

Echtzeitsysteme - Übungen

Analyse von Ausführungszeiten

Sommersemester 2024

Eva Dengler Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)

Lehrstuhl Informatik 4 (Systemsoftware)

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



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Rekapitulation: Worst-Case Execution Time

Ausflug: Cache-Analyse

- Grundlagen

- Beispiel: LRU-Cache

WCET-Analyse auf dem EZS-Board

- GPIOs

- aiT

Rekapitulation: Worst-Case Execution Time

Ausflug: Cache-Analyse

Grundlagen

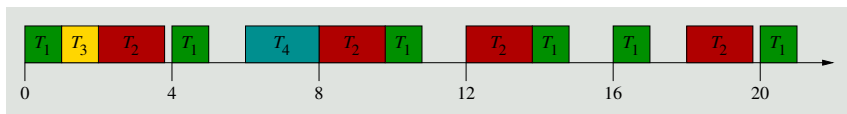
Beispiel: LRU-Cache

WCET-Analyse auf dem EZS-Board

GPIOs

aiT

Worst-Case Execution Time



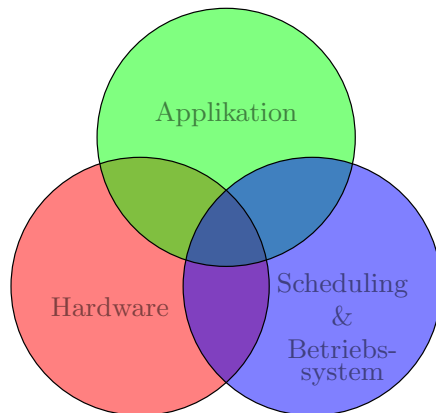
- Eine entscheidende Größe für:

- 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, ...

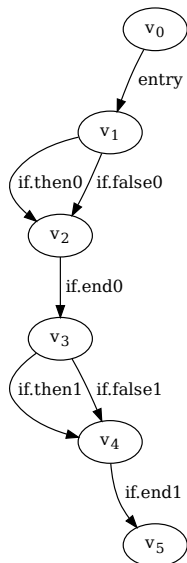
```

1 void func(int a) { // entry
2   if (a % 2)
3     f(); // if.then0
4   ++a; // if.end0
5
6   if(a % 2)
7     g(); // if.then1
8   ... // if.end1
9 }

```

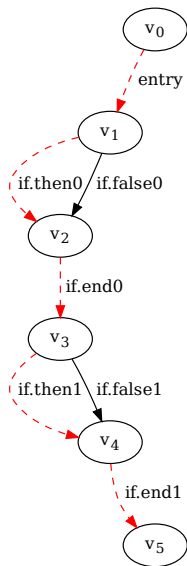
- T-Graph aus Kontrollflussgraph abgeleitet
- Worst Case == maximaler Fluss durch T-Graph
- Nebenbedingungen des Flussproblems:
 - $freq(entry) = freq(if.then0) + freq(if.false0)$
 - $freq(if.then0) + freq(if.false0) = freq(if.end0)$
 - ...
- Nebenbedingungen werden für Integer Linear Program (ILP) verwendet: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

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



```
1 void func(int a) { // entry
2   if (a % 2)
3     f(); // if.then0
4   ++a; // if.end0
5
6   if(a % 2)
7     g(); // if.then1
8   ... // if.end1
9 }
```

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



☞ 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 [2]

- Ergebnis: $e_{\text{getop}} = 80$ Zyklen

- Annahmen: Obere Schranke..

... für jede Instruktion

... der Sequenz durch Summation

⚠ Äußerst pessimistisch und zum Teil falsch

- **Falsch** für mit Laufzeitanomalien behaftete Systeme

- (intuitive) Annahmen über Worst-Case-Verhalten verletzt
- Lokales Maximum führt nicht zwingend zu globalem Maximum

- **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

Rekapitulation: Worst-Case Execution Time

Ausflug: Cache-Analyse

Grundlagen

Beispiel: LRU-Cache

WCET-Analyse auf dem EZS-Board

GPIOs

aiT

««««< Updated upstream ===== >>>>> Stashed changes

☞ Cache: ein kleiner, schneller Zwischenspeicher

- Zugriffszeiten variieren je nach Zustand des Caches enorm:

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

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

⚠ **Hits** sind schneller als **Misses**: $e_m \gg e_h$

→ 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*)

Zeilenersetzungsstrategie

- 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



- 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

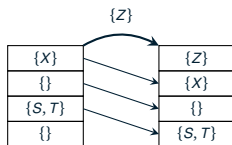
- ☞ must- und may-Analyse approximieren diese konkrete Semantik:
- must** Obergrenze des Alters \rightsquigarrow Unterapproximation des Inhalts
 - Obergrenze \leq Assoziativität \rightsquigarrow garantiert im Cache
 - may** Untergrenze des Alters \rightsquigarrow Überapproximation des Inhalts
 - Untergrenze $>$ Assoziativität \rightsquigarrow garantiert nicht im Cache

Beispiel: LRU-Cache, Zugriff auf eine Speicherstelle

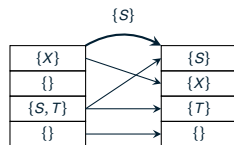
- Annäherung des Cache-Verhaltens durch must- und may-Approximation: Aktualisierung von Inhalt und Verwaltungsinformation

must-
Approximation

Potential Cache Miss

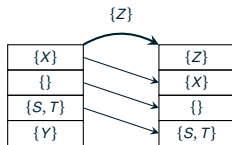


Definitive Cache Hit

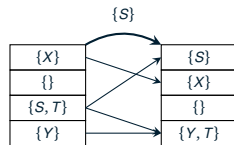


may-
Approximation

Definitive Cache Miss

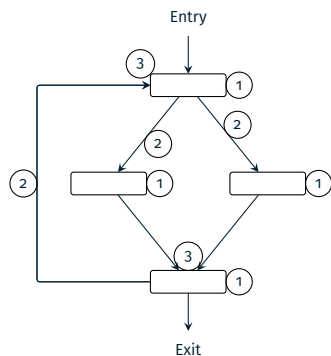


Potential Cache Hit



Wie funktioniert nun die Cache-Analyse?

- ☞ Die Analyse ist eine Datenflussanalysen [1, Kapitel 8]

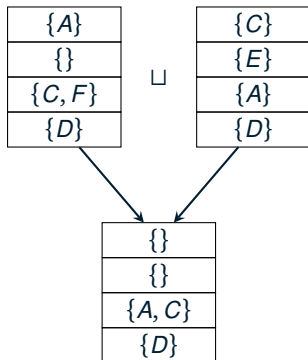


1. sammle Information in den Grundblöcken
 - Speicherzugriffe (s. Folie 13)
 - man bestimmt die Übertragungsfunktion (engl. *transfer function*) des Grundblocks
2. die Information wird über ausgehende Kanten weiterverteilt
 - Eingabe für die Übertragungsfunktion der folgenden Grundblöcke
3. fließt der Kontrollfluss wieder zusammen, wird auch die Information verschmolzen
 - Verschmelzungsoperatoren

- ☞ Verschmelzungsoperatoren für must- und may-Analyse

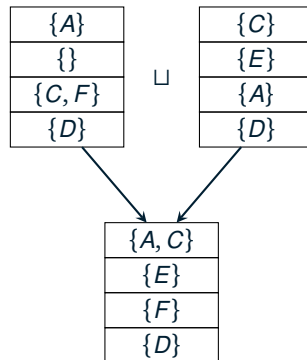
Verschmelzungsoperatoren für must- und may-Analyse

must-Analyse



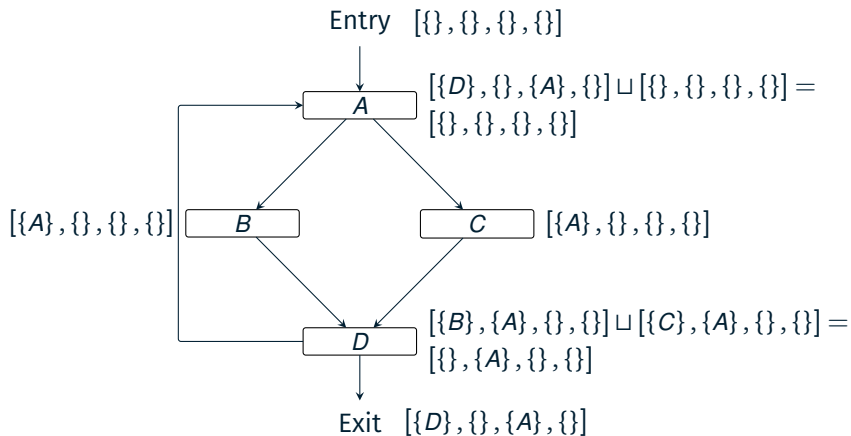
„Schnittmenge + max. Alter“

may-Analyse



„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 Ersetzungszähler
 - must-Analyse **kaum**, may-Analyse **überhaupt nicht brauchbar**
 - Beispiel: Motorola Coldfire 5307
- ☞ Keine belastbaren Aussagen zum STM32F429

Rekapitulation: Worst-Case Execution Time

Ausflug: Cache-Analyse

Grundlagen

Beispiel: LRU-Cache

WCET-Analyse auf dem EZS-Board

GPIOs

aiT

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

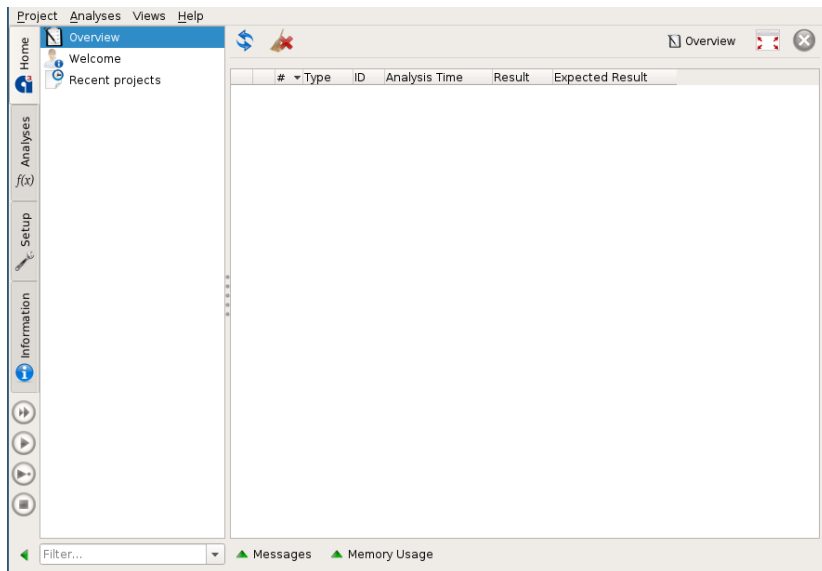


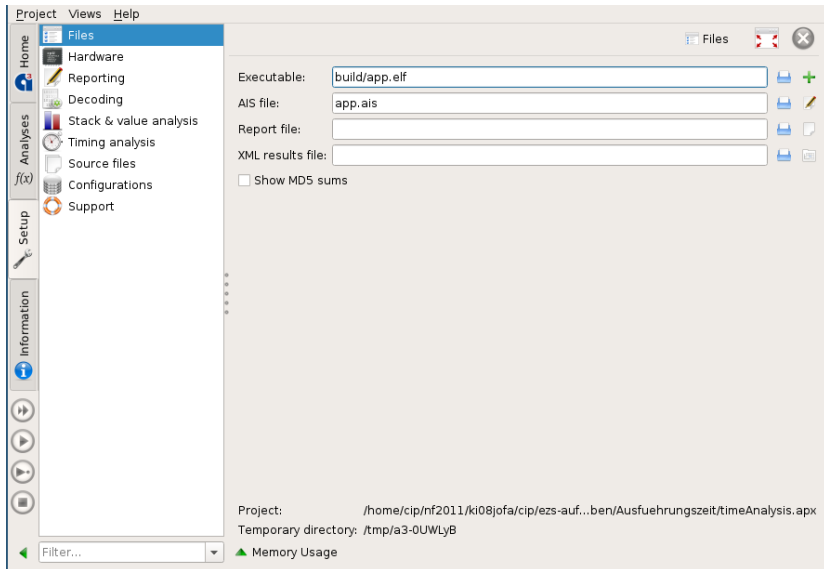
Ansteuerung

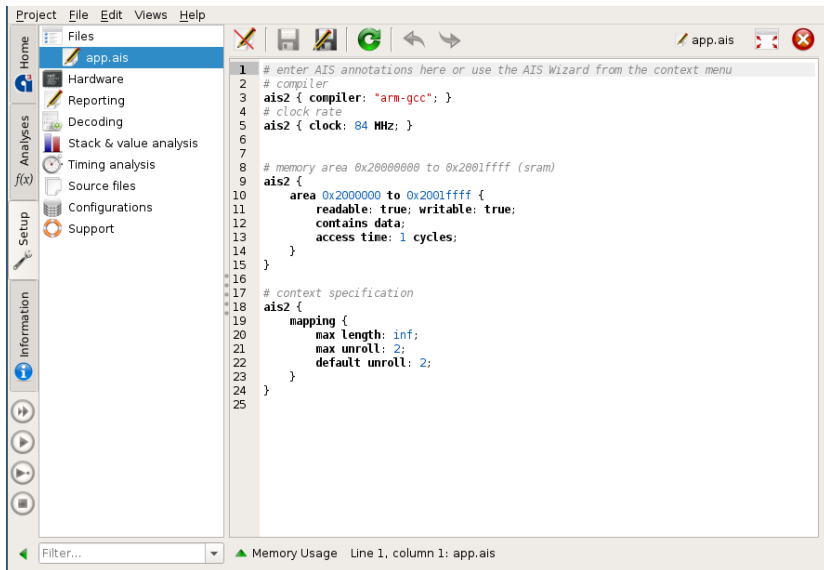
```
void ezs_gpio_set(bool) //PD12
```

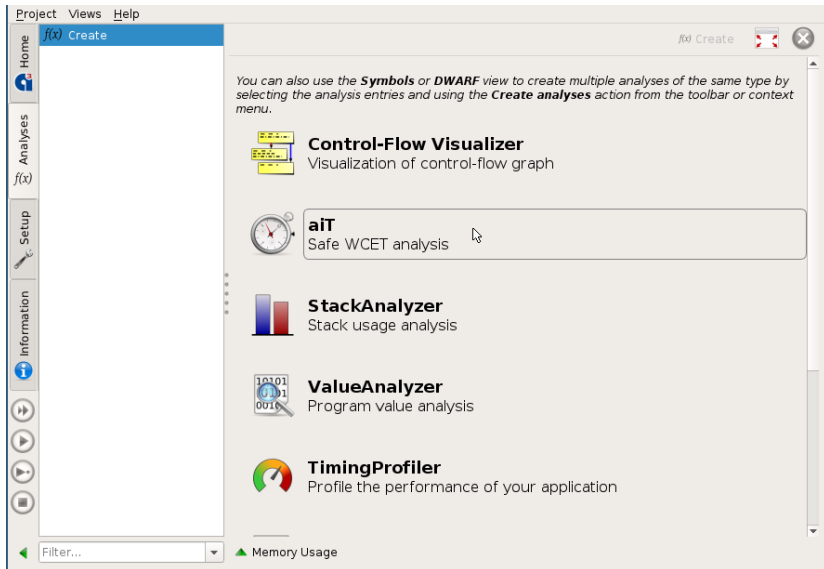
Auswertung

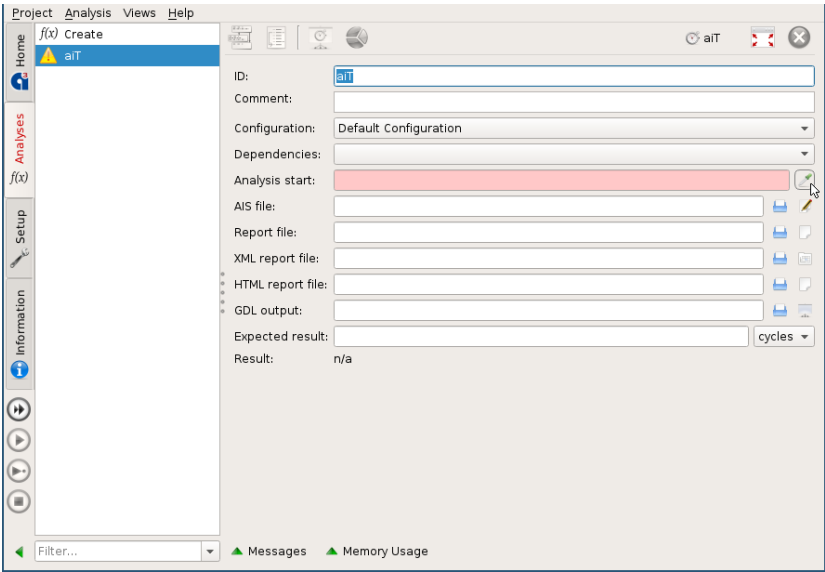
↳ Oszilloskop

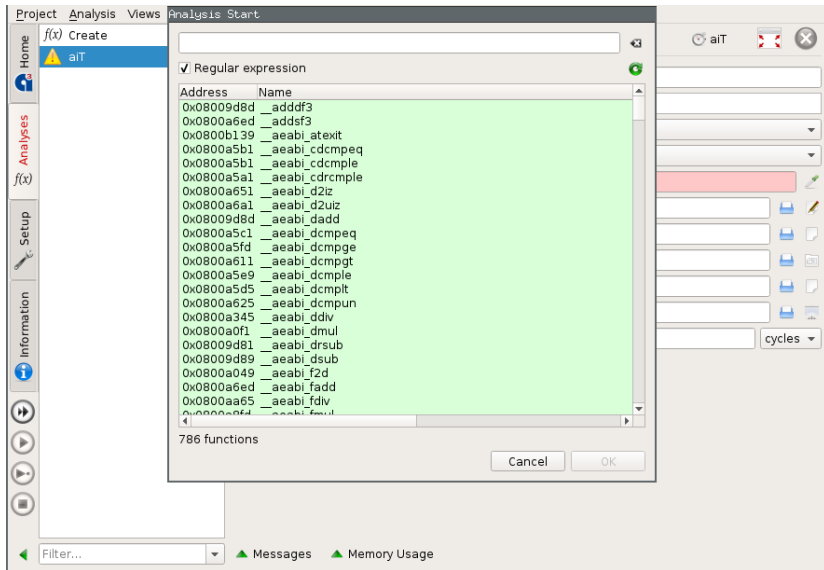








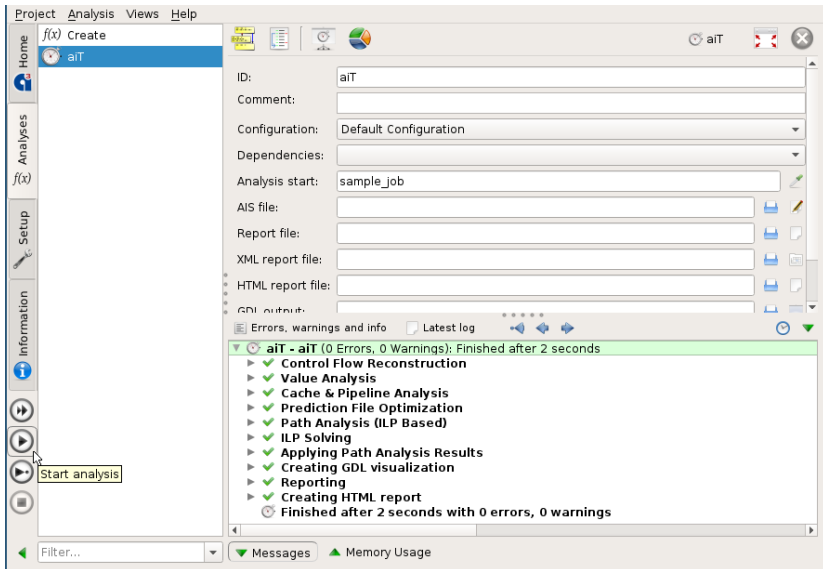




The screenshot displays the AbsInt aiT software interface. On the left is a sidebar with a menu containing: Home, Analyses (with 'Timing analysis' selected), f(x), Setup, and Information. Below the menu are navigation icons: Home, Back, Forward, Stop, and Refresh. At the bottom of the sidebar is a search bar labeled 'Filter...'. The main window has a title bar with 'Project Views Help' and a 'Timing analysis' window icon. The interface is divided into several sections:

- Cache Analysis:** Instruction cache: Always hit; Data cache: Always hit.
- Pipeline Analysis:** WCET computation mode: Global worst-case (with note 'applies only to aiT analyses'); Threshold for applying default memory regions: 1024 kB; Skip timing analysis for main entry if additional starts are defined (unchecked); Generate pipeline basic block statistics (unchecked); Interactive pipeline visualization (checked).
- Path Analysis:** Default loop bound: 4; Default recursion bound: 4; Path analysis variant: ILP based; ILP solver: clpsolve.
- TimeWeaver:** Restrict loop bounds with trace results (unchecked); Process complete traces only (unchecked).

At the bottom of the main window, there are icons for Search and Memory usage, and a small green checkmark icon.



The screenshot displays the AbsInt aiT software interface. The main window, titled "Analysis graph", shows the following analysis results:

- Computed Worst Case for Entry 'sample_job': 2.12 μ s
- Cache Statistics:
 - L1 Instruction Cache: max 31 hits, max 5 misses

Below the text, a call graph visualizes the execution flow with the following nodes and their durations:

- sample_job: 1.62 μ s
- initialize_adc: 83.334 ns
- sample_adc: 0.417 μ s

The call graph shows 'sample_job' as the root node, which branches into 'initialize_adc' and 'sample_adc'.

The bottom status bar shows the following log messages:

- Call Graph - aiT (0 Errors, 0 Warnings): Finished after 1 second
- Control Flow Reconstruction
- Creating GDL visualization
- Finished after 1 second with 0 errors, 0 warnings

The interface includes a sidebar with navigation options: Home, Analyses, Setup, and Information. The "Analyses" section is currently selected, showing options like "Control-Flow graph", "Analysis graph", "Disassembly", and "Statistics". The "Analysis graph" option is highlighted. The bottom of the window has a "Filter..." dropdown and buttons for "Messages" and "Memory Usage".

The screenshot shows the AbsInt aiT software interface. The main window is titled "Statistics" and displays a table of analysis results. A tooltip "Display analysis statistics" is visible over the Statistics icon in the toolbar. The table lists routines and their performance metrics.

Routine	Calls	Self [cycles]	Self [ns]	Self [ms]
sample_job	1	136	1619.05	
sample_adc	1	35	416.67	
initialize_adc	1	7	83.33	

Below the table, the "Errors, warnings and info" section shows a "Call Graph - aiT" with the following details:

- Call Graph - aiT (0 Errors, 0 Warnings): Finished after 1 second
- Control Flow Reconstruction
- Creating GDL visualization
- Finished after 1 second with 0 errors, 0 warnings

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

Beispiele:

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

- Manche Codestücke sind nicht analysierbar
→ Ausführungszeit annotieren

⚠ Natürlich nur sinnvoll, wenn WCET bereits bekannt

Beispiel:

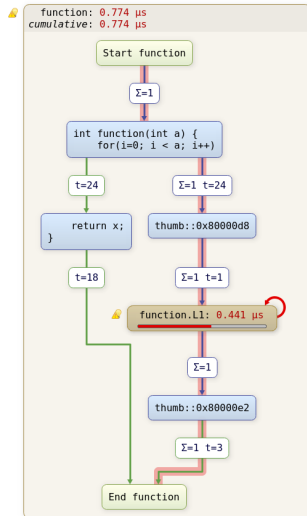
```
routine "even"{
  not analyzed;
  takes: 150 cycles;
}

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


- 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 "function.L1" mapping {  
  default unroll: 100;  
}
```



- Grenzen von Hand spezifizieren

Beispiele:

```
loop "function.L1" { bound: 0 .. 10 end; }  
loop "function.L1" { bound: 10 begin; }  
loop "function.L1" { bound: 10 .. inf end; }  
loop "function.L1" { takes 20 ms; }
```

- Grenzen in Abhängigkeit von Registern spezifizieren

Beispiele:

```
loop "function.L1" {  
  bound: 0 .. floor((reg("r0") - reg("r1")) / 4);  
}
```

The screenshot shows a code editor window with a warning annotation. The annotation text is: `ais2 { loop "function.L1" { bound: 0 .. <int>; #mapping defa...`. A context menu is open over the annotation, listing various actions such as 'Copy', 'Show in call graph', and 'Add annotation'. Below the code editor, a log window displays the results of an analysis, including a summary: 'aiT - aiT (0 Errors, 1 Warning): Finished on 2018-11-08 at 09:30:35 after analyzing for 3 seconds'. The log also lists several analysis steps with their status (e.g., 'Control Flow Reconstruction', 'Value Analysis', 'Cache & Pipeline Analysis', 'Prediction File Optimization', 'Path Analysis (ILP Based)', 'ILP Solving', 'Reporting', 'Creating HTML report') and a final summary: 'Finished on 2018-11-08 at 09:30:35 after analyzing for 3 seconds with 0 errors, 1 warning'.

☞ Die Herausforderung ist nicht die Syntax, sondern das Finden (präziser) Schleifengrenzen

Problem:

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

Lösung:

```
// Annotations-"Variable" rmax definieren  
routine "A" { enter with: user("rmax") = 10; }  
routine "B" { enter with: user("rmax") = 20; }  
// "Variable" in Annotation nutzen  
loop "R.L1" { bound: 0 .. user("rmax"); }
```

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

Problem:

```
int fib(int n) {  
    if (n <= 1)  
        return n;  
    return fib(n-1)  
        + fib(n-2);  
}
```

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>, 2010.

Lecture Notes.