

# Verlässliche Echtzeitsysteme

Verifikation nicht-funktionaler Eigenschaften

**Peter Ulbrich**

Lehrstuhl für Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität Erlangen-Nürnberg

<https://www4.cs.fau.de>

29. Juni 2017



# Wiederholung: Verifikationsverfahren

## Abstraktion

Beispiel	Testfall	Clang Address Sanitizer	Clang Static Analyser	Astrée	framaC, Isabelle
Ebene	$\mu$ -Code	$\mu$ -Code + Instrumentierung	Zwischensprache / AST <sup>1</sup>	Abstrakte Programmsemantik	Abstrakte Semantik + Kontrakt
Technik	return()	assert()	Symbolische Ausführung / Erreichbarkeitsanalyse	Abstrakte Interpretation	Abstrakte Interpretation / Verifikation
Konkrete Maschine	CPU	CPU	Compiler	Model Checker	Model Checker

dynamisch      statisch      unsicher      sicher

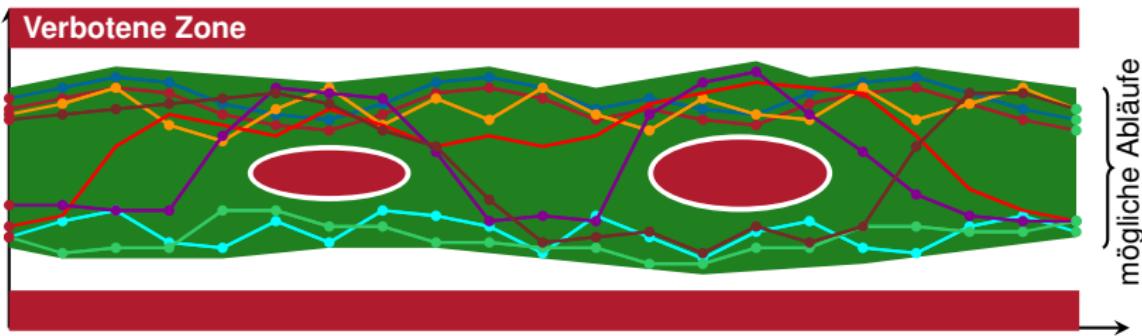
## ■ Statisch versus dynamisch

- Nutzung der konkreten/abstrakten Programmsemantik (siehe Folien VIII/15 ff)
- Konkrete Ausführung (Maschine) hängt jedoch von der Betrachtungsebene ab!

## ■ Sicher versus unsicher

- Vollständigkeit der Analyse (sicher  $\mapsto$  100 %, siehe Folien VIII/20 ff)
- Steht im Bezug zu einer bestimmten Spezifikation (z.B. C-Standard bei Astrée)





- Abstrakte Interpretation (engl. *abstract interpretation*)
  - Betrachtet eine abstrakte Semantik (engl. *abstract semantics*)
    - Sie umfasst alle Fälle der konkreten Programmsemantik
  - Sicherheitszonen beschreiben fehlerhafte Zustände
  - Ist die abstrakte Semantik sicher  $\Rightarrow$  konkrete Semantik ist sicher

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
  - **Abstrakte Interpretation:**  
Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
  - **Abstrakte Interpretation:**  
Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)



Dies ist **notwendig** jedoch **nicht hinreichend**

- Einfluss nicht-funktionaler Eigenschaften der Ausführungsumgebung
  - Anwendung ist in die Umwelt eingebettet!
  - Exemplarisch: **Speicherverbrauch** und **Laufzeit**

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund

- **Abstrakte Interpretation:**

- Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)



Dies ist **notwendig** jedoch **nicht hinreichend**

- Einfluss nicht-funktionaler Eigenschaften der Ausführungsumgebung

- Anwendung ist in die Umwelt eingebettet!

- Exemplarisch: **Speicherverbrauch** und **Laufzeit**



**Einhaltung** bestimmter **nicht-funktionaler Eigenschaften** garantieren?

- Speicherverbrauch: **Worst-Case Stack Usage** (WCSU, siehe 7 ff)

- Laufzeit: **Worst-Case Execution Time** (WCET, siehe 14 ff)

- Messung versus statische Analyse



## 1 Speicherverbrauch

- Überblick
- Messbasierte Bestimmung
- Analytische Bestimmung

## 2 Ausführungszeit

- Überblick
- Messbasierte WCET-Analyse
- Statische WCET-Analyse

## 3 Zusammenfassung



### ☞ Betrachtung des Speicherverbrauchs nach Lokalität

#### ■ Festwertspeicher (engl. *Read Only Memory, ROM*)

- Umfasst die Übersetzungseinheiten (**Funktionen** und **Konstanten**)
- **Architekturabhängig** (Wortbreite, Optimierungsstufe, Inlineing, ...)
- Größe ist dem Compiler/Linker **statisch bekannt**:

```
gcc -Wl,-Map,PROGRAM.map *.o -o PROGRAM
```



### Betrachtung des Speicherverbrauchs nach Lokalität

#### ■ Festwertspeicher (engl. *Read Only Memory, ROM*)

- Umfasst die Übersetzungseinheiten (**Funktionen** und **Konstanten**)
- **Architekturabhängig** (Wortbreite, Optimierungsstufe, Inlineing, ...)
- Größe ist dem Compiler/Linker **statisch bekannt**:

```
gcc -Wl,-Map,PROGRAM.map *.o -o PROGRAM
```

#### ■ Direktzugriffspeicher (engl. *Random Access Memory, RAM*)

- In eingebetteten Systemen typischerweise statisch allokiert (**globale Variablen** & **Stapspeicher**-Konfiguration)
- Permanenter Verbrauch (**architekturabhängig**) ebenso **statisch bekannt**



### Betrachtung des Speicherverbrauchs nach Lokalität

#### ■ Festwertspeicher (engl. *Read Only Memory, ROM*)

- Umfasst die Übersetzungseinheiten (**Funktionen** und **Konstanten**)
- **Architekturabhängig** (Wortbreite, Optimierungsstufe, Inlineing, ...)
- Größe ist dem Compiler/Linker **statisch bekannt**:

```
gcc -Wl,-Map,PROGRAM.map *.o -o PROGRAM
```

#### ■ Direktzugriffsspeicher (engl. *Random Access Memory, RAM*)

- In eingebetteten Systemen typischerweise statisch allokiert (**globale Variablen** & **Stapspeicher**-Konfiguration)
- Permanenter Verbrauch (**architekturabhängig**) ebenso **statisch bekannt**

## Dynamischer Speicher in eingebetteten Systemen

Wird typischerweise auf den **Stapspeicher** (engl. *Stack*) abgebildet



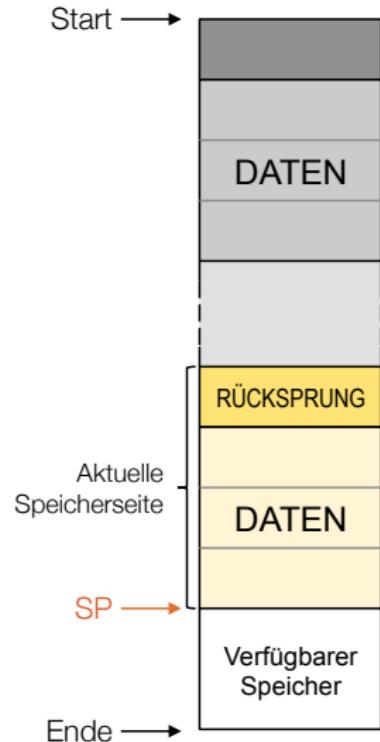
# Der Stapselspeicher (Stack)

Dynamische Nutzung von Speicher ist eingebetteten Systemen



Stapselspeicher wird verwendet für:

- Lokale Variablen und Zwischenwerte
- Funktionsparameter
- Rücksprungadressen



# Der Stapselspeicher (Stack)

Dynamische Nutzung von Speicher ist eingebetteten Systemen

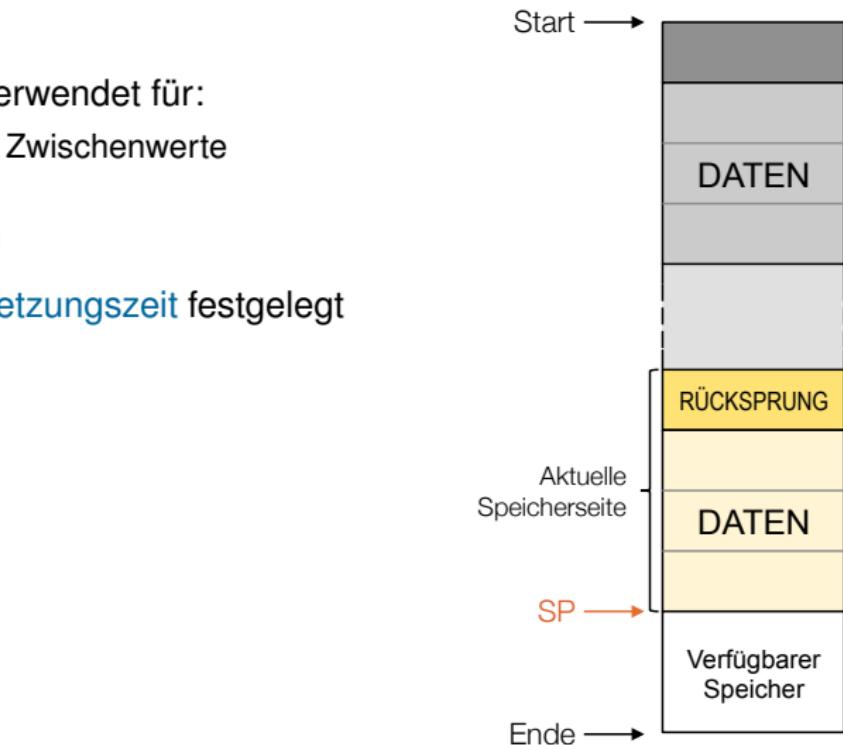


Stapselspeicher wird verwendet für:

- Lokale Variablen und Zwischenwerte
- Funktionsparameter
- Rücksprungadressen



Größe wird zur Übersetzungszeit festgelegt



# Der Stapselspeicher (Stack)

Dynamische Nutzung von Speicher ist eingebetteten Systemen



Stapselspeicher wird verwendet für:

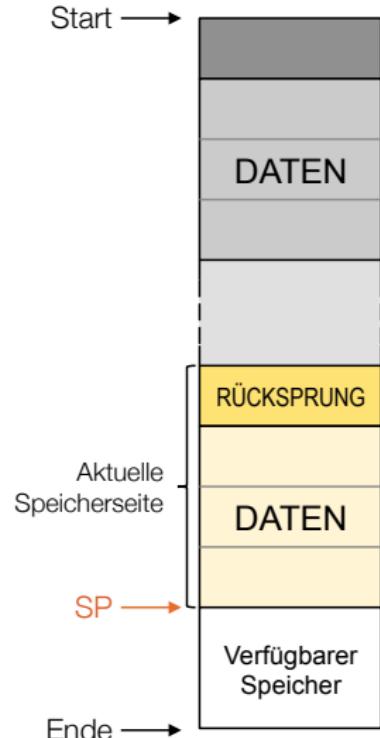
- Lokale Variablen und Zwischenwerte
- Funktionsparameter
- Rücksprungadressen



Größe wird zur Übersetzungszeit festgelegt

## Fehlerquelle Stapselspeicher

- Unterdimensionierung  $\leadsto$  Überlauf
- Größenbestimmung  $\approx$  Halteproblem



# Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [7]



## ■ Elektronisches Stellwerk

- Hersteller: Siemens
- Simis-3216 (i486)
- Inbetriebnahme: 12. März 1995
- Kosten: 62,6 Mio DM
- Ersetzte 8 Stellwerke (1911-52)



# Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [7]



## ■ Elektronisches Stellwerk

- Hersteller: Siemens
- Simis-3216 (i486)
- Inbetriebnahme: 12. März 1995
- Kosten: 62,6 Mio DM
- Ersetzte 8 Stellwerke (1911-52)



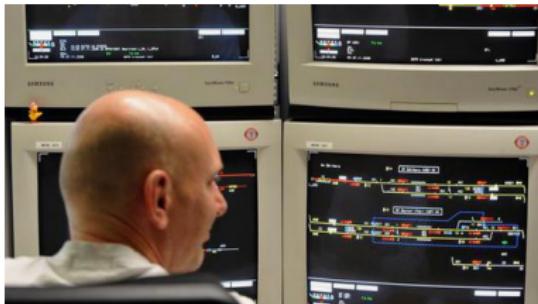
## Dynamische Verwaltung der Stellbefehle auf dem Stapelspeicher

- Initial 3.5 KiB  $\sim$  zu klein schon für normalen Verkehr
- Fehlerbehandlungsroutine fehlerhaft  $\sim$  Endlosschleife
- Notabschaltung durch Sicherungsmaßnahmen (fail-stop)



# Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [7]



## ■ Elektronisches Stellwerk

- Hersteller: Siemens
- Simis-3216 (i486)
- Inbetriebnahme: 12. März 1995
- Kosten: 62,6 Mio DM
- Ersetzte 8 Stellwerke (1911-52)



Dynamische Verwaltung der Stellbefehle auf dem Stapelspeicher

- Initial 3.5 KiB  $\sim$  zu klein schon für normalen Verkehr
- Fehlerbehandlungsroutine fehlerhaft  $\sim$  Endlosschleife
- Notabschaltung durch Sicherungsmaßnahmen (fail-stop)

Ausfall am Tag der Inbetriebnahme

Kein Schienenverkehr für **2 Tage**, 2 Monate Notfahrplan



pu

# Bestimmung des Stapelspeicherverbrauchs



Überabschätzung führt zu unnötigen Kosten



# Bestimmung des Stapelspeicherverbrauchs



Überabschätzung führt zu unnötigen Kosten



Unterabschätzung des Speicherverbrauchs führt zu Stapelüberlauf

- Schwerwiegendes und komplexes Fehlermuster
- Undefiniertes Verhalten, **Datenfehler** oder Programmabsturz
- Schwer zu finden, reproduzieren und beheben!





Überabschätzung führt zu unnötigen Kosten



Unterabschätzung des Speicherverbrauchs führt zu Stapelüberlauf

- Schwerwiegendes und komplexes Fehlermuster
- Undefiniertes Verhalten, **Datenfehler** oder Programmabsturz
- Schwer zu finden, reproduzieren und beheben!

## ■ Voraussetzungen für sinnvolle Analyse

- Zyklische Ausführungspfade vermeiden
- Keine **Rekursion**, **Funktionszeiger**, **dynamischer Speicher**





Überabschätzung führt zu unnötigen Kosten



Unterabschätzung des Speicherverbrauchs führt zu Stapelüberlauf

- Schwerwiegendes und komplexes Fehlermuster
- Undefiniertes Verhalten, **Datenfehler** oder Programmabsturz
- Schwer zu finden, reproduzieren und beheben!

## ■ Voraussetzungen für sinnvolle Analyse

- Zyklische Ausführungspfade vermeiden
- Keine **Rekursion**, **Funktionszeiger**, **dynamischer Speicher**



Analyse gängiger Compiler

- gcc -fstack-usage ist **nicht genug**
- Richtwert bei der Entwicklung einzelner Funktionen

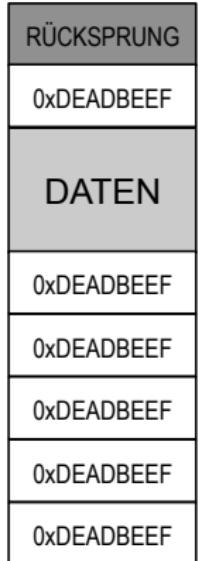


# Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie VII/19 ff.)

## ■ Messung (Water-Marking, Stack Canaries)

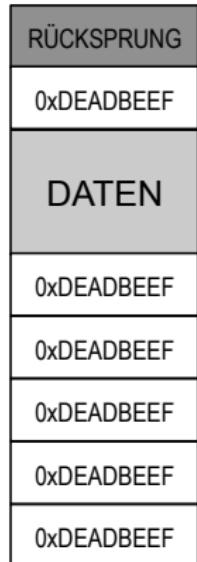
- Stapelspeicher wird **vorinitialisiert** (z.B. 0xDEADBEEF)
- Maximaler Verbrauch **der Ausführung**  
    ~ höchste Speicherstelle ohne Wasserzeichen
- Auf Rücksprungadressen anwendbar



# Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie VII/19 ff.)

- **Messung** (Water-Marking, Stack Canaries)
  - Stapelspeicher wird **vorinitialisiert** (z.B. 0xDEADBEEF)
  - Maximaler Verbrauch **der Ausführung**  
→ höchste Speicherstelle ohne Wasserzeichen
  - Auf Rücksprungadressen anwendbar
- Systemüberwachung zur Laufzeit
  - Verfahren gut geeignet zur dynamischen Fehlererkennung
  - **Stack Check** (o.ä.) in AUTOSAR, OSEK, ...



# Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie VII/19 ff.)

- **Messung** (Water-Marking, Stack Canaries)
  - Stapelspeicher wird **vorinitialisiert** (z.B. 0xDEADBEEF)
  - Maximaler Verbrauch **der Ausführung**  
→ höchste Speicherstelle ohne Wasserzeichen
  - Auf Rücksprungadressen anwendbar
- Systemüberwachung zur Laufzeit
  - Verfahren gut geeignet zur dynamischen Fehlererkennung
  - **Stack Check** (o.ä.) in AUTOSAR, OSEK, ...

- ⚠ Keine Aussagen zum maximalen Speicherverbrauch
- Liefert nur den konkreten Verbrauch der Messungen
  - **Fehleranfällig** und **aufwendig**
  - Keine Garantien möglich!

RÜCKSPRUNG
0xDEADBEEF
DATEN
0xDEADBEEF



```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```



Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf



```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```



Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf

- Speicherausrichtung (engl. *alignment*) von Variablen und Parametern
  - Abhängig von Binärschnittstelle (engl. *Application Binary Interface, ABI*)
  - In diesem Beispiel 16 Byte (und mehr)



```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```



Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf

- Speicherausrichtung (engl. *alignment*) von Variablen und Parametern
  - Abhängig von Binärschnittstelle (engl. *Application Binary Interface, ABI*)
  - In diesem Beispiel 16 Byte (und mehr)
- Aufrufort der Funktion unbekannt
  - Segmentierung kann zu nahen und fernen Aufrufen führen
  - Rücksprungadressen unterschiedlicher Größen



```
1 unsigned int function(unsigned char a, unsigned char b) {  
2     unsigned int c;  
3     unsigned char d;  
4     /* code */  
5     return c;  
6 }
```



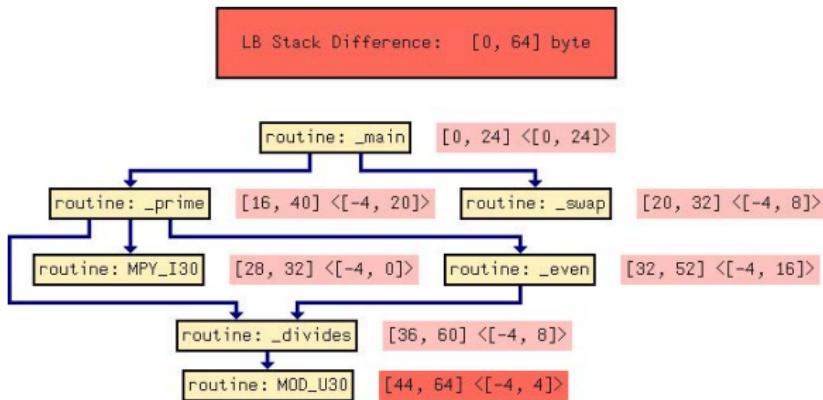
Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf

- Speicherausrichtung (engl. *alignment*) von Variablen und Parametern
  - Abhängig von **Binärschnittstelle** (engl. *Application Binary Interface, ABI*)
  - In diesem Beispiel 16 Byte (und mehr)
- Aufrufort der Funktion unbekannt
  - Segmentierung kann zu nahen und fernen Aufrufen führen
  - Rücksprungadressen unterschiedlicher Größen
- Inline-Ersetzung der Funktion (kein Stapelverbrauch für Aufruf)



# Bestimmung des maximalen Stapelspeicherverbrauchs

Durch abstrakte Interpretation des Programmcodes [1, 5]



- Statische Analyse des **Kontrollfluss**- und **Aufrufgraphen**
  - Pufferüberlauf als weitere Form von Laufzeitfehler
  - Vorgehen analog zum Korrektheitsnachweis
- Weist **Abwesenheit** von Pufferüberläufen nach
  - Pfadanalyse  $\leadsto$  Maximaler Speicherverbrauch
  - Ausrollen von Schleifen (siehe Folie X/32)
  - Partitionierung und Werteanalyse (siehe Folie X/33)

## 1 Speicherverbrauch

- Überblick
- Messbasierte Bestimmung
- Analytische Bestimmung

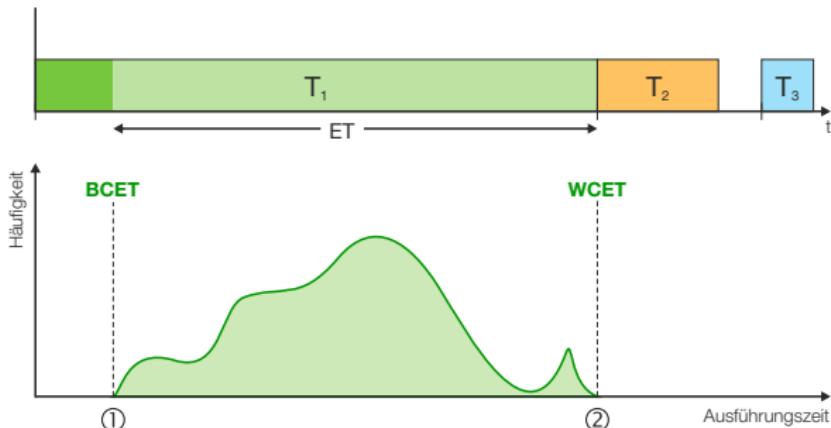
## 2 Ausführungszeit

- Überblick
- Messbasierte WCET-Analyse
- Statische WCET-Analyse

## 3 Zusammenfassung

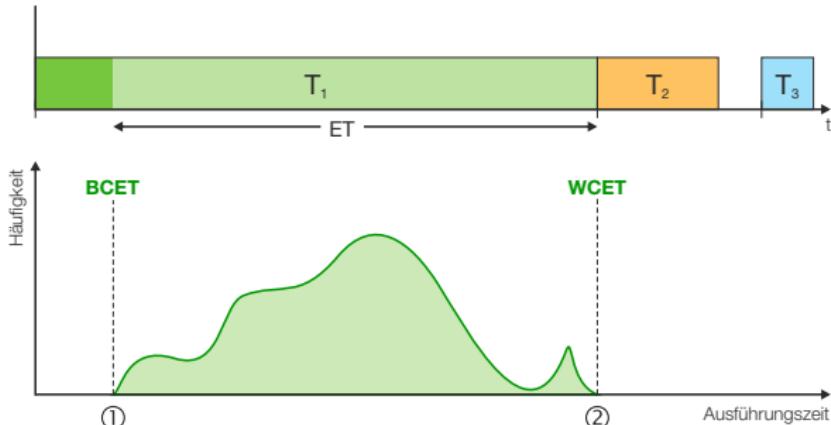


# Die maximalen Ausführungszeiten



- Alle sprechen von der maximalen Ausführungszeit ( $e$ )
  - *Worst Case Execution Time (WCET)*  $e_i$  (vgl. [6] Folie III-2/28)

# Die maximalen Ausführungszeiten



- Alle sprechen von der maximalen Ausführungszeit ( $e$ )
  - *Worst Case Execution Time (WCET)*  $e_i$  (vgl. [6] Folie III-2/28)
- Tatsächliche Ausführungszeit bewegt sich zwischen:
  - 1 bestmöglicher Ausführungszeit (*Best Case Execution Time, BCET*)
  - 2 schlechtest möglicher Ausführungszeit (besagter *WCET*)



# Bestimmung der WCET – eine Herausforderung

Wovon hängt die maximale Ausführungszeit 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;  
}
```



## 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 von dessen Inhalt
- ⚠ Exakte Vorhersage ist kaum möglich
- Größe und Inhalt von a[ ] kann zur Laufzeit variieren
  - Welches ist der längste Pfad?



# Bestimmung der WCET – eine Herausforderung

Wovon hängt die maximale Ausführungszeit 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 von dessen Inhalt
- ⚠ Exakte Vorhersage ist kaum möglich
  - Größe und Inhalt von a[ ] kann zur Laufzeit variieren
  - Welches ist der längste Pfad?

## ■ Maschinenprogrammebene:

- Ausführungszeit der Elementaroperationen (ADD, LOAD, ...)
- ⚠ 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, ...





- Idee: Prozessor selbst ist das präziseste Hardware-Modell
  - Dynamische Ausführung und Beobachtung der Ausführungszeit



- ☞ Idee: Prozessor selbst ist das präziseste Hardware-Modell
  - Dynamische Ausführung und Beobachtung der Ausführungszeit
- Messbasierte WCET-Analyse:
  - Intuitiv und gängige Praxis in der Industrie
  - Weiche/feste Echtzeitsysteme erfordern keine sichere WCET
  - Einfach umzusetzen, verfügbar und anpassbar
    - Verschafft leicht Orientierung über die tatsächliche Laufzeit
    - Geringer Aufwand zur Instrumentierung (Plattformwechsel)
    - Eingeschränkte Verfügbarkeit statischer Analysewerkzeuge (HW-Plattform)
  - Sinnvolle Ergänzung zur statischen WCET-Analyse (Folie X/19ff)
    - Validierung statisch bestimmter Werte
    - Ausgangspunkt für die Verbesserung der statischen Analyse

- ☞ Idee: Prozessor selbst ist das präziseste Hardware-Modell
  - Dynamische Ausführung und Beobachtung der Ausführungszeit
- Messbasierte WCET-Analyse:
  - Intuitiv und gängige Praxis in der Industrie
  - Weiche/feste Echtzeitsysteme erfordern keine sichere WCET
  - Einfach umzusetzen, verfügbar und anpassbar
    - Verschafft leicht Orientierung über die tatsächliche Laufzeit
    - Geringer Aufwand zur Instrumentierung (Plattformwechsel)
    - Eingeschränkte Verfügbarkeit statischer Analysewerkzeuge (HW-Plattform)
  - Sinnvolle Ergänzung zur statischen WCET-Analyse (Folie X/19ff)
    - Validierung statisch bestimmter Werte
    - Ausgangspunkt für die Verbesserung der statischen Analyse



Das Richtigste zu messen ist das Problem!



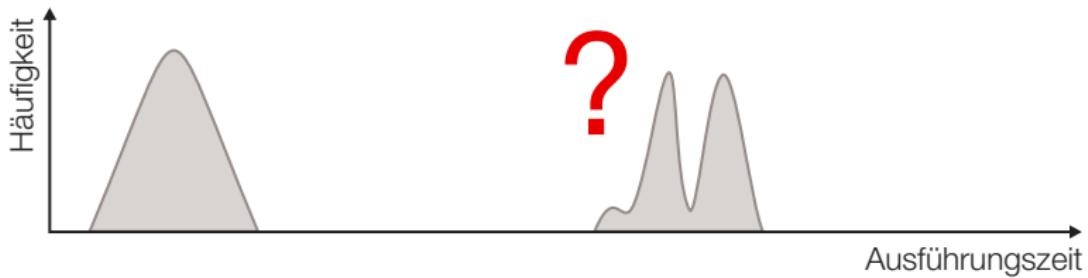
# Herausforderungen der Messung

- ☞ Messungen umfassen stets das **Gesamtsystem**
  - Hardware, Betriebssystem, Anwendung(en), ...
- ⚠ **Fluch** und **Segen**



# Herausforderungen der Messung

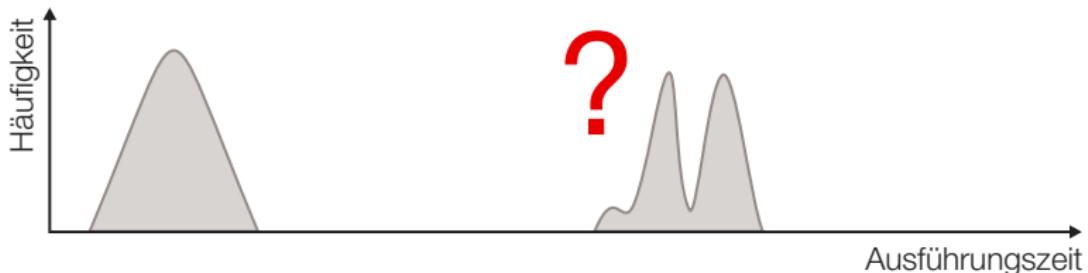
- ☞ Messungen umfassen stets das **Gesamtsystem**
  - Hardware, Betriebssystem, Anwendung(en), ...
- ⚠ **Fluch und Segen**
- Mögliches Ergebnis einer Messung:



# Herausforderungen der Messung

- Messungen umfassen stets das **Gesamtsystem**
  - Hardware, Betriebssystem, Anwendung(en), ...
- ⚠ **Fluch und Segen**

## ■ Mögliches Ergebnis einer Messung:

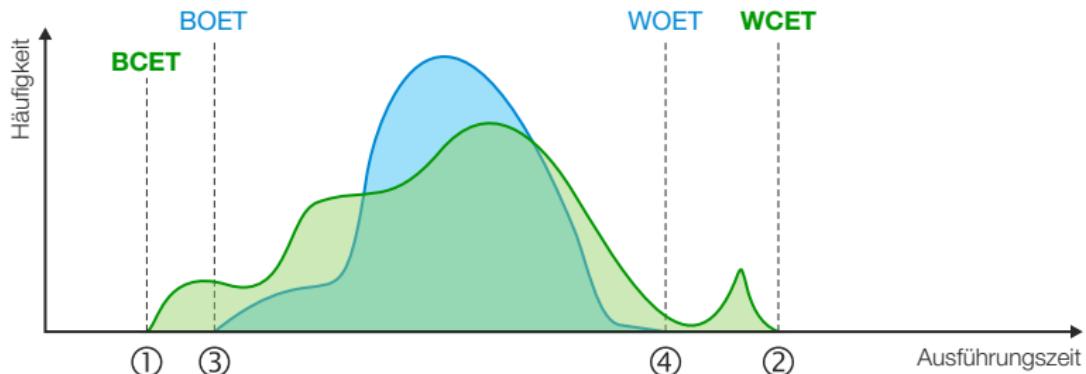


## Probleme und Anomalien

- Nebenläufige Ereignisse unterbinden → Verdrängung
- Gewählte Testdaten führen nicht unbedingt zum **längsten Pfad**
- Seltene Ausführungsszenarien → Ausnahmefall
- Abschnittsweise WCET-Messung ↗ Globalen WCET
- Wiederherstellung des **Hardwarezustandes** schwierig/unmöglich

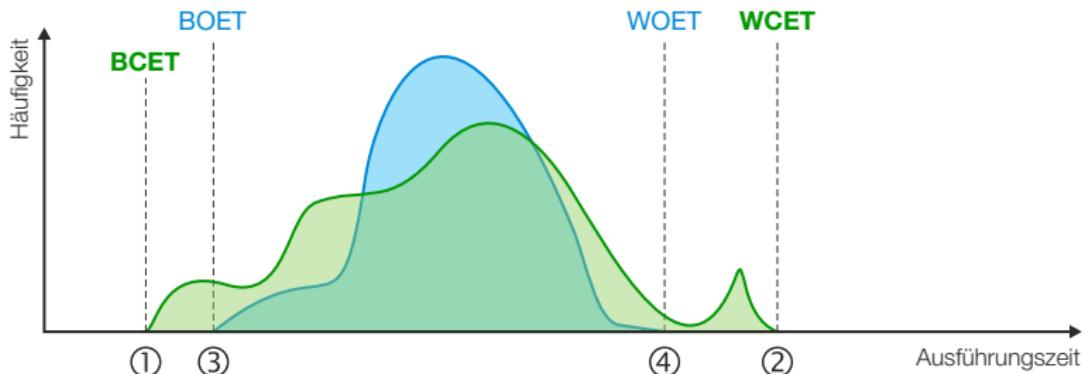


# Aussagekraft messbasierter WCET-Analyse



- Dynamische WCET-Analyse liefert **Messwerte**:
  - 3 Bestmögliche beobachtete Ausführungszeit  
(Best Observed Execution Time, **BOET**)
  - 4 Schlechtest mögliche beobachtete Ausführungszeit  
(Worst Observed Execution Time, **WOET**)

# Aussagekraft messbasierter WCET-Analyse



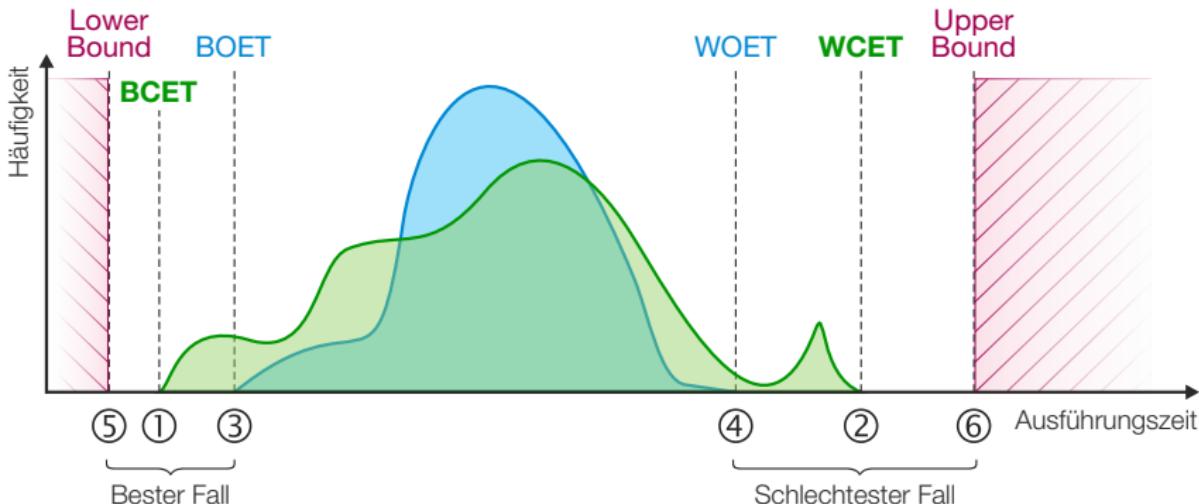
- Dynamische WCET-Analyse liefert **Messwerte**:

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

⚠️ Messbasierte Ansätze unterschätzen die WCET meistens



# 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: Längster Pfad

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;  
}
```



# Problem: Längster Pfad

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;  
    }  
}
```

Aufruf: bubbleSort(a, size)

- Durchläufe, Vergleiche und Vertauschungen (engl. **Swap**)
- a = {1, 2}, size = 2  
→ D = 1, V = 1, **S = 0**;
- a = {1, 3, 2}, size = 3  
→ D = 3, V = 3, **S = 1**;
- a = {3, 2, 1}, size = 3  
→ D = 3, V = 3, **S = 3**;



# Problem: Längster Pfad

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;  
}
```

Aufruf: bubbleSort(a, size)

- Durchläufe, Vergleiche und Vertauschungen (engl. **Swap**)
- a = {1, 2}, size = 2  
→ D = 1, V = 1, **S = 0**;
- a = {1, 3, 2}, size = 3  
→ D = 3, V = 3, **S = 1**;
- a = {3, 2, 1}, size = 3  
→ D = 3, V = 3, **S = 3**;



Für den **allgemeinen Fall** nicht berechenbar → Halteproblem

- Wie viele Schleifendurchläufe werden benötigt?



In Echtzeitsystemen ist dieses Problem häufig lösbar

- Kanonische Schleifenkonstrukte beschränkter Größe → `max(size)`
- Pfadanalyse → Nur **maximale Pfadlänge** von belang



## Berechnung 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:

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



## Berechnung 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



# Berechnung 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:** Vereinfachung der konkreten Pfadsemantik

- Abstraktion und Abbildung auf ein Flussproblem  
(vgl. Abstrakte Semantik, VIII/20 ff)
- Flussprobleme sind mathematisch gut untersucht
- Im folgenden zwei Lösungswege: Timing Schema und IPET

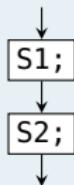


# Lösungsweg<sub>1</sub>: Timing Schema

Eine einfache Form der Sammelsemantik (siehe Folie VIII/25)

Sequenzen  $\leadsto$  Hintereinanderausführung

S1();  
S2();



# Lösungsweg<sub>1</sub>: Timing Schema

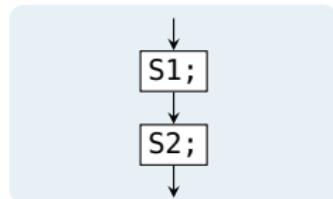
Eine einfache Form der Sammelsemantik (siehe Folie VIII/25)

Sequenzen  $\leadsto$  Hintereinanderausführung

S1();  
S2();

Summation der WCETs:

$$e_{seq} = e_{S1} + e_{S2}$$



# Lösungsweg<sub>1</sub>: Timing Schema

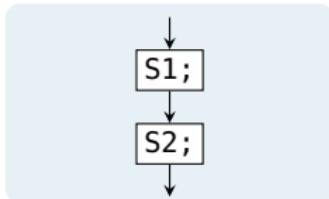
Eine einfache Form der Sammelsemantik (siehe Folie VIII/25)

Sequenzen  $\leadsto$  Hintereinanderausführung

```
S1();  
S2();
```

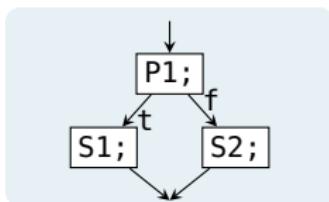
Summation der WCETs:

$$e_{seq} = e_{S1} + e_{S2}$$



Verzweigung  $\leadsto$  Bedingte Ausführung

```
if(P1())  
    S1();  
else S2();
```



# Lösungsweg<sub>1</sub>: Timing Schema

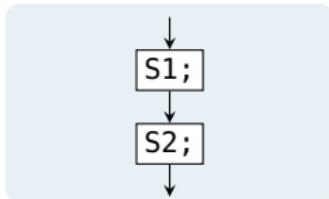
Eine einfache Form der Sammelsemantik (siehe Folie VIII/25)

Sequenzen  $\leadsto$  Hintereinanderausführung

```
S1();  
S2();
```

Summation der WCETs:

$$e_{seq} = e_{S1} + e_{S2}$$

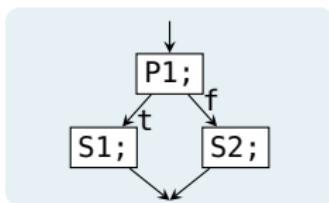


Verzweigung  $\leadsto$  Bedingte Ausführung

```
if(P1())  
  S1();  
else S2();
```

Maximale Gesamtausführungszeit:

$$e_{cond} = e_{P1} + \max(e_{S1}, e_{S2})$$



# Lösungsweg<sub>1</sub>: Timing Schema

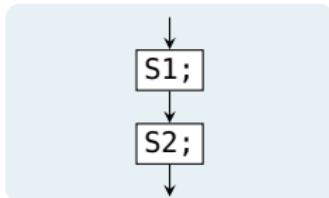
Eine einfache Form der Sammelsemantik (siehe Folie VIII/25)

Sequenzen  $\leadsto$  Hintereinanderausführung

```
S1();  
S2();
```

Summation der WCETs:

$$e_{seq} = e_{S1} + e_{S2}$$

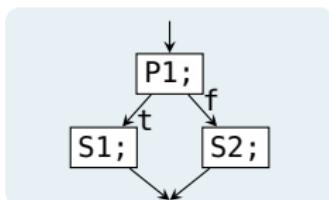


Verzweigung  $\leadsto$  Bedingte Ausführung

```
if(P1())  
  S1();  
else S2();
```

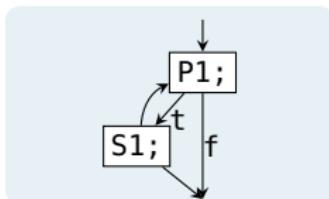
Maximale Gesamtausführungszeit:

$$e_{cond} = e_{P1} + \max(e_{S1}, e_{S2})$$



Schleifen  $\leadsto$  Wiederholte Ausführung

```
while(P1())  
  S1();
```



# Lösungsweg<sub>1</sub>: Timing Schema

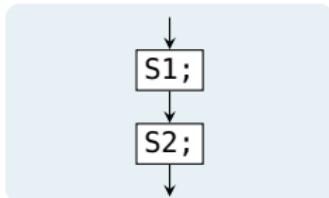
Eine einfache Form der Sammelsemantik (siehe Folie VIII/25)

Sequenzen  $\leadsto$  Hintereinanderausführung

```
S1();  
S2();
```

Summation der WCETs:

$$e_{seq} = e_{S1} + e_{S2}$$

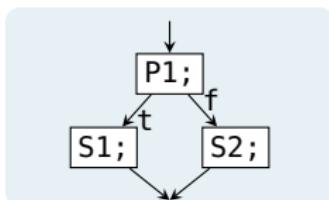


Verzweigung  $\leadsto$  Bedingte Ausführung

```
if(P1())  
  S1();  
else S2();
```

Maximale Gesamtausführungszeit:

$$e_{cond} = e_{P1} + \max(e_{S1}, e_{S2})$$

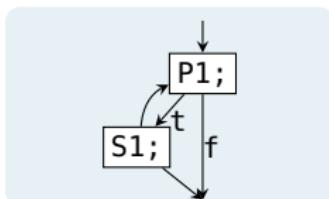


Schleifen  $\leadsto$  Wiederholte Ausführung

```
while(P1())  
  S1();
```

Schleifendurchläufe berücksichtigen:

$$e_{loop} = e_{P1} + n(e_{P1} + e_{S1})$$



## ■ Eigenschaften

- Traversierung des abstrakten Syntaxbaums (AST) **bottom-up**
  - An den Blättern beginnend bis zur Wurzel
- **Aggregation** der maximale Ausführungszeit nach festen Regeln
  - Für Sequenzen, Verzweigungen und Schleifen



## ■ Eigenschaften

- Traversierung des abstrakten Syntaxbaums (AST) **bottom-up**
  - An den Blättern beginnend bis zur Wurzel
- **Aggregation** der maximale Ausführungszeit nach festen Regeln
  - Für Sequenzen, Verzweigungen und Schleifen

## ■ Vorteile

- + Einfaches Verfahren mit geringem Berechnungsaufwand
- + Skaliert gut mit der Programmgröße

## ■ Eigenschaften

- Traversierung des abstrakten Syntaxbaums (AST) **bottom-up**
  - An den Blättern beginnend bis zur Wurzel
- **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-lokaler Codeteile lassen sich nicht berücksichtigen
  - Schwierige Integration mit einer separaten Hardware-Analyse
- Nichtrealisierbare Pfade (infeasible paths) nicht ausschließbar
  - ~ unnötige Überapproximation





Explizite Pfadanalyse ohne Vereinfachung nicht handhabbar



Lösungsansatz<sub>2</sub>: Nutzung impliziter Pfadaufzählungen  $\leadsto$  **Implicit Path Enumeration Technique (IPET)** [3]

<sup>1</sup><http://gurobi.com/>



Explizite Pfadanalyse ohne Vereinfachung nicht handhabbar



Lösungsansatz<sub>2</sub>: Nutzung impliziter Pfadaufzählungen  $\leadsto$  Implicit Path Enumeration Technique (IPET) [3]

■ **Vorgehen:** Transformation des Kontrollflussgraphen in ein ganzzahliges, lineares Optimierungsproblem (ILP)

- 1 Bestimmung des **Zeitanalysegraphs** aus dem Kontrollflussgraphen
- 2 Abbildung auf ein **lineare Optimierungsproblem**
- 3 Annotation von **Flussrestriktionen**
  - Nebenbedingungen im Optimierungsproblem
- 4 Lösung des Optimierungsproblems (z.B. mit **gurobi**<sup>1</sup>)



Globale Vereinfachung des Graphen statt lokaler Aggregierung



---

<sup>1</sup><http://gurobi.com/>

## Der Zeitanalysegraph (engl. *timing analysis graph*)

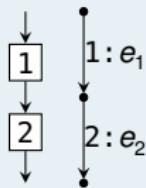
- Ein **Zeitanalysegraph (T-Graph)** ist ein gerichteter Graph mit einer Menge von Knoten  $\mathcal{V} = \{V_i\}$  und Kanten  $\mathcal{E} = \{E_i\}$ 
  - Mit genau einer **Quelle** und einer **Senke**
  - Jede Kante ist Bestandteile eines Pfads  $P$  von der Senke zur Kante
  - Jeder Kante wird ihre WCET  $e_i$  zugeordnet



# Der Zeitanalysegraph (engl. *timing analysis graph*)

- Ein **Zeitanalysegraph (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**
  - Jede Kante ist Bestandteile eines Pfads  $P$  von der Senke zur Kante  
→ Jeder Kante wird ihre WCET  $e_i$  zugeordnet

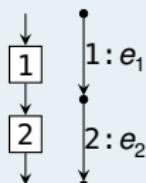
## Sequenz



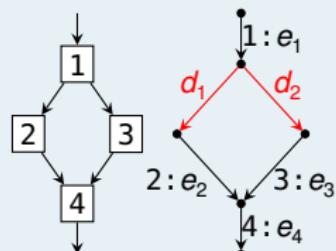
# Der Zeitanalysegraph (engl. *timing analysis graph*)

- Ein **Zeitanalysegraph (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**
  - Jede Kante ist Bestandteile eines Pfads  $P$  von der Senke zur Kante
  - Jeder Kante wird ihre WCET  $e_i$  zugeordnet
  - ⚠ Verzweigungen benötigen **Dummy-Kanten**  $d_i$

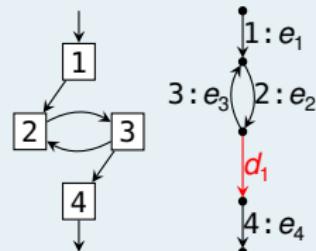
Sequenz



Verzweigung



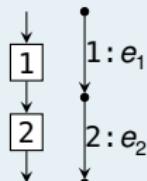
Schleife



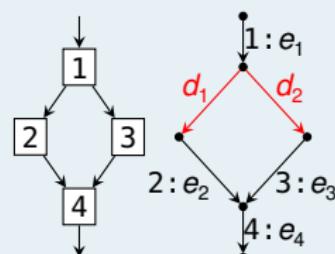
# Der Zeitanalysegraph (engl. *timing analysis graph*)

- Ein **Zeitanalysegraph (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**
  - Jede Kante ist Bestandteile eines Pfads  $P$  von der Senke zur Kante
  - Jeder Kante wird ihre WCET  $e_i$  zugeordnet
  - ⚠ Verzweigungen benötigen **Dummy-Kanten**  $d_i$

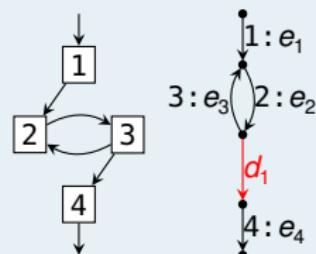
## Sequenz



## Verzweigung



## Schleife



Graphentheorie annotiert Kosten klassischerweise **an Kanten**



# Ganzzahliges Lineares Optimierungsproblem



Zielfunktion: Maximierung des gewichteten Flusses

$$\text{WCETe} = \max_{(f_1, \dots, f_e)} \sum_{E_i \in \mathcal{E}} f_i e_i$$

→ der Vektor  $(f_1, \dots, f_e)$  maximiert die Ausführungszeit



# Ganzzahliges Lineares Optimierungsproblem

- ☞ Zielfunktion: Maximierung des gewichteten Flusses

$$\text{WCETe} = \max_{(f_1, \dots, f_e)} \sum_{E_i \in \mathcal{E}} f_i e_i$$

→ der Vektor  $(f_1, \dots, f_e)$  maximiert die Ausführungszeit

- ☞ Nebenbedingungen: Garantieren tatsächlich mögliche Ausführungen
  - Flusserhaltung für jeden Knoten des T-Graphen

$$\sum_{E_j^+ = V_i} f_j = \sum_{E_k^- = V_i} f_k$$

- Flussrestriktionen für alle Schleifen des T-Graphen, z.B.

$$f_2 \leq (\text{size} - 1) f_1$$

- Rückkehrkante kann nur einmal durchlaufen werden:  $f_{E_e} = 1$



- Betrachtet implizit alle Pfade des Kontrollflussgraphen
  - Erzeugung des Zeitanalysegraphen
  - Überführung in ganzzahliges lineares Optimierungsproblem



- Betrachtet implizit alle Pfade des Kontrollflussgraphen
  - Erzeugung des Zeitanalysegraphen
  - Überführung in ganzzahliges lineares Optimierungsproblem
- **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

- Betrachtet implizit alle Pfade des Kontrollflussgraphen
  - Erzeugung des Zeitanalysegraphen
  - Überführung in ganzzahliges lineares Optimierungsproblem
- **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**
  - Lösen eines ILP ist im Allgemeinen **NP-hart**
  - Flussrestriktionen sind kein Allheilmittel
    - Beschreibung der Ausführungsreihenfolge ist problematisch



- ☞ Ausführungszeit von Elementaroperationen ist **essentiell**
- Die Berechnung ist alles andere als einfach, ein Beispiel:

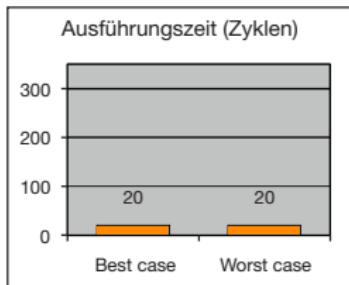
```
1 /* x = a + b */  
2 LOAD r2, _a  
3 LOAD r1, _b  
4 ADD r3,r2,r1
```



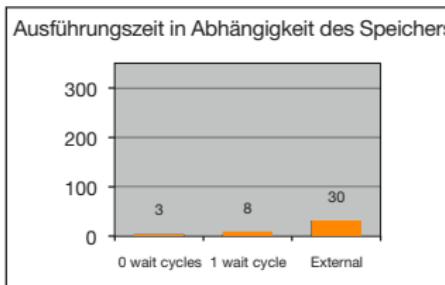
- ☞ Ausführungszeit von Elementaroperationen ist **essentiell**
- Die Berechnung ist alles andere als einfach, ein Beispiel:

```
1 /* x = a + b */  
2 LOAD r2, _a  
3 LOAD r1, _b  
4 ADD r3,r2,r1
```

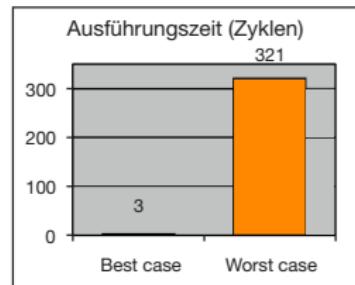
68K (1990)



MPC 5xx (2000)



PPC 755 (2001)



Quelle: Christian Ferdinand [2]



Laufzeitbedarf ist hochgradig **Hardware- und kontextspezifisch**



# WCET eines Code-Schnipsels?

Werte der Grundblöcke sind Eingabe für die Flussanalyse



## Grundproblem: Ausführungszyklen von Instruktionen zählen

```
_getop:  
    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 [3]

- Ergebnis:  $e_{\_getop} = 80$  Zyklen
- Annahmen:
  - Obere Schranke für jede Instruktion
  - Obere Schranke der Sequenz durch Summation



# WCET eines Code-Schnipsels?

Werte der Grundblöcke sind Eingabe für die Flussanalyse

## ☞ Grundproblem: Ausführungszyklen von Instruktionen zählen

```
_getop:  
    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 [3]

- Ergebnis:  $e_{\text{getop}} = 80$  Zyklen
- Annahmen:
  - Obere Schranke für jede Instruktion
  - Obere Schranke der Sequenz durch Summation



## 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 teilt sich in verschiedene Phasen
  - Aufteilung ist nicht dogmenhaft festgeschrieben



- ☞ 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

- ☞ 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 anschließend berücksichtigt
  - 3 Pfad-Analyse und WCET-Berechnung



## 1 Speicherverbrauch

- Überblick
- Messbasierte Bestimmung
- Analytische Bestimmung

## 2 Ausführungszeit

- Überblick
- Messbasierte WCET-Analyse
- Statische WCET-Analyse

## 3 Zusammenfassung



- **Dynamische Messung**  $\leadsto$  Beobachtung
  - Speicherverbrauch
    - Water-Marking  $\leadsto$  Füllstand des statischen Stapels zur Laufzeit
    - Überwachung durch Betriebssystem (Wächter)
  - Ausführungszeit
    - Durch (strukturiertes) Testen der Echtzeitanwendung
    - Betrachtung des Gesamtsystems (Software und Hardware)

- **Dynamische Messung**  $\leadsto$  Beobachtung
  - Speicherverbrauch
    - Water-Marking  $\leadsto$  Füllstand des statischen Stapels zur Laufzeit
    - Überwachung durch Betriebssystem (Wächter)
  - Ausführungszeit
    - Durch (strukturiertes) Testen der Echtzeitanwendung
    - Betrachtung des Gesamtsystems (Software und Hardware)
- **Statische Analyse**  $\leadsto$  Bestimmung einer oberen Schranke
  - Speicherverbrauch
    - Analyse des Kontroll- und Aufrufgraphen
    - Beachtung der Ausführungsbedingungen (ABI)
  - Ausführungszeit
    - **Makroskopisch**: *Was macht das Programm?*
    - **Mikroskopisch**: *Was passiert in der Hardware?*



- [1] Ferdinand, C. ; Heckmann, R. ; Franzen, B. :  
Static memory and timing analysis of embedded systems code.  
In: *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems*, 2007, S. 07–04
- [2] Ferdinand, C. ; Heckmann, R. ; Wolff, H.-J. ; Renz, C. ; Parshin, O. ; Wilhelm, R. :  
Towards model-driven development of hard real-time systems.  
In: *Model-Driven Development of Reliable Automotive Services*.  
Springer, 2008, S. 145–160
- [3] Puschner, P. :  
*Zeitanalyse von Echtzeitprogrammen*.  
Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Technische Universität Wien, Institut für Technische Informatik, Diss., 1993
- [4] Puschner, P. ; Huber, B. :  
*Zeitanalyse von sicherheitskritischen Echtzeitsystemen*.  
[http://ti.tuwien.ac.at/rts/teaching/courses/wcet, 2012. –](http://ti.tuwien.ac.at/rts/teaching/courses/wcet, 2012.)  
Lecture Notes

- [5] Regehr, J. ; Reid, A. ; Webb, K. :  
    *Eliminating Stack Overflow by Abstract Interpretation.*  
    In: *ACM Trans. Embed. Comput. Syst.* 4 (2005), Nov., Nr. 4, 751–778.  
    <http://dx.doi.org/10.1145/1113830.1113833>. –  
    DOI 10.1145/1113830.1113833. –  
    ISSN 1539–9087
- [6] Ulbrich, P. :  
    *Echtzeitsysteme.*  
    [http://www4.cs.fau.de/Lehre/WS16/V\\_EZS/](http://www4.cs.fau.de/Lehre/WS16/V_EZS/), 2016
- [7] Weber-Wulff, D. :  
    *More on German Train Problems.*  
    <http://catless.ncl.ac.uk/Risks/17.02.html>.  
    Version: 04 1995