Echtzeitsysteme

Exkurs: Nicht-blockierende Synchronisation & WCET-Analyse

Lehrstuhl Informatik 4

26. Januar 2012

Gliederung

Fragestellungen

- Blockierende Synchronisation ist problembehaftet!
  - Nichtblockierende Synchronisation bietet einen Ausweg
    - Verzicht auf die Aussperrung anderer Jobs
    - nicht alle Betriebsmittel sind nichtblockierend synchronisierbar
  - grober Überblick über
    - grundlegende Funktionsweise nichtblockierender Synchronisation
    - und ihr Zusammenspiel mit dem Thema Rechtzeitigkeit
  - Auszug aus der Vorlesung „Betriebssystemtechnik“ [7, Kapitel VII-2]
- Alle sprechen von der WCET – aber wo kommt sie eigentlich her?
  - Wie geht man mit der Abhängigkeit von Eingabedaten um?
  - Welche Rolle spielt der Prozessor beim Thema WCET?

Gliederung

Wiederholung: Synchronisation Considered Harmful

Verklemmungen, Prioritätsumkehr, Prioritätsverletzung (s. Folie VII/12 ff.)

Blockierende Synchronisation ist Voraussetzung vieler Probleme und Laufzeitanomalien bei mehrseitiger Synchronisation

Verklemmungen $\rightsquigarrow$ unmöglich ohne „hold & wait“
- ohne Blockade entstehen keine Verklemmungen (engl. *deadlocks*)

$\rightsquigarrow$ Verklemmungsvorbeugung [6, VIII-60]

Prioritätsumkehr $\rightsquigarrow$ unmöglich, falls keine Blockade stattfindet
- erfordert die Ausführung eines niedrig priorisierten Jobs anstelle eines ausführungsbereiten, höher priorisierten Jobs

$\rightsquigarrow$ der nieder-priore Job blockiert den hoch-prioren Job

Prioritätsverletzung $\rightsquigarrow$ oft Folge von blockierender Synchronisation
- der Betriebsmittelvergabe zuzuführende, blockierte Jobs werden in einer ungeeignet Warteschlange verwaltet

☞ kritisches Element ist die Blockierung

$\rightsquigarrow$ nichtblockierende Synchronisation würde diese Probleme lösen

Grundidee

hardware-basierte Implementierung kritischer Abschnitte

Koordinierung sich gegebenenfalls überlappender Aktivitäten, ohne gleichzeitige Jobs auszusperren
- parallele und pseudoparallele Programmausführung erlauben

$\rightsquigarrow$ Sperren auf spezialisierte Instruktionen des Prozessors begrenzen

CISC $\mapsto$ TAS, FAA, CAS oder CMPXCHG
RISC $\mapsto$ LL/SC
sonst $\mapsto$ Befehle zum Lesen/Schreiben von Speicherwörtern

☞ nur diese Instruktionen werden atomar ausgeführt
- nicht die Abfolge von Instruktionen ist unteilbar
- auf Mehrkern- oder auch auf Mehrprozessorebene
- abseits dieser Instruktionen ist eine verschränkte Ausführung erlaubt und erwünscht

Beispiel: CAS - Compare and Swap

Atomare, bedingte Wertzuweisung

CAS als Maschinenprogramm (s. Folie [7, VII-2/8]

```
int cas(atom_t *ref, word_t exp, word_t val) {
    bool done;

    LOCK(&ref->lock);
    if (done = (ref->data == exp)) ref->data = val;
    UNLOCK(&ref->lock);

    return done;
}
```

```
typedef struct {
    word_t data;
    lock_t lock;
} atom_t;
```

`ref` die Adresse des atomar, bedingt zu ändernden Speicherworts

`exp` der unter der Adresse `ref` erwartete alte Wert

`val` der unter der Adresse `ref` zu speichernde neue Wert
- die Speicherung erfolgt nur, wenn der unter `ref` gespeicherte Wert dem erwarteten Wert `exp` gleicht
- lag Gleichheit vor, liefert die Funktion *true*, anderenfalls *false*

Beispiel: CAS - Compare and Swap (Forts.)

Elementaroperation der Befehlssatzebene:
- unteilbar: Unterbrechungs- *und* Zugriffssperre (Datenbus)
- implementiert eine Transaktion für Uni- und Multiprozessoren
- gebräuchlich als Einzel- (CAS) und Doppelwortvariante (DCAS)
- eingeführt mit IBM System/370 [3, S. 123]

Ausführung mit einem atomaren „*read-modify-write*“-Zyklus:
1. Lesen eines Datums aus dem Arbeitsspeicher
2. bedingte Modifikation des gelesenen Datums
3. bedingtes Zurückschreiben des Datums in den Arbeitsspeicher

Beachte: CAS lässt sich mittels Sperren nachbilden

Uniprozessoren	• Unterbrechungs- oder Verdrängungssperre
Multiprozessoren	• Umlaufsperre (engl. *spinlock*)

Bedingte Wertzuweisung: CAS als Spezialbefehl (x86)

`ZF = (eax == *ref) ? (*ref = val, true) : (eax = *ref, false)`

```
int cas(word_t *ref, word_t exp, word_t val) {
    unsigned char aux;

    __asm__ __volatile__(
        "lock\n\t"                      /* prefix next instruction */
        "cmpxchgl %2,%1\n\t"            /* (ref) == exp ? (ref) = val */
        "sete %0"                       /* extend ZF into aux */
        : "=q" (aux), "=m" (*ref)
        : "r" (val), "m" (*ref), "a" (exp)  /* %eax loaded with exp */
        : "memory");

    return aux;
}
```

- `lock` — setzt die Bussperre für den nachfolgenden Befehl
  - zwingend für Multi(kern)prozessorsysteme, optional sonst
- `cmpxchgl` — *compare & exchange*: ZF = *true* → Speicherung erfolgt
- `sete` — definiert Variable `done` mit dem Wert der ZF-Flagge

Nichtblockierende Synchronisation mit CAS

erledige nichtblockierende Synchronisation mit CAS:
  wiederhole
    lokale Kopie = lese(&globale Variable);
    lokaler Wert = berechne(lokale Kopie,...);
  solange CAS(&globale Variable,lokale Kopie,lokaler Wert) scheitert;
basta.

pros
- Probleme blockierender Synchronisation werden vermieden:
  - keine Verklemmungen, Prioritätsumkehr, Prioritätsverletzung
- Tolerierung beliebiger Überlappungsmuster
- Robustheit: keine hängenden Sperren bei Prozessabbrüchen
- in funktionaler Hinsicht wiederverwendbar und komponierbar

cons
- Gefahr von Aushungerung (engl. *starvation*)
- Wiederverwendung sequentieller Altsoftware unmöglich
- Entwicklung nebenläufiger Varianten im Regelfall nicht trivial

Stapel: *last in*, *first out* (LIFO)

Verwendung z.B. für stapelorientierte Fadeneinplanung [1], die Fäden nach LCFS (Abk. für (engl.) *last come, first served*) verarbeitet

`dos` — Abk. für (engl.) *devoid of synchronization*

```
void dos_push(chain_t *head, chain_t *item) {
    item->link = head->link;
    head->link = item;
}
```

```
chain_t *dos_pull(chain_t *head) {
    chain_t *node

    if ((node = head->link) != 0)
        head->link = node->link;

    return node;
}
```

push ⊨ Bereitstellung eines Fadens
pull ⊨ Auswahl eines Fadens

```
void dos_zero(chain_t *head) {
    head->link = 0;
}
```

Listenelement

```
typedef struct chain chain_t;

struct chain {
    chain_t *link;
};
```


Stapel: Nichtblockierend synchronisiert

Element durch ein wettlauftolerantes Maschinenprogramm am Kopf der Liste einfügen (*push*) und entnehmen (*pull*)

globale Variable Verkettungszeiger des Kopfes (`head->link`)

```
void lf_push(chain_t *head, chain_t *item) {
    do item->link = head->link;                 /* is elected head */
    while (!CAS(&head->link, item->link, item));  /* try push item */
}
```

lokale Kopie Verkettungszeiger des neuen Elements (`item->link`)
lokaler Wert wird jeweils nicht explizit berechnet

```
chain_t *lf_pull(chain_t *head) {
    chain_t *node;

    do if ((node = head->link) == 0) break;       /* access head */
    while (!CAS(&head->link, node, node->link));  /* try pull node */

    return node;
}
```

lokale Kopie das zurückzuliefernde Element (`node`)

Stapel: Plausibilitätskontrolle

- kritischer Datenbestand ist `head->link`, der Listenkopf

push # das durch `item` adressierte Element ist noch nicht in der Liste und es wird auch nicht mehrfach in diese Liste eingetragen
- der Listenkopf ist in `item->link` als Kopie vermerkt
- neuer Listenkopf wäre *item*, dessen Bindung CAS versucht
- do `...while` terminiert, falls `item` als Kopf gebunden wurde

pull
- der Listenkopf ist in `node` als (lokale) Kopie vermerkt
- CAS versucht den neuen Listenkopf `node->link` zu binden
- do `...while` terminiert, falls `node->link` gebunden wurde

Beachte
- auch wenn mehrere Fäden auf denselben Kopfzeiger gleichzeitig zugreifen, wird CAS für nur einen Faden die Manipulation zulassen
- je nach Nutzung von *push* und *pull* droht das „ABA-Problem“ [2]

Laufzeitverhalten nichtblockierender Synchronisation

Zwar spielt die Blockadezeit einer nichtblockierend synchronisierten Datenstrukturoperation kaum eine Rolle, aber die laufzeittechnische Einordnung des CAS-basierter Algorithmen ist schwierig (s. Folie IX/10):

- ⇝ Wie lange dauert es, die Operation erfolgreich abzuschließen?
- ⇝ Wieviele Schleifendurchläufe werden benötigt?
  - Wie viele Versuche benötigt man, bis die CAS-Operation glückt?
  - Jeder Versuch erfordert es, eine lokale Kopie zu ziehen und einen neuen lokalen Wert zu berechnen.
- ⇝ Wie lange dauert es, bis die CAS-Operation glückt?
  - Hängt vom Grad der Konkurrenz um die betreffende Speicherstelle ab.
  - Niedrig priorisierte Jobs benötigen u.U. sehr viele Versuche.

☞ Ist (und falls ja, in welchem Umfang) Fortschritt bei der Ausführung dieser Operation garantiert?

Fortschrittseigenschaften

Aussagen, inwiefern eine erfolgreiche CAS-Operation garantiert werden kann

Behinderungsfreiheit (engl. *obstruction-freedom*)
- isoliert ausgeführte Operationen erzielen Fortschritt
  - sie darf nicht von kollidierenden Operationen überlappt werden
  - ⇝ wechselseitige Konflikte können jeglichen Fortschritt verhindern

⇝ EZS: Ausschluss wechselseitiger Konflikte notwendig

Sperrfreiheit (engl. *lock-freedom*) impliziert Hinderungsfreiheit
- mindestens eine kollidierende Operation erzielt Fortschritt
  - ⇝ andere Operation können aushungern (engl. *starvation*)

⇝ EZS: Aushungern ist ein Problem
- ist aber prinzipiell konform z.B. zu Vorrangsteuerung

Wartefreiheit (engl. *wait-freedom*) impliziert Sperrfreiheit
- alle kollidierenden Operationen erzielen Fortschritt
  - häufig implementiert durch sog. Helfer-Schemata
  - ⇝ hoch-priore Jobs erledigen auch Operationen niedrig-priorer Jobs mit

⇝ EZS: Helfer-Schemata implizieren Prioritätsverletzungen
- ihr Einfluss lässt sich aber nach oben begrenzen

Dinge, die man beachten sollte!

Nichtblockierende Synchronisation: eine kleine Packungsbeilage

- Nichtblockierende Synchronisation ist nicht immer möglich!
  - Unteilbare, nur exklusiv belegbare Betriebsmittel kann man nur blockierend synchronisieren!
  - Hardware fällt häufig in diese Kategorie
    - Kommunikationsmedien, Busse, Aktoren, Sensoren, ...
    - eine explizite Zustandsverwaltung ermöglicht aber eine geteilte Nutzung solcher Betriebsmittel ⇝ z.B. der Prozessor
  - Datenstrukturen (genauer: Speicherstellen) eignen sich bestens für nichtblockierende Synchronisation
- Betrachtung der Rechtzeitigkeit ist eine globale Angelegenheit!
  - Fortschritt niedrig priorisierter Jobs kann durch hoch-priore Jobs zunichte gemacht werden ⇝ Gefahr der Aushungerung

Gliederung

Auf der Suche nach dem *e*

Die maximale Ausführungszeit ist eine unabkömmliche Information für die Ablaufplanung

Statische Ablaufplanung ordnet Jobs Zeitintervallen zu

⇝ diese Zeitintervalle müssen ausreichend groß sein

- mindestens so groß, wie die maximale Ausführungszeit e des Jobs

Planbarkeitsanalyse basiert auf Abschätzungen ...

- der maximalen CPU-Auslastung:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + \frac{b_i^{np}}{\min(D_i, p_i)} \leq 1 \quad ; i = 1, 2, \ldots, n$$

- der maximalen Antwortzeit:

$$\omega_i(t) = e_i + b_i^{np} + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k; 0 < t \leq p_i$$

die ohne die maximale Ausführungszeit e nicht auskommen

- selbst die Blockadezeit b^{np} ist eine maximale Ausführungszeit
- nämlich die des längsten kritischen Abschnitts

Warum ist es so schwierig, *e* zu bestimmen?

Anders: Wovon hängt die maximale Ausführungszeit eigentlich ab?

Beispiel: Bubblesort

```
void bubbleSort(int a[],int size) {
  int i,j;

  for(i = 0;i < (size-1);++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

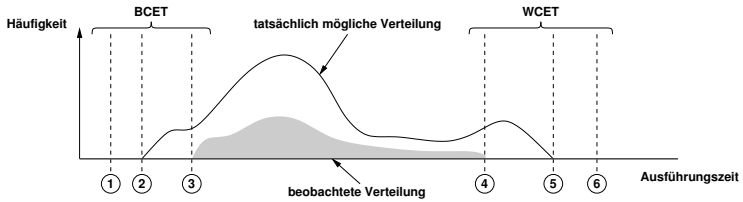
- sowohl die Größe als auch der Inhalt des Felds kann zur Laufzeit variieren

Die Maschinenprogrammebene liefert Dauer der Elementaroperationen:

- wie lange dauert ein `ADD`, ein `LOAD`, ...
- ist prozessorabhängig und für moderne Prozessoren sehr schwierig
  - Pipeline ⇝ Wie ist der Zustand der Pipeline an einer Instruktion?
  - Cache ⇝ Liegt die Instruktion/das Datum im schnellen Cache?
  - Out-of-Order-Execution, Branch-Prediction, Hyper-Threading, ...

Aufgabenstellung

Die maximale Ausführungszeit nach oben abschätzen



Bereits für relativ einfache Programme ergibt sich eine Bandbreite möglicher Programmlaufzeiten, besondere Bedeutung haben

- bestmögliche Ausführungszeiten (engl. **b***est* **c***ase* **e***xecution* **t***ime*)
  1. geschätzt, 2. tatsächlich und 3. beobachtet
- und maximale Ausführungszeiten (engl. **w***orst* **c***ase* **e***xecution* **t***ime*)
  4. beobachtet, 5. tatsächlich und 6. geschätzt

☞ Ziel ist eine sichere Abschätzung der WCET

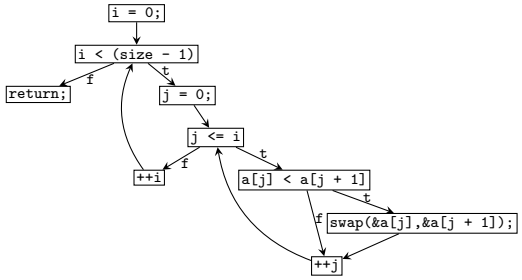
- und den Abstand zur tatsächlichen WCET klein zu halten

Den längsten Weg durch ein Programm finden

Die möglichen Wege lassen sich durch Kontrollflussgraphen beschreiben

Beispiel: Bubblesort

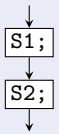
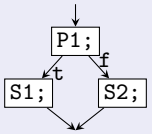
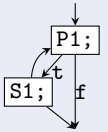
```
void bubbleSort(int a[],int size) {
  int i,j;

  for(i = 0;i < (size-1);++i) {
    for(j = 0;j <= i;++j) {
      if(a[j] > a[j+1]) {
        swap(&a[j],&a[j+1]);
      }
    }
  }

  return;
}
```



Ein Kontrollflussgraphen (engl. *control flow graph*) ist ein gerichteter Graph und setzt sich aus Grundblöcken (engl. *basic blocks*) zusammen

- Grundblöcke sind sequentielle „Code-Schnipsel“
  - hier wird gearbeitet $\rightsquigarrow$ Grundblöcke verbrauchen Rechenzeit
- Kanten im Kontrollflussgraphen $\rightsquigarrow$ Sprünge zwischen Grundblöcken

WCET entlang des Kontrollflussgraphen akkumulieren

Sequenz	Verzweigung	Schleife
		
$e_{S1} + e_{S2}$	$e_{P1} + \max(e_{S1}, e_{S2})$	$e_{P1} + n(e_{P1} + e_{S1})$

Kopfschmerzen bereiten allenfalls Schleifen – Schleifendurchläufe

- Anzahl der Schleifendurchläufe n bestimmt maßgeblich die WCET
- eine automatisierte Bestimmung gelingt in vielen Fällen nicht
  - Abhängigkeiten zu Benutzereingaben oder Sensorwerten
  - $\rightsquigarrow$ für zuverlässige und hinreichend präzise obere Schranken ist tiefgehendes Anwendungswissen erforderlich $\rightsquigarrow$ Annotationen
- verwundbarer Punkt im Hinblick auf Laufzeitüberschreitungen
  - $\rightsquigarrow$ Kompromittierung des Schleifenzählers durch einen Laufzeitfehler

Wie berechnet man den längsten Pfad?

. . . und bestimmt so auch gleicht die maximale Ausführungszeit

Lösungsansatz: Fasse die Bestimmung der WCET als Flussproblem auf [5]

- die Maximierung des Flusses in einem gerichteten Graphen ist ein gut untersuchtes Problem aus dem Bereich der Graphentheorie
- mithilfe ganzzahliger linearer Programmierung (engl. *integer linear programming*) lässt sich dieses Problem zudem effektiv lösen

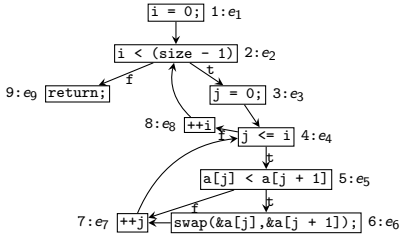
Vorgehen: Transformiere den Kontrollflussgraphen in ein lineares Optimierungsproblem und löse es

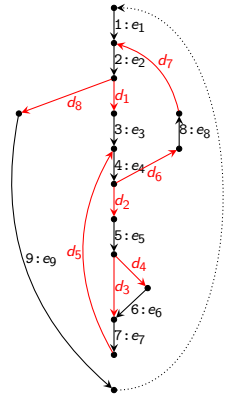
1. bestimme einen Zeitanalysegraph aus dem Kontrollflussgraphen
2. formuliere das lineare Optimierungsproblem
3. bestimme die Flussrestriktionen des Zeitanalysegraphen
   - dies sind die Nebenbedingungen im Optimierungsproblem
4. löse des Optimierungsproblem (z.B. mit lpsolve[1])

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

Der Zeitanalysegraph (engl. *timing analysis graph*)

- Grundblöcke werden auf Kanten abgebildet
  - Grundblöcke sind nummeriert mit WCET e_i
  - Pseudokanten d_i für Schleifen/Verzweigungen
  - eine Rückwärtskante von der Quelle zur Senke





- trägt die Kante i den Fluss f_i, wird sie f_i-mal durchlaufen
  - Ziel ist die Maximierung des gewichteten Flusses: $\sum_{i \in E} f_i e_i$
  - E ist die Menge der Kanten im Zeitanalysegraphen
  - Pseudokanten und die Rückwärtskante haben kein Gewicht: $e = 0$

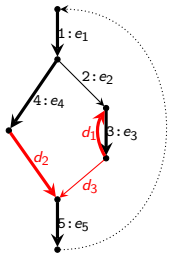
Zirkulationen und Flussrestriktionen

Eine Abbildung $f : E \mapsto \mathcal{R}$ heißt Zirkulation, falls sie den Fluss erhält

- sie ordnet jeder Kante einen Fluss f_i zu
- der eingehende Fluss eines Knotens ist gleich dem ausgehenden Fluss

Flussrestriktionen schließen Zirkulationen ungültiger Abarbeitungen aus

- Formulierung als Nebenbedingungen des Optimierungsproblems
- Beschränkung der maximalen Anzahl von Schleifendurchläufen



- $f_1 = f_2 + f_4$ wird durch die Zirkulation garantiert
- gültige Zirkulation: $\{1, 4, d_2, 5\} \cup \{3, d_1\} \cup \{\text{ret}\}$
  - $\rightsquigarrow$ aber keine gültige Abarbeitung
  - ret ist die Rückwärtskante
- Flussrestriktion $f_3 \leq 5 f_2$ löst dieses Problem
  - wird 2 nicht abgearbeitet, so gilt $f_3 \leq 5 \cdot 0 = 0$
  - hier: Beschränkung auf 5 Schleifendurchläufe
  - $\rightsquigarrow$ Nebenbedingung des Optimierungsproblems

Wie lange dauern die „sequentiellen Code-Schnipsel“

Die WCETs e_i der einzelnen Grundblöcke ist Eingabe für die Flussanalyse

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 [5]

- Ergebnis: $e_{\texttt{\_getopt}} = 80$ Zyklen
- Annahmen:
  - obere Schranke für jede Instruktion
  - die obere Schranke der Sequenz bestimmt man durch Summation

Problem: Vorgehen ist 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, ... haben großen Anteil an der verfügbaren Rechenleistung heutiger Prozessoren
- die WCET jeder einzelnen Instruktion als Grundlage zu verwenden ignoriert diese Maßnahmen

Beispiel: Cache-Analyse [8, Kapitel 22]

Cache: ein kleiner, schneller Zwischenspeicher, Zugriffszeiten auf Daten/Instruktionen variieren je nach Zustand des Caches enorm:

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

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

Hits sind schneller als *Misses*: $e_m \gg e_h$ (> 100 Taktzyklen möglich)

Hilfreich ist zu wissen, ob z.B. eine Instruktion im Cache ist, oder nicht:

must, die Instruktion ist garantiert im Cache

- $\rightsquigarrow$ man kann immer die schnellere Ausführungszeit e_h annehmen
  - wird für die Vorhersage von Treffern verwendet

may, die Instruktion ist vielleicht im Cache

- $\rightsquigarrow$ 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

- $\rightsquigarrow$ erster Zugriff ist ein Fehlschlag, alle weiteren sind Treffer
- $\rightsquigarrow$ erster Zugriff: e_m, weitere Zugriffe: e_h

Beispiel: LRU-Cache, 4-fach assoziativ

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

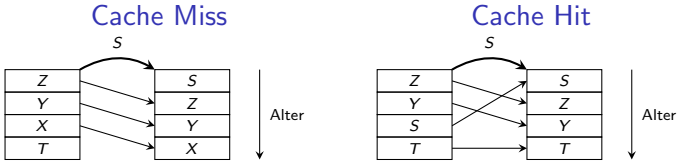
Caches werden häufig in Sätze (engl. *cache set*) unterteilt

- ein *n*-fach assoziativer Cache besitzt pro Satz *n* Cache-Blöcke

$\rightsquigarrow$ 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 Cache



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

must Obergrenze des Alters $\rightsquigarrow$ Unterapproximation des Inhalts

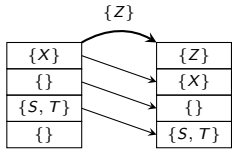
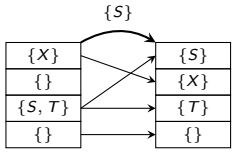
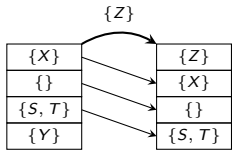
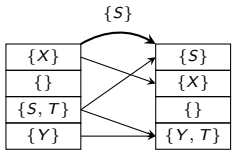
- Obergrenze $\leq$ Assoziativität $\rightsquigarrow$ Element ist garantiert im Cache

may Untergrenze des Alters $\rightsquigarrow$ Überapproximation des Inhalts

- Untergrenze > Assoziativität $\rightsquigarrow$ Element ist garantiert nicht im Cache

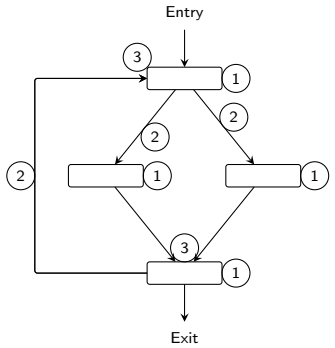
Beispiel: LRU-Cache, Zugriff auf eine Speicherstelle

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

	Potential Cache Miss	Definitive Cache Hit
must-Approximation		
	Definitive Cache Miss	**Potential Cache Hit**
may-Approximation		

Wie funktioniert nun die Cache-Analyse?

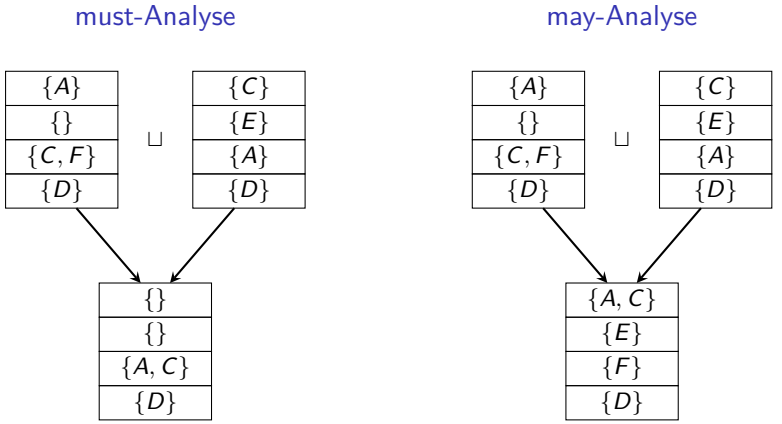
Die Analyse ist eine Datenflussanalysen [4, Kapitel 8]



1. sammle Information in den Grundblöcken
   - Speicherzugriffe (s. Folie X/29)
   - 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

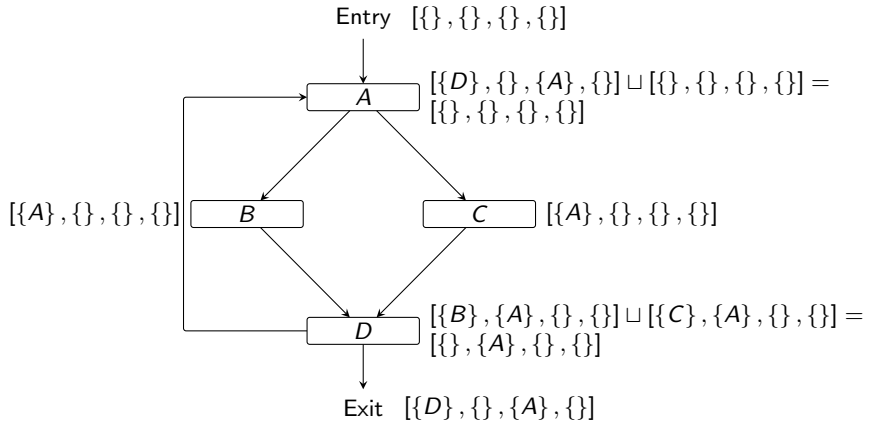
☞ Verschmelzungsoperatoren für must- und may-Analyse

Verschmelzungsoperatoren für must- und may-Analyse



„Schnittmenge + max. Alter“ „Vereinigungsmenge + min. Alter“

Beispiel: must-Analyse für LRU



☞ Hier ist leider keine Vorhersage von Treffern möglich ☹

⇝ Tip: ein einfaches, virtuelles Ausrollen der Schleife hilft weiter!

Gliederung

Resümee

Blockierende Synchronisation ist problembehaftet
- Verklemmungen, Prioritätsumkehr, Prioritätsverletzung

⇝ aber manchmal auch unumgänglich
- unteilbare, nur exklusiv belegbare Betriebsmittel

Nichtblockierende Synchronisation kann diese Probleme lösen
- unteilbare Spezialinstruktionen statt atomarer Befehlssequenzen

⇝ Prozessorunterstützung erforderlich ↦ TAS, CAS, FAA, LL/SC

Bauschema für gewisse nichtblockierende Algorithmen
- Aufbau als Transaktion ⇝ lesen, manipulieren, commit (= CAS)
  - leider klappt der Abschluss der Transaktion nicht immer gleich ☹
- Fortschrittsgarantien: Behinderungs-, Sperr- und Wartefreiheit

Bestimmung der WCET durch statische Analyse
- Ziel: sichere Abschätzung nach oben ohne allzu viel Pessimismus
- Flussanalyse finde die längsten Pfade durch ein Programm
- Hardwareanalyse bestimmt die WCET einzelner Grundblöcke
  - kämpft mit dem Prozessor: Pipeline, Cache, Branch-Prediction, ...

Literaturverzeichnis

[1] Baker, T. P.:
Stack-Based Scheduling of Realtime Processes.
In: *Real-Time Systems* 3 (1991), Nr. 1, S. 67–99

[2] Herlihy, M. :
Wait-Free Synchronization.
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr. 1, S. 124–149

[3] IBM Corporation (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
White Plains, NY, USA: IBM Corporation, Sept. 1 1975. –
GA22-7000-4, File No. S/370-01

[4] Muchnick, S. S.:
Advanced compiler design and implementation.
San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1997. –
ISBN 1–55860–320–4

Literaturverzeichnis (Forts.)

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

[6] Schröder-Preikschat, W. :
Softwaresysteme 1.
www4.informatik.uni-erlangen.de/Lehre/SS07/V_SOS1, 2007. –
Lecture Notes

[7] Schröder-Preikschat, W. :
Betriebssystemtechnik.
http://www4.informatik.uni-erlangen.de/Lehre/SS11/V_BST, 2011. –
Lecture Notes

[8] Wilhelm, R. :
Embedded Systems.
http:
//react.cs.uni-sb.de/teaching/embedded-systems-10-11/lecture-notes.html,
2010. –
Lecture Notes