Seminar
Analyse von Programmausführungszeiten
CAU Kiel, WS 2002/03, Prof. Dr. R. v. Hanxleden

Vortrag zum Thema:
Low-Level-Analyse I: Caches

Vorgetragen am 02.12.02 von Ingo Schiller

0. Überblick

1. Einführung
2. ILP Formulierung und Pfadanalyse
3. Direct-Mapped Instruction Cache Analyse
4. Implementierung und Realisierung
5. Schlussfolgerung und Bewertung
6. Ergebnisse
7. Referenzen

1. Einführung

- Warum macht man eine WCET-Analyse?
- Woraus besteht die WCET-Analyse?
- Was beeinflußt die WCET?
- Bisherige Ansätze und Defizite
- Welchen Einfluß haben Caches auf die WCET?

Warum macht man eine WCET-Analyse?

- Oft genaue Kenntnis der WCET erforderlich:
  - WCET muß Deadlines unterschreiten, die Spezifikation erfüllen.
  - Wichtig um beim Design die Trennung HW/SW herzustellen.
- Programmausführungszeiten variieren von Ausführung zu Ausführung trotz gleichem System und Eingabewerten.
- Eine genaue WCET-Kenntnis ist wichtig für effiziente Auslastung des Systems.
- Scheduling-Verfahren benötigen i.d.R. Kenntnis der WCET.

Woraus besteht die WCET-Analyse?

- Programmpfadanalyse
- Mikroarchitekturelle Modellierung

In der Spezifikation einer SW, speziell eines eingebetteten RT-Systems, müssen ganz genaue Deadlines für zeitkritische Vorgänge festgelegt werden. Um diese in jedem Fall einhalten zu können muss eine Analyse des Systems und der SW durchgeführt werden um sicherzustellen, dass das System der Spezifikation genügt.

Da bei jeder Ausführung der Systemzustand oder bei RT-Systemen auch die Umwelt verschieden sein können muss durch Analyse die WCET festgestellt werden und kann nicht durch “Ausprobieren” empirisch ermittelt werden.

Aus Gründen der Wirtschaftlichkeit muss eine hohe Systemauslastung erreicht werden.

Was beeinflußt die WCET?

- Systemzustand
- Eingabewerte

Bei modernen Prozessoren:

- Pipelining
- Caching

Caching hat dabei den größten Anteil weshalb es hier hauptsachlich betrachtet wird.

Für eingebettete Echtzeitsysteme kommt einweites Spektrum an Prozessoren zum Einsatz. Für einfache Aufgaben begnügt man sich aus Kostengründen noch häufig mit einfachen Prozessoren ohne Speicherhierarchie. Jedoch nimmt der Anteil an Prozessoren mit Caches stark zu, z.B. wenn ein Prozessor aus dem PC/Workstation-Bereich verwendet wird.

Einfluß von Caches auf die WCET

- Einfache mikroarchitekturelle Modellierung

  → Viel Pessimismus bzgl. der WCET weil der Geschwindigkeitsgewinn des Caches nicht berücksichtigt wird.
- Caches reduzieren die Ausführungszeiten durch Zwischenspeicherung in ihrem schnellen Speicher signifikant.

  → Caches müssen adäquat modelliert werden um genauere WCET zu erhalten.

Die Verwendung von Caches resultiert nicht immer in einem Geschwindigkeitsgewinn:

Falls immer ein Cache-Miss auftritt, kann ein Cache sogar verlangsamend wirken, da immer eine gesamte Speicherzeile geladen wird. (Deswegen wird in WCET-kritischen Anwendungen auch häufig der Cache deaktiviert.) Der Cache zielt, wie die meisten architekturellen Maßnahmen für moderne Prozessoren, auf die Minimierung der AverageCET, nicht WorstCET.

Bisherige Ansätze und Defizite

- Klassischer Ansatz:

  Untersuchung aller Kombinationen von Systemzustand und Eingabewerten und messen der Ausführungszeit.

  → Bei komplexen Programmen unmöglich.
- Analyse unter Annahme eines einfachen Hardware Modells.

  → Sehr pessimistisch.
- Klassifizierung jeder Instruktion bzgl. des Caches als:

  first miss, always hit, always miss und ähnliche Kategorien.

  → Schnelle aber ungenaue Cache-Analyse.

Um einen Einblick in die wirkliche Neuerung zu erhalten die dieser Vortrag zum Thema hat, betrachten wir klassische nicht zufriedenstellende Ansätze.

Beim einfachen HW-Modell gilt: Ausführungszeit jeder Instruktion ist konstant und gleich der WCET der Instruktionen, keine Beachtung des Caches.

2. ILP Formulierung und Pfadanalyse

Die WCET-Analyse teilt sich in 2 wesentliche Teile:
Die Pfadanalyse und die
Mikroarchitekturelle Analyse der darunter liegenden Hardware.
Im Folgenden betrachten wir:

- Pfadanalyse von Programmen.
- Ausführungszeitberechnung (Klassisch)
- ILP- (Integer Linear Programming) Formulierung.

Pfadanalyse

- Die Pfadanalyse stellt die Ausführungsreihenfolge der Instruktionen (bei WCET) fest.
- Annahme: Jede Instruktion hat konstante Ausführungszeit.
- Kein durchsuchen aller Programmpfade, sondern analytische Feststellung der maximalen Anzahl der auszuführenden Instruktionen unter WCET-Bedingungen. (siehe frühere Arbeiten von [Li and Malik 1995])

Siehe frühere Arbeiten der Autoren. Über die Pfadanalyse existiert eine eigene Arbeit die hier zu weit führen würde.

Die Ausführungsreihenfolge wird bestimmt, die als der WORST CASE bzgl. der Ausführungszeit bestimmt wurde.

Ausführungszeitberechnung

- Unterteilung des Programms in Blöcke, sog. „Basic Blocks“.
- Instruktionen in einem „Basic Block“ werden immer zusammen ausgeführt, daher müssen sie immer gleich oft ausgeführt werden und können als Einheit betrachtet werden.

  x_i = Ausführungszähler von Block B_i
  c_i = Ausführungszeit von Block B_i

x ist die ANZAHL wie oft ein Block ausgeführt wird.

c ist die Dauer, wie lange die Ausführung eines Blocks benötigt.

Ausführungszeitberechnung

- Gesamte Ausführungszeit des Programms:

$$t_G = \sum_{i=1}^{N} c_i x_i$$

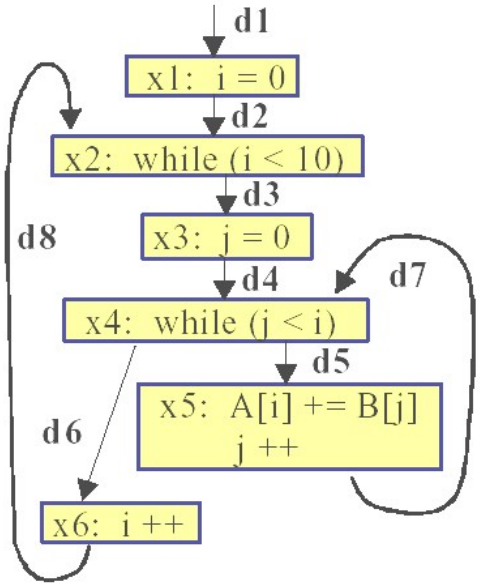
N = Anzahl der Blöcke, c_i konstant für jeden Block i.

Die hier berechnete Ausführungszeit ist noch sehr ungenau und und meist ist sie wesentlich zu hoch.

Aus der Unterteilung in Basic Blocks und der Bestimmung der Ausführungszeit der Blöcke kann man mittels oberer Formel die gesamte Ausführungszeit des Programms berechnen.

Beispiel 1

```
1: x1  i = 0;
2: x2  while (i <      10) {
3: x3       j = 0;
4: x4       while (j < i) {
5: x5           A[i] += B[j];
6:              j++;
7:          }
8: x6       i++;
9:      }
```



Structural Constraints:
x1 = d1 = d2
x4 = d4 + d7 = d5 +d6

Functionality Constraints:
10x1 = x3
0x3 ≤ x5 ≤ 9x3
45•x1 = x5

Chen 2000

Aus Vortag von Kaiyu Chen Princeton University 12/15/00

URL: http://campuscgi.princeton.edu/~znhuang/mescal/ppt/21.pdf

Links das Programm und rechts der zugehörige CFG(Control Flow Graph).

Das Programm ist unterteilt in 6 Blöcke x1 bis x6.

Die Variablen d1 bis d8 stammen aus der Programmpfadanalyse und beschreiben die Anzahl der Übergänge zwischen den Blöcken, bzw. wie oft ein Pfad durchlaufen wird.

Daraus lassen sich Ungleichungen bilden die das Verhalten des Programms beschreiben. Diese „Constraints“ Beschränkungen werden im Folgenden bei der Analyse der WCET und der ILP-Formulierung benötigt.

ILP Formulierung (ILP = Integer Linear Programming)

- Werte von x_i und d_i werden beschränkt von der Programmstruktur (Pfadanalyse) und den möglichen Werten der Programmvariablen.
- Formulierung dieser Beschränkungen (constraints) als lineare Ungleichungen.
- Das Problem transformiert sich in ein ILP-Problem.
- Ein ILP-“Solver“ kann dann eine WCET unter den gegebenen Bedingungen berechnen.

Ein ILP-“Solver“ kann auf Basis der Beschränkungen und der Funktion der gesamten Ausführungszeit WCET bestimmen.

ILP Formulierung

Linearen Beschränkungen bestehen aus 2 Teilen:

1. Programmstrukturellen Beschränkungen

   Structural constraints

   (aus dem CFG(control flow graph))

2. Programmfunktionalen Beschränkungen

   Functionality constraints

   (Durch den Programmierer bestimmt: Schleifen, Bedingungen...)

Was ist ein ILP-Solver?

Ein Programm das als Eingabe erhält:

- Eine sog. „objective function“, die entweder minimiert oder maximiert werden soll,
- Eine Reihe von Ungleichungen (unsere Beschränkungen „constraints“), unter denen die Minimierung oder Maximierung stattfinden soll.

Besonderheiten:

- Bei einem ILP (I=Integer) dürfen die Variablen nur Integer-Werte annehmen.
- „Linear“ bedeutet dabei, dass die Beschränkungen linear formuliert werden müssen.

Siehe http://www.gnome.org/projects/gnumeric/doc/solver.html für Details.

Auf Basis dieser Ungleichungen kann ein ILP-Solver die Funktion minimieren oder maximieren.

3. Direct-Mapped Instruction Cache Analyse

Direct-Mapped: Jeder Adresse wird eine Cache-Zeile eindeutig zugeordnet. (Siehe Appendix)
Um Cache-Analyse in unser ILP Modell einzubinden, müssen wir unsere Ausführungszeitberechnung modifizieren.

Dazu müssen wir die Beschränkungen des Caches (linear cache constraints) hinzufügen.

Mit einem Cache resultiert jede Ausführung einer Instruktion entweder in einem Cache-„Hit“ oder in einem Cache-“Miss“.

Dies führt zu unterschiedlichen Ausführungszeiten!
→ Das bisherige mikroarchitekturelle Modell beschreibt diese Situation nicht mehr gut genug.

Cache Hit: Instruktion ist im Cache vorhanden.

Cache Miss: Instruktion ist nicht im Cache vorhanden.

Modifizierte Kostenfunktion

- Instruktionszähler x_i wird geteilt in Zähler von Cache-Hits und Cache-Misses.
- Bestimmung dieser Zähler und der Ausführungszeiten für beide Fälle:
  → eine genauere Grenze für die Ausführungszeit des Programms ist bestimmbar.
- Unterteilung der Basic Blocks in „l-Blocks“ von der Größe einer Cache-Zeile. Jeder Basic Block hat n l-Blocks.
- Dabei unterscheiden wir konfliktierende und nicht-konfliktierende l-Blocks.

Formung von l-Blocks von der Größe einer Cache-Zeile. Die Basic Blocks werden durch die Größe der Cache-Zeilen zerstückelt.

Instruktionen in einem l-Block werden immer zusammen in dieser Reihenfolge ausgeführt.

Da der Cache Controller immer eine ganze Zeile Code in den Cache lädt ist ein l-Block entweder ganz im Cache oder gar nicht.

Für jedes Set von konfliktierenden l-Blocks wird ein CCG(Cache Conflict Graph) angelegt aus dem weitere Constraints abgeleitet werden.

Modifizierte Kostenfunktion

Neuer Instruktionszähler:

$$x_i = x_{i,j}^{hit} + x_{i,j}^{miss}$$

$$j = 1,2,\ldots, n_i$$

Die neue gesamte Ausführungszeit (Kostenfunktion) beträgt nun:

$$t_G = \sum_{i=1}^{N} \sum_{j=1}^{n_i} (c_{i,j}^{hit} x_{i,j}^{hit} + c_{i,j}^{miss} x_{i,j}^{miss})$$

Wobei $c_{i,j}$ die „Kosten“ für Cache-Hits bzw. Cache-Misses bedeuten.

Für jedes Set von konfliktierenden l-Blöcken wird nun ein CCG (Cache Conflict Graph) gezeichnet. Dabei wird von jedem l-Block zu jedem anderen ein Pfad gezeichnet und mit P(i,j,k,l) beschriftet, wenn im Programm in solcher Pfad möglich ist. Die Werte von P geben dabei an wie oft der Programmfluss diesen Pfad durchläuft.

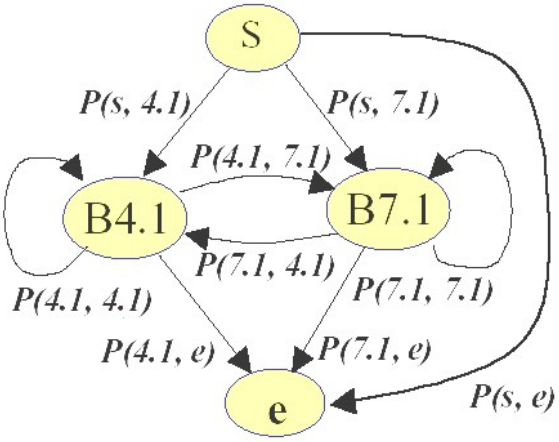
Beispiel 2

```
   /* k >= 0 */
x1 s = k;
x2 while (k < 10) {
x3   if (ok)
x4     j++;
x5   else {
x6     j = 0;
x7     ok = true;
     }
x8   k++;
   }
x9 r = j;
```

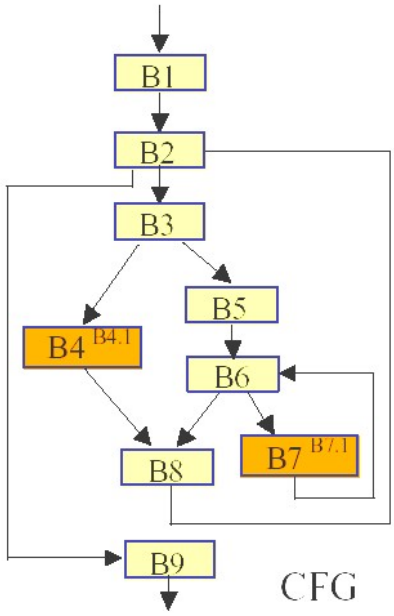
(Beispiel-Sourcecode für den CCG)

Chen 2000

Beispiel 2



Cache conflict graph



CFG

Constraints:

$p(s, 4.1) + p(s, 7.1) + p(s, e) = 1 \qquad p(s, 7.1) + p(4.1, 7.1) \leq x5$

$x4 = p(s,4.1) + p(4.1,4.1) + p(7.1,4.1) = p(4.1,e) + p(4.1,4.1) + p(4.1,7.1)$

$p(4.1,4.1) \leq x_4^{hit} \leq p(s,4.1) + p(4.1,4.1) \qquad x_4^{hit} + x_4^{miss} = x4$

Chen 2000

Ein CCG für die konfliktierenden l-Blocks B4.1 und B7.1. S ist Startknoten und e Endknoten. Aus dem CFG und dem CCG werden die Cache-Constraints ermittelt.

Die Werte von P(...) gibt dabei die Anzahl der Übergänge im Graph an, bzw. Wie oft ein Block in den Cache geladen werden muss. Über die x Variablen werden sie mit den anderen Ungleichungen verbunden.

Beschränkungen bzgl. des Caches

Diese Beschränkungen werden verwendet um die Werte der $X_{i,j}$ also der Hit-und Miss-Zähler zu beschränken.

Dabei sind konfliktierende und nicht-konfliktierende l-Blocks zu unterscheiden.

Bei nicht-konfliktierenden l-Blocks ist z.B. $x_{i,j}^{miss} \leq 1$,

Bei nicht-konfliktierenden l-Blocks die aber in der gleichen

Cache-Zeile stehen gilt entsprechend: $x_{i,j}^{miss} + x_{k,l}^{miss} \leq 1.$

Konfliktierende l-Blocks: ein l-Block wird durch den anderen bei Ausführung ersetzt.

Nicht-konfliktierende l-Blocks: wird ein l-Block einmal in den Cache geladen dann bleibt er dort resistent.

2.Punkt: Cache Controller lädt immer eine ganze Zeile in den Cache. Deshalb ist die gesamte miss-Anzahl <=1.

Beschränkungen bzgl. des Caches

Die Werte der P-Variablen müssen sozusagen „von Hand“ beschränkt werden. Denn die P-Variablen erhalten viel zu hohe Werte, da bedingte Schleifendurchgänge unberücksichtigt bleiben.

Die Beschränkung wird durch: $0 \leq P_{(i,j,u,v)} \leq \min(x_i, x_u)$

mit in die anderen Beschränkungen eingebunden.

Das verwendete Programm „Cinderella“ stellt die Möglichkeit zur Verfügung diese Einstellungen nach Analyse des Programms vorzunehmen.

4. Implementierung und Realisierung

- Verfahren von den Autoren (Li and Malik) realisiert im Tool „Cinderella“.
- Ausgeführt auf einem 32Bit RISC-Prozessor von Intel , mit einem direct-mapped instruction cache mit 32x16-byte Zeilen.
- Es benötigt als Input die auszuführende Datei und konstruiert automatisch die CFGs und CCGs.
- Der User muss dann Schleifen-Beschränkungen eingeben und kann zusätzliche Ausführungspfad-Information eingeben.

Informationen zu „Cinderella“ findet man unter:

http://www.ee.princeton.edu/~yauli/cinderella-3.0/

5. Ergebnisse

Table I. Benchmark Examples: Descriptions, Source File Line and i960KB Binary Code Sizes

Program	Description	Lines	Bytes
check data	Check if any of the elements in an array is negative, from Park [1992]	23	88
circle	Circle drawing routine, from Gupta [1993]	100	1,588
des	Data Encryption Standard	192	1,852
dhry	Dhrystone benchmark	761	1,360
djpeg	Decompression of 128 × 96 color JPEG image	857	5,408
fdct	JPEG forward discrete cosine transform	300	996
fft	1024-point Fast Fourier Transform	57	500
line	Line drawing routine, from Gupta [1993]	165	1,556
matcnt	Summation of 2 100 × 100 matrices, from Arnold [Arnold, Mueller, Whalley, and Harmon 1994]	85	460
matcnt2	Matcnt with inlined functions	73	400
piksrt	Insertion sort of 10 elements	19	104
sort	Bubble sort of 500 elements, from Arnold [Arnold, Mueller, Whalley, and Harmon 1994]	41	152
sort2	Sort with inlined functions	30	148
stats	Calculate the sum, mean and variance of two 1,000-element arrays, from Arnold [Arnold, Mueller, Whalley, and Harmon 1994]	100	656
stats2	Stats with inlined functions	90	596
whetstone	Whetstone benchmark	196	2,760

In Table I sehen wir die Programme die verwendet wurden um die Implementierung im Tool „Cinderella“ zu testen und um die theoretischen Ansätze zu validieren. Dazu sehen wir die Länge des Programms in Zeilen und Binärcodegröße des Programms auf der Testmaschine.

Ergebnisse

Table II. Estimated WCETs of Benchmark Programs. Estimated WCETs and Measured WCETs In Units of Clock Cycles

Program	Measured WCET	Estimated WCET	Ratio
check data	4.30×10^2	4.91×10^2	1.14
circle	1.45×10^4	1.54×10^4	1.06
des	2.44×10^5	3.70×10^5	1.52
dhry	5.76×10^5	7.57×10^5	1.31
djpeg	3.56×10^7	7.04×10^7	1.98
fdct	9.05×10^3	9.11×10^3	1.01
fft	2.20×10^6	2.63×10^6	1.20
line	4.84×10^3	6.09×10^3	1.26
matcnt	2.20×10^6	5.46×10^6	2.48
matcnt2	1.86×10^6	2.11×10^6	1.13
piksrt	1.71×10^3	1.74×10^3	1.02
sort	9.99×10^6	27.8×10^6	2.78
sort2	6.75×10^6	7.09×10^6	1.05
stats	1.16×10^6	2.21×10^6	1.91
stats2	1.06×10^6	1.24×10^6	1.17
whetstone	6.94×10^6	10.5×10^6	1.51

In Table 2 sehen wir die gemessene WCET, d.h. den größten Wert aus einem zufälligen Set von Systemzustand und Eingabevariablen, die durch unsere Analyse-Methode erwartete WCET und das Verhältnis der beiden. Kleineres Verhältnis bedeutet dabei eine genauere Voraussage.

Die schlechten Voraussagen bei sort, matcnt und stats sind deswegen so schlecht, weil die „register windows“ nicht modelliert wird.

Programme sort2, matcnt2 und stats2 haben Funktionsaufrufe inline und dies bringt eine viel bessere Schätzung.

Ergebnisse

Table III. Estimated Worst-Case Number of Cache Misses of Benchmark Programs. The instruction cache is 512–byte direct-mapped, its line size is 16 bytes

Program	DineroIII simulation	Est. worst case cache misses	Ratio
circle	443	458	1.03
des	3,872	4,188	1.08
dhry	8,304	8,304	1.00
djpeg	230,861	316,394	1.37
fdct	63	63	1.00
line	99	101	1.02
stats	47	47	1.00
stats2	44	44	1.00
whetstone	18,678	18,678	1.00

Floating Point Operationen in fft und whetstone sind in ihrer Ausführungszeit abhängig von den Daten. Deswegen ist hier die Schätzung nicht so gut. Angenommen wurde für jede Instruktion die WCET der Instruktion. Diese ist 30-40% höher als durchschnittlich.

Table III zeigt die Anzahl der Cache – Misses, die simulierten und die geschätzten und das Verhältnis der beiden. Die Ergebnisse sind durchweg gut. Einzig bei djpeg ist eine größere Abweichung zu erkennen.

6. Schlussfolgerung und Bewertung

- Es wurde ein Verfahren vorgestellt mit dem man eine genaue WCET von Programmen vorhersagen kann unter Berücksichtigung eines direct-mapped Caches.
- Dabei wird das Problem in ein ILP-Problem transformiert und von einem ILP-Solver gelöst.
- Dabei wird eine explizite Pfadnummerierung vermieden.
- Der User kann sogar zusätzliche Pfadinformation angeben um eine noch genauere WCET zu erhalten.

Schlussfolgerung und Bewertung

- Bisher nur Modellierung eines direct-mapped instruction caches. (Autoren versprechen Erweiterung auf set-assoziative Caches, doch bisher keine Ergebnisse.)
- Daher sehr beschränkte Einsatzmöglichkeiten.
- Anzahl der „constraints“ steigen bei größeren Programmen stark an. (z.B. djpeg ~2.500!!)
- Jedoch wird die WCET durch diese Methode recht genau vorhergesagt und bietet für kritische Echtzeitprogramme eine Möglichkeit der genauen Voraussage der WCET und damit einer effizienten Auslastung der Systeme.

7. Referenzen

- Perfomance Estimation of Embedded Software with Instruction Cache Modelling

  (Yau-Tsun Steven Li, Sharad Malik and Andrew Wolfe, Princeton University)
- Real-Time Memory Management: Compile-Time Techniques and Run-Time Mechanisms that Enable the Use of Caches in Real-Time Systems

  (Bruce L.Jacob and Shuvra S.Bhattacharyya, University of Maryland, College Patrk)

Appendix

- Anschließend optionale Folien zu Punkten, die nicht im Vortrag behandelt werden …
- Darin:
  - Was sind Caches?
  - Was sind Echtzeitsysteme?
  - Funktionsaufrufe bei WCET-Analyse.

Was sind Caches?

- Bestehen aus schnellem Speicher, vergleichbar mit dem Speicher dessen Inhalt sie cachen, meist aus SRAM.
- Jedoch ist nicht jeder schnelle Speicher ein Cache. Caches halten Kopien von Daten, nicht die Daten selber. (vgl. DSP: aus SRAM, aber halten keine Kopien von Daten.)
- Die Adressierungsmethode ist transparent. (Adressierung entspricht der Adressierung des Hauptspeichers.)

Was sind Caches?

- Wenn die Kopie der Daten im Cache mit den originalen Daten übereinstimmen soll muss dies explizit über Software geregelt werden.
- Ein Programm muss nicht über die Existenz eines Caches bescheid wissen um ihn zu benutzen.
- Alles andere ist kein Cache!

Was sind Caches?

- Zuweisungsstrategien für Caches:
  - Direct-Mapped:

    Jeder Adresse wird eine Cache-Zeile eindeutig zugeordnet.

    (Blockadresse)module(Cachegröße=Anzahl von Zeilen)
  - Voll-Assoziativ:

    Alle Daten können im ganzen Cache gespeichert werden.
  - Set-Assoziativ:

    Kompromiss zwischen direct-mapped und voll-assoziativ. Jede Adresse kann in einem bestimmten Set von Cache-Zeilen gespeichert werden.

Was sind Echtzeit-Systeme?

Systeme bei denen die Korrektheit des Ergebnisses nicht nur vom logischen Resultat, sondern auch vom Zeitpunkt an dem das Resultat geliefert wird, abhängt.

(System mit Deadline)

Eingebettete Systeme: Systeme mit sehr eingeschränkten Funktionen, die nur zu einem ganz bestimmten Zweck und Nutzen produziert werden.

Oftmals keine Programmierung auf dem System möglich weshalb eine Portierung auf das jeweilige System erfolgen muss.

Eine vohersagbare Ausführungszeit und garantierte WCET sind essentiell.

Funktionsaufrufe

- Alle Funktionsaufrufe innerhalb des betrachteten Programms werden behandelt als wären sie „inline“.
- Variablen von aufgerufenen Funktionen erhalten zur Unterscheidung ein Suffix „.f_k“.
- So werden Funktionsaufrufe mit in die ILP-Formulierung eingebunden.
- Dabei muss bei mehrmaligem Aufruf einer Funktion besonders auf Cache-Konflikte geachtet werden.