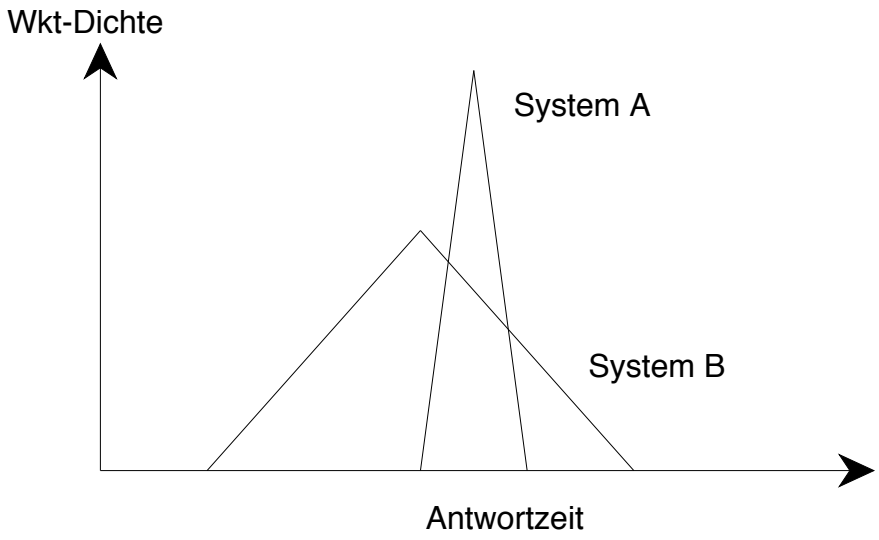
Performance Maße für Echtzeitysteme



- Verhalten von System A ist besser vorhersagbar als das von System B
- System B ist schneller

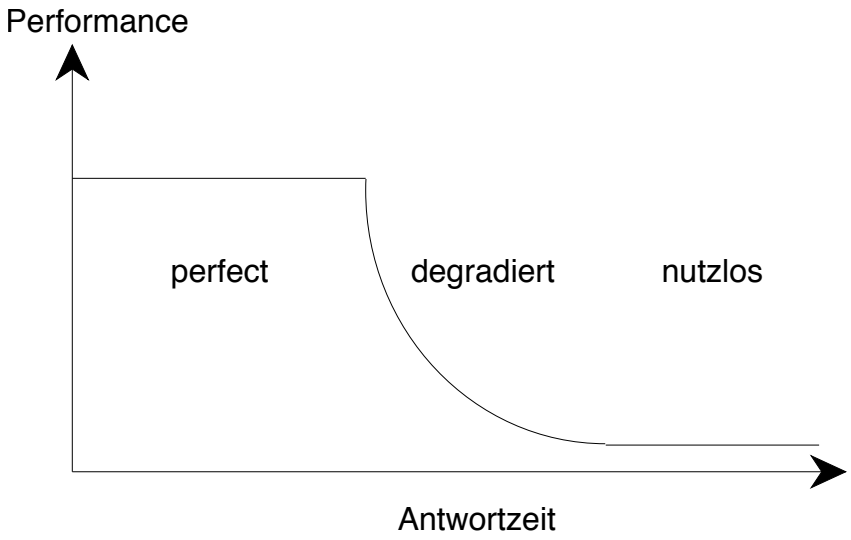
->B ist besser als A bezüglich **mittlerer Antwortzeit**

aber:

- sei Performance $P = a_1M + a_2V$; wobei M mittlere Antwortzeit und V Varianz seien
- a_1 und a_2 werden vom Benutzer bestimmt

->kein formaler Weg, diese Parameter zu bestimmen
-> kein einfacher Vergleich von A und B möglich

Darstellung von Zeichen auf dem Bildschirm:



- Benutzer kann nicht zwischen 5μs und 10 μs Verzögerung beim Echo von Zeichen unterscheiden

-> beide Antwortzeiten liefern gleiche Performance

Applikation bestimmt Bewertung eines Systems

- Computer A und B benötigen für jede Instruktion gleichviele Zyklen
- A hat speziellen Arrayprozessor; kann 256x256 arrays in 4 Takten berechnen
- B hat 10 MHz Taktfrequenz; A nur 5 MHz
- Antwortzeit als Performancemaß

-> es gibt kein **besseres** System

Vernünftig klingendes Performancemaß kann fehlleiten

- MIPS-Rate: A < B (Faktor 1.5)
- A hat komplexen Instruktionssatz; eine Instruktion braucht 1.8 Zyklen
- B hat einfacheren Instruktionssatz; eine Instruktion braucht 1.2 Zyklen
- Leider ist dasselbe Programm auf Maschine B doppelt so lang wie auf A (Maschinencode)

-> Benutzersicht: B ist langsamer als A

Eigenschaften für Performancemaße

1) Effiziente Kodierung relevanter Informationen

2) Objektive Basis für die Bewertung eines Testkandidaten bezüglich einer gegebenen Applikation

3) Liefert objektive Optimierungskriterien

4) Präsentiert verifizierbare Fakten

Traditionelle Maße

Reliability: Wahrscheinlichkeit, daß das System über ein betrachtetes Intervall fehlerfrei arbeitet.

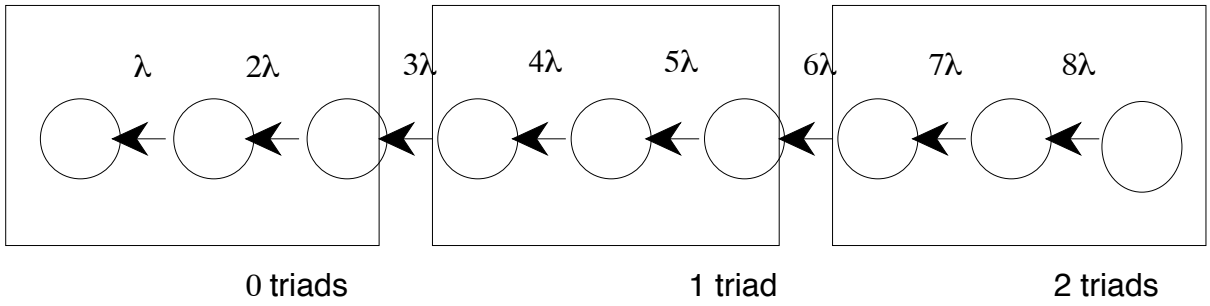
Availability: Bruchteil der Zeit, in der das System innerhalb eines betrachteten Zeitraumes verfügbar war.

Throughput: durchschnittliche Zahl von Instruktionen, die das System pro Zeiteinheit abarbeiten kann.

Was bedeutet: System ist verfügbar (funktioniert) ?

- bei einfachen Einprozessorrechnern ?
- bei fehlertoleranten/redundanten Systemen ?

3-fach redundantes, fehlertolerantes System:
TMR + voting



Markov-Diagramm: Zustände = # funktionierender Prozessoren

- 8 Prozessoren: 2 *triads* + 2 *spares*
- System sei rekonfigurierbar
- Prozessoren werden nicht repariert
- Fehler sind unabhängig, Rate λ
- Performance ist abhängig von Zahl funktionierender *triads*

Reliability $P_{fail} = \sum_{i \in FAIL} P_i(t)$

- $P_i(t)$ ist die Wahrscheinlichkeit, daß sich System zum Zeitpunkt t im Zustand i befindet.
- Menge FAIL umfaßt alle Zustände, die als fehlerhaft definiert sind

FAIL = {0,1,2} - beide *triads* ausgefallen
FAIL = {0,1,2,3,4,5} - mindestens 1 *triad* ausgefallen

- Durchsatz kann in Abhängigkeit der verfügbaren Prozessoren definiert werden; für Verfügbarkeit existieren Wahrscheinlichkeiten

Problem: Zusammenspiel von Hardware, Systemsoftware und Applikation nicht reflektiert

Definition der Menge fehlerhafter Zustände ist für komplexe Systeme schwierig

Reliability beschäftigt sich entweder mit Hard- oder Software.

Availability ist nutzlos als Maß für Einhalten von deadlines.

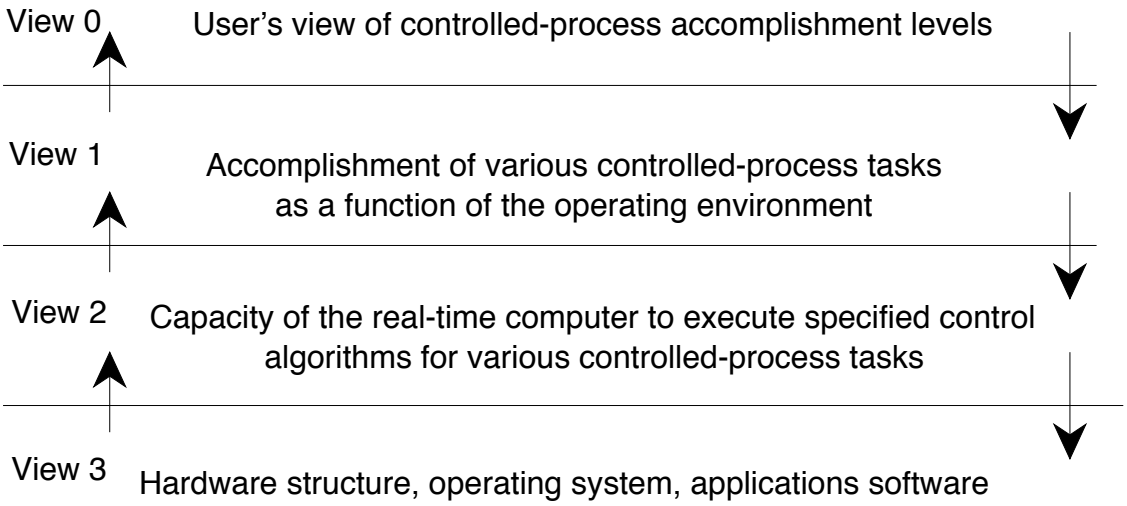
Performability

mißt *performance* in Bezug auf den *real-time process*, den ein Echtzeitrechner steuert.

Gesteuerter Prozeß hat eine Reihe von *accomplishment levels* - das sind vom Benutzer beobachtbare *performance levels*.

Auf jedem Niveau sind eine Reihe von *control tasks* auszuführen.

Performability ist ein Wahrscheinlichkeitsvektor, der beschreibt mit welcher Wahrscheinlichkeit die einzelnen *accomplishment levels* erreicht werden.



- durch passende Definition der *accomplishment levels* kann *performability* auf die traditionellen Leistungsmaße reduziert werden

  accomplishment level = processed jobs / time unit
  -> throughput

View 0: Benutzersicht
View 1: *environmental conditions*, *performance* des gesteuerten Prozeß
Sicht des Control-Ingenieurs, *tasks* und *deadlines* sind bekannt
View 2: benötigte Leistung/Ressourcen des Computers, um *tasks* abzuarbeiten
Sicht des Rechnerarchitekten
View 3: *hardware: failure rates, operating system: scheduler, system software: failure rates*
Sicht des Rechnerarchitekten

- *information hiding, abstraction*

Beispiel: Landung eines Flugzeuges

accomplishment levels aus Sicht des Benutzers:

- A_0: sichere Landung am Ziel
- A_1: Umleitung, sichere Landung
- A_2: *crash*

crash am Ziel oder entferntem Flugplatz gleichschlimm
Zustandsbeschreibung View 0:

$$a_0 = \begin{cases} 0\text{: Flugzeug nicht umgeleitet} \\ 1\text{: Flugzeug umgeleitet} \end{cases}$$

$$b_0 = \begin{cases} 0\text{: Flugzeug stürzt nicht ab} \\ 1\text{: Flugzeug stürzt ab} \end{cases}$$

Abbildung auf View 0-Ebene:

A_0 { (0,0) }; A_1 { (1,0) }; A_2 { (0,1), (1,1) }

operating environment: Wetter am Ziel
nötige *control task*: *automatic landing feature* (AL)
Zustandsbeschreibung View 1:

$$a_1 = \begin{cases} 0\text{: gute Sicht am Ziel} \\ 1\text{: schlechte Sicht} \end{cases}$$

$$b_1 = \begin{cases} 0\text{: AL funktioniert während Landung} \\ 1\text{: Fehler an AL vor Landephase} \\ 2\text{: Fehler an AL während Landephase} \end{cases}$$

$c_1 = \begin{cases} 0: \text{flugkritische Komponten funktionieren} \\ 1: \text{Fehler an kritischen Komponenten} \end{cases}$

Abbildung von View 0 Zuständen auf View 1 Zustände:

(0,0) <-> { (0,0,0), (0,1,0), (0,2,0), (1,0,0) }
(0,1) <-> { (0,0,1), (0,1,1), (0,2,1), (1,0,1), (1,2,0), (1,2,1) }
(1,0) <-> { (1,1,0) }
(1,1) <-> { (1,1,1) }

Wetter und Zustand der mechanischen Komponenten des Flugzeugs entziehen sich unserer Kontrolle, also:
Zustandsbeschreibung View2:

$a_2 = \begin{cases} 0: & \text{Computer hat genügend Ressourcen um AL-Job während Landephase auszuführen} \\ 1: & \text{Computer hat zu irgendeinem Zeitpunkt der Landephase ungenügend Ressourcen} \\ 2: & \text{Computer hat zu Beginn der Landephase genügend Ressourcen, erleidet aber Abarbeitungsfehler während der Landung} \end{cases}$

- weitere Zustandsvariable, die Sicht am Landeort spezifiert.

Performability:

- Vektor ($P(A_1)$, $P(A_2)$, $P(A_3)$)
- wobei: $P(A_i)$ Wahrscheinlichkeit ist, daß Computer accomplishment level A_i erreicht

Abschätzung von Programmlaufzeit

- schwierig, Forschungsgegenstand

Einflußfaktoren:

- *Source code*: tuning
- *Compiler*: source to machine code mapping is not unique
- *Machine architecture*: # processors, memory - processor interconnect, I/O devices
  # registers beeinflußt Abarbeitungszeit
  cache organization
  RAM: refresh-Raten
- *Betriebssystem*: implementiert scheduling, Speicher-verwaltung, interrupt-Behandlung

Ideal: Tool (Code, Comp, Arch) -> Exec-Time

Ansatz: Lassen Programm einfach oft/lange genug laufen
und messen Abarbeitungszeit

Leider: viele verschiedene Eingabewerte
Programme laufen nicht in Isolation
Komplexität

Quellcodeanalyse

```
L1:  a := b * c;
L2:  b := d + e;
L3:  d := a - f;
```

- ganz einfaches Beispiel
- keine Sprünge
- ein Eintritts-/Austrittspunkt
- $T_{exec} = T_{exec}(L1) + T_{exec}(L2) + T_{exec}(L3)$

Compiler übersetzt L1 in:

```
L1.1 Get address of c
L1.2 Load c
L1.3 Get address of b
L1.4 Load b
L1.5 Multiply
L1.6 Store into a
```

Maschinenarchitektur beeinflußt Abarbeitungszeit für diese Intruktionen

- keine pipeline, nur 1 memory-port:

  $T_{exec} = \sum^{6}_{i=1} T_{exec}(L_i)$

- keine Interrupts
- Variablen noch nicht in Registern
- Dauer der Multiplikation datenabhängig

Erhalten nur eine lose, obere Grenze für Abarbeitungszeit

```
L4: while (P) do
L5:    Q1;
L6:    Q2;
L7:    Q3;
L8: end while;
```

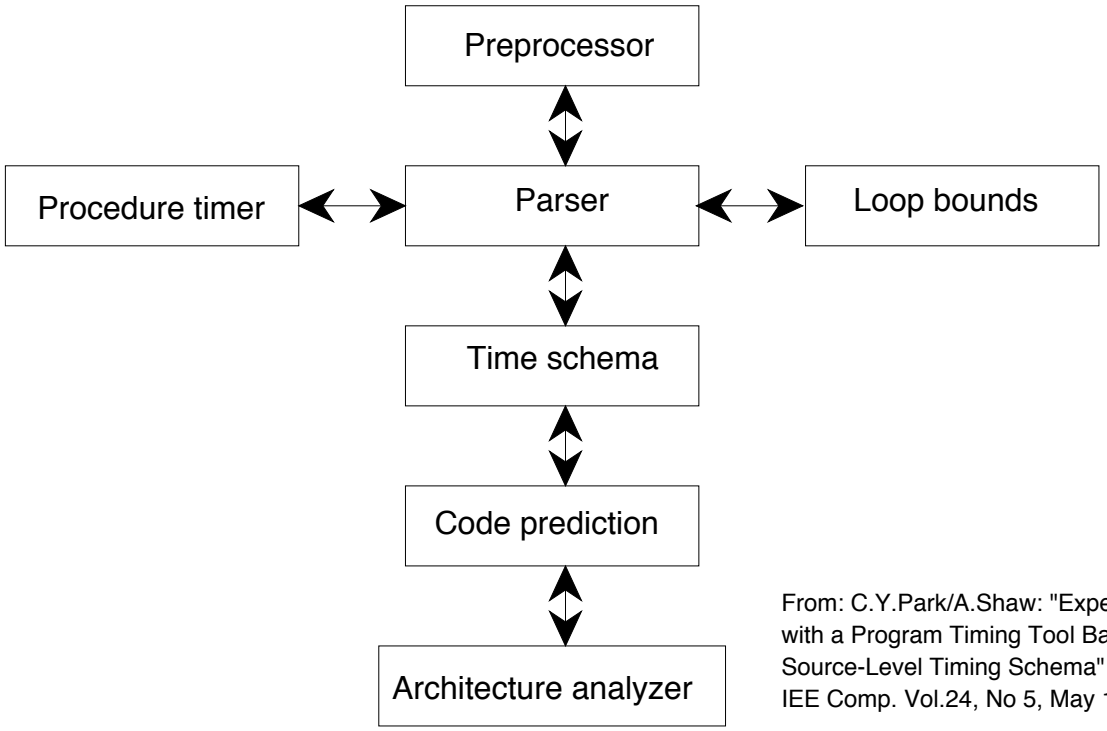
- brauchen obere/untere Grenzen für Schleifendurchläufe

```
L9: if B1 then S1;
    else if B2 then S2;
    else if B3 then S3;
    else S4;
    end if;
```

- Abarbeitungszeit abhängig von B1, B2, B3, B4
- B1 false, B2 true:

$$T_{exec} = T(B1) + T(B2) + T(S2) + T(JMP);$$

- Abschätzunge wesentlich komplexer wenn Interrupts zugelassen werden.



From: C.Y.Park/A.Shaw: "Experiments with a Program Timing Tool Based on Source-Level Timing Schema"
IEE Comp. Vol.24, No 5, May 1991

Parser: analyzes input of C program, statement by statement

Procedure timer: manages a table of procedure names and their execution times

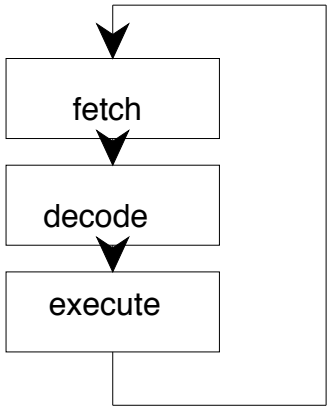
Loop bounds: interacts with the user to obtain loop bounds

Time schema: analyzes a statement into atomic blocks and computes execution time of the statement using the times of the atomic blocks computed incode prediction.

Code prediction: predicts the exact code for a given atomic block by looking up the assembly code prepared by the preprocessor and computes the time of the block using instruction execution times provided by the architecture analyzer

Berücksichtigung von Pipelining

- von Neumann-Modell:



hier: Berechnung von Ausführungszeiten schwierig, aber möglich.

- moderne Rechner benutzen *pipelining*
- Abhängigkeiten zwischen simultan verarbeiteten Instruktionen:

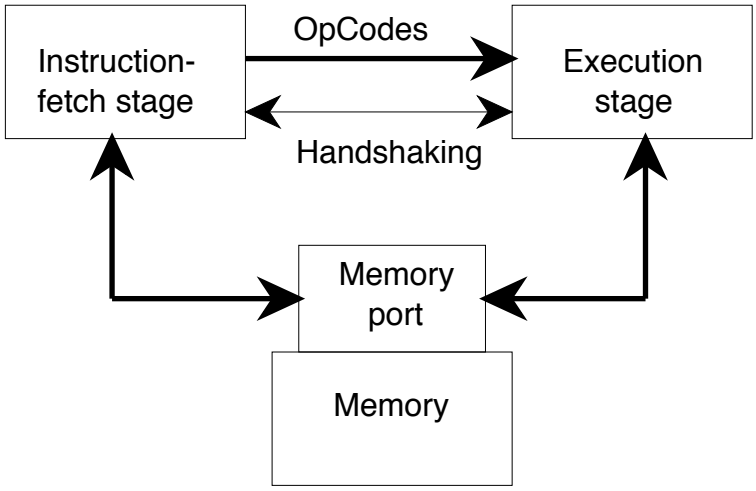
    - Instruktion I_i benötigt Ergebnis von I_j
    - bedingte Sprünge
    - Interrupts

- Sprünge:

    - Stop *prefetch*-Op. bis Bedingung ausgewertet ist (Performance degradiert)
    - Raten, welcher Zweig genommen wird; entsprechende *prefetch*-Op. ausführen

Zweistufige Pipeline

- erste Stufe lädt Instruktionen in *prefetch*-Puffer
- zweite Stufe behandelt alles andere, inklusive Operanden
- beide Stufen müssen auf Hauptspeicher zugreifen
- *handshaking, 1 cycle delay*, wenn zweite Stufe auf Speicher zugreifen muß
- zweite Stufe hat höhere Priorität; non-präemptiv



- es wird kein Cache benutzt
- alle Variablen residieren im Hauptspeicher
- keine page faults
- keine *preemption/ interrupts*
- *Execute*-Stufe ist keine *pipeline*

- Analyse läuft auf mühsame Betrachtung aller Fälle hinaus.
- wir beschränken uns hier auf einfachen, unverzweigten Code.

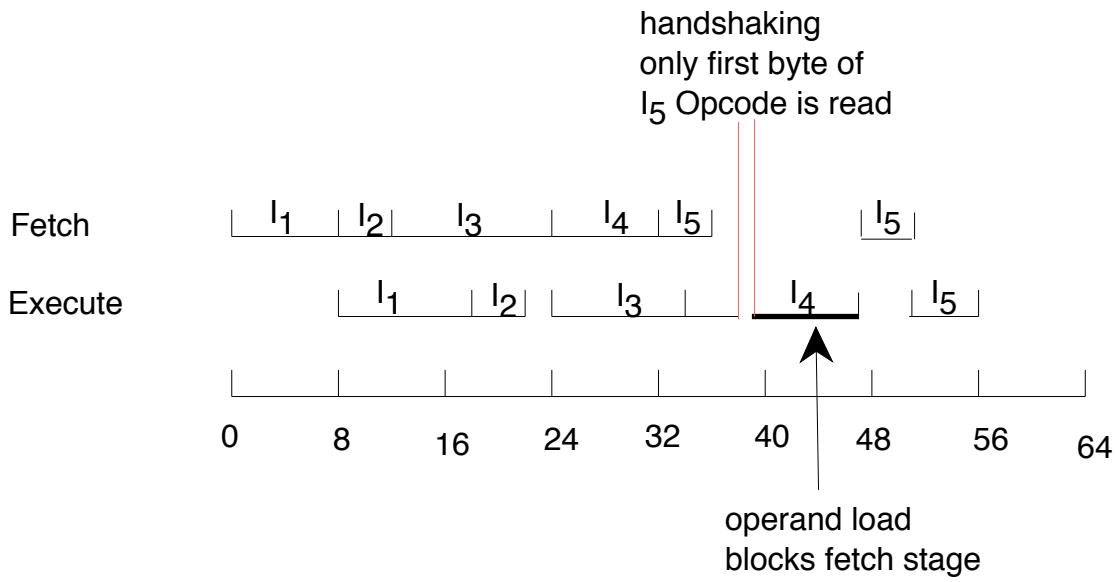
Sei nun:

n_i - Ausführungszeit für I_i ohne Speicherzugriffe
v_i - Größe des OpCodes in Bytes
r_i - Anzahl der Lesezugriffe für Operanden von I_i
w_i - Zahl der von I_i verlangten Schreibzugriffe

Ein Speicherzugriff benötige 4 Zyklen.

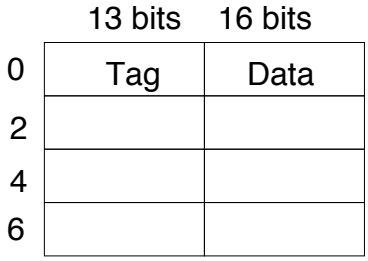
Instruktion	n_i	v_i	r_i	w_i
I_1	10	2	0	0
I_2	4	1	0	0
I_3	10	3	0	0
I_4	2	2	2	2
I_5	5	2	0	0

- Abarbeitung aller Instruktionen aus unserem Beispiel okkupiert den Zeitabschnitt [0,56].

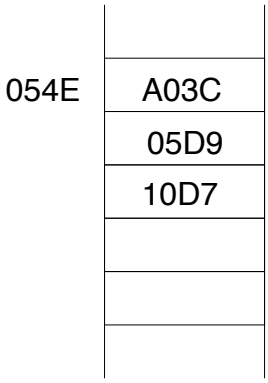


Caches

- caches erschweren Abschätzungen für Ausführungszeit
- Vorhersage von cache-Inhalten ist schwierig



direct-mapped cache



main memory contents

16 bits operands/instructions
LOOP starts at 02EC

```
LOOP    Add     (R1)+, R0
        Dec     R2
        BNE     LOOP
```

initial: R0 <- 0
R1 <- 054E
R3 <- 3

after pass 1

005E	BNE
005D	Add
005D	Dec

after pass 2

005E	BNE
005D	Add
005D	Dec

after pass 3

005E	BNE
00AA	10D7
005D	Add
005D	Dec

02EC = 0000 0010 1110 1100 (Add)

5 — tag — D 4 — cache line

054E = 0000 0101 0100 1110 (Operand)

02EE = 0000 0010 1110 1110 (Dec - overwrites Op in line 6)

- Wieviele Hauptspeicherzugriffe in diesem Beispiel?
- Ausführungszeit der Speicherzugriffe, wenn
  - HS-Zugriff: 200ns
  - cache-Zugriff: 30ns dauert?

schwierig:

- bedingte Sprünge
- Präemption
  -> Idee: Unterteilung des cache nach Tasks

- Strategic Memory Allocation for Real-Time (SMART) cache
  - exklusive Partitionen
  - *shared*-Bereich
- kritische Tasks benutzen nur exklusive Partitionen

Wieviele Partitionen sollten einer Task zugewiesen werden?
- NP-hartes diskretes Optimierungsproblem
- greedy-Algorithmus, der Tasks bewertet nach
  - ihrer Laufzeit (unter Benutzung von k cache-Partitionen) und
  - ihrer Ausführungsfrequenz
- cache-Partitionen werden nach Wertigkeit vergeben

Virtual Memory

- Benutzung ist zu vermeiden!
- Abschätzungen für die Behandlung von page-faults sind kaum möglich.