



A. Koch

Eingebettete Prozessorarchitekturen

3. Compilierung für VLIW-Prozessoren

Andreas Koch

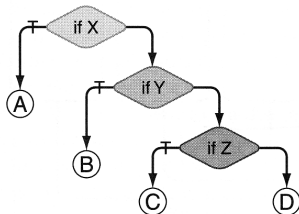
FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Wintersemester 2007/2008

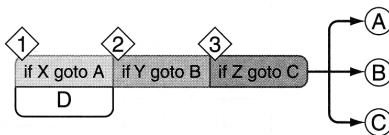
Mehrwege-Sprünge



CONTROL-FLOW GRAPH



SCHEDULED MULTIWAY BRANCH



A. Koch

- Auch bedingte Sprünge parallel ausführbar
- Anforderung: Priorisierte Abarbeitungsfolge in Instruktionwort
 - Entsprechend der ursprünglichen sequentiellen Abarbeitungsfolge
- Sinnvoll?
 - Durchschnittlich alle 5-7 Instruktionen ein Sprung
 - **Maximaler** ILP also nur 5-7 ohne Mehrwege-Sprünge



- Führe Operationen aus, bevor sie gebraucht werden
- Realisierung: Verschiebe Operationen **vor** Sprungbedingung
- Dürfen bei Fehlspekulation keinen Effekt haben, z.B.
 - Dürfen keine Register überschreiben, deren Daten später noch benötigt werden
 - STOREs häufig nicht spekulativ ausführbar
- Problem: Ausnahmebehandlung (Exceptions)
 - Zugriff auf ungültige Speicheradresse, Division durch 0, etc.
 - Ein Ansatz: **Korrekte** Programme dürfen kein anderes Verhalten zeigen
 - Bei spekulativer Ausführung Ausnahmen **unterdrücken**
 - Wären in korrektem Programm ja nicht aufgetreten
 - Ansatz von Multiflow, auch in Lx vorhanden

Kontrollspekulation

Beispiel



A. Koch

Ohne Spekulation

```
br $b0, L1
;;
add $r5, $r10, $r11
add $r6, $r7, $r8
;;
stw 0[$r5], $r6
;;
L1:
```

3 Takte, 1.33 Operationen
pro Takt

Mit Spekulation

```
add $r5, $r10, $r11
add $r6, $r7, $r8
br $b0, L1
;;
stw 0[$r5], $r6
;;
L1:
```

2 Takte, 2.0 Operationen pro
Takt

Kontrollspekulation

Beispiel für Auftreten von Ausnahmen



Annahme: 1 Takt zwischen CMP und BR, 2 Takte für LD

A. Koch

```
# if (p != 0) *p += 2
cmpeq $b0 = $r5, 0
xnop 1
;;
br    $b0, L1
;;
ldw   $r1, 0[$r5]
xnop 1
;;
add   $r1, $r1, 2
;;
stw   0[$r5], $r1
;;
L1:
```

Ohne Spekulation

- LOAD kann nicht vorgezogen werden
- Sonst bei Zugriff auf Adresse 0: Segmentation Violation

➡ Programmabbruch

7 Takte, 0.7 Operationen pro Takt

Kontrollspekulation

Beispiel für Unterdrücken von Ausnahmen mit *Silent Instructions*



Annahme: 1 Takt zwischen CMP und BR, 2 Takte für LD

A. Koch

```
# if (p != 0) *p += 2
cmpeq $b0, $r5, 0
ldw.d $r1 = 0[$r5]
xnop 1
;;
add $r1, $r1, 2
br $b0, L1
;;
stw 0[$r5], $r1
;;
L1:
```

4 Takte, 1.25 Operationen
pro Takt

Mit Spekulation

- ldw.d: *dismissible load*
- Löst bei illegaler Adresse **keine** Ausnahme aus
- Liefert aber unbestimmte Daten
- Damit vorziehbar

Datenspekulation



A. Koch

- Operation auch dann ausführen, wenn sie möglicherweise inkorrekt abläuft
- Fehler muss erkannt und korrigiert werden (hoffentlich nur selten!)
- Betrifft im wesentlichen Reihenfolge von LOADs und STOREs

Ohne Datenspekulation

```
mpy $r1, $r2, $r3
stw 8[$r7] = $r1
ldw $r4 = 16[$r5]
add $r6 = $r4, 1
```

Mit Datenspekulation

```
ldw $r4 = 16[$r5]
mpy $r1, $r2, $r3
xnop 1;;
stw 8[$r7] = $r1
add $r6 = $r4, 1
```

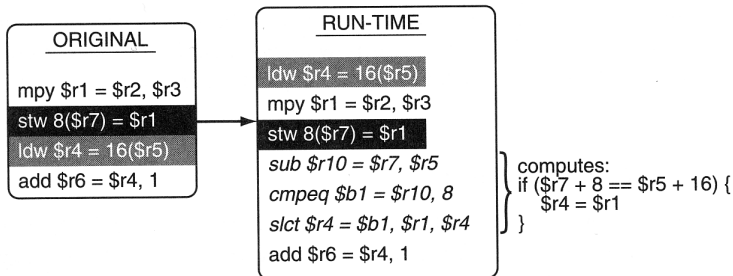
- Was bei `8[$r7] = 15[$r5]` ?
- Liest **veralteten** Wert!

Datenspekulation

Umgehung mit Software



A. Koch



- Sehr zeitaufwendig
- Datenspekulation nur sinnvoll mit Hardware-Unterstützung
 - Intel Itanium *Advanced Load Table* (ALAT)
 - LSQ, LSIDs, Multi-Value-Cache, ...
- Teilweise auch zur Compile-Zeit möglich
 - *static memory disambiguation*



Basisblock (BB)

Längste Folge von Anweisungen ohne Kontrollfluß.

Beispiel:

```
a := b + 42;  
if (a > 23) then  
  c := a - 46;  
  d := b * 15;  
else  
  c := a + 46;  
  d := 0  
  q := false;  
endif
```

Basisblöcke:

```
a := b + 42;
```

```
c := a - 46;  
d := b * 15;
```

```
c := a + 46;  
d := 0  
q := false;
```

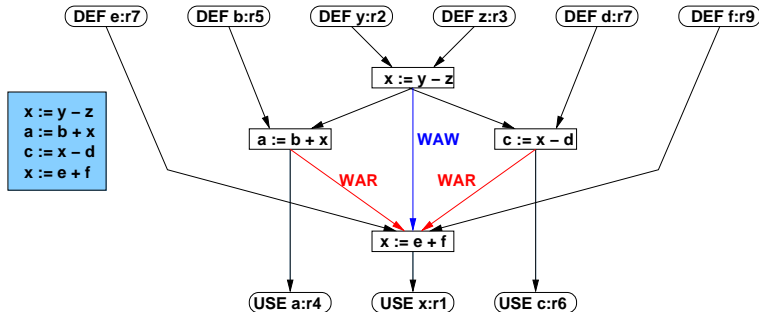
Datenabhängigkeitsgraph

(*data dependence graph*) Ablaufplanung innerhalb von Basisblöcken



- Betrachte jeden Basisblock einzeln
- Knoten sind primitive Operationen (Assembler-Ebene)
- Kanten für die drei Arten von Datenabhängigkeiten (RAW, WAR, WAW)
- Kantengewichte größer 1 können längere Rechenzeiten darstellen (mehr als einen Takt)

A. Koch

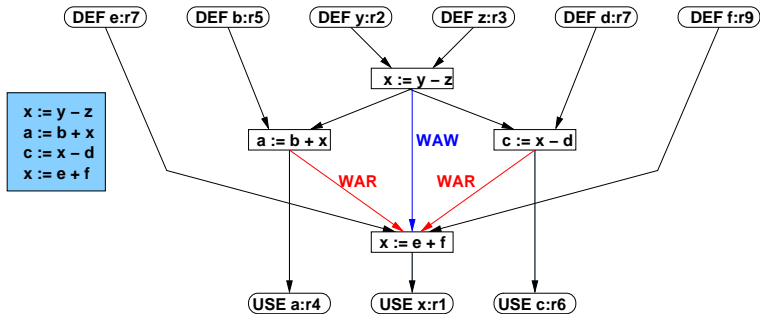


Datenabhängigkeitsgraph

Kanten geben zeitliche Reihenfolge vor



A. Koch



Möglicher Ablaufplan des Assembler-Codes (unoptimiert)

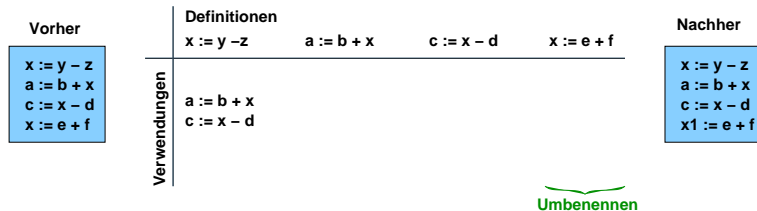
```
sub $r1 = $r2, $r3
;;
add $r4 = $r5, $r1
sub $r6 = $r1, $r8
;;
add $r1 = $r7, $r9
# 3 Takte
```

Transformationen auf Datenabhängigkeitsgraph

Umbenennung zum Auflösen von WAW und WAR-Abhängigkeiten



A. Koch



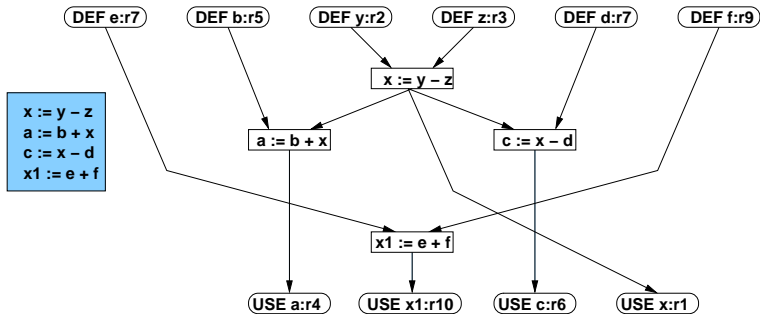
- Zuordnung von Definitionen einer Variable/Register zu deren Lesern
- Innerhalb einer Spalte kann eine Variable umbenannt werden
- Hier: Ein x zu $x1$ umbenennen

Datenabhängigkeitsgraph

Neue Ablaufplanung



A. Koch



Nun möglicher Ablaufplan des Assembler-Codes

```
sub $r1 = $r2, $r3
add $r10 = $r7, $r9
;;
add $r4 = $r5, $r1
sub $r6 = $r1, $r8
;; # 2 Takte
```

Einschränkungen



A. Koch

- Problem: I.d.R. nur 5-7 Anweisungen in Basisblock
- Wenig Spielraum für Parallelisierung
- Auch bei Auflösung von WAR und WAW-Abhängigkeiten

➡ Idee: Größere Bereiche bearbeiten

Darstellung von ganzen Prozeduren

Zunächst: Durch Kontrollflußgraphen (*control flow graph, CFG*)



A. Koch

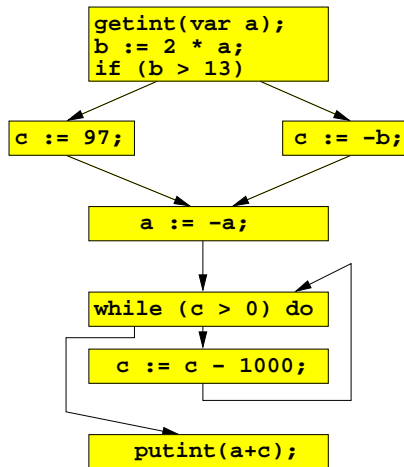
- Beschreibt Kontrollfluß (Verzweigungen) zwischen Basisblöcken
 - Knoten sind Basisblöcke
 - Kanten sind Sprünge zwischen Blöcken
- Am Ende jedes Basisblocks steht nun **genau eine** Verzweigung
 - Unbedingter Sprung
 - Bedingter Sprung mit einem oder mehreren Sprungzielen
 - Sprungziel ist immer ein Blockanfang

Beispiel Kontrollflußgraph 1



A. Koch

```
getint(var a);  
b := 2 * a;  
if (b > 13) then  
  c := 97;  
else  
  c := -b;  
a := -a;  
while (c > 0) do  
  c := c - 1000;  
putint(a+c);
```



Idee: Finde Regionen von Basisblöcken



A. Koch

- Bearbeite diese dann zusammen
- Chance auf mehr parallele Operationen
- Welche Basisblöcke nehmen?
- Gängiger Ansatz: Wahrscheinlich aufeinanderfolgende
- Motivation: Konzentriere Optimierung/Parallelisierung auf **wahrscheinlichsten Weg** durch Prozedur/Programm

Profiling

Ohne Programmausführung



A. Koch

- Woher Daten über Ausführungswahrscheinlichkeiten bekommen?
- Manuelle Annotation
 - Programmierer charakterisiert jede Verzweigung manuell
- Schätzungen anhand Programmtext
 - “Jede Schleife wird 100x durchlaufen”
 - Bulldog (1985, Prä-Multiflow Compiler)
 - Geht heute etwas genauer (statisches Profiling)

Profiling

Messungen am laufenden Programm



A. Koch

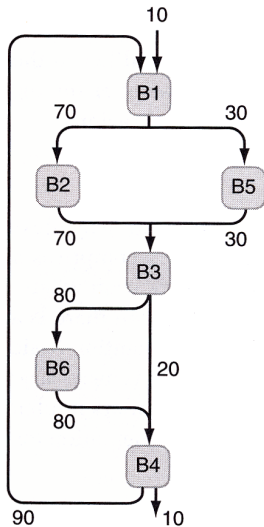
- Instrumentierung: Programm wird automatisch um Meßpunkte erweitert
- Dann Ausführen des instrumentierten Programms
- Schreibt bei Erreichen eines Meßpunktes Profiling-Daten in Datei
- Anschliessend Daten auswerten
- Problem: Auswahl der Eingabedaten beeinflusst Aussagekraft
 - Bei eingebetteten Systemen nicht so kritisch
 - Eng begrenztes Aufgabengebiet, repräsentative Eingabedaten

Annotierter CFG

Ausführungszahlen $\text{count}((v, w))$ an Kanten: *point profile*



A. Koch



Spuren (Traces)

Weg durch annotierten CFG finden



A. Koch

Spur (engl. *trace*)

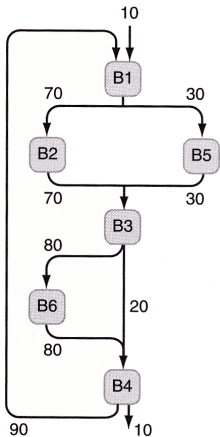
- **Zusammenhängende** Abfolge von Basisblöcken im CFG
- Kann **mehrere** Eingangs- und Ausgangskanten haben
- **Innerhalb** der Spur dürfen keine Zyklen existieren

- Vorwärtssprünge innerhalb der Spur sind erlaubt
- Die gesamte Spur darf Teil eines Zyklus sein
 - Z.B. wenn sie selbst der Schleifenkörper ist

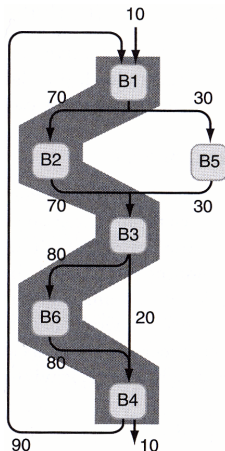
Beispielspur



Annotierter CFG



Eine Spur





Ab hier auch Material aus

Bulldog: A Compiler for VLIW Architectures

von

John R. Ellis

Konstruktion von Spuren

Wahrscheinlichste Wege durch annotierten CFG finden



A. Koch

- 1 Finde einen Startknoten v mit größter Ausführungsanzahl (Summe der eingehenden Kanten)
- 2 Suche vom Ende v der Spur nach Kante (v, w) mit
 - w ist noch nicht Teil einer Spur **und**
 - $\text{count}((v, w))$ ist maximal für alle Kanten (v, x) **und**
 - $\text{count}((v, w))$ ist maximal für alle Kanten (x, w)
- 3 Wenn solche Kante gefunden, nimm Block w als neues Ende zur Spur hinzu
- 4 Wenn nicht, suche vom Anfang der Spur rückwärts

Konstruktion von Spuren

Diskussion



A. Koch

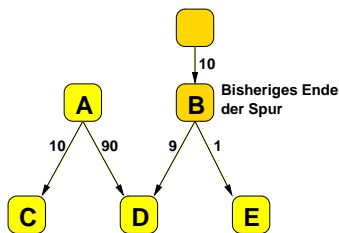
- Wenn keine Kanten mehr dazukommen (weder vorwärts noch rückwärts) ...
- ... diese Spur schliessen
- Nach Startknoten für nächste Spur suchen
- Effekt: Ganzer CFG wird mit (immer kleiner werdenden) Spuren überdeckt

Beispiel für Kantenauswahl

Idee: Gegenseitige größte Wahrscheinlichkeit von Knoten



A. Koch



- Annahme: Ende der Spur ist **B**, es wird vorwärts gesucht
- Aus Sicht von **B** ist **(B,D)** eine lohnende Kante
- Aus Sicht von **D** wäre allerdings **(A,D)** lohnender
- Es gibt also keine Kante mit **gegenseitiger** größter Wahrscheinlichkeit
- Die Spur endet mit **B**
- Nun rückwärts vom Anfang der Spur suchen

Trace Scheduling

Ablaufplanung auf Spuren-Ebene



A. Koch

- Nun jede Spur einzeln ablaufplanen und binden
- Beginnend bei erster (=wichtigster) Spur
- Erinnerung **Ablaufplanung** (Kanonik CMS)
 - Zeitliche Zuordnung jeder Operation an konkreten Zeitschritt
 - Unter Berücksichtigung
 - aller Abhängigkeiten
 - verfügbarer Ressourcen (Recheneinheiten)
- Erinnerung **Bindung** (Kanonik CMS)
 - Zuordnung jeder Operation an konkrete Ausführungseinheit
 - Wichtig insbesondere dann, wenn es unterschiedliche Einheiten gibt

Darstellung einer Spur



A. Koch

- Als Prozedurabhängigkeitsgraph (PDG)
 - *procedure dependence graph*
 - Manchmal auch *program dependence graph* genannt
- Erweiterung des Datenabhängigkeitsgraphen
- Nun zusätzliche Knoten für bedingte Sprünge
 - Haben Datenabhängigkeit zu ihrer Bedingung
- Zusätzliche Kanten für Kontrollabhängigkeiten
 - Von bedingten Sprüngen zu davon abhängigen Operationen
 - Bei fehlender Kontrollkante: spekulative Ausführung möglich
 - Beachte: Bei uns **nicht** erlaubt bei STOREs

Beispiel Trace-Scheduling

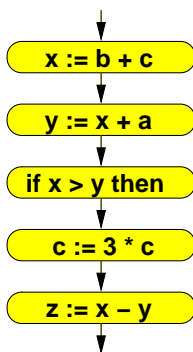
Vom Programm zur Spur



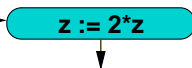
Programm

```
x := b + c
y := x + a
if x > y then
  c := 3 * c
  z := x - y
else
  z := 2 * z
...
endif
... := z
```

On-Trace



Off-Trace



- Am Ende der Spur ist hier nur noch **z** relevant
- Andere Variablen können vernachlässigt werden
- Der Zweig **z := 2 * z** ist unwahrscheinlicher
- ... und damit nicht auf der Spur

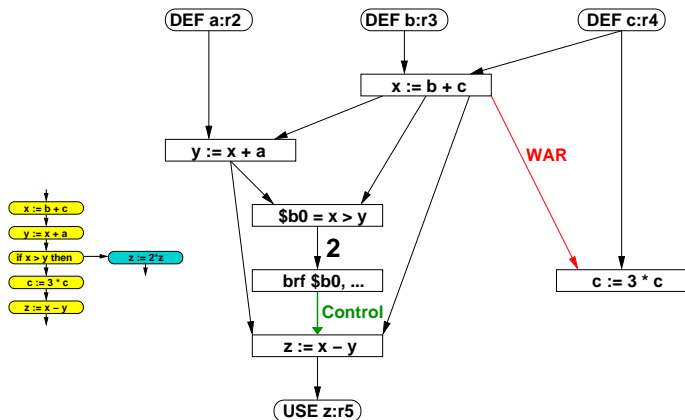
A. Koch

Beispiel Trace-Scheduling

Von der Spur zum PDG



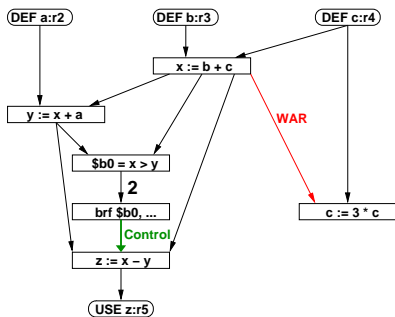
A. Koch



- **if/then/else** spalten in Vergleich/bedingten Sprung
- $z := x - y$ darf **nicht** vor Verzweigung gezogen werden
- Ist aber nicht datenabhängig: Kontrollkante einziehen!

Beispiel Trace-Scheduling

Vom PDG zum Ablaufplan (=VEX Programm)



```
add $r10 = $r3, $r4
mpy $r4 = $r4, 3
;;
add $r11 = $r10, $r2
;;
cmpgt $b0 = $r10, $r11
xnop 1
;;
brf $b0, ...
;;
sub $r5 = $r10, $r11
```

A. Koch

- Multiplikation spekulativ vorgezogen
- 2 Takte von Berechnung der Bedingung bis Sprung
- Kontrollkante verhindert Vorziehen von $z := x - y$
- Aufwendige Bindung bei VEX nicht erforderlich
 - Jeweils gleiche Recheneinheiten (ALU, MULT)



- w sei Operation im Programm **hinter** Verzweigung v
- w ist von v **kontrollabhängig**, wenn ...
 - ... die Zielvariable d von w auf dem anderen Zweig von v **gelesen** wird
 - ... **ohne** dass d dort vor dem Lesen redefiniert wird
 - Dann sicherstellen: w wird erst **nach** v ausgeführt
- Andere Terminologie
 - Manchmal auch Write-after-Conditional-Read genannt (WACR)
 - Compiler: d ist **live** auf anderem Ast der Verzweigung v

Diskussion spekulative Ausführung



A. Koch

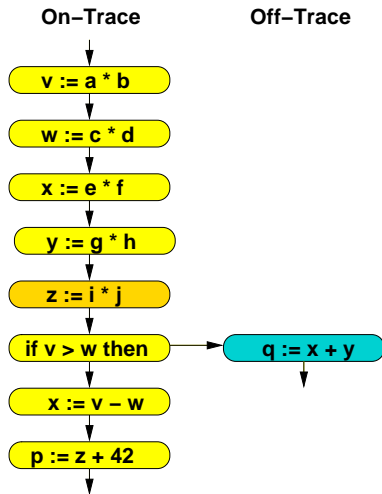
- Multiplikation hatte WAR-Abhängigkeit
- Wurde aber trotzdem parallel ausgeführt!
- Warum?
- VEX-Ausführungsmodell
 - Alle Operationen einer Instruktionen lesen zuerst Werte
 - Schreiben von Ergebnissen findet erst **danach** statt
- WAR auch bei paralleler Ausführung eingehalten!

Diskussion Verschiebung von Operationen



Annahme: Nur 1 schneller Multiplizierer mit 1 Takt Laufzeit

A. Koch



```
mpy    $r11 = $r1, $r2 #v
;;
mpy    $r12 = $r3, $r4 #w
;;
cmpgt  $b0 = $r11, $r12
mpy    $r13 = $r5, $r6 #x
;;
brf    $b0, ...
mpy    $r14, $r7, $r8 #y
;;
mpy    $r15, $r9, $r10 #z
sub    $r13, $r11, $r12#x
;;
add    $r17, $r15, 42 #p
```

Hier Verschiebung auch
hinter Verzweigung!

Stand der Dinge

Reicht das bisherige Vorgehen aus?



A. Koch

- Erreicht bisher
 - Jede Spur intern ablaufgeplant
 - Ggf. Operationen an Recheneinheiten gebunden
- Ablaufplanung kann **Reihenfolge** der Operationen ändern
 - Verschiebung relativ zum ursprünglichen Programm
 - Kann Einfluss auf **andere** Spuren haben
 - Hatten wir bisher vernachlässigt
 - Ausrede: Die Variablen werden dort nicht mehr gelesen
 - Ist aber in Realität komplizierter

➡ Schaden reparieren durch **Kompensationscode**



- Mehrwegesprünge und Spekulation
- Darstellung von Programmen durch
 - Datenabhängigkeitsgraph
 - Kontrollflußgraph
 - Prozedurabhängigkeitsgraph
- Profiling
- Spuren
- Trace-Scheduling
- Kompensationscode fehlt aber noch!