - Rumpf: $C_1: ++j;$
 - Durchläufe: $\text{size} - 1$
 - $e_{L_2} = e_{P_2} + (\text{size} - 1)(e_{P_2} + e_{C_1} + e_{++j})$

Timing Schema: Eigenschaften, Vor- und Nachteile

Eigenschaften

- Traversierung des abstrakten Syntaxbaums **bottom-up**
 - d. h. an den Blättern beginnend, bis man zur Wurzel gelangt
- **Aggregation** der maximale Ausführungszeit nach festen Regeln
 - für Sequenzen, Verzweigungen und Schleifen

Vorteile

- + einfaches Verfahren mit geringem Berechnungsaufwand
- + skaliert gut mit der Programmgröße

Nachteile

- Informationsverlust durch Aggregation
 - Korrelationen (z. B. sich ausschließende Zweige) nicht-lokalen Code Teile lassen sich nicht berücksichtigen
 - Schwierige Integration mit einer separaten Hardware-Analyse
- Nichtrealisierbare Pfade (infeasible paths) nicht ausschließbar
~ unnötige Überapproximation

Pfadbasierte Bestimmung der WCET

Mit der Anzahl f_i der Ausführungen einer Kante E_i bestimmt man die WCET e durch Summation der Ausführungszeiten des längsten Pfades:

$$e = \max_P \sum_{E_i \in P} f_i e_i$$

☞ **Problem:** erfordert die **explizite Aufzählung aller Pfade**

~ das ist algorithmisch nicht handhabbar

☞ **Lösung:** fasse die Bestimmung der WCET als **Flussproblem** auf

~ der **maximale Fluss** durch das durch den Graphen gegebene Netzwerk führt zur gesuchten WCET

~ Flussprobleme sind mathematisch gut untersucht und lassen sich durch **lineare Ganzzahlprogrammierung** lösen

Lösungsansatz₂: Implicit Path Enumeration Technique

Lösungsansatz: Fasse die Bestimmung der WCET als Flussproblem auf [1] (Implicit Path Enumeration Technique, IPET)

Vorgehen: Transformiere den Kontrollflussgraphen in ein ganzzahliges, lineares Optimierungsproblem (ILP) und löse es

- ❶ bestimme einen **Zeitanalysegraph** aus dem Kontrollflussgraphen
- ❷ formuliere das lineare Optimierungsproblem
- ❸ bestimme die **Flussrestriktionen** des Zeitanalysegraphen
 - dies sind die Nebenbedingungen im Optimierungsproblem
- ❹ löse des Optimierungsproblem (z.B. mit `lpsolve`¹)

¹<http://lpsolve.sourceforge.net/>

Der Zeitanalysegraph (engl. *timing analysis graph*)

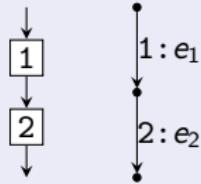
Ein Zeitanalysegraph (oder kurz **T-Graph**) ist ein gerichteter Graph mit einer Menge von Knoten $\mathcal{V} = \{V_i\}$ und Kanten $\mathcal{E} = \{E_i\}$.

- mit genau einer **Quelle** und einer **Senke**
 - Knoten, aus denen/in die nur Kanten entspringen/münden
- jede Kante ist Bestandteile eines Pfads P von der Senke zur Kante
 - solche ein Pfad P entspricht einer möglichen Abarbeitung
- jeder Kante wird ihre WCET e_i zugeordnet

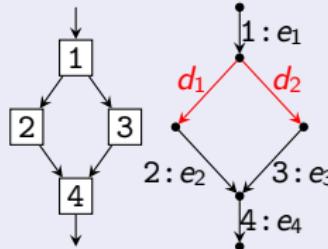
Grundblöcke des Kontrollflussgraphen werden auf Kanten abgebildet

- für Verzweigungen benötigt man **Dummy-Kanten** d_i

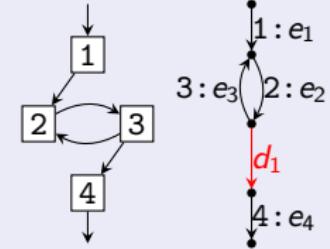
Sequenz



Verzweigung



Schleife



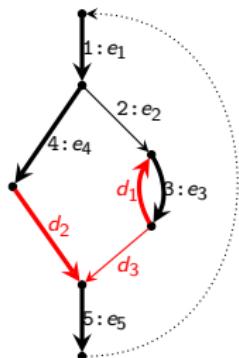
Zirkulationen

Eine Abbildung $f : \mathcal{E} \mapsto \mathcal{R}$ heißt **Zirkulation**, falls sie den Fluss erhält

- jeder Kante wird die **Zahl der Ausführungen** f_i als Fluss zugeordnet
- **Flusserhaltung**: jeder Knoten wird gleich oft betreten und verlassen
 - erfordert die Einführung einer Rückkehrkante E_e mit $f_e = 1$

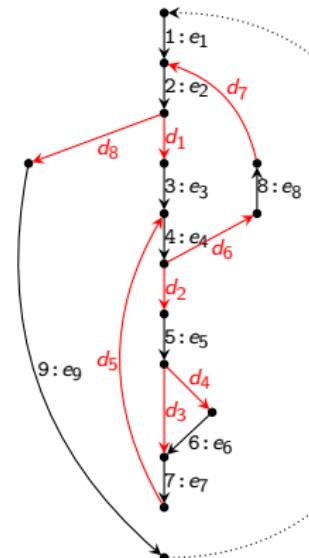
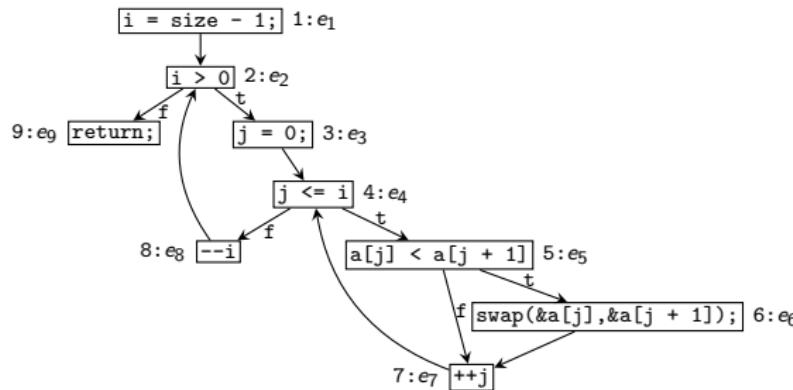
Flussrestriktionen schließen Zirkulationen ungültiger Abarbeitungen aus

- Formulierung als **Nebenbedingungen** des Optimierungsproblems
- Beschränkung der maximalen Anzahl von Schleifendurchläufen



- $f_1 = f_2 + f_4$ wird durch die Zirkulation garantiert
- gültige Zirkulation: $\{E_1, E_4, d_2, E_5, E_e\} \cup \{E_3, d_1\}$
 ↳ aber **keine gültige Abarbeitung**
- Flussrestriktion $f_3 \leq 5f_2$ löst dieses Problem
 - wird E_2 nicht abgearbeitet, so gilt $f_3 \leq 5 \cdot 0 = 0$
 - hier: Beschränkung auf 5 Schleifendurchläufe
- Nebenbedingung des Optimierungsproblems

Beispiel: Bubblesort



- Flussrestriktionen, die sich aus Schleifen ergeben:
 - „äußere Schleife“: $f_2 \leq (size - 1)f_1$
 - „innere Schleife“: $f_4 \leq (size - 1)f_3$
- Flussrestriktionen, die sich aus Verzweigungen ergeben:
 - bedingte Vertauschung: $f_{d_3} + f_6 = f_7$

IPET: Eigenschaften, Vor- und Nachteile

- betrachte mögliche Abarbeitungen des Kontrollflussgraphen
- dabei werden alle Pfade implizit in Betracht gezogen
 - zunächst wird aus dem Kontrollflussgraph ein T-Graph erzeugt
 - dieser wird in ganzzahliges lineares Optimierungsproblem überführt

Vorteile

- + Möglichkeit komplexer Flussrestriktionen
 - z. B. sich ausschließende Äste aufeinanderfolgender Verzweigungen
- + Nebenbedingungen für das ILP sind leicht aufzustellen
- + viele Werkzeuge zur Lösung von ILPs verfügbar

Nachteile

- das Lösen eines ILP ist im Allgemeinen **NP-hart**
- auch Flussrestriktionen sind kein Allheilmittel
 - Beschreibung der Ausführungsreihenfolge ist problematisch

Gliederung

- 1 Übersicht
- 2 Problemstellung
- 3 Dynamische WCET-Analyse (Messbasiert)
- 4 Statische WCET-Analyse
- 5 Hardware-Analyse – Die Maschinenprogrammebene
- 6 Zusammenfassung

Wie lange dauern die „sequentiellen Code-Schnipsel“

Die WCETs e_i der einzelnen Grundblöcke ist Eingabe für die Flussanalyse

Grundproblem: Ausführungszyklen von Instruktionen zählen

```
_getopt:
    link    a6 ,#0          ; 16 Zyklen
    moveml #0x3020 ,sp@-   ; 32 Zyklen
    movel   a6@(8) ,a2      ; 16 Zyklen
    movel   a6@(12) ,d3     ; 16 Zyklen
```

Quelle: Peter Puschner [1]

- Ergebnis: $e_{\text{getopt}} = 80$ Zyklen
- Annahmen:
 - obere Schranke für jede Instruktion
 - die obere Schranke der Sequenz bestimmt man durch Summation

Problem: Vorgehen ist **äußerst pessimistisch** und **zum Teil falsch**
falsch für Prozessoren mit **Laufzeitanomalien**

- WCET der Sequenz > Summe der WCETs aller Instruktionen
pessimistisch für **moderne Prozessoren**
- Pipeline, Cache, Branch Prediction, Prefetching, ... haben großen
 Anteil an der verfügbaren Rechenleistung heutiger Prozessoren
- blanke Summation einzelner WCETs ignoriert diese Maßnahmen

Hardware-Analyse

Hardware-Analyse teilt sich in verschiedene Phasen

- Aufteilung ist nicht dogmenhaft festgeschrieben

Integration von Pfad- und Cache-Analyse

- ① Pipeline-Analyse
 - Wie lange dauert die Ausführung der Instruktionssequenz?
- ② Cache- und Pfad-Analyse sowie WCET-Berechnung
 - Cache-Analyse wird direkt in das Optimierungsproblem integriert

Separate Pfad- und Cache-Analyse

- ① Cache-Analyse
 - kategorisiert Speicherzugriffe mit Hilfe einer Datenflussanalyse
- ② Pipeline-Analyse
 - Ergebnisse der Cache-Analyse werden direkt berücksichtigt
- ③ Pfad-Analyse und WCET-Berechnung

Cache-Analyse [3, Kapitel 22]

Cache: ein kleiner, schneller Zwischenspeicher, Zugriffszeiten auf Daten/Instruktionen variieren je nach Zustand des Caches enorm:

Treffer (engl. *hit*), Daten/Instruktion sind im Cache $\sim e_h$

Fehlschlag (engl. *miss*), Daten/Instruktion sind nicht im Cache $\sim e_m$

Hits sind schneller als *Misses*: $e_m \gg e_h$ (> 100 Taktzyklen möglich)

Folgende Eigenschaften von Caches haben Einfluss auf seine Analyse

- Typ
- Cache für **Instruktionen**
 - Cache für **Daten**
 - kombinierter Cache für **Instruktionen und Daten**

- Auslegung
- direkt abgebildet (engl. *direct mapped*)
 - vollassoziativ (engl. *fully associative*)
 - satz- oder mengenassoziativ (engl. *set associative*)

Seitenersetzungsstrategie

- engl. *(pseudo) least recently used*, (Pseudo-)LRU
- engl. *(pseudo) first in first out*, (Pseudo-)FIFO

Ergebnisse der Cache-Analyse

Hilfreich ist, zu wissen, ob z.B. eine Instruktion im Cache ist, oder nicht:

must, die Instruktion ist *garantiert im Cache*

- ~ man kann immer die schnellere Ausführungszeit e_h annehmen
- wird für die Vorhersage von Treffern verwendet

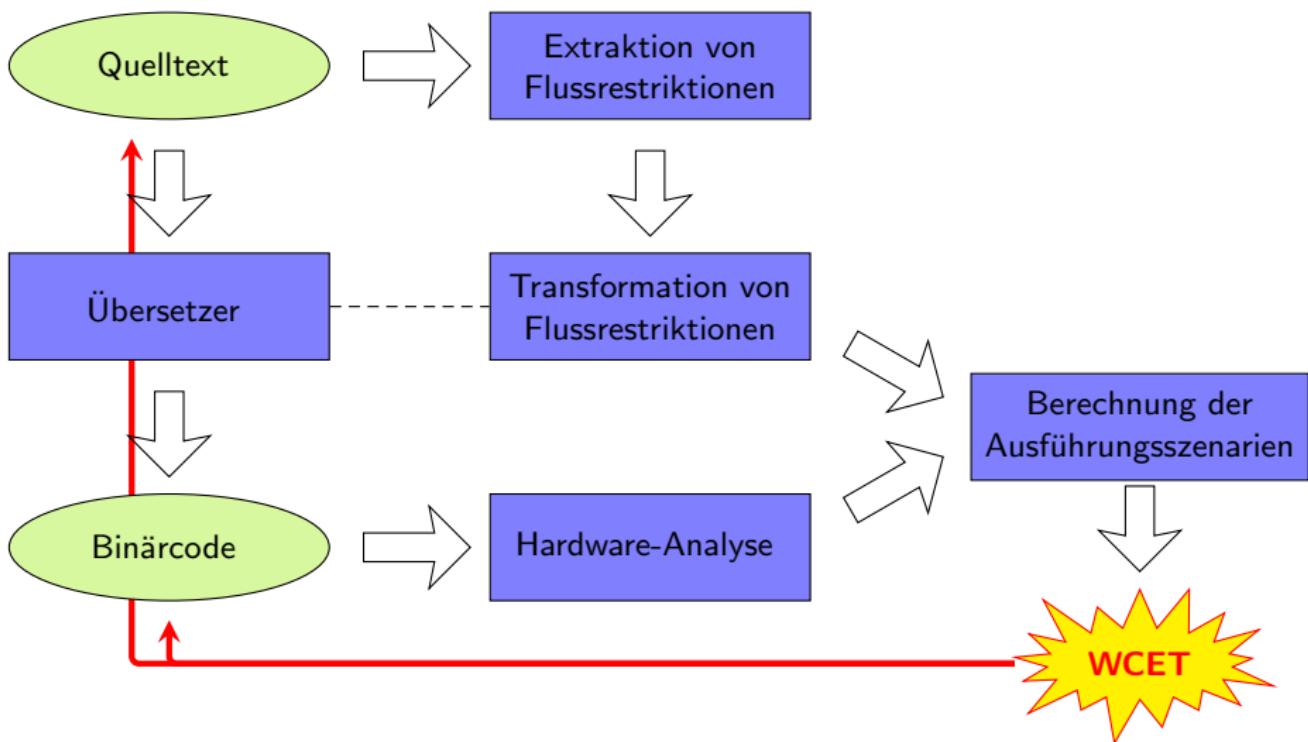
may, die Instruktion ist *vielleicht im Cache*

- ~ ist dies nicht der Fall, muss man die Ausführungszeit e_m annehmen
- wird für die Vorhersage von Fehlschlägen verwendet

persistent, die Instruktion *verbleibt im Cache*

- ~ erster Zugriff ist ein Fehlschlag, alle weiteren sind Treffer
- ~ erster Zugriff: e_m , weitere Zugriffe: e_h
 - ist besonders für Schleifen interessant, die den Cache „füllen“

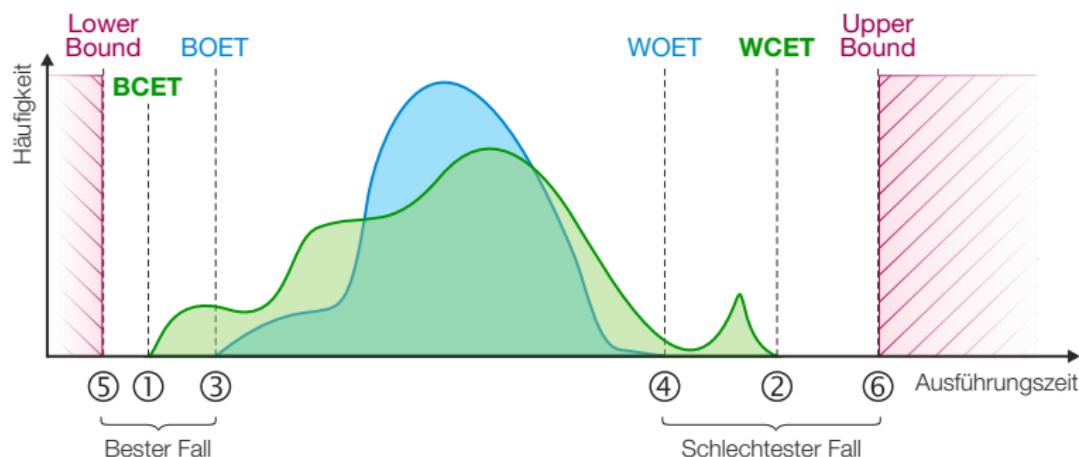
Werkzeugkette für die WCET-Analyse [2]



Gliederung

- 1 Übersicht
- 2 Problemstellung
- 3 Dynamische WCET-Analyse (Messbasiert)
- 4 Statische WCET-Analyse
- 5 Hardware-Analyse – Die Maschinenprogrammebene
- 6 Zusammenfassung

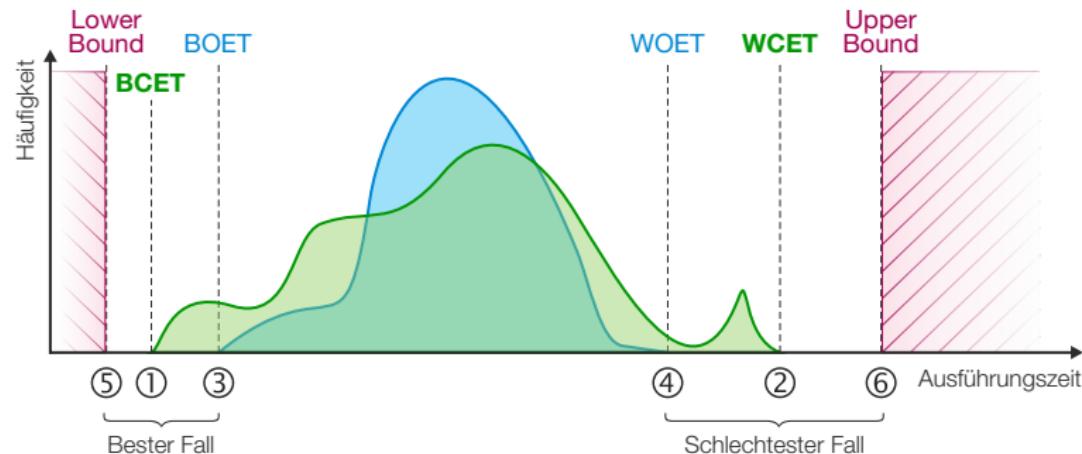
Resümee



WCET-Bestimmung gliedert sich grob in zwei Teilprobleme

- Programmiersprachenebene (makroskopisch) \leadsto finde die längsten Pfade durch ein Programm
 - Maschinenprogrammebene (mikroskopisch) \leadsto bestimmte die WCET der Elementaroperationen
- ☞ Tatsächliche Ausführungszeit: BCET / WCET

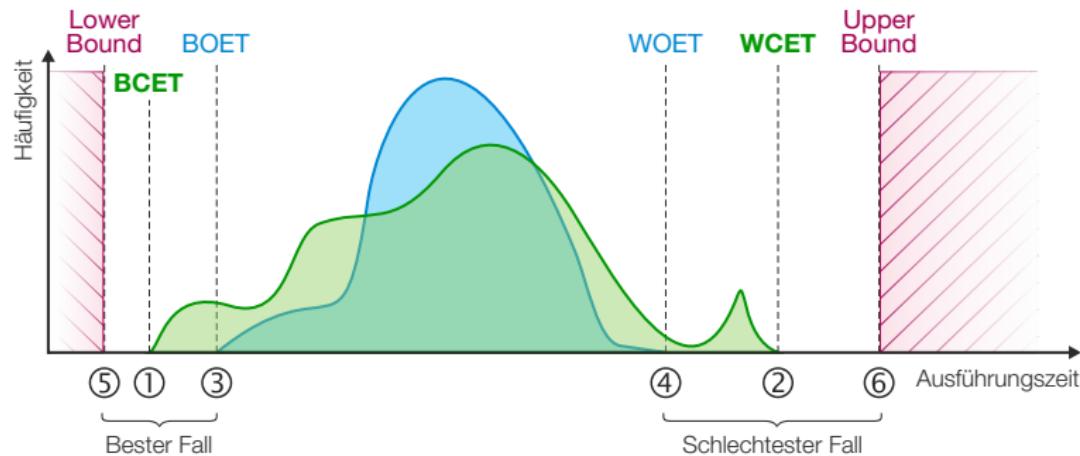
Resümee



Dynamische Analyse \sim beobachtet die Ausführungszeit

- Messung bezieht beide Ebenen mit ein
- Vollständige Messung im Allgemeinen **nicht möglich** \sim **Unterapproximation**
- 👉 Gemessene Ausführungszeit: BOET / WOET

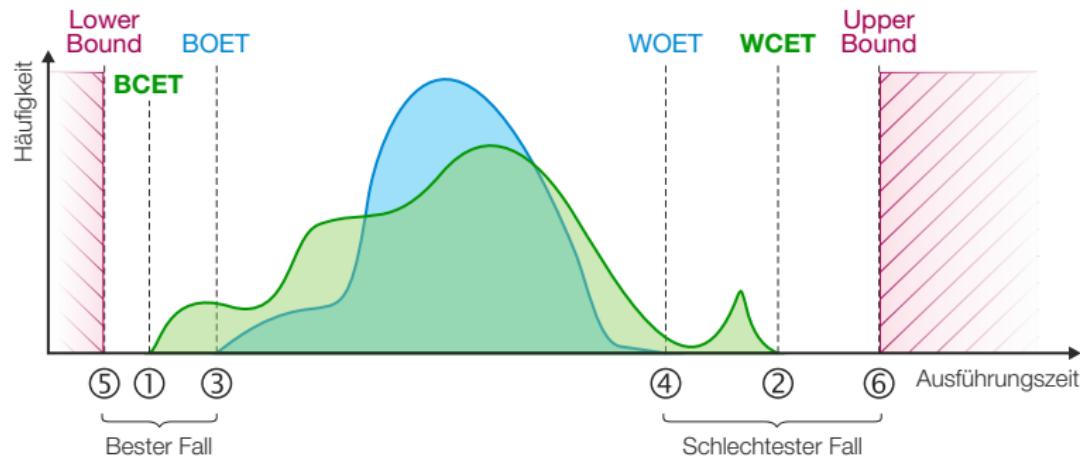
Resümee



Statische Analyse \leadsto schätzt die Ausführungszeit

- Pfadanalyse (Programmiersprachenebene) \leadsto Abstraktion (Timing Schema vs. IPET)
- Gibt pessimistische Schranken an \leadsto Überapproximation
-  Geschätzte Ausführungszeitgrenzen: Lower- / Upper Bound

Resümee



Hardware-Analyse \leadsto Eingaben für die WCET-Berechnung

- Hauptaufgaben: Cache- und Pipeline-Analyse
- must-Approximation und may-Approximation
- Werkzeugunterstützung kombiniert Ebenen und macht die WCET-Analyse handhabbar

Literaturverzeichnis

- [1] PUSCHNER, P. :
Zeitanalyse von Echtzeitprogrammen.
Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Technische Universität Wien, Institut für
Technische Informatik, Diss., 1993
- [2] PUSCHNER, P. ; HUBER, B. :
Zeitanalyse von sicherheitskritischen Echtzeitsystemen.
<http://ti.tuwien.ac.at/rts/teaching/courses/wcet>, 2012. –
Lecture Notes
- [3] WILHELM, R. :
Embedded Systems.
<http://react.cs.uni-sb.de/teaching/embedded-systems-10-11/lecture-notes.html>,
2010. –
Lecture Notes