Verlässliche Echtzeitsysteme

Verifikation nicht-funktionaler Eigenschaften

Peter Ulbrich

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)

`www4.informatik.uni-erlangen.de`

02. Juni 2015

Übersicht und Problemstellung

Abstrakte Interpretation und Design-by-Contract sind nicht genug!

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
  - Abstrakte Interpretation:
    Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)
  - Design-by-Contract:
    Korrektheitsaussagen über Vor- und Nachbedingungen (funktional)

Übersicht und Problemstellung

Abstrakte Interpretation und Design-by-Contract sind nicht genug!

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
  - Abstrakte Interpretation:
    Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)
  - Design-by-Contract:
    Korrektheitsaussagen über Vor- und Nachbedingungen (funktional)

⚠ Dies ist notwendig jedoch nicht hinreichend

→ Einfluss nicht-funktionaler Eigenschaften der Ausführungsumgebung

- Anwendung ist in die Umwelt eingebettet!
- Exemplarisch: Speicherverbrauch und Laufzeit

Übersicht und Problemstellung

Abstrakte Interpretation und Design-by-Contract sind nicht genug!

- Bislang stand Verifikation des korrekten Verhaltens im Vordergrund
  - Abstrakte Interpretation:
    Abwesenheit von Laufzeitfehlern (Sprachstandard, nicht-funktional)
  - Design-by-Contract:
    Korrektheitsaussagen über Vor- und Nachbedingungen (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)
- Laufzeit: Worst-Case Execution Time (WCET)

→ Messung versus statische Analyse

Gliederung

Speicherverbrauch im Überblick

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
```

Speicherverbrauch im Überblick

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 & Stapelspeicher-Konfiguration)
  - Permanenter Verbrauch (architekturabhängig) ebenso statisch bekannt

Speicherverbrauch im Überblick

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 & Stapelspeicher-Konfiguration)
  - Permanenter Verbrauch (architekturabhängig) ebenso statisch bekannt

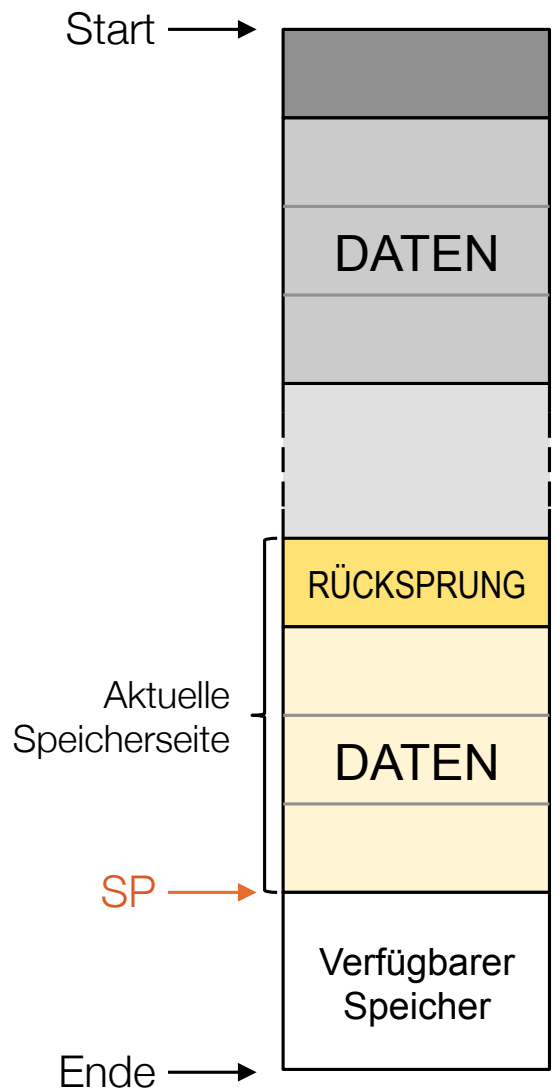
Dynamischer Speicher in eingebetteten Systemen

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

Der Stapelspeicher (Stack)

Dynamische Nutzung von Speicher ist eingebetteten Systemen

- Stapelspeicher wird verwendet für:
  - Lokale Variablen und Zwischenwerte
  - Funktionsparameter
  - Rücksprungadressen

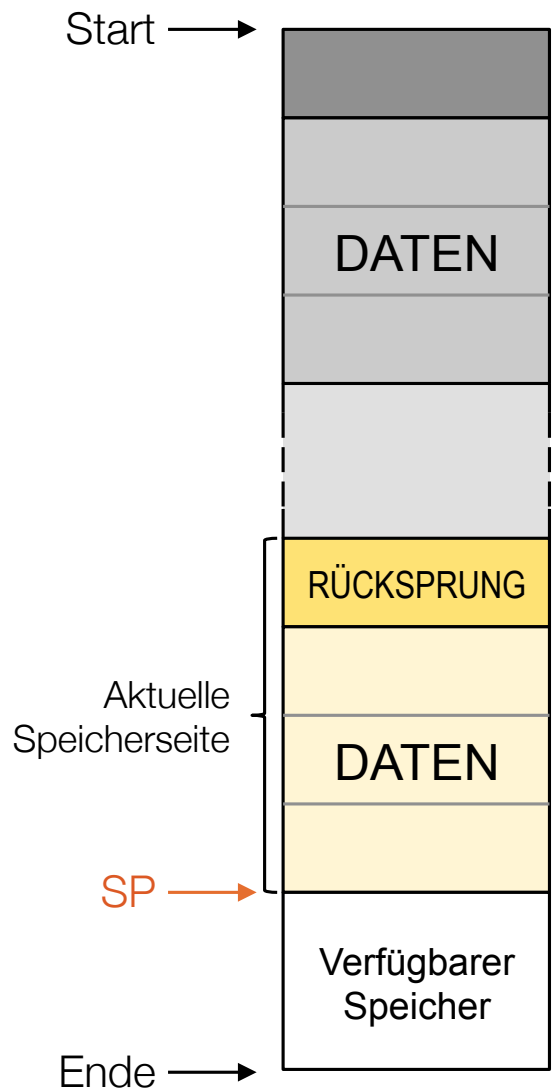


Der Stapelspeicher (Stack)

Dynamische Nutzung von Speicher ist eingebetteten Systemen

- Stapelspeicher wird verwendet für:
  - Lokale Variablen und Zwischenwerte
  - Funktionsparameter
  - Rücksprungadressen

⚠ Größe wird zur Übersetzungszeit festgelegt



Der Stapelspeicher (Stack)

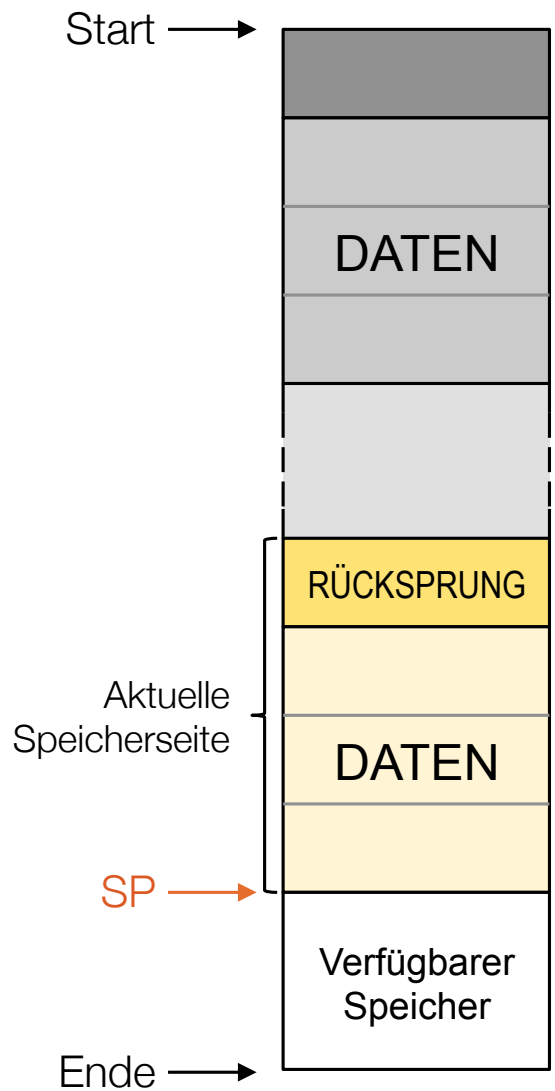
Dynamische Nutzung von Speicher ist eingebetteten Systemen

- Stapelspeicher wird verwendet für:
  - Lokale Variablen und Zwischenwerte
  - Funktionsparameter
  - Rücksprungadressen

⚠ Größe wird zur Übersetzungszeit festgelegt

Fehlerquelle Stapelspeicher

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



Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [6]

- **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 [6]

- **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 $\rightsquigarrow$ zu klein schon für normalen Verkehr
- Fehlerbehandlungsroutine fehlerhaft $\rightsquigarrow$ Endlosschleife
- Notabschaltung durch Sicherungsmaßnahmen (fail-stop)

Problem: Maximaler Speicherverbrauch

Fallbeispiel: Stellwerk Hamburg-Altona [6]

- **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 $\rightsquigarrow$ zu klein schon für normalen Verkehr
- Fehlerbehandlungsroutine fehlerhaft $\rightsquigarrow$ Endlosschleife
- Notabschaltung durch Sicherungsmaßnahmen (fail-stop)

Ausfall am Tag der Inbetriebnahme

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

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!

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!

- Voraussetzungen für sinnvolle Analyse
  - Zyklische Ausführungspfade vermeiden
  - Keine Rekursion, Funktionszeiger, dynamischer Speicher

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!

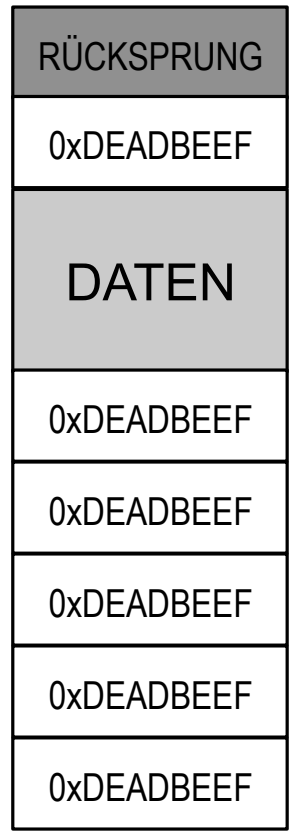
- Voraussetzungen für sinnvolle Analyse
  - Zyklische Ausführungspfade vermeiden
  - Keine Rekursion, Funktionszeiger, dynamischer Speicher

⚠ Analyse gängiger Compiler
- `gcc -fstack-usage` ist nicht genug
- Richtwert bei der Entwicklung einzelner Funktionen

Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie IV/10 ff.)

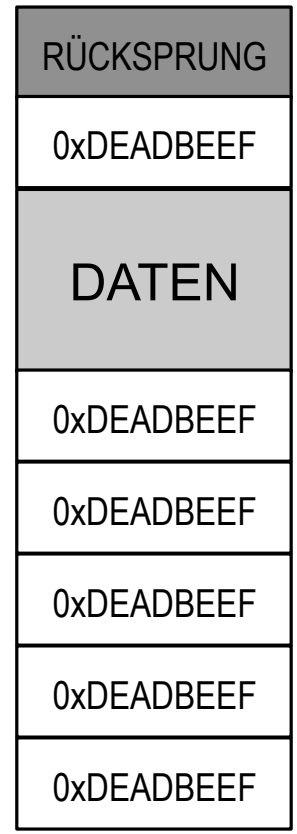
- Messung (Water-Marking, Stack Canaries)
  - Stapelspeicher wird vorinitialisiert (z.B. `0xDEADBEEF`)
  - Maximaler Verbrauch der Ausführung
    $\rightsquigarrow$ höchste Speicherstelle ohne Wasserzeichen
  - Auf Rücksprungadressen anwendbar



Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie IV/10 ff.)

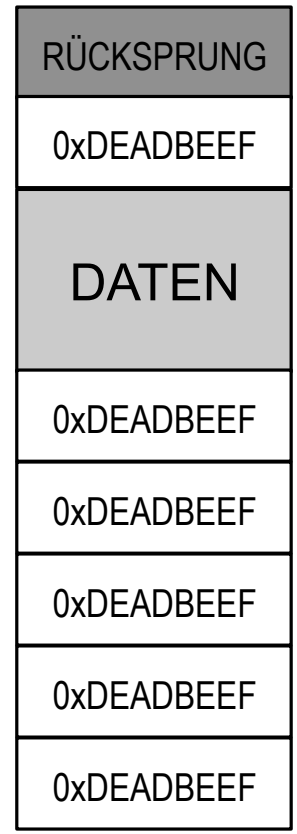
- Messung (Water-Marking, Stack Canaries)
  - Stapelspeicher wird vorinitialisiert (z.B. `0xDEADBEEF`)
  - Maximaler Verbrauch der Ausführung
    $\rightsquigarrow$ höchste Speicherstelle ohne Wasserzeichen
  - Auf Rücksprungadressen anwendbar
- Systemüberwachung zur Laufzeit
  - Verfahren gut geeignet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



Messung des Stapelspeicherverbrauchs

Analog zum dynamischen Testen (siehe Folie IV/10 ff.)

- Messung (Water-Marking, Stack Canaries)
  - Stapelspeicher wird vorinitialisiert (z.B. `0xDEADBEEF`)
  - Maximaler Verbrauch der Ausführung
    $\rightsquigarrow$ 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!

Herausforderung Analyse

Wenn Zählen so einfach wäre ...

```
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

Herausforderung Analyse

Wenn Zählen so einfach wäre ...

```
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)

Herausforderung Analyse

Wenn Zählen so einfach wäre ...

```
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

Herausforderung Analyse

Wenn Zählen so einfach wäre ...

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

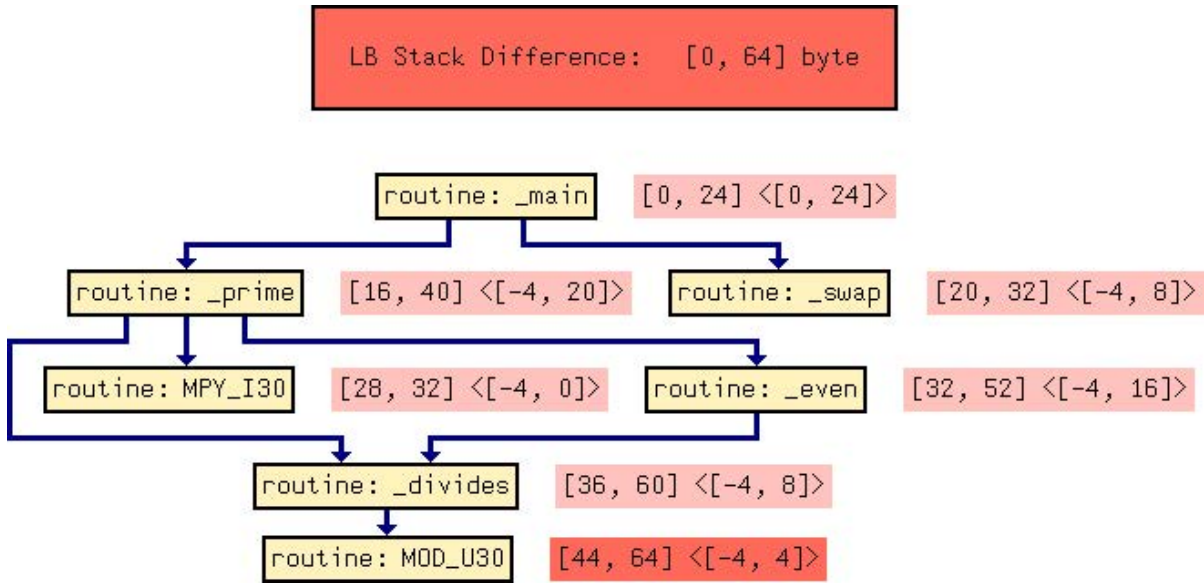
⚠ Ausführungsbedingungen bestimmen tatsächlichen Speicherbedarf

- 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

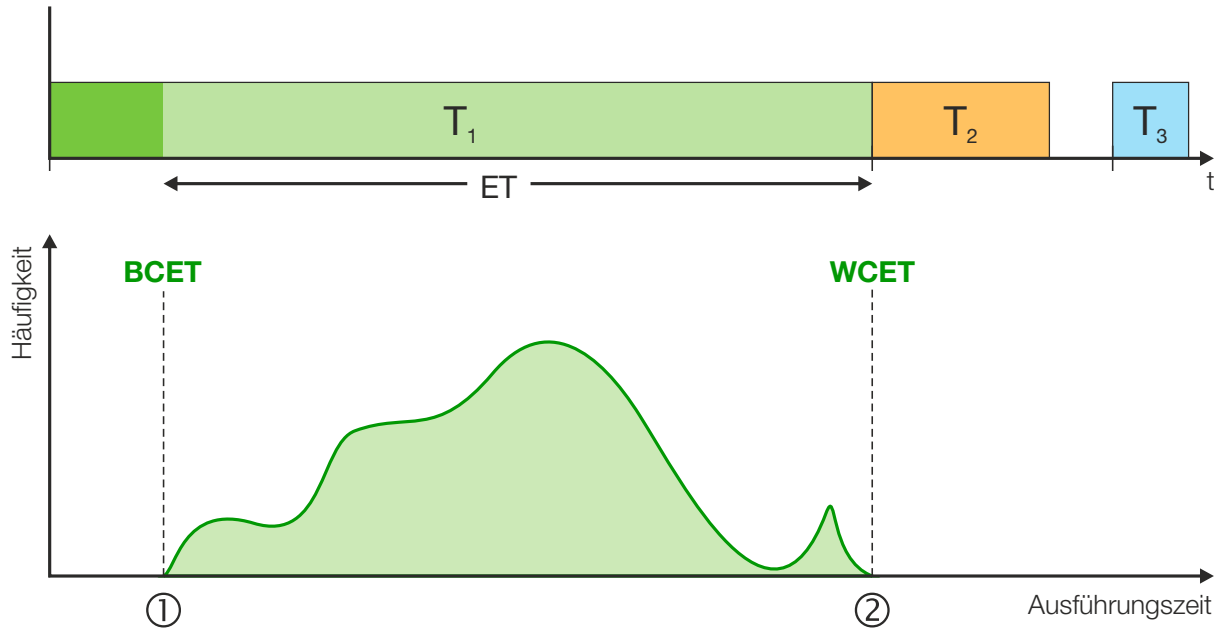
Durch abstrakte Interpretation des Programmcodes [1, 4]



- Statische Analyse des Kontrollfluss- und Aufrufgraphen
  - Pufferüberlauf als weitere Form von Laufzeitfehler
  - Vorgehen analog zum Korrektheitsnachweis
- Weist Abwesenheit von Pufferüberläufen nach
  - Pfadanalyse $\rightsquigarrow$ Maximaler Speicherverbrauch
  - Ausrollen von Schleifen (siehe Folie VI/5)
  - Partitionierung und Werteanalyse (siehe Folie VI/6)

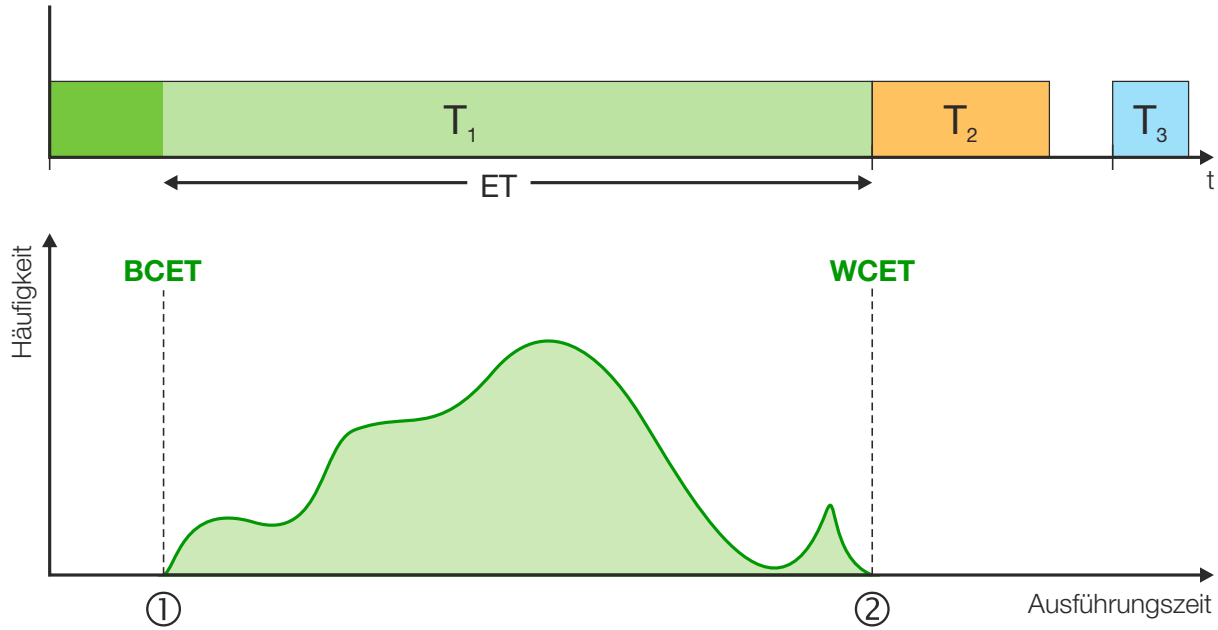
Gliederung

Die **maximalen** Ausführungszeit



- Alle sprechen von der maximalen Ausführungszeit (e)
  - **W***orst* **C***ase* **E***xecution* Time (WCET) e_i (vgl. [5] Folie III-2/28)
  - Unabdingbares Maß für zulässigen Ablaufplan (vgl. [5] Folie III-2/33)

Die **maximalen** Ausführungszeit



- Alle sprechen von der maximalen Ausführungszeit (e)
  - **W***orst* **C***ase* **E***xecution* Time (WCET) e_i (vgl. [5] Folie III-2/28)
  - Unabdingbares Maß für zulässigen Ablaufplan (vgl. [5] Folie III-2/33)
- Tatsächliche Ausführungszeit bewegt sich zwischen:
  1. bestmöglicher Ausführungszeit (**B***est* **C***ase* **E***xecution* **T***ime*, BCET)
  2. schlechtest möglicher Ausführungszeit (besagter WCET)

Warum ist es so schwierig, die WCET 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;
}
```

Warum ist es so schwierig, die WCET 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

Warum ist es so schwierig, die WCET 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

Maschinenprogrammebene liefert Dauer der Elementaroperationen:

- die Ausführungsdauer von `ADD`, `LOAD`, ...
- ist prozessorabhängig und für moderne Prozessoren sehr schwierig
  - Cache $\rightsquigarrow$ Liegt die Instruktion/das Datum im schnellen Cache?
  - Pipeline $\rightsquigarrow$ Wie ist der Zustand der Pipeline an einer Instruktion?
  - Out-of-Order-Execution, Branch-Prediction, Hyper-Threading, ...

Messbasierte WCET-Analyse [3]

Idee: der Prozessor selbst ist das präziseste Hardware-Modell

- Führe das Programm aus und beobachte die Ausführungszeit!

Messbasierte WCET-Analyse [3]

- 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 Zeitschranken
  - 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

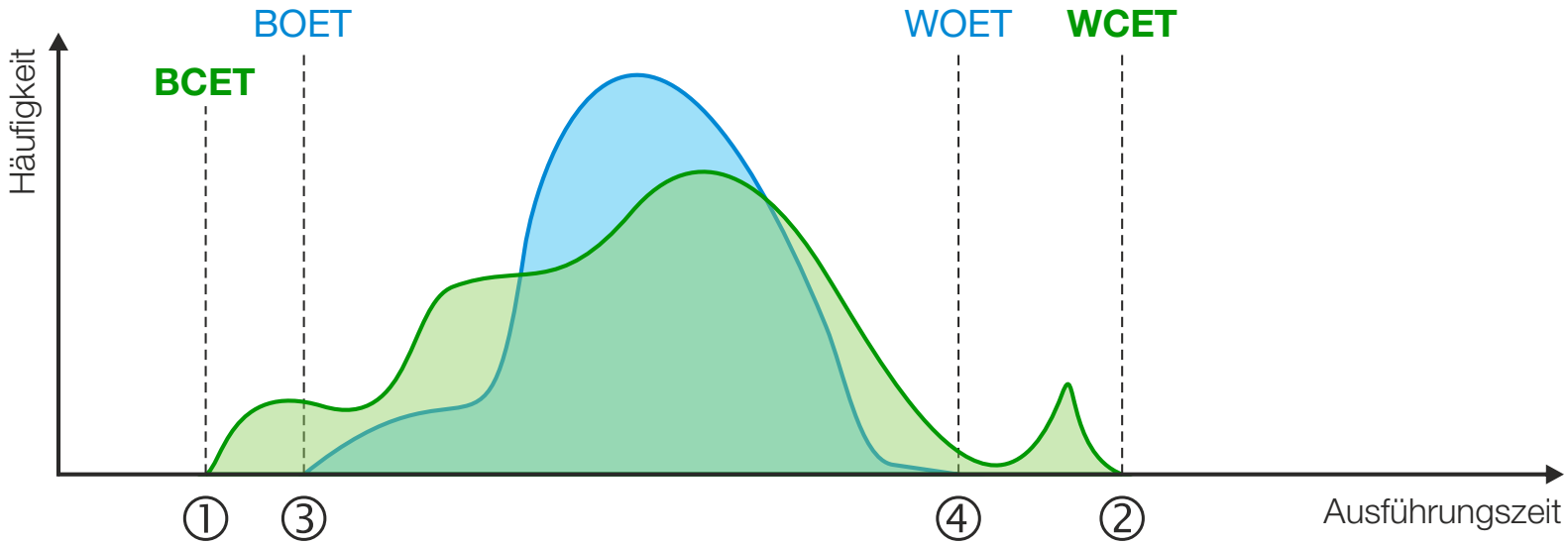
Messbasierte WCET-Analyse [3]

- 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 Zeitschranken
  - 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“

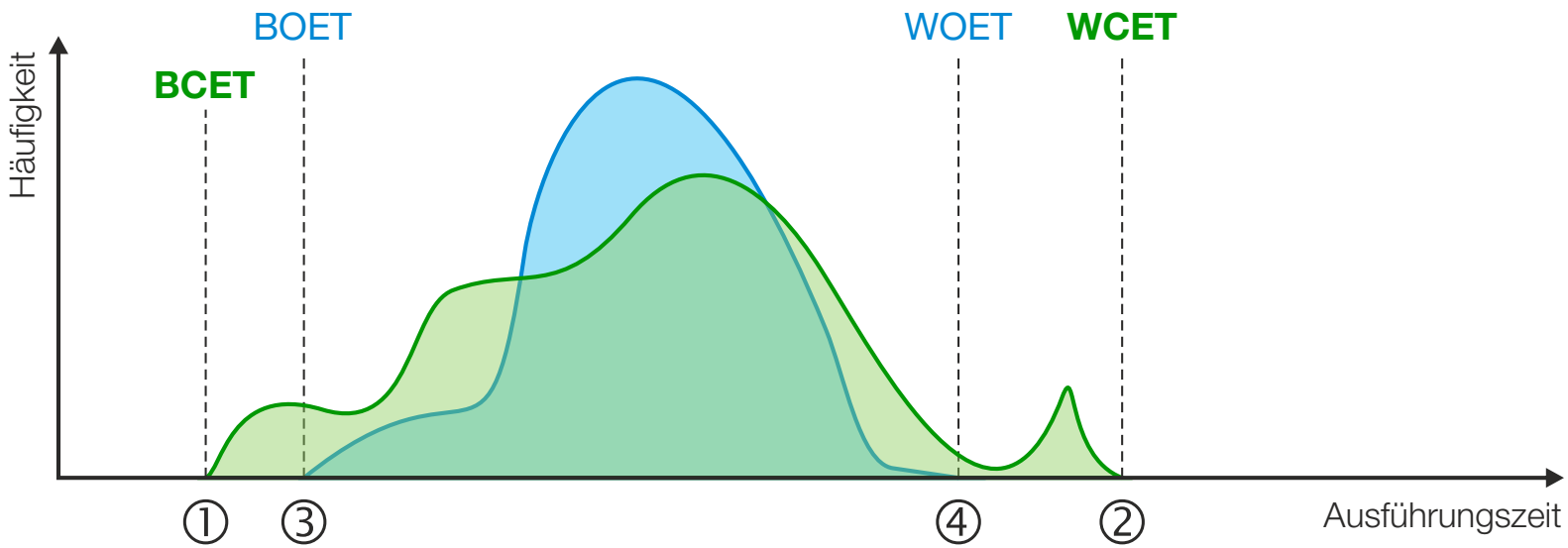
- ⇝ z. B. immer Pfade vermessen (d. h. Ablauf und Zeit)
- ⇝ auf einen definierten Startzustand achten

Messbasierte WCET-Analyse [3]



- 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 WCET-Analyse [3]

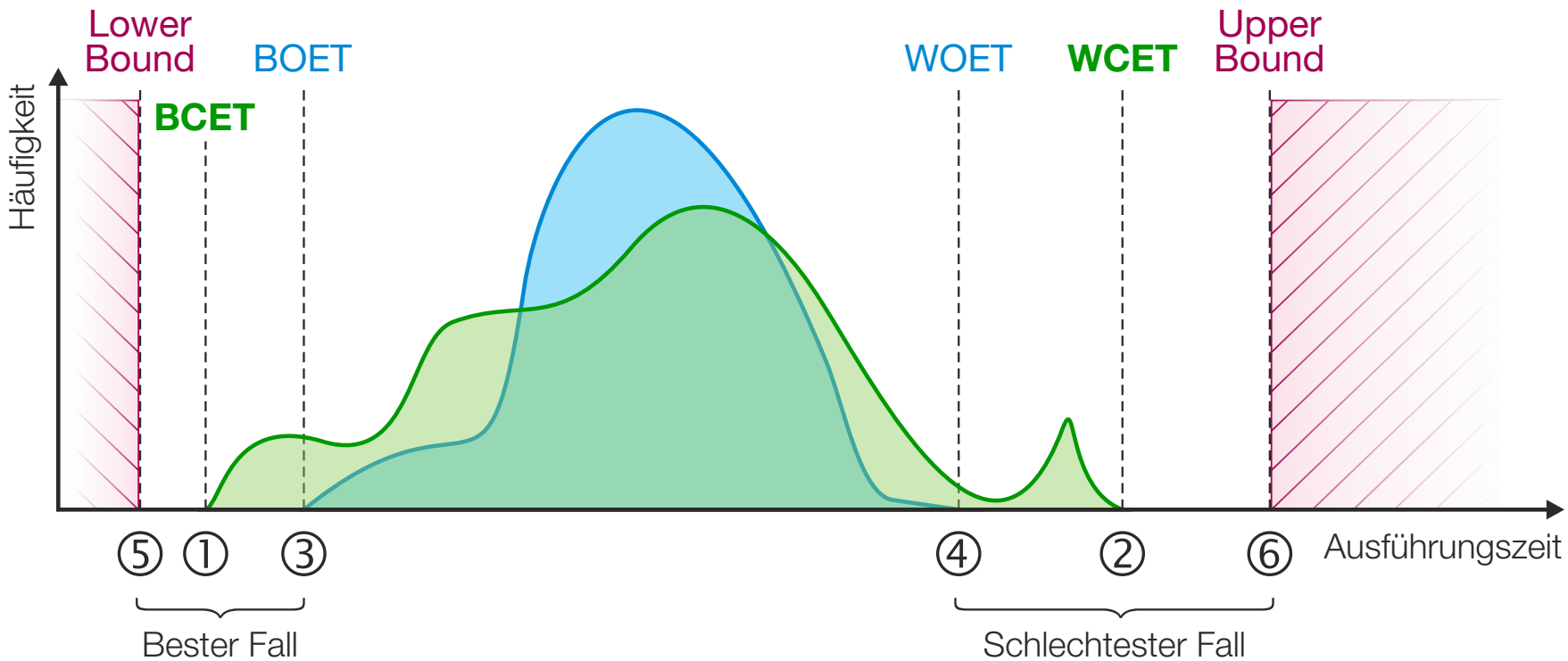


- Probleme messbasierter Ansätze
  - in der Praxis ist es unmöglich alle relevanten Pfade zu betrachten
  - gewählte Testdaten führen nicht unbedingt zum längsten Pfad
  - seltene Ausführungsszenarien werden nicht abgedeckt
  - abschnittsweise WCET-Messung $\not\rightsquigarrow$ globalen WCET
  - Wiederherstellung des Hardwarezustandes schwierig/unmöglich

⚠ Messbasierte Ansätze unterschätzen die WCET meistens

☞ Systematischere Analysetechniken sind vonnöten

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 Upper Bound $\geq$ 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)`

- Anzahl von **D**urchläufen, **V**ergleichen und Vertauschungen (engl. **S***wap*)
- `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 $= 3$;

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)`

- Anzahl von **D**urchläufen, **V**ergleichen und Vertauschungen (engl. **S***wap*)
- `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
  - Wie viele Schleifendurchläufe werden benötigt?

☞ Wiederrum statische Analyse des Kontrollfluss- und Aufrufgraphen

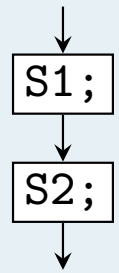
- Pfadanalyse $\rightsquigarrow$ Nur maximale Pfadlänge von belang
- Ausrollen von Schleifen (siehe Folie VI/5)

Lösungsweg$_1$: Timing Schema

Eine einfache Form der Sammelsemantik (siehe Folie V/15)

Sequenzen $\rightsquigarrow$ Hintereinanderausführung

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



Lösungsweg$_1$: Timing Schema

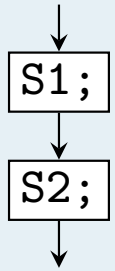
Eine einfache Form der Sammelsemantik (siehe Folie V/15)

Sequenzen $\rightsquigarrow$ Hintereinanderausführung

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

Summation der WCETs:

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



Lösungsweg$_1$: Timing Schema

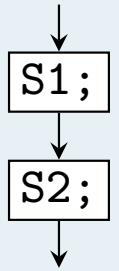
Eine einfache Form der Sammelsemantik (siehe Folie V/15)

Sequenzen $\rightsquigarrow$ Hintereinanderausführung

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

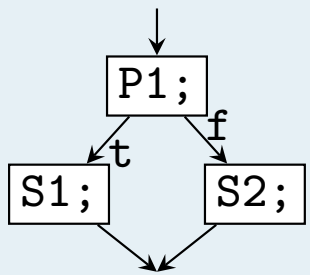
Summation der WCETs:

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



Verzweigung $\rightsquigarrow$ bedingte Ausführung

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



Lösungsweg$_1$: Timing Schema

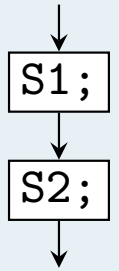
Eine einfache Form der Sammelsemantik (siehe Folie V/15)

Sequenzen $\rightsquigarrow$ Hintereinanderausführung

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

Summation der WCETs:

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

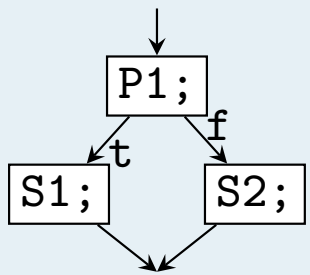


Verzweigung $\rightsquigarrow$ bedingte Ausführung

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

Abschätzung der Gesamtausführungszeit:

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



Lösungsweg$_1$: Timing Schema

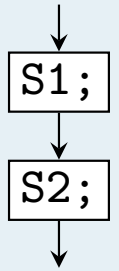
Eine einfache Form der Sammelsemantik (siehe Folie V/15)

Sequenzen $\rightsquigarrow$ Hintereinanderausführung

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

Summation der WCETs:

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

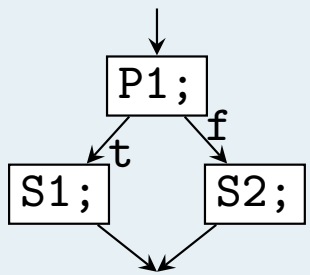


Verzweigung $\rightsquigarrow$ bedingte Ausführung

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

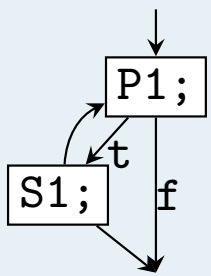
Abschätzung der Gesamtausführungszeit:

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



Schleifen $\rightsquigarrow$ wiederholte Ausführung

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



Lösungsweg$_1$: Timing Schema

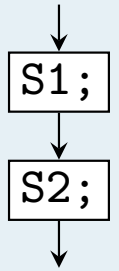
Eine einfache Form der Sammelsemantik (siehe Folie V/15)

Sequenzen $\rightsquigarrow$ Hintereinanderausführung

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

Summation der WCETs:

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

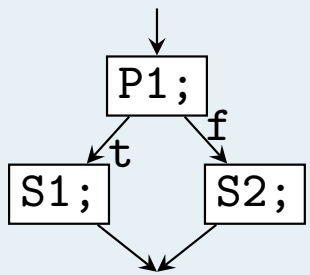


Verzweigung $\rightsquigarrow$ bedingte Ausführung

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

Abschätzung der Gesamtausführungszeit:

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

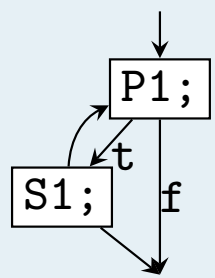


Schleifen $\rightsquigarrow$ wiederholte Ausführung

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

Schleifendurchläufe berücksichtigen:

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



Timing Schema: Eigenschaften, Vor- und Nachteile

- Eigenschaften
  - Traversierung des abstrakten Syntaxbaums (AST) 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

Timing Schema: Eigenschaften, Vor- und Nachteile

- Eigenschaften
  - Traversierung des abstrakten Syntaxbaums (AST) 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

Timing Schema: Eigenschaften, Vor- und Nachteile

- Eigenschaften
  - Traversierung des abstrakten Syntaxbaums (AST) 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-lokaler Codeteile lassen sich nicht berücksichtigen
    - Schwierige Integration mit einer separaten Hardware-Analyse
  - Nichtrealisierbare Pfade (infeasible paths) nicht ausschließbar
    $\rightsquigarrow$ 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:

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

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

$\rightsquigarrow$ Das ist algorithmisch nicht handhabbar

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

- $\rightsquigarrow$ Das ist algorithmisch nicht handhabbar

Lösung: Fasse die Bestimmung der WCET als Flussproblem auf

- $\rightsquigarrow$ Der maximale Fluss durch das durch den Graphen gegebene Netzwerk führt zur gesuchten WCET
- $\rightsquigarrow$ Flussprobleme sind mathematisch gut untersucht und lassen sich durch lineare Ganzzahlprogrammierung lösen

Lösungsansatz$_2$: Implicit Path Enumeration Technique

☞ Lösungsansatz: Bestimmung der WCET als Flussproblem auffassen

$\rightsquigarrow$ Implicit Path Enumeration Technique (IPET) [2]

[1] http://lpsolve.sourceforge.net/

Lösungsansatz$_2$: Implicit Path Enumeration Technique

☞ Lösungsansatz: Bestimmung der WCET als Flussproblem auffassen
$\rightsquigarrow$ Implicit Path Enumeration Technique (IPET) [2]

- 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 lpsolve[1])

[1] `http://lpsolve.sourceforge.net/`

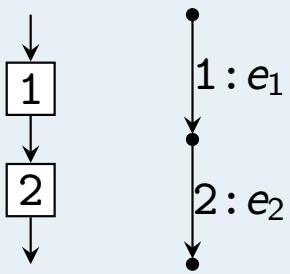
Der Zeitanalysegraph (engl. *timing analysis graph*)

- 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
    - 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

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
    - 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

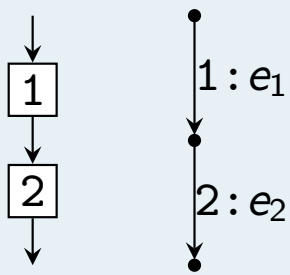
Sequenz



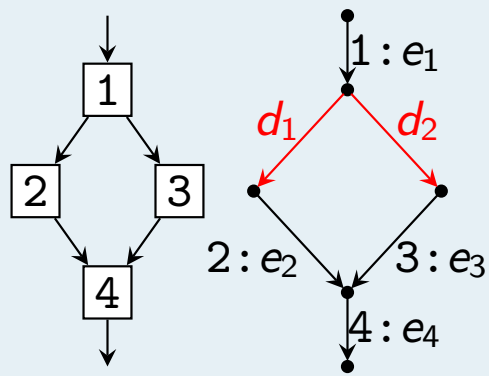
Der Zeitanalysegraph (engl. *timing analysis graph*)

- 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
    - 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

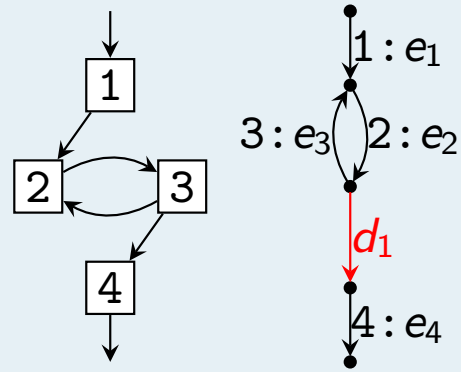
Sequenz



Verzweigung



Schleife



Ganzzahliges Lineares Optimierungsproblem

Zielfunktion: Maximierung des gewichteten Flusses

$$\mathrm{WCET}\, e = \max_{(f_1,\ldots,f_e)} \sum_{E_i \in \mathcal{E}} f_i e_i$$

$\rightsquigarrow$ der Vektor $(f_1, \ldots, f_e)$ maximiert die Ausführungszeit

Ganzzahliges Lineares Optimierungsproblem

Zielfunktion: Maximierung des gewichteten Flusses

$$\text{WCET}\, e = \max_{(f_1,\ldots,f_e)} \sum_{E_i \in \mathcal{E}} f_i e_i$$

$\rightsquigarrow$ der Vektor $(f_1, \ldots, f_e)$ 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.

$$f_2 \leq (size - 1) f_1$$

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

IPET: Eigenschaften, Vor- und Nachteile

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

IPET: Eigenschaften, Vor- und Nachteile

- 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

IPET: Eigenschaften, Vor- und Nachteile

- 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

Problem Ausführungszeit von Elementaroperationen

- Kenntnis der Ausführungszeit von Elementaroperationen ist essentiell

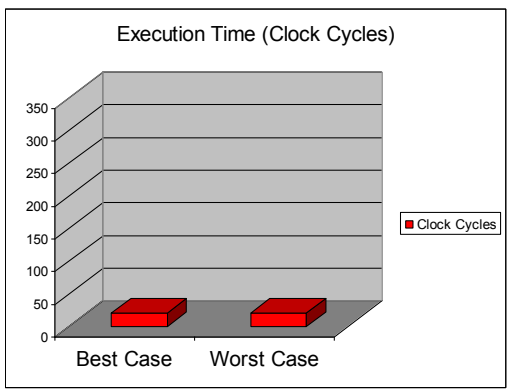
Problem Ausführungszeit von Elementaroperationen

Die Crux mit der Hardware

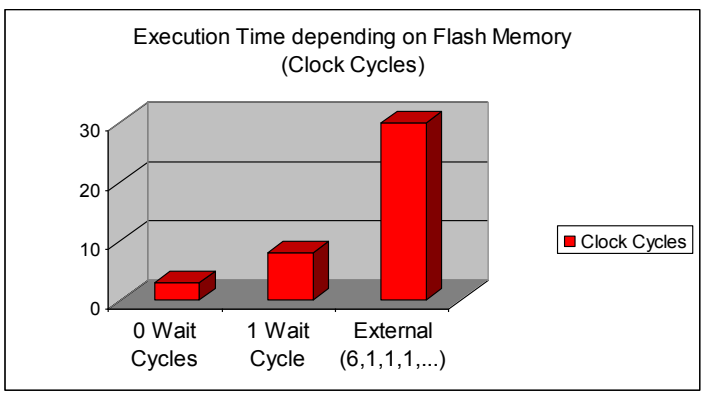
- Kenntnis der Ausführungszeit von Elementaroperationen ist essentiell
- Die Berechnung ist alles andere als einfach, ein Beispiel:

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

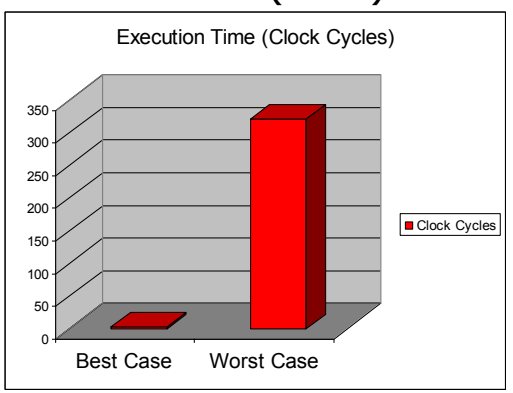
68K (1990)



MPC 5xx (2000)



PPC 755 (2001)



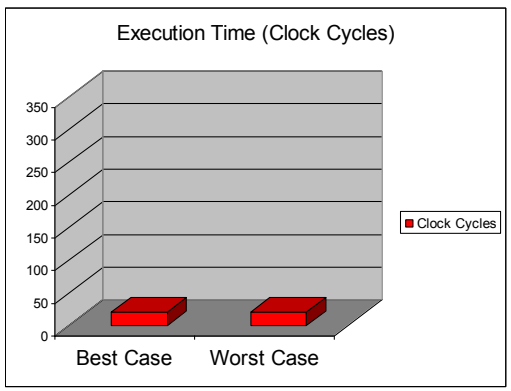
Problem Ausführungszeit von Elementaroperationen

Die Crux mit der Hardware

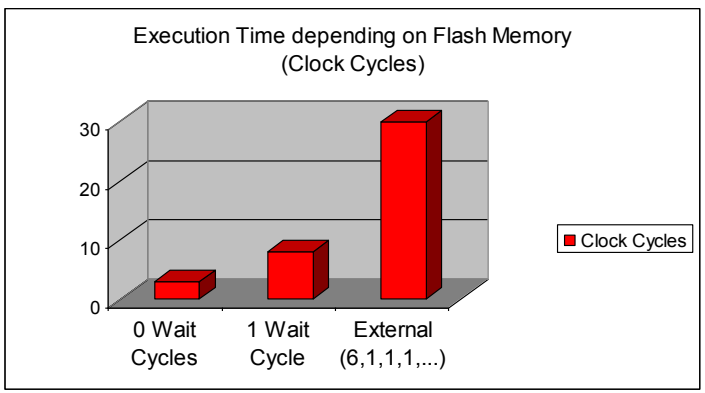
- Kenntnis der Ausführungszeit von Elementaroperationen ist essentiell
- Die Berechnung ist alles andere als einfach, ein Beispiel:

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

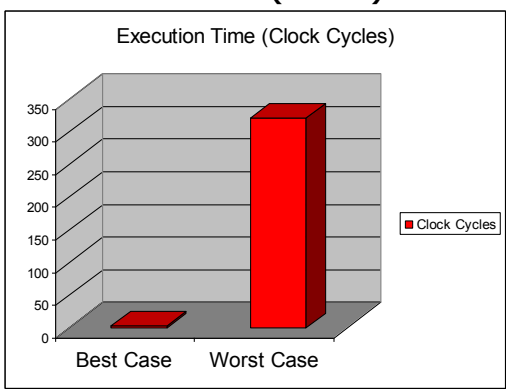
68K (1990)



MPC 5xx (2000)



PPC 755 (2001)



Laufzeitbedarf ist hochgradig Hardware- und kontextspezifisch

Hardware-Analyse

Hardware-Analyse teilt sich in verschiedene Phasen

- Aufteilung ist nicht dogmenhaft festgeschrieben

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

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 anschließend berücksichtigt
  3. Pfad-Analyse und WCET-Berechnung

Gliederung

Zusammenfassung

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

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)
- 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?*

Gliederung

Zusammenfassung der letzten Vorlesung (Redundanz)

Fehlertypen $\mapsto$ Toleranz von SDCs und DUEs

Redundanz $\mapsto$ hat mehrere Dimensionen

- Grundvoraussetzung für Fehlertoleranz
- Redundanz in Struktur, Funktion, Information, oder Zeit
- Fehlererkennung, -diagnose, -eindämmung, -maskierung

Replikation $\mapsto$ koordinierter Einsatz struktureller Redundanz

- Replikation der Eingaben, Abstimmung der Ausgaben
- Replikate für fail-silent, fail-consistent, malicious
- Zeitliche und räumliche Isolation einzelner Replikate

Hardwarebasierte Replikation $\mapsto$ Umfassend und teuer

- Dreifache Auslegung, toleriert Fehler im Wertbereich
- Zuverlässigkeit von Replikat und Gesamtsystem

Softwarebasierte Replikation $\mapsto$ Flexibel aber eingeschränkt

- Process Level Redundancy reduziert Kosten von TMR, zulasten eines geringeren Schutzes

Diversität $\mapsto$ versucht Gleichtaktfehler auszuschließen

Literaturverzeichnis

[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] Puschner, P. :
Zeitanalyse von Echtzeitprogrammen.
Treitlstr. 1-3/182-1, 1040 Vienna, Austria, Technische Universität Wien, Institut für Technische Informatik, Diss., 1993

[3] Puschner, P. ; Huber, B. :
Zeitanalyse von sicherheitskritischen Echtzeitsystemen.
http://ti.tuwien.ac.at/rts/teaching/courses/wcet, 2012. –
Lecture Notes

[4] 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

Literaturverzeichnis (Forts.)

[5] Ulbrich, P. :
Echtzeitsysteme.
http://www4.cs.fau.de/Lehre/WS14/V_EZS/, 2014

[6] Weber-Wulff, D. :
More on German Train Problems.
http://catless.ncl.ac.uk/Risks/17.02.html.
Version: 04 1995