

Echtzeitsysteme

Übungen zur Vorlesung

Analyse von Ausführungszeiten

Florian Schmaus **Peter Wägemann**

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
<https://www4.cs.fau.de>

16. November 2017



Übersicht

1 Rekapitulation: Worst-Case Execution Time

2 Ausflug: Cache-Analyse

- Grundlagen
- Beispiel: LRU-Cache

3 WCET-Analyse auf dem EZS-Board

- GPIOs
- aiT



Übersicht

1 Rekapitulation: Worst-Case Execution Time

2 Ausflug: Cache-Analyse

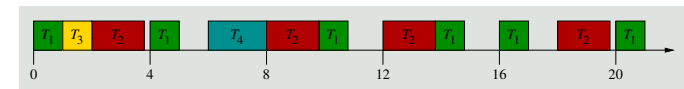
- Grundlagen
- Beispiel: LRU-Cache

3 WCET-Analyse auf dem EZS-Board

- GPIOs
- aiT



Worst-Case Execution Time



■ Eine entscheidende Größe für:

- Statische **Ablaufplanung**
- **Planbarkeitsanalyse**
- **Übernahmeprüfung**
- ...

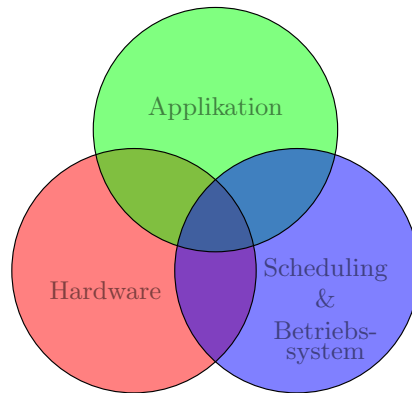


Es geht um den **schlimmsten Fall** (engl. *worst case*)

→ Obere Schranke für **alle** Fälle



Wiederholung: Einflüsse auf die Ausführungszeit



- 1 **Applikation:** Eingabedaten, ...
- 2 **Hardware:** Caches, Pipelining, ...
- 3 **Scheduling:** Höherpriorie Aufgaben, Interrupts, Overheads, ...



WCET-Analyse – Flusssensitive Informationen

Beispiel

```

1 void func(int a) { // entry
2   if (a % 2) {
3     f(); // if.then0
4   }
5   ++a; // if.end0

6   if(a % 2) {
7     g(); // if.then1
8   }
9   ... // if.end1
10 }
    
```

- T-Graph aus Kontrollflussgraph abgeleitet
- Worst Case == maximaler Fluss durch T-Graph
- Nebenbedingungen des Flussproblems:
 - $freq(entry) = freq(if.then0) + freq(if.false0)$
 - $freq(if.then0) + freq(if.false0) = freq(if.end0)$
 - ...
- Nebenbedingungen werden für Integer Linear Program (ILP) verwendet:

Zielfunktion: $max : cost(entry) \cdot freq(entry) + cost(if.then0) \cdot freq(if.then0) + \dots$



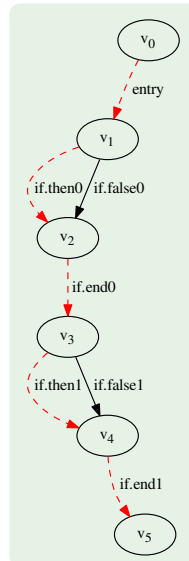
WCET-Analyse – Flusssensitive Informationen

Pessimistische Annahmen der IPET

```

1 void func(int a) { // entry
2   if (a % 2) {
3     f(); // if.then0
4   }
5   ++a; // if.end0

6   if(a % 2) {
7     g(); // if.then1
8   }
9   ... // if.end1
10 }
    
```



- Für jeden Basis Block: WCET notwendig
- Schleifengrenzen notwendig
- Struktureller Ansatz: *nicht kontextsensitiv*
- Im Beispi: *beide Pfade* aufgenommen
⇒ **pessimistische Annahme**
- *Nachträgliche* Reduktion dieser Überabschätzung
- **abstrakte Interpretation** → VEZS



WCET eines Basis Blocks?

Wiederholung aus der Vorlesung



Grundproblem: Ausführungszyklen von Instruktionen zählen

```

1 _getop:
2   link    a6,#0           // 16 Zyklen
3   moveml  #0x3020,sp@-    // 32 Zyklen
4   movel   a6@8,a2         // 16 Zyklen
5   movel   a6@12,d3        // 16 Zyklen
    
```

■ Ergebnis: $e_{getop} = 80$ Zyklen

■ Annahmen:

- Obere Schranke für jede Instruktion
- Obere Schranke der Sequenz durch Summation

Quelle: Peter Puschner [2]



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
- **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 Rekapitulation: Worst-Case Execution Time
- 2 Ausflug: Cache-Analyse
 - Grundlagen
 - Beispiel: LRU-Cache
- 3 WCET-Analyse auf dem EZS-Board
 - GPIOs
 - aiT



Grundlagen der Cache-Analyse [3, Kapitel 22]

Wiederholung aus der Vorlesung

- Cache: ein kleiner, schneller Zwischenspeicher
 - Zugriffszeiten variieren je nach Zustand des Caches enorm:
 - Treffer (engl. *hit*), Daten/Instruktion sind im Cache $\leadsto e_h$
 - Fehlschlag (engl. *miss*), Daten/Instruktion sind nicht im Cache $\leadsto e_m$
- ⚠ Hits sind schneller als Misses: $e_m \gg e_h$
 - Strafe liegt schnell bei > 100 Taktzyklen
- Eigenschaften von Caches mit Einfluss auf deren 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



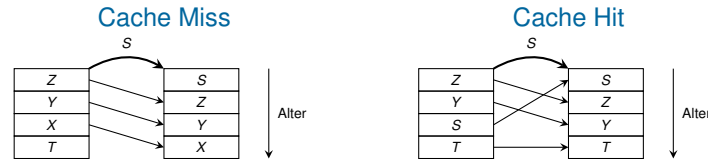
Ergebnisse der Cache-Analyse

- Wissen ob eine Instruktion / ein Datum 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“



Beispiel: LRU-Cache, 4-fach assoziativ

LRU = „least recently used“ – Das älteste Element fliegt raus!



- Caches werden häufig in **Sätze** (engl. *cache set*) unterteilt
 - Ein n -fach assoziativer Cache besitzt pro Satz n Cache-Blöcke
 - Aufnahme von n konkurrierende Speicherstellen pro Satz möglich
 - Inhalt und Verwaltungsinformation (bei LRU das Alter des Blocks) werden sowohl bei Treffern als auch bei Fehlschlägen aktualisiert

→ Konkrete Semantik des Caches

must-Analyse und **may-Analyse** approximieren diese konkrete Semantik:

must Obergrenze des Alters $\leadsto$ Unterapproximation des Inhalts

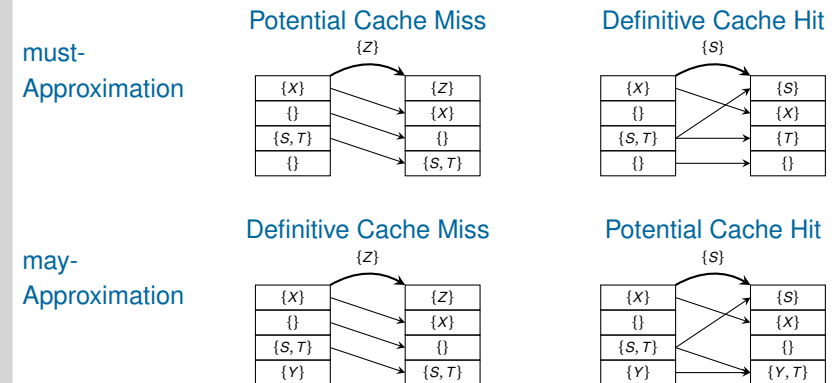
- Obergrenze $\leq$ Assoziativität $\leadsto$ garantiert im Cache

may Untergrenze des Alters $\leadsto$ Überapproximation des Inhalts

- Untergrenze $>$ Assoziativität $\leadsto$ garantiert nicht im Cache

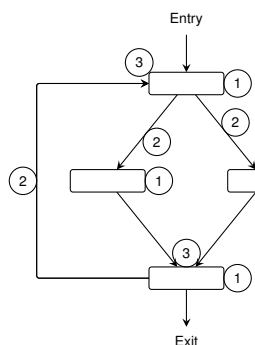
Beispiel: LRU-Cache, Zugriff auf eine Speicherstelle

- Annäherung des Cache-Verhaltens durch **must-** und **may-**Approximation:
Aktualisierung von Inhalt und Verwaltungsinformation



Wie funktioniert nun die Cache-Analyse?

Die Analyse ist eine **Datenflussanalysen** [1, Kapitel 8]



- sammle Information in den Grundblöcken

- Speicherzugriffe (s. Folie V/14)
- man bestimmt die **Übertragungsfunktion** (engl. *transfer function*) des Grundblocks

- die Information wird über ausgehende Kanten weiterverteilt

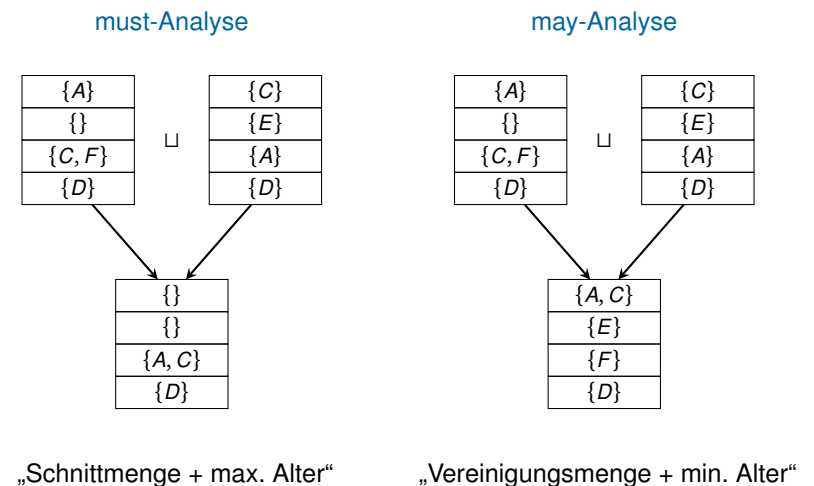
- Eingabe für die Übertragungsfunktion der folgenden Grundblöcke

- fließt der Kontrollfluss wieder zusammen, wird auch die Information verschmolzen

→ Verschmelzungsoperatoren

Verschmelzungsoperatoren für **must-** und **may-**Analyse

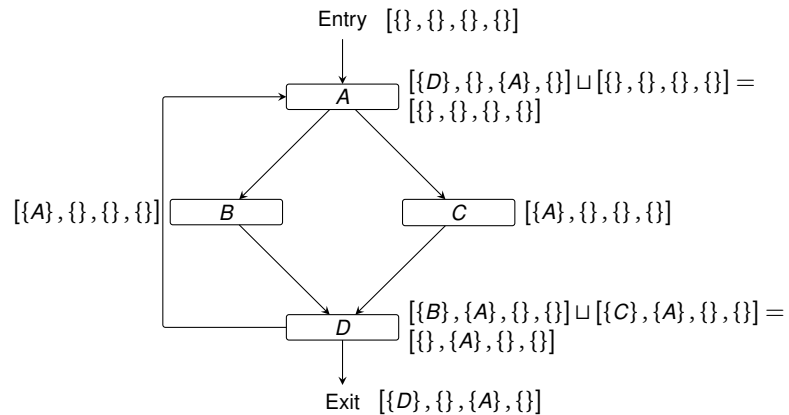
Verschmelzungsoperatoren für **must-** und **may-**Analyse



„Schnittmenge + max. Alter“

„Vereinigungsmenge + min. Alter“

Beispiel: must-Analyse für LRU



Hier ist leider keine Vorhersage von Treffern möglich ☹️



Praxisrelevante Cache-Implementierungen



Cache-Analyse mithilfe einer Datenflussanalyse funktioniert für **mengenassoziative Caches mit LRU** sehr gut

→ Zugriffe auf unterschiedliche Cache-Zeilen beeinflussen sich nicht

■ Beispiel TriCore: 2-fach assoziativer LRU-Cache



Es kommen auch andere Strategien zum Einsatz:

→ Im Durchschnitt ähnliche Leistung wie LRU, **weniger vorhersagbar**

■ **Pseudo-LRU**

- Cache-Zeilen werden als Blätter eines Baums verwaltet
- must-Analyse **eingeschränkt brauchbar**, may-Analyse **unbrauchbar**
- Beispiel: PowerPC 750/755

■ **Pseudo-Round-Robin**

- 4-fach mengenassoziativer Cache mit **einem** 2-bit Ersetzungsähler
- must-Analyse **kaum**, may-Analyse **überhaupt nicht brauchbar**
- Beispiel: Motorola Coldfire 5307



Keine belastbaren Aussagen zum STM32F104



Übersicht

1 Rekapitulation: Worst-Case Execution Time

2 Ausflug: Cache-Analyse

- Grundlagen
- Beispiel: LRU-Cache

3 WCET-Analyse auf dem EZS-Board

- GPIOs
- aiT



GPIO

General Purpose Input/Output

- Pins eines Mikrochips zur **freien Verwendung**
- Konfigurierbar als Ein-/Ausgang
- Teilweise pegelfest bis 5 V
→ Mikrocontroller-Handbuch lesen ☺️
- Zugriff über
 - spezielle Speicheradressen
 - Spezialanweisungen

Ansteuerung

`void ezs_gpio_set ( bool )`

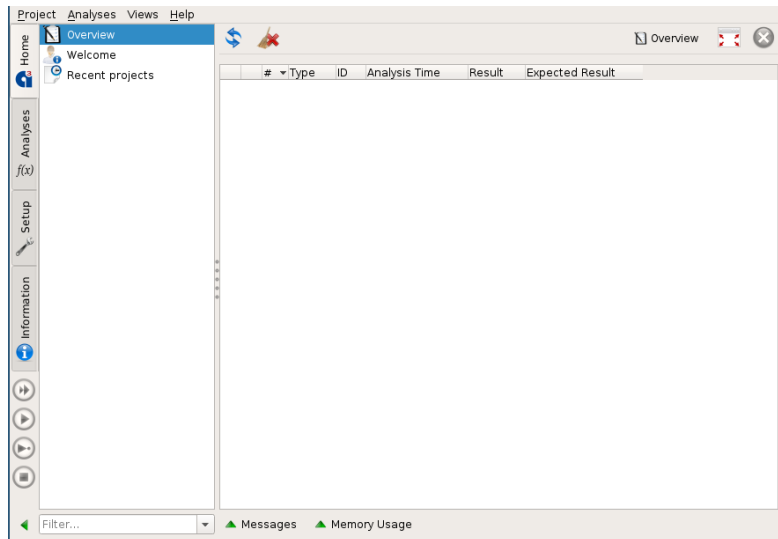
Auswertung

Oszilloskop



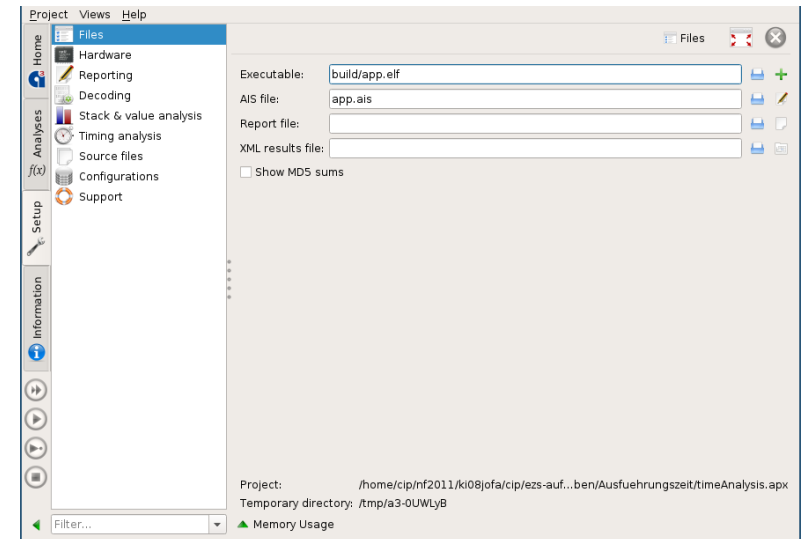
Startansicht

AbsInt aiT



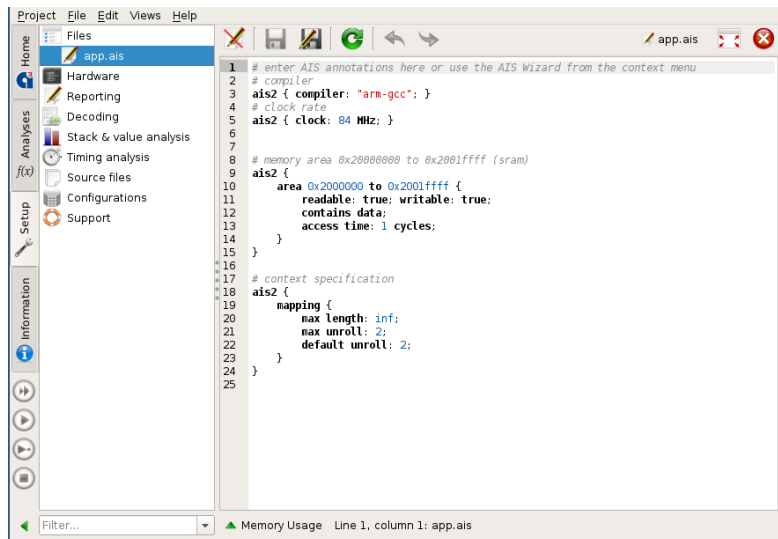
Projektdateien

AbsInt aiT



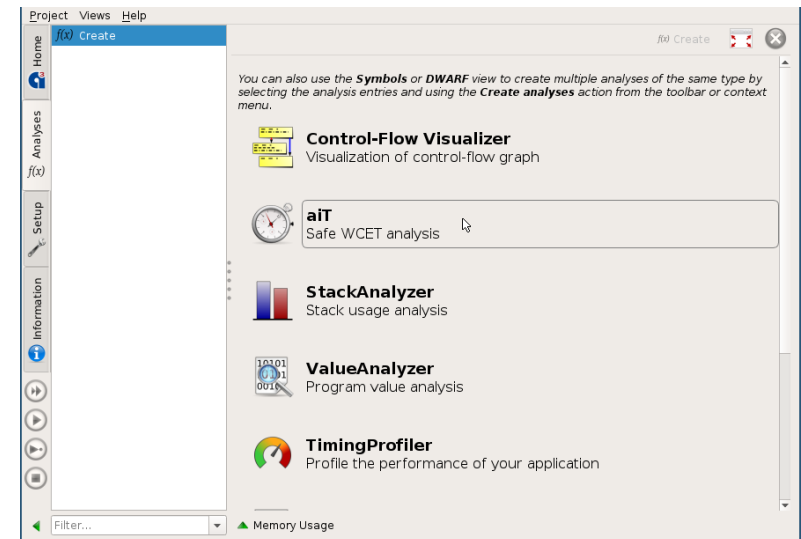
Projektdateien annotieren

AbsInt aiT



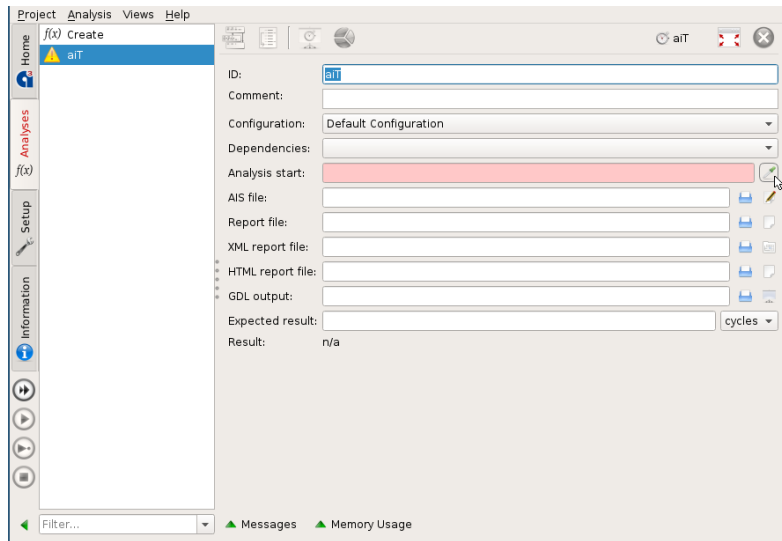
Neue Analyse anlegen

AbsInt aiT



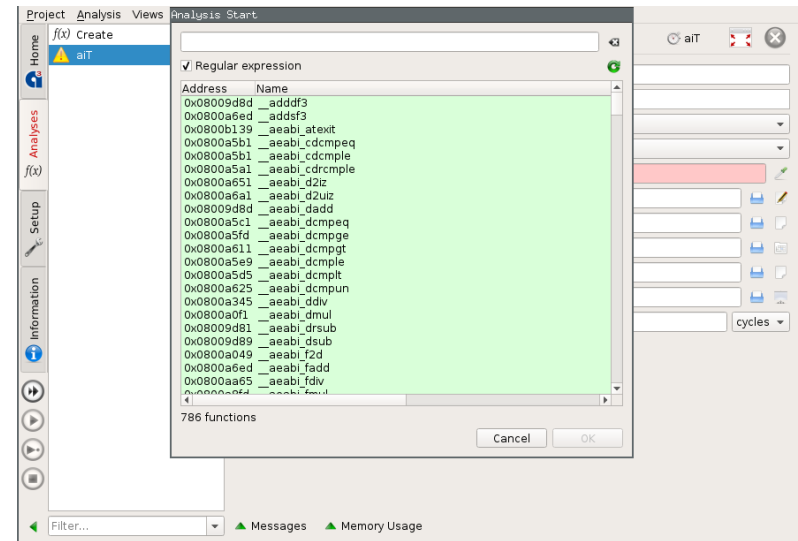
Neue Analyse anlegen

AbsInt aiT



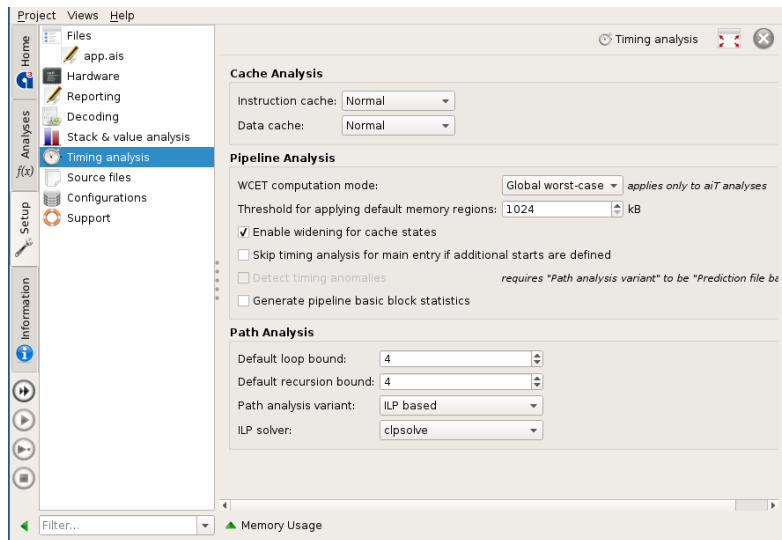
Neue Analyse anlegen

AbsInt aiT



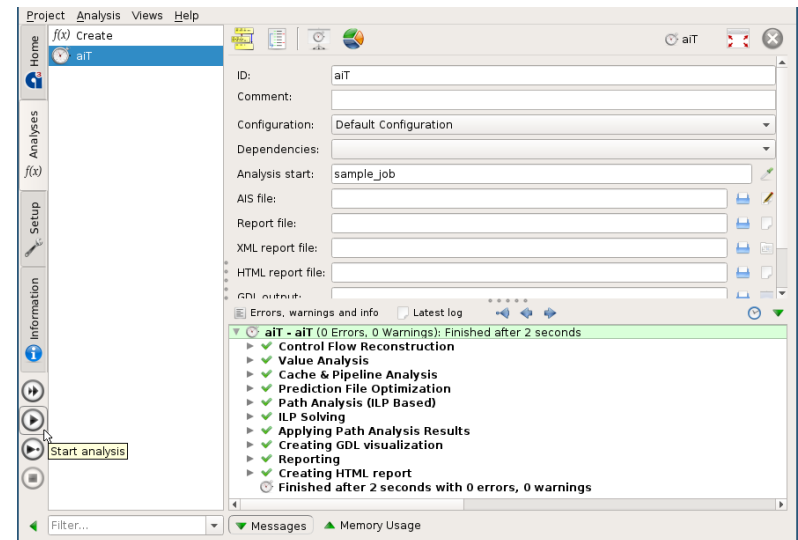
Analyseparameter

AbsInt aiT



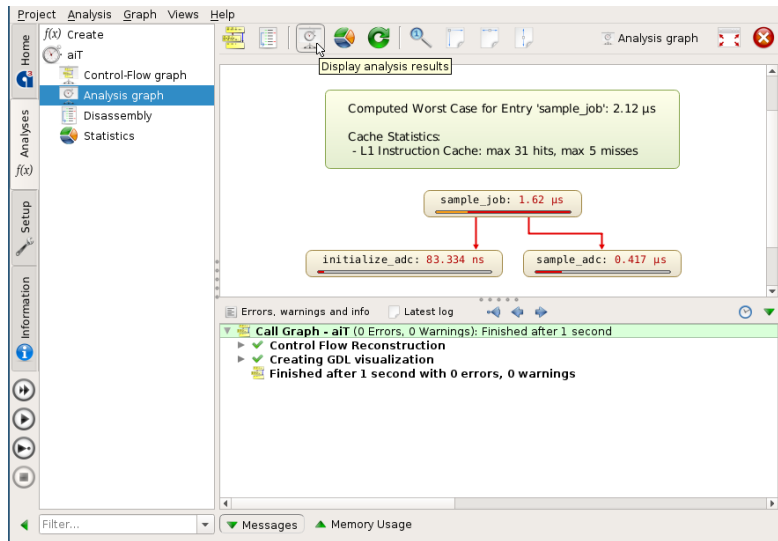
Analyse starten

AbsInt aiT



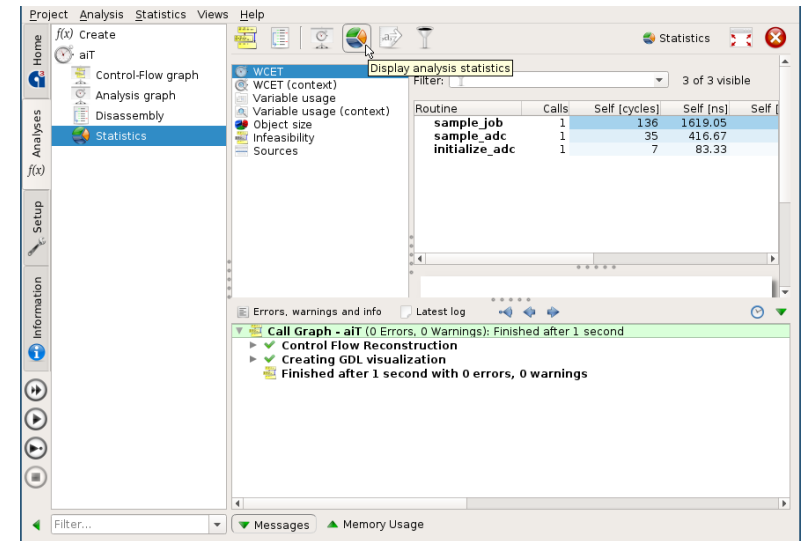
Analyse untersuchen

AbsInt aiT



Analyse untersuchen

AbsInt aiT



Annotationen

CPU-Takt

- aiT gibt Zeitmessungen zunächst nur in Takten aus
- Taktrate angeben $\leadsto$ tatsächliche Zeit

Beispiele:

```
clock: 84MHz;  
clock: 83.95 .. 84.05 MHz;
```

Annotationen

Zeit annotieren

- Manche Codestücke sind nicht analysierbar
 - Ausführungszeit annotieren
- ⚠ Natürlich nur sinnvoll, wenn WCET bereits bekannt

Beispiel:

```
routine "even"{  
    not analyzed;  
    takes: 150 cycles;  
}  
  
# exclude code as far as specified program points  
instruction ProgramPoint snippet {  
    continue at: ProgramPoint1 , PP2 , ... , PPn;  
    not analyzed;  
    takes: 10 cycles;  
}
```


Annotationen

Schleifen automatisch analysieren

- Genaue Anzahl von Schleifendurchläufen zu bestimmen ist teuer
- ☞ aiT versucht standardmäßig nur zwei Durchläufe zu interpretieren
- Lohnt sich jedoch manchmal
- ☞ aiT mehr Freiheiten für die Analyse geben

Beispiel:

```
loop "sort.L1" mapping { default unroll: 100; }
```



Annotationen

Schleifen von Hand begrenzen

- Grenzen von Hand spezifizieren

Beispiele:

```
loop "sort.L2" { bound: 0 .. 10 end; }  
loop "sort.L2" { bound: 10 begin; }  
loop "sort.L2" { bound: 10 .. inf end; }  
loop "sort.L2" { takes 20 ms; }
```

- Grenzen in Abhängigkeit von Registern spezifizieren

Beispiele:

```
loop "sort.L2" {  
    bound: 0 .. floor((reg("r0") - reg("r1")) / 4);  
}
```



Annotationen

Kontextsensitive Schleifenannotationen

Problem:

```
if (C) {  
    A(); // Vorbedingungen für Schleife in R()  
    R();  
} else {  
    B(); // Andere Vorbedingungen für R()  
    R();  
}
```

Lösung:

```
routine "A" { enter with: user("rmax") = 10; }  
routine "B" { enter with: user("rmax") = 20; }  
loop "R.L1" { bound: 0 .. user("rmax"); }
```



Annotationen

Rekursion

- aiT ist oft nicht in der Lage Rekursionen zu analysieren
→ Grenzen von Hand spezifizieren

Beispiele:

```
routine "fib" { recursion bound: 0 .. 10; }  
routine "fib" { recursion bound: 10; }  
routine "fib" { recursion bound: 5 .. 10; }
```

- Weitere Annotationen im Hilfe-Menü des aiT
→ „AIS2 quick reference“



- [1] Steven S. Muchnick.
Advanced compiler design and implementation.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.
- [2] Peter Puschner.
Zeitanalyse von Echtzeitprogrammen.
PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1,
1040 Vienna, Austria, 1993.
- [3] Reinhard Wilhelm.
Embedded systems.
[http:
//react.cs.uni-sb.de/teaching/embedded-systems-10-11/lecture-notes.html](http://react.cs.uni-sb.de/teaching/embedded-systems-10-11/lecture-notes.html),
2010.
Lecture Notes.

