

# Optimierende Compiler

## Datenflussanalyse

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen  
Informatik, TU Darmstadt

Sommersemester 2010

# Organisatorisches

Ab jetzt auszugsweise Material aus

Advanced Compiler Design and Implementation

von Steven S. Muchnick, erschienen 1997 bei  
Morgan-Kaufman

# Copy Propagation

- Viele Algorithmen legen Zwischenvariablen an
  - $a := x + y; t1 = a;$
- Zwischenvariablen
  - benötigen viel Speicher, viele Register
  - verursachen viele Kopieranweisungen  $y := x$
  - sind in vielen Fällen unnötig

➔ Beseitigen durch

- 1 Copy Propagation (→ Muchnick 12.5)
- 2 Dead Code Elimination

# Idee Copy Propagation (CP)

A. Koch

- Versuche zur Compile-Zeit Aussagen über Laufzeitverhalten zu machen
  - “Simulation” des Programmes
- Falls möglich, benutze immer Originalvariable statt Kopie
  - Eingabe:  $a := x + y; b := x + y;$
  - Nach SSA/Red.Elim./AST:  $a := x + y; t1 := a; b := t1;$
  - Nach CP:  $a := x + y; t1 := a; b := a;$
  - Nach Dead Code-Elimination:  $a := x + y; b := a;$
- Vorgehen
  - Stelle fest, wenn Originalvariablen zwischen ihrer Berechnung ...
  - ... und Ihrer Verwendung **nicht** überschrieben werden

Lokale und globale Phasen

Speichere: Zuordnung von Originalvariablen  $w$  an Kopien  $v$   
für eine Zuweisung  $v := w$

A. Koch

Tupel  $(v, w)$

- Zielvariable  $v$
- Originalvariable  $w$

## ACP (*available copies*)

Die Menge der verfügbaren Kopieranweisungen **ACP** sind  
all die  $(v, w)$ , bei denen weder  $v$  noch  $w$  zwischen Definition  
und der betrachteten Stelle des Programmes überschrieben  
wurden.

Realisierung von ACP bestimmt Gesamtlaufzeit des Verfahrens abhängig von Anzahl von Kopieranweisungen  $n$ .

- Lineare Suche:  $O(n^2)$
- Baumstruktur:  $O(n \log n)$
- Hash:  $O(n)$



Hilfsfunktion: Liefere zu verwendenden Operand für `opnd`,  
ggf. ausgetauscht durch eine in ACP vorhandene  
Originalvariable

A. Koch

```
func Copy_Value(opnd, ACP) : Var
  Operand          opnd;
  Set<Pair<Var,Var>> ACP; // Menge der (v,w)
begin
  Pair<Var,Var> acp;      // ein (v,w)
  foreach acp in ACP do
    if opnd.kind == VARIABLE && opnd.name == acp.first() then
      return acp.second(); // gefunden, verwende Originalvar.
    endif
  endfor
  return opnd.name; // Ziel nicht gefunden, alter Opnd. zurück
end
```

Hilfsfunktion: Entferne eine überschriebene Variable  $v$  aus ACP

A. Koch

```
proc Remove_ACP(ACP, varname)
  var Set<Pair<Var,Var>> ACP;
  Var varname;
begin
  Set<Pair<Var,Var>> temp = ACP.copy(); // Löschen bei Iterat.
  Pair<Var,Var> acp; // Paar (v,w)
  foreach acp in temp do
    if acp.first() == varname || acp.second() == varname then
      ACP.remove(acp);
    endif
  endfor
end
```

```
proc Local_Copy_Prop(b)
  Block b;
begin
  Set<Pair<Var,Var>> ACP = Set.empty();
  Instruction i;
  foreach i in b.instructions() do
    if (i instanceof Expression) then // benutzende Auftreten
      if (i == "a + b") then // Bin.Exp.
        i.opnds.a.name := Copy_Value(i.opnds.a.name, ACP);
        i.opnds.b.name := Copy_Value(i.opnds.b.name, ACP);
      else if (i == "-a") then // Un.Exp.
        i.opnds.a.name := Copy_Value(i.opnds.a.name, ACP);
      else if (i == "f(a)") then // List.Exp.
        i.opnds.a.name := Copy_Value(i.opnds.a.name, ACP);
      else if ... // andere lesende Instruktionsarten
      endif
    else if (i == "LHS := RHS") then // Zuweisung
      Remove_ACP(ACP, i.LHS.name); // entferne übersch. Var.
      if (RHS instanceof Var && LHS != RHS) then // Kopie?
        ACP.add(new Pair(LHS, RHS));
      endif
    endif
  endfor
end
```

A. Koch

# Beispiel Lokale CP

| Position | Code Before          | ACP  | Code After           |
|----------|----------------------|--|----------------------|
|          |                      | $\emptyset$                                      |                      |
| 1        | $b \leftarrow a$     |  | $b \leftarrow a$     |
|          |                      | $\{\langle b, a \rangle\}$                       |                      |
| 2        | $c \leftarrow b + 1$ |  | $c \leftarrow a + 1$ |
|          |                      | $\{\langle b, a \rangle\}$                       |                      |
| 3        | $d \leftarrow b$     |  | $d \leftarrow a$     |
|          |                      | $\{\langle b, a \rangle, \langle d, a \rangle\}$ |                      |
| 4        | $b \leftarrow d + c$ |  | $b \leftarrow a + c$ |
|          |                      | $\{\langle d, a \rangle\}$                       |                      |
| 5        | $b \leftarrow d$     |  | $b \leftarrow a$     |
|          |                      | $\{\langle d, a \rangle, \langle b, a \rangle\}$ |                      |

- Basiert auf Datenflussanalyse
  - Welche Kopieranweisungen erreichen Verwendungen ihrer LHS intakt?
  - **Intakt:** Weder LHS noch RHS überschrieben!
- Erweiterte Darstellung  $(v, w, b, p)$ 
  - $b$  ist Block der Zuweisung  $v := w$
  - $p$  ist Position der Zuweisung  $v := w$  innerhalb des Blockes  $b$  (z.B. Nummer der Anweisung)

## COPY( $b$ )

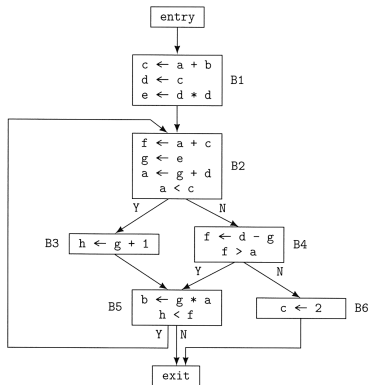
Menge der  $(v, w, b, p)$ , bei denen bei einer Kopieranweisung  $v := w$  im Block  $b$  weder  $v$  noch  $w$  vor Ende des Blockes Ziel einer Zuweisung sind.

## KILL( $b$ )

Menge der  $(t, u, d, q)$  mit  $d \neq b$ , bei denen  $t$  und/oder  $u$  in Block  $d$  Ziel einer Zuweisung sind.

# Beispiel: COPY und KILL

## Beispiel



## Mengen

|               |                       |
|---------------|-----------------------|
| $COPY(entry)$ | $= \emptyset$         |
| $COPY(B1)$    | $= \{(d, c, B1, 2)\}$ |
| $COPY(B2)$    | $= \{(g, e, B2, 2)\}$ |
| $COPY(B3)$    | $= \emptyset$         |
| $COPY(B4)$    | $= \emptyset$         |
| $COPY(B5)$    | $= \emptyset$         |
| $COPY(B6)$    | $= \emptyset$         |
| $COPY(exit)$  | $= \emptyset$         |
| $KILL(entry)$ | $= \emptyset$         |
| $KILL(B1)$    | $= \{(g, e, B2, 2)\}$ |
| $KILL(B2)$    | $= \emptyset$         |
| $KILL(B3)$    | $= \emptyset$         |
| $KILL(B4)$    | $= \emptyset$         |
| $KILL(B5)$    | $= \emptyset$         |
| $KILL(B6)$    | $= \{(d, c, B1, 2)\}$ |
| $KILL(exit)$  | $= \emptyset$         |

A. Koch

## CPIN( $b$ )

Menge von Kopieranweisungen  $(t, u, d, q)$ , die zu Beginn des Blocks  $b$  intakt sind.

## CPOUT( $b$ )

Menge von Kopieranweisungen  $(t, u, d, q)$ , die am Ende eines Blocks  $b$  intakt sind.



## Vorgehensweise bei Aufstellen der Gleichungen

A. Koch

- Nur solche Kopieranweisungen sind am Anfang eines Blockes verfügbar ...
- ... die an **allen** Enden von Vorgängern verfügbar waren
- Startwerte für iterative Lösung
  - $CPIN(entry)$ : Startblock hat keine Kopieranweisungen zur Verfügung
  - $CPIN(b), b \neq entry$ : Alle anderen Blöcke haben **alle** in der ganzen Prozedur auftretenden Kopieranweisungen zur Verfügung
    - Wird schrittweise eingeschränkt

$$\text{CPIN}(b) = \bigcap_{d \in \text{pred}(b)} \text{CPOUT}(d)$$

$$\text{CPOUT}(b) = \text{COPY}(b) \cup (\text{CPIN}(b) - \text{KILL}(b))$$

mit Initialisierung

$$\text{CPIN}(\text{entry}) = \emptyset$$

$$\text{CPIN}(b) = \bigcup_{d \in \text{Blocks}} \text{COPY}(d) \text{ ,für } b \neq \text{entry}$$

# Beispiel: Initialisierung

COPY(entry) =  $\emptyset$   
COPY(B1) =  $\{\langle d, c, B1, 2 \rangle\}$   
COPY(B2) =  $\{\langle g, e, B2, 2 \rangle\}$   
COPY(B3) =  $\emptyset$   
COPY(B4) =  $\emptyset$   
COPY(B5) =  $\emptyset$   
COPY(B6) =  $\emptyset$   
COPY(exit) =  $\emptyset$

CPIN(entry) =  $\emptyset$   
CPIN(B1) =  $\{\langle d, c, B1, 2 \rangle, \langle g, e, B2, 2 \rangle\}$   
CPIN(B2) =  $\{\langle d, c, B1, 2 \rangle, \langle g, e, B2, 2 \rangle\}$   
CPIN(B3) =  $\{\langle d, c, B1, 2 \rangle, \langle g, e, B2, 2 \rangle\}$   
CPIN(B4) =  $\{\langle d, c, B1, 2 \rangle, \langle g, e, B2, 2 \rangle\}$   
CPIN(B5) =  $\{\langle d, c, B1, 2 \rangle, \langle g, e, B2, 2 \rangle\}$   
CPIN(B6) =  $\{\langle d, c, B1, 2 \rangle, \langle g, e, B2, 2 \rangle\}$   
CPIN(exit) =  $\emptyset$

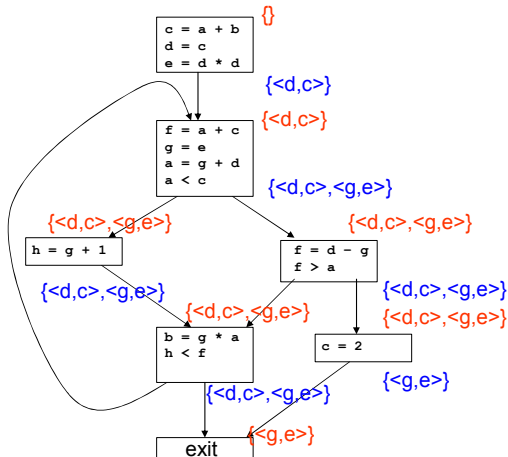
# Einfaches iteratives Lösungsverfahren

```
worklist := cfg.getBlocks();  
  
while ( worklist.notEmpty() ) {  
    b := worklist.removeFirst();  
    recompute CPin(b);  
    recompute CPout(b);  
    if (CPout(b) changed)  
        worklist.add(b.getSuccessors());  
}
```

# Beispiel: Ergebnis der iterativen Berechnung

Rot: CPIN, Blau: CPOUT

A. Koch

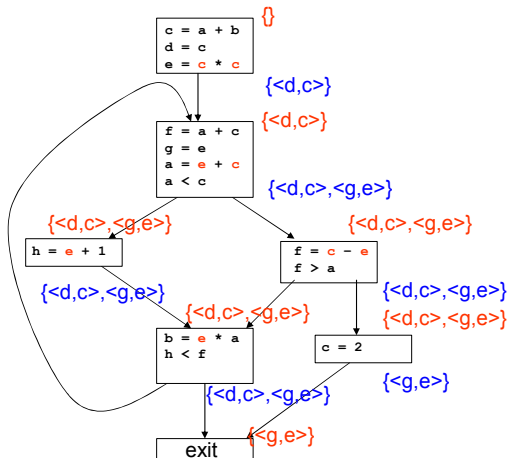


- Werte berechnete Daten nun pro Block aus
- Vorgehen: `Local_Copy_Prop` beginnt nun **nicht** mehr mit leerer ACP-Menge
- ... sondern: Initialisiere ACP-Menge für Block  $b$  aus  $CPIN(b)$
- Analog zu: VN auf EBB und Region (DVNT),  $\rightarrow$  7. Block

# Beispiel: Ergebnis der globalen CP

Rot: CPIN, Blau: CPOUT

A. Koch



# Iterative Datenflussanalyse



- **Treffe Aussagen**
  - ... über Laufzeitverhalten von Programm
  - ... zur Compile-Zeit
- **Mittel der Wahl**
  - Gleichungssysteme
  - Lösungsverfahren: Hier iterative, gibt aber auch andere
- **Anwendung**
  - Finde Anwendungsstellen von Optimierungen
  - Beweise, das Anwendung sicher ist

Weiteres Beispiel: *Live Variables*

# Live Variables

## Live Variable

Eine Variable  $v$  ist *lebendig* (*live*) an einer Stelle  $p$  im Programm genau dann, wenn es im CFG einen Pfad von  $p$  zu einer Verwendung von  $v$  gibt, auf dem  $v$  *nicht* definiert wird.

- Nur live Variables müssen in Prozessorregistern gehalten werden
- Können bei der SSA-Konstruktion zur Eliminierung von Phi-Funktionen dienen
- Können zur Erkennung von uninitialisierten Variablen dienen
  - Lokale Variable ist live bei Prozedureintritt
- Können Basis direkter Optimierungen sein
  - Store-Anweisungen nur für live Variables, überflüssig für andere

## LIVEOUT( $b$ )

Menge aller Variablen, die bei **Austritt** aus Block  $b$  live sind.

A. Koch

Damit Berechnung durch Gleichungssystem.

1. Teil

$\text{LIVEOUT}(b_n) = \emptyset$ , mit  $b_n$  Endknoten des CFG

Bei Prozedurende sind alle (lokalen) Variablen nicht mehr live.

- Beschränkung auf Prozedurebene
- Bei uns vereinfacht: Parameter nicht betrachtet

## 2. Teil: Rekursive Definition für innere Knoten

$$\text{LIVEOUT}(b) = \bigcup_{m \in \text{succ}(b)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

- Rechnet **rückwärts** von Nachfolger zu Vorgängerknoten
- $\text{UEVAR}(m)$  (*upwards exposed*): Vor ihrer Definition in Block  $m$  benutzte Variablen
- $\text{VARKILL}(m)$  sind alle im Block  $m$  definierten Variablen

$$\text{LIVEOUT}(b) = \bigcup_{m \in \text{succ}(b)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

A. Koch

- $\text{LIVEOUT}(m)$  sind alle Variablen, die live am Anfang von Nachfolgerblöcken  $m$  sind
- Variable muss nur auf **einem** Pfad live sein ( $\rightarrow \cup$ )
- Jeder Nachfolgerknoten  $m$  trägt Variablen bei
  - In  $m$  benutzte Variablen, die vorher nicht redefiniert werden ( $\text{UEVAR}(m)$ )
  - Variablen die
    - $m$  selbst live verlassen ( $\text{LIVEOUT}(m)$ )
    - ... und in  $m$  nicht redefiniert werden ( $\text{VARKILL}(m)$ )

- 1 CFG aufbauen
  - Kennen wir bereits, für strukturierte Sprachen einfach
  - Falls nötig um einen eindeutigen Endknoten anreichern
- 2 Per-Block Daten vorberechnen (UEVAR und VARKILL)
- 3 Iterativen Fixpunkt-Algorithmus für LIVEOUT anwenden



$UEVAR(b) := \emptyset$   
 $VARKILL(b) := \emptyset$

```
for  $i := 1$  to number of operations in block  $b$  do  
  parse operation  $i$  into " $LHS := RHS$ "  
  for  $v \in$  variables referenced in  $RHS$  do  
    if  $v \notin VARKILL(b)$  then  
       $UEVAR(b) := UEVAR(b) \cup \{v\}$   
       $VARKILL(b) := VARKILL(b) \cup \{variable(LHS)\}$ 
```

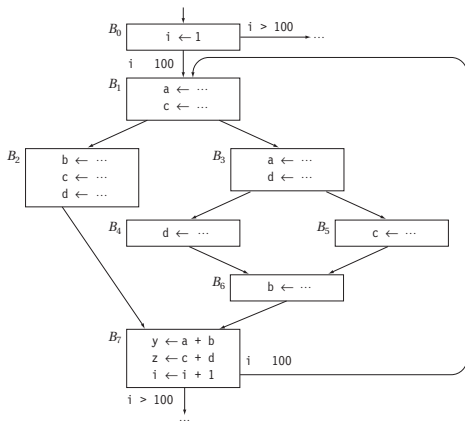
A. Koch

Hier vereinfacht: Nur Zuweisungen in Block  
Analoges Vorgehen für andere Operationen, unterscheide

- Lesen (RHS) von Variablen
- Schreiben (LHS) von Variablen

# Live Variables - Beispiel 1

A. Koch



|   | $B_0$                  | $B_1$               | $B_2$            | $B_3$               | $B_4$                  | $B_5$                  | $B_6$                  | $B_7$               |
|---|------------------------|---------------------|------------------|---------------------|------------------------|------------------------|------------------------|---------------------|
| <b>UEVAR</b>                                  | $\emptyset$            | $\emptyset$         | $\emptyset$      | $\emptyset$         | $\emptyset$            | $\emptyset$            | $\emptyset$            | $\{a, b, c, d, i\}$ |
| <b><math>\overline{\text{VARKILL}}</math></b> | $\{a, b, c, d, y, z\}$ | $\{b, d, i, y, z\}$ | $\{a, i, y, z\}$ | $\{b, c, i, y, z\}$ | $\{a, b, c, i, y, z\}$ | $\{a, b, d, i, y, z\}$ | $\{a, c, d, i, y, z\}$ | $\{a, b, c, d\}$    |

```
 $N := \text{number of blocks} - 1$   
for  $i := 0$  to  $N$  do  
    LIVEOUT( $i$ ) :=  $\emptyset$   
    changed := true  
    while changed do  
        changed := false  
        for  $i := 0$  to  $N$  do  
            recompute LIVEOUT( $i$ )  
            if LIVEOUT( $i$ ) changed then  
                changed := true
```

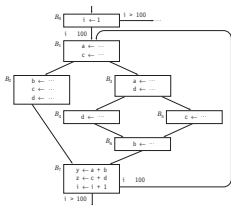
# Live Variables - Beispiel 2

LIVEOUT( $b$ ) =

$$\bigcup_{m \in \text{succ}(b)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

A. Koch

|                             | $B_0$                  | $B_1$               | $B_2$            | $B_3$               | $B_4$                  | $B_5$                  | $B_6$                  | $B_7$               |
|-----------------------------|------------------------|---------------------|------------------|---------------------|------------------------|------------------------|------------------------|---------------------|
| UEVAR                       | $\emptyset$            | $\emptyset$         | $\emptyset$      | $\emptyset$         | $\emptyset$            | $\emptyset$            | $\emptyset$            | $\{a, b, c, d, i\}$ |
| $\overline{\text{VARKILL}}$ | $\{a, b, c, d, y, z\}$ | $\{b, d, i, y, z\}$ | $\{a, i, y, z\}$ | $\{b, c, i, y, z\}$ | $\{a, b, c, i, y, z\}$ | $\{a, b, d, i, y, z\}$ | $\{a, c, d, i, y, z\}$ | $\{a, b, c, d\}$    |



Quelle: C&T, pp.442-443

| Iteration | LIVEOUT( $n$ ) |               |                     |                  |                  |                  |                     |             |
|-----------|----------------|---------------|---------------------|------------------|------------------|------------------|---------------------|-------------|
|           | $B_0$          | $B_1$         | $B_2$               | $B_3$            | $B_4$            | $B_5$            | $B_6$               | $B_7$       |
| 0         | $\emptyset$    | $\emptyset$   | $\emptyset$         | $\emptyset$      | $\emptyset$      | $\emptyset$      | $\emptyset$         | $\emptyset$ |
| 1         | $\emptyset$    | $\emptyset$   | $\{a, b, c, d, i\}$ | $\emptyset$      | $\emptyset$      | $\emptyset$      | $\{a, b, c, d, i\}$ | $\emptyset$ |
| 2         | $\emptyset$    | $\{a, i\}$    | $\{a, b, c, d, i\}$ | $\emptyset$      | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\{i\}$     |
| 3         | $\{i\}$        | $\{a, i\}$    | $\{a, b, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\{i\}$     |
| 4         | $\{i\}$        | $\{a, c, i\}$ | $\{a, b, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\{i\}$     |
| 5         | $\{i\}$        | $\{a, c, i\}$ | $\{a, b, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\{i\}$     |

# Diskussion

Vor Benutzung berücksichtigen:

- Terminiert die Analyse?
- Beantwortet das berechnete Ergebnis die gestellte Frage?
- Wie schnell läuft die Analyse?

Im folgenden Diskussion am Beispiel LIVEOUT.

- LIVEOUT Mengen wachsen monoton, beginnend bei  $\emptyset$
- Sie können nie schrumpfen
- Bei maximaler Größe umfasst eine LIVEOUT-Menge **alle** Variablen
- Da es nur endlich viele Variablen gibt, sind die LIVEOUT-Mengen beschränkt
- Die Iteration bricht also nach endlicher Zeit immer ab
  - Irgendwann ändert sich nichts mehr
  - Worst-case: Alle LIVEOUT-Mengen umfassen alle Variablen

- LIVEOUT berechnet lokale Eigenschaft
  - Zwischen Block und seinen Nachfolgern
- Vereinigt Ergebnisse der Nachfolger
  - Wenn  $v$  live auf irgendeiner Nachfolgekante ist, dann  $v$  in LIVEOUT
- Kann Zusammenhang zwischen lokalen Eigenschaften und der Definition von Live Variables hergestellt werden?
  - Diese ist ja über alle Pfade definiert!
- Beweis über Verbandalgebra (*lattice algebra*)
  - Hier nicht behandelt ( $\rightarrow$  Kam/Ullman JACM 1976)



- Überlegung: Das Ergebnis der iterativen Lösung des Datenflußproblems ist **unabhängig** von der Bearbeitungsreihenfolge der Blöcke
- Die Reihenfolge beeinflusst aber die nötige Anzahl von Iterationen
- Also: Suche nach schnellerer Abarbeitungsreihenfolge
- Idee: Bei Vorgehen ...
  - ... rückwärts (LIVEOUT): Besuche so viele **Nachfolger** eines Knotens wie möglich, bevor der Knoten selbst besucht wird

## Verschiedene Möglichkeiten für Abarbeitungsreihenfolgen

- Vorwärts: z.B. Breadth-First-Search, aber besser **Reverse Post-Order** (RPO)

A. Koch

## Beispiel: Reverse Post-Order

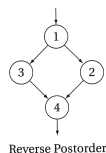
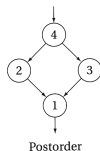
### Step1: PostOrder

```
proc main() ≡  
  count ← 1  
  Visit(Entry)  
end
```

```
proc Visit(v) ≡  
  mark v as visited  
  foreach successor s of v not yet visited  
    Visit(s)  
end  
  PostOrder(v) ← count ++  
end
```

### Step 2: rPostOrder

```
foreach v ∈ V do  
  rPostOrder(v) ← |V| - PostOrder(v)  
end
```

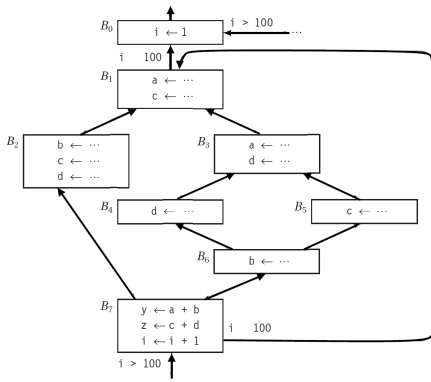


# Effizienz der Datenflußanalyse 3

Rückwärts

- z.B. Depth-First Search
- besser RPO auf **reversem CFG** (Kanten umgekehrt)

A. Koch



Post-Order auf rev. CFG:  $B_0, B_1, B_2, B_3, B_5, B_4, B_6, B_7$

RPO auf rev. CFG:  $B_7, B_6, B_5, B_4, B_2, B_3, B_1, B_0$

Abspeichern als Permutation in Array  $P = [7, 6, 5, 4, 2, 3, 1, 0]$

A. Koch

```
 $N :=$  number of blocks - 1  
for  $i := 0$  to  $N$  do  
  LIVEOUT( $i$ ) :=  $\emptyset$   
  changed := true  
  while changed do  
    changed := false  
    for  $i := 0$  to  $N$  do  
      recompute LIVEOUT( $P[i]$ )  
      if LIVEOUT( $P[i]$ ) changed then  
        changed := true
```

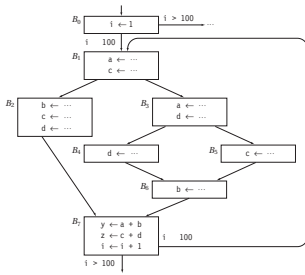
LIVEOUT( $b$ ) =

$$\bigcup_{m \in \text{succ}(b)} \text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)})$$

A. Koch

|                             | $B_0$                  | $B_1$               | $B_2$            | $B_3$               | $B_4$                  | $B_5$                  | $B_6$                  | $B_7$                       |
|-----------------------------|------------------------|---------------------|------------------|---------------------|------------------------|------------------------|------------------------|-----------------------------|
| UEVAR                       | $\emptyset$            | $\emptyset$         | $\emptyset$      | $\emptyset$         | $\emptyset$            | $\emptyset$            | $\emptyset$            | $\{a, b, c\}$<br>$\{d, i\}$ |
| $\overline{\text{VARKILL}}$ | $\{a, b, c, d, y, z\}$ | $\{b, d, i, y, z\}$ | $\{a, i, y, z\}$ | $\{b, c, i, y, z\}$ | $\{a, b, c, i, y, z\}$ | $\{a, b, d, i, y, z\}$ | $\{a, c, d, i, y, z\}$ | $\{a, b, c, d\}$            |

Reihenfolge: B7, B6, B5, B4, B2, B3, B1, B0

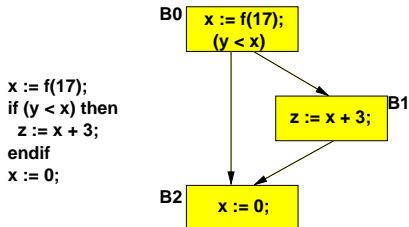


| Iteration | LIVEOUT( $n$ ) |               |                     |                  |                  |                  |                     |             |
|-----------|----------------|---------------|---------------------|------------------|------------------|------------------|---------------------|-------------|
|           | $B_0$          | $B_1$         | $B_2$               | $B_3$            | $B_4$            | $B_5$            | $B_6$               | $B_7$       |
| 0         | $\emptyset$    | $\emptyset$   | $\emptyset$         | $\emptyset$      | $\emptyset$      | $\emptyset$      | $\emptyset$         | $\emptyset$ |
| 1         | $\{i\}$        | $\{a, c, i\}$ | $\{a, b, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\emptyset$ |
| 2         | $\{i\}$        | $\{a, c, i\}$ | $\{a, b, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\{i\}$     |
| 3         | $\{i\}$        | $\{a, c, i\}$ | $\{a, b, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, c, d, i\}$ | $\{a, b, c, d, i\}$ | $\{i\}$     |

Konvergiert jetzt in 3 Iterationen (statt 5)!

- Fundamentale Annahme bei Datenflußberechnung:
- **Alle** Blöcke können ausgeführt werden

# Gegenbeispiel zur Annahme 1



- $x$  ist Live in B0, da es in B1 gelesen werden kann
- $x$  wird aber in B2 Killed
- Falls B1 nie ausgeführt wird, ist  $x$  nicht Live außerhalb von B0

Falls der Compiler beweisen kann, dass immer  $y \geq x \dots$

- würde die Anweisung  $z := x+3$  nie ausgeführt werden
- Falls dann auch noch der Aufruf  $f(17)$  keine Seiteneffekte hat
- ... können Blocks B0 und B1 komplett entfernt werden

Kann aber nicht allgemein gelöst werden ( $\rightarrow$  Halteproblem)!



- LIVEOUT: Wird immer über **alle** Nachfolger berechnet
- Berechnet wird so nur eine **Zusammenfassung** der tatsächlich möglichen Abläufe

## Probleme bei Arrays

- Zugriff  $\mathbf{A}[i, j, k]$  auf ein einzelnes Element
- Datenflussanalyse kennt aber keine konkreten Werte für  $i, j, k$
- Abstraktion: Betrachte **gesamtes** Array als **eine** Variable
  - $\dots := \mathbf{A}[i, j, k]$  zählt als Verwendung des **gesamten** Arrays
  - $\mathbf{A}[i, j, k] := \dots$  zählt als Definition des **gesamten** Arrays

Benutzung dieser ungenauen Ergebnisse muß **konservativ** erfolgen!

A. Koch

- Fehlabschätzungen dürfen Korrektheit der Analyse in Bezug auf die gesuchte Aussage nicht beeinflussen
- Beispiele
  - Kann der Wert von  $A[i, j, k]$  nach Schreibzugriff auf  $A[1, m, n]$  verworfen werden?
  - ... Nein, denn der Schreibzugriff KILLED **nicht** notwendigerweise  $A[i, j, k]$ !
  - Könnte der Wert von  $A[i, j, k]$  nach Schreibzugriff auf  $A[1, m, n]$  beschädigt werden?
  - ... Ja, denn der Schreibzugriff **könnte** jedes Element von A verändern!

## Analoge Problematik bei Zeigern

A. Koch

- Zuweisung via Zeiger kann potentiell **jede** Variable beeinflussen
- Kann weite Teile der Datenflussanalyse unbrauchbar machen
- Wird schlimmer bei Adressarithmetik (wie in C)
  - Nun nicht nur auf einzelne Variablen, sondern beliebig im Speicher
- Wird etwas besser bei fester Typisierung (keine Wandlung möglich)
  - Nun nur noch Variablen vom Typ des Zeigers betroffen

## Prozeduren

A. Koch

- Auch bei Beschränkung der Analyse auf eine Prozedur
  - Jeder Prozeduraufruf **kann** verändern (abhängig von Sprache):
    - Nur Var-Parameter
    - Nicht-Lokale Variablen
    - Globale Variablen
    - Bei Unterstützung von Zeigern: Gesamten Speicherinhalt
  - Unterprozeduren verkomplizieren die Situation noch
- ➔ Analyse muss “worst case” Annahmen machen

# Sammlung von Datenflußproblemen

- Available Expressions
- $AVAIL(b)$ : Menge der Ausdrücke, die Block  $b$  erreichen
- Genauer im 7. Block (Einführung in Code-Optimierung)
- **Vorwärtsgerichteter** Fluß über berechnete **Ausdrücke**
- Konkrete Anwendung:  
Global Common Subexpression Elimination

Eine Definition  $d$  einer Variablen  $v$  **erreicht** eine Operation  $i$  genau dann, wenn  $v$  in  $i$  gelesen wird und  $v$  auf einem Pfad von  $d$  zu  $i$  **nicht** redefiniert wird.

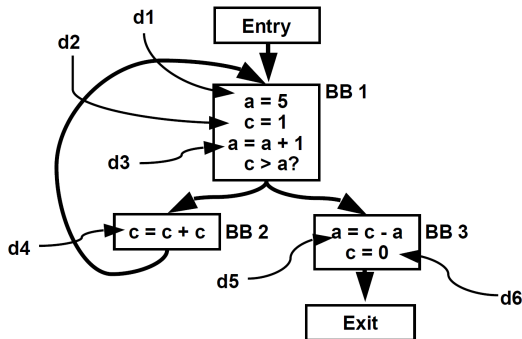
- $\text{REACHES}(b)$ : Menge der Definitionen, die Block  $b$  erreichen.
- **Vorwärtsgerichteter** Fluß über **Zuweisungen an Variablen**
- *Reaching Definitions*



# Erreichende Definitionen 2

DEDEF( $b$ ) (*downward exposed definitions*): Definitionen in  $b$ , die nicht vor Blockende überschrieben werden

A. Koch



$$\text{DEDEF}(BB1) = \{d2, d3\}$$

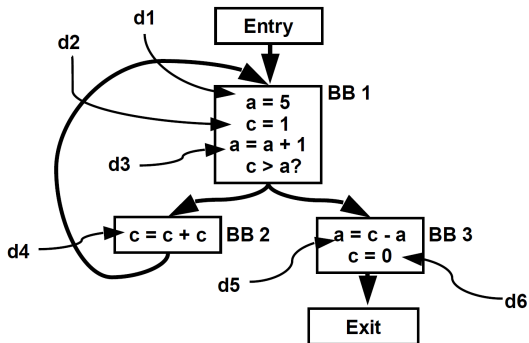
$$\text{DEDEF}(BB2) = \{d4\}$$

$$\text{DEDEF}(BB3) = \{d5, d6\}$$

# Erreichende Definitionen 3

$DEFKILL(b)$ : Im Block  $b$  überschriebene Definitionen anderer Blöcke aus der Menge **aller** Definitionen in der Prozedur

A. Koch



$$DEFKILL(BB1) = \{d5, d4, d6\}$$

$$DEFKILL(BB2) = \{d2, d6\}$$

$$DEFKILL(BB3) = \{d1, d3, d2, d4\}$$

## Datenflußgleichungen

$$\text{REACHES}(b_0) = \emptyset$$

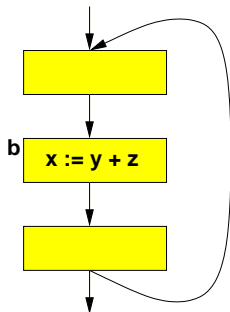
$$\text{REACHES}(b) = \bigcup_{d \in \text{preds}(b)} (\text{DEDEF}(d) \cup (\text{REACHES}(d) \cap \overline{\text{DEFKILL}(d)}))$$

- Lösung mit iterativem Fixpunktverfahren
- Startwerte:  $\text{REACHES}(b) = \emptyset$  für alle  $b$

Anwendungsbeispiel: Anweisung  $x := y + z$   
in Schleifen-Body  $b$

A. Koch

- Falls alle  $\text{REACHES}(b)$  für  $y$  und  $z$  **außerhalb** der Schleife
- ... kann gesamte Berechnung von  $x$  **vor** die Schleife gezogen werden
- Loop-Invariant Code Motion



## Sehr Rege (*very busy*)

A. Koch

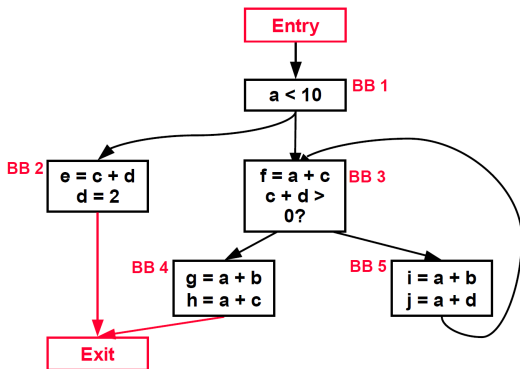
Ein Ausdruck  $e$  ist **sehr rege** am Ende eines Blocks  $b$ , wenn er in allen Nachfolgern von  $b$  evaluiert und benutzt wird, und das einmalige Evaluieren von  $e$  am Ende von  $b$  das gleiche Ergebnis hätte wie die erstmalige Evaluation von  $e$  in den Nachfolgern von  $b$ .

- $\text{VERYBUSY}(b)$ : Menge der **Ausdrücke**, die am Ende von  $b$  sehr rege sind
- **Rückwärtsgerichteter** Fluß über **Ausdrücke**
- *Very Busy Expressions*

# Sehr rege Ausdrücke 2

$UEEXPR(b)$  (*upwards exposed expressions*): In  $b$  **vor**  
Überschreiben ihrer Operanden benutzte Ausdrücke.

A. Koch



$$UEEXPR(BB1) = \emptyset$$

$$UEEXPR(BB2) = \{c + d\}$$

$$UEEXPR(BB3) = \{a + c, c + d\}$$

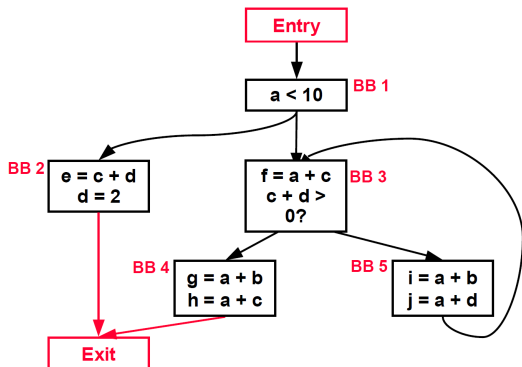
$$UEEXPR(BB4) = \{a + b, a + c\}$$

$$UEEXPR(BB5) = \{a + b, a + d\}$$

# Sehr rege Ausdrücke 3

EXPRKILL( $b$ ): Im Block  $b$  durch Überschreiben der Operanden unbrauchbar gemachte Ausdrücke

A. Koch



$$\text{EXPRKILL}(BB1) =$$

$\emptyset$

$$\text{EXPRKILL}(BB2) =$$

$\{a + d, c + d\}$

$$\text{EXPRKILL}(BB3) =$$

$\emptyset$

$$\text{EXPRKILL}(BB4) =$$

$\emptyset$

$$\text{EXPRKILL}(BB5) =$$

$\emptyset$

## Datenflußgleichungen

$$\text{VERYBUSY}(b_n) = \emptyset$$

$$\text{VERYBUSY}(b) =$$

$$\bigcap_{d \in \text{succ}(b)} (\text{UEEXPR}(d) \cup (\text{VERYBUSY}(d) \cap \overline{\text{EXPRKILL}(d)}))$$

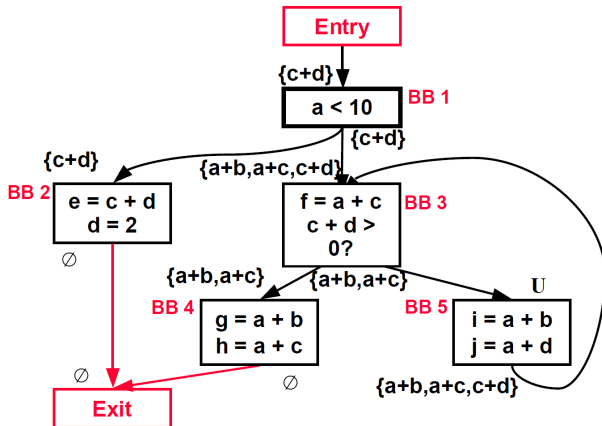
- Lösung mit iterativem Fixpunktverfahren
- Startwert für  $b \neq b_n$ :  $\text{VERYBUSY}(b) = U$ , mit  $U$  Menge aller Ausdrücke in Prozedur



# Sehr rege Ausdrücke 5

## Konkretes Beispiel

A. Koch



- Anwendung zur Optimierung: Code Hoisting
- Ersetze Evaluationen der sehr regen Ausdrücke in Nachfolgern
- ... durch eine Evaluation in Vorgänger
- Macht Code nicht (direkt) schneller, aber **kleiner**

# Verallgemeinerung

## Ein Pfad-Vorwärts: Reaching definitions

$$\text{REACHES}(b) = \bigcup_{d \in \text{preds}(b)} (\text{DEDEF}(d) \cup (\text{REACHES}(d) \cap \overline{\text{DEFKILL}(d)}))$$

A. Koch

## Ein Pfad-Rückwärts: Live variables

$$\text{LIVEOUT}(b) = \bigcup_{m \in \text{succ}(b)} (\text{UEVAR}(m) \cup (\text{LIVEOUT}(m) \cap \overline{\text{VARKILL}(m)}))$$

## Alle Pfade-Vorwärts: Available expressions ( $\rightarrow$ 7. Block)

$$\text{AVAIL}(b) = \bigcap_{d \in \text{preds}(b)} (\text{DEEXPR}(d) \cup (\text{AVAIL}(d) \cap \overline{\text{EXPRKILL}(d)}))$$

## Alle Pfade-Rückwärts: Very busy expressions

$$\text{VERYBUSY}(b) = \bigcap_{d \in \text{succ}(b)} (\text{UEEXPR}(d) \cup (\text{VERYBUSY}(d) \cap \overline{\text{EXPRKILL}(d)}))$$

- Sehr ähnliche Struktur der Gleichungen
  - $f(x) = c_1 \text{ op}_1 (x \text{ op}_2 c_2)$
- Wie ausnutzen?
- Lösung **aller** solcher Datenflußprobleme
- **Data Flow Framework**
- Akzeptiert  $c_1, c_2, \text{op}_1, \text{op}_2$ , Konfluenzoperator als Parameter
- Lösen dann für Fixpunkt
- Vorteil: Nur ein Algorithmus muß mit viel Sorgfalt implementiert werden
- Kann dann alle vergleichbaren Probleme lösen

Es gibt aber auch Datenflußprobleme mit anderer Struktur!

- *Constant Propagation*
- Weiterführung von Constant Folding
- Nun hinweg über Anweisungsgrenzen und Merge Points
- Darstellung durch Paare  $(v, c)$ 
  - $v$  ist Variable
  - $c$  ist entweder Konstante, oder  $\perp$  (unbekannter Wert)
- $\text{CONSTANTS}(b)$  sind alle bisher gesammelten Aussagen zu Beginn des Blocks  $b$
- Damit darstellbar:
  - Keine Aussage über  $v$  machbar:  $(v, c) \notin \text{CONSTANTS}(b)$
  - $v$  ist konstant mit Wert  $c$ :  $(v, c) \in \text{CONSTANTS}(b)$
  - $v$  hat unbekanntem (potentiell variablen) Wert:  $(v, \perp) \in \text{CONSTANTS}(b)$

- Anfangs ist  $\text{CONSTANTS}(b) = \emptyset$
- Dann in Reihenfolge Anweisungen in jedem Block  $b$  untersuchen

Für  $x := y$

```
if  $(x,c) \in \text{CONSTANTS}(b)$  do  
     $\text{CONSTANTS}(b) := \text{CONSTANTS}(b) - \{(x,c)\}$   
if  $(y,c) \in \text{CONSTANTS}(b)$  do  
     $\text{CONSTANTS}(b) := \text{CONSTANTS}(b) \cup \{(x,c)\}$ 
```

Für  $x := y \text{ op } z$

A. Koch

**if**  $(x,c) \in \text{CONSTANTS}(b)$  **do**

$\text{CONSTANTS}(b) := \text{CONSTANTS}(b) - \{(x,c)\}$

**if**  $(y,c_1) \in \text{CONSTANTS}(b) \wedge (z,c_2) \in \text{CONSTANTS}(b)$  **do**

$\text{CONSTANTS}(b) := \text{CONSTANTS}(b) \cup \{(x, c_1 \text{ op } c_2)\}$

- Mit  $\perp \text{ op } x = x \text{ op } \perp = \perp$
- Analog  $x := y \text{ op } \text{Const}$ .
- Hier auch Sonderregeln möglich
  - $c \cdot 0 = 0, c - c = 0, c \cdot 1 = c, \dots$
- Transformation von  $\text{CONSTANTS}(b)$  in Block  $b$ :  
 $\text{CONSTANTS}_{\text{out}}(b) = F_b(\text{CONSTANTS}_{\text{in}}(b))$



Bei Überschreiten von Blockgrenzen:  
Mehrere Aussagen  $C_{out,b}$  treffen zusammen

A. Koch

Konfluenzoperator ist  $\wedge$  (*meets*, Infimum, Durchschnitt)

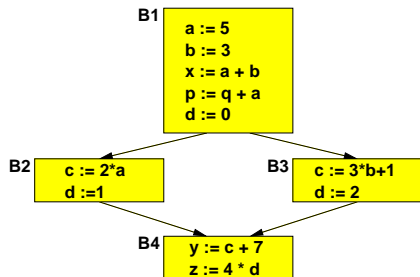
## Definition Meets-Operator

$$(v, c_1) \wedge (v, c_2) = \begin{cases} (v, c_1) & : \text{wenn } c_1 = c_2 \\ (v, \perp) & : \text{sonst} \end{cases}$$

Damit vorwärtsgerichtetes Datenflußproblem formulierbar:

$$\text{CONSTANTS}(b) = \bigwedge_{d \in \text{preds}(b)} F_d(\text{CONSTANTS}(d))$$

# Konstanten propagieren 5



Cout(B1)={(a,5),(b,3),(d,0),(x,8)}

Cout(B2)={(a,5),(b,3),(c,10),(d,1),(x,8)}

Cout(B3)={(a,5),(b,3),(c,10),(d,2),(x,8)}

Cout(B4)={(a,5),(b,3),(c,10),(d,⊥),(y,17),(x,8),(z,⊥)}

- Beachte: **Keine** Aussage über **p** möglich
- Grund: Keine Aussage über **q** möglich

- $\text{CONSTANTS}(b)$  kann groß werden, ist aber endlich
- Nur drei **aufeinanderfolgende** Zustände einer Variable
  - 1 Keine Aussage
  - 2 Konstant
  - 3 Variabel
- Relevanz
  - Hier nur Beispiel für ungewöhnlicheres DF-Problem
- Besser: Sparse Conditional Constant Propagation
  - Ignoriert Einfluß nicht-ausführbarer Blöcke

# Zusammenfassung

- Aufräumen nach Optimierung: Copy Propagation
- Iterative Datenflußanalyse
  - Live Variables
  - Erreichende Definitionen
  - Sehr rege Ausdrücke
  - Konstanten propagieren
- Diskussion
  - Reihenfolge
  - Schwächen
  - Gemeinsamkeiten